Notes for CSCI 303-01, Algorithms

Robert Michael Lewis
Department of Computer Science
College of William & Mary

Spring 2026

Version: March 28, 2026 13 : 51

Contents

Preface

These are supplemental notes for CSCI 303, the standard undergraduate algorithms course at William & Mary.

Following [16], we will use 😱 to indicate technical material that may be skipped on first reading. We will also borrow the practice of Bourbaki[1] and Donald Knuth[2] and use the metric road sign for "dangerous curve" to indicate an important point (or opinion) the reader should note:

 Do not mix handguns and tequila.

The use of two warning signs indicates an even more important point:

 Do not even think about putting ketchup on hot dogs.

[1] Nicolas Bourbaki is the pseudonym of a group of mathematicians founded in France in 1935. The group took its name from Charles Denis Bourbaki, a French general of the Franco–Prussian War.

[2] Donald Knuth (1938–), American computer scientist.

Chapter 0

Mathematical background

This chapter reviews some mathematical results we will need later.

0.1 Logarithms

We will use lg to denote the base-2 logarithm. This usage is common in computer science, if not entirely standard, and is the prescription of the *Chicago Manual of Style*.[1] We will use the standard notation ln for the natural logarithm.

The following proposition says that the logarithms in two different bases are related by a constant factor, so in this sense it does not matter which base we use.

Proposition 0.1.1. *Suppose $a, b, c > 0$. Then*

$$\log_a c = \log_a b \ \log_b c.$$

Proof We have

$$c = a^{\log_a c},$$
$$c = b^{\log_b c}.$$

Taking the base-a logarithm of both expressions yields

$$\log_a c = \log_a c,$$
$$\log_a c = \log_a b^{\log_b c} = \log_b c \ \log_a b,$$

so

$$\log_a c = \log_a b \ \log_b c.$$

■

As a result of this proposition we have

$$\ln c = \log_e c = \log_e 2 \ \lg c = \ln 2 \lg c. \tag{0.1.1}$$

Since any two logarithms are related by a constant factor, we are free to choose a base that is convenient (typically base-2).

Here is another useful relationship that we will see when solving recurrence relations.

Proposition 0.1.2. *Suppose $a, b, c > 0$. Then*

$$a^{\log_b c} = c^{\log_b a}. \tag{0.1.2}$$

Proof We have

$$a^{\log_b c} = (b^{\log_b a})^{\log_b c} = b^{\log_b a \ \log_b c} = b^{\log_b c \ \log_b a} = (b^{\log_b c})^{\log_b a} = c^{\log_b a}.$$

■

Example 0.1.3. *If $a = 7$ and $b = 2$, then $7^{\log_2 c} = c^{\log_2 7}$.*

[1] The ISO 80000 standard for mathematical notation specifies lb for the binary logarithm, but I have never seen this used.

0.2 Geometric series

A geometric series is a sum of the form

$$1 + r + r^2 + r^3 + \cdots + r^n$$

where $r \neq 1$. Let $S_n = 1 + r + r^2 + r^3 + \cdots + r^n$ and observe that

$$rS_n = r + r^2 + r^3 + r^4 + \cdots + r^{n+1}.$$

Thus,

$$S_n - rS_n = 1 - r^{n+1},$$

whence

$$S_n = \frac{1 - r^{n+1}}{1 - r} = \frac{r^{n+1} - 1}{r - 1}.$$

We summarize this in the following proposition.

Proposition 0.2.1 (Geometric series). *If $r \neq 1$, then*

$$1 + r + r^2 + r^3 + \cdots + r^n = \frac{r^{n+1} - 1}{r - 1}.$$

The special case $r = 2$ yields the familiar formula

$$1 + 2 + 2^2 + 2^3 + \cdots + 2^n = 2^{n+1} - 1.$$

0.3 Sums of powers of integers

Let $S_k(n)$ denote the sum of the k-th powers of the first n integers:

$$S_k(n) = \sum_{i=1}^{n} i^k.$$

It is straightforward to obtain bounds on this sum by viewing it as a rectangular rule approximation of the integral of x^k. We illustrate this in Figures 0.3.1–0.3.2 for $k = 2$ and $n = 7$. From Figure 0.3.1 we see that

$$1^k + 2^k + \cdots + n^k < \int_0^{n+1} x^k \, dx = \frac{(n+1)^{k+1}}{k+1}. \tag{0.3.1}$$

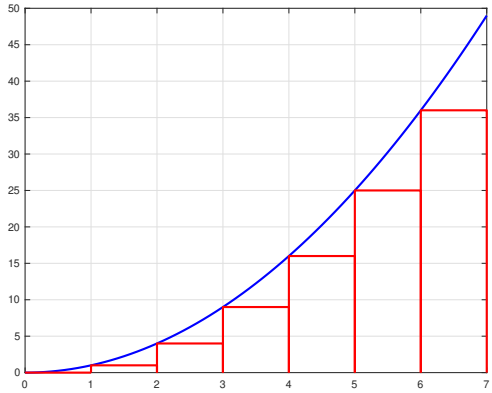


Figure 0.3.1: $1^k + 2^k + \cdots + n^k$ is an underestimate of $\int_1^{n+1} x^k \, dx$.

Meanwhile, from Figure 0.3.2 it is easy to see that

$$1^k + 2^k + \cdots + n^k \geq \int_0^n x^k \, dx = \frac{n^{k+1}}{k+1} \tag{0.3.2}$$

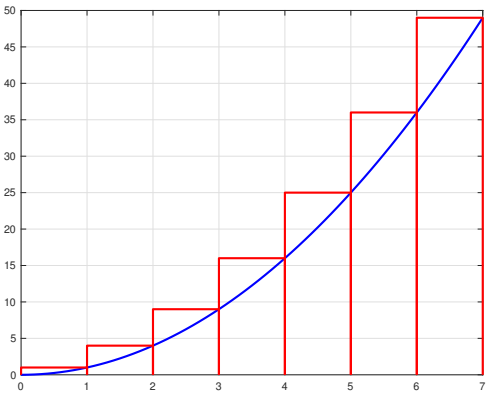


Figure 0.3.2: $1^k + 2^k + \cdots + n^k$ is an overestimate of $\int_0^n x^k \, dx$.

We summarize (0.3.1) and (0.3.2) as follows.

Proposition 0.3.1. *For $k \geq 1$,*

$$\frac{n^{k+1}}{k+1} < \sum_{i=1}^{n} i^k < \frac{(n+1)^{k+1}}{k+1}.$$

For instance, when $k = 1$ we have the well-known formula

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

It is straightforward to check that Proposition 0.3.1 holds in this case:

$$\frac{n^2}{2} \leq \frac{n(n+1)}{2} \leq \frac{(n+1)^2 - 1}{2} = \frac{n(n+2)}{2}.$$

There is an extensive literature on exact formulae for $S_k(n)$. We examine one general approach. Because $S_k(1) = 1$ and $S_k(n+1) = S_k(n) + (n+1)^k$ we surmise that $S_k(n)$ is a polynomial in n. Moreover, since

$$S_k(n) \leq n \cdot n^k = n^{k+1}$$

for all n, we see that $S_k(n)$ is a polynomial of degree of at most $k+1$. This suggests the following approach to deriving formulae for $S_k(n)$. Consider

$$S_1(n) = \sum_{i=1}^{n} i.$$

We posit that this is a polynomial of degree 2:

$$S_1(n) = an^2 + bn + c.$$

Can we find a, b, c that makes this formula work? We must have

$$S_1(n+1) = a(n+1)^2 + b(n+1) + c = S_1(n) + (n+1) = an^2 + bn + c + (n+1),$$

so

$$a(n^2 + 2n + 1) + b(n + 1) + c = an^2 + bn + c + (n + 1)$$
$$a(2n + 1) + b = n + 1,$$

or

$$(2a - 1)n + (a + b - 1) = 0.$$

For this equation to hold for all n we must have

$$2a - 1 = 0$$
$$a + b - 1 = 0,$$

whence

$$a = b = \frac{1}{2}.$$

Finally, since

$$S_1(1) = \sum_{i=1}^{1} i = 1 = an^2 + bn + c = \frac{1}{2} + \frac{1}{2} + c,$$

we see that $c = 0$. Thus we arrive at the familiar formula

$$S_1(n) = \sum_{i=1}^{n} i = \frac{1}{2}n^2 + \frac{1}{2}n = \frac{n(n + 1)}{2}.$$

In general, for the sum of the k-th powers of the first n integers we would assume that $S_k(n)$ is a polynomial of degree $k + 1$. For example, for the sum of squares of the first n integers we would begin with the hypothesis that this sum is a polynomial of degree 2:

$$S_2(n) = an^3 + bn^2 + cn + d.$$

We then use what we know about $S_2(n)$ to determine a, b, c, d.

0.4 Approximations of $\ln n!$

We will be interested in approximations of $\ln n!$. Our first result is the following:

Theorem 0.4.1.

$$\lim_{n\to\infty} \frac{\ln n!}{n \ln n} = 1.$$

In this sense we have the approximation $\ln n! \approx n \ln n$ for large n.

Proof Since

$$\ln n! = \sum_{n=1}^{n} \ln n$$

and $\ln x$ is strictly increasing, we have

$$\sum_{1}^{n} \ln n \leq \int_{1}^{n} \ln x \, dx \leq \sum_{2}^{n+1} \ln n = \sum_{1}^{n} \ln n + \ln(n + 1).$$

Now compute:

$$\int_1^n \ln x \, dx = (x \ln x - x)\Big|_1^n = n \ln n - n + 1$$

Thus,

$$\ln n! \le n \ln n - n + 1 \le \ln n! + \ln(n+1).$$

Now divide by $n \ln n$:

$$\frac{\ln n!}{n \ln n} \le \frac{n \ln n - n + 1}{n \ln n} \le \frac{\ln n!}{n \ln n} + \frac{\ln(n+1)}{n \ln n}.$$

Taking the limit as $n \to \infty$, we see that

$$\lim_{n\to\infty} \frac{\ln n!}{n \ln n} \le 1 \le \lim_{n\to\infty} \frac{\ln n!}{n \ln n},$$

whence

$$\lim_{n\to\infty} \frac{\ln n!}{n \ln n} = 1.$$

■

The approximation in Theorem 0.4.1 is not particularly close unless n is quite large:

n	$\ln n!/(n \ln n)$
10	0.66
100	0.79
1,000	0.86
10,000	0.89
100,000	0.91
1,000,000	0.93

Table 0.1: Comparison of $\ln n!$ and $n \ln n$.

Stirling's approximation for $n!$ is a sharper result. For a short proof, see [34].

Theorem 0.4.2 (Stirling's approximation).

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\, e^{-n} n^n} = 1.$$

That is,

$$n! \approx \sqrt{2\pi n}\, e^{-n} n^n$$

when n is large.

Stirling's approximation is surprisingly good even for small values of n, with the relative error rapidly decreasing to less than 1%:

n	$n!$	$\sqrt{2\pi n}e^{-n}n^n$	relative error
1	1	0.92	0.078
2	2	1.92	0.040
3	6	5.84	0.027
4	24	23.51	0.021
5	120	118.02	0.017
6	720	710.08	0.014
7	5040	4980.40	0.012
8	40320	39902.40	0.010
9	362880	359536.87	0.009
10	3628800	3598695.62	0.008

Table 0.2: Comparison of $n!$ and Stirling's approximation.

0.5 The harmonic series

The harmonic series is

$$\sum_{k=1}^{\infty} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots$$

Let H_n denote the partial sum

$$H_n = \sum_{k=1}^{n} \frac{1}{k}.$$

H_n is called the n-th harmonic number.

You doubtless saw in a calculus course that $H_n \to \infty$ as $n \to \infty$. Interestingly, $H_n \approx \ln n$ for large n in the following sense.

Theorem 0.5.1.

$$\lim_{n\to\infty} (H_n - \ln n) = \gamma = 0.57721566\ldots$$

Theorem 0.5.1 says that even though H_n and $\ln n$ are going to infinity with n, their difference is converging to the modest quantity γ. The quantity γ is known as **Euler's constant** or the **Euler–Mascheroni constant**. Euler's constant remains a rather mysterious number. For instance, it is not known if γ is rational or irrational, much less transcendental.

Before we prove this theorem we introduce the **zeta function**. For an integer $m \geq 2$ define

$$\zeta(m) = \sum_{j=1}^{\infty} \frac{1}{j^m}.$$

Since $m \geq 2$, this is an absolutely convergent series, so $\zeta(m)$ is finite. In addition, ζ is a decreasing function of m: if $m < m'$, then $\zeta(m) > \zeta(m')$. For instance,

$$\zeta(2) = \sum_{j=1}^{\infty} \frac{1}{j^2} > \sum_{j=1}^{\infty} \frac{1}{j^7} = \zeta(7).$$

Aside. More generally, for $s \in \mathbb{C}$ with $\operatorname{Re} s > 1$ the **Riemann zeta function** is defined to be

$$\zeta(s) = \sum_{j=1}^{\infty} \frac{1}{j^s}.$$

This function plays an important role in analytic number theory and a very famous unsolved problem in mathematics known as the Riemann Hypothesis.

PROOF The proof we give here follows Euler's original argument (1731). Define

$$H_{n,m} = \sum_{j=1}^{n} \frac{1}{j^m}, \quad m \geq 2.$$

Note that

$$\lim_{n\to\infty} H_{n,m} = \sum_{j=1}^{\infty} \frac{1}{j^m} = \zeta(m).$$

The Newton–Mercator series says that for $-1 < x \leq 1$,

$$\ln(1+x) = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{x^k}{k},$$

so

$$\begin{aligned}
\ln 2 &= 1 - \frac{1}{2}\left(\frac{1}{1}\right)^2 + \frac{1}{3}\left(\frac{1}{1}\right)^2 - \frac{1}{4}\left(\frac{1}{1}\right)^3 + \cdots \\
\ln \frac{3}{2} &= \frac{1}{2} - \frac{1}{2}\left(\frac{1}{2}\right)^2 + \frac{1}{3}\left(\frac{1}{2}\right)^2 - \frac{1}{4}\left(\frac{1}{2}\right)^3 + \cdots \\
\ln \frac{4}{3} &= \frac{1}{3} - \frac{1}{2}\left(\frac{1}{3}\right)^2 + \frac{1}{3}\left(\frac{1}{3}\right)^2 - \frac{1}{4}\left(\frac{1}{3}\right)^3 + \cdots \\
\ln \frac{5}{4} &= \frac{1}{4} - \frac{1}{2}\left(\frac{1}{4}\right)^2 + \frac{1}{3}\left(\frac{1}{4}\right)^2 - \frac{1}{4}\left(\frac{1}{4}\right)^4 + \cdots \\
&\vdots = \vdots
\end{aligned}$$

If we sum the first n equations, on the left-hand side we obtain

$$\ln 2 + \ln \frac{3}{2} + \ln \frac{4}{3} + \ln \frac{5}{4} + \cdots + \ln \frac{n+1}{n} = \ln\left(2 \times \frac{3}{2} \times \frac{4}{3} \times \frac{5}{4} \times \cdots \times \frac{n+1}{n}\right) = \ln(n+1).$$

Meanwhile, on the right-hand side we obtain

$$H_n - \frac{1}{2}H_{n,2} + \frac{1}{3}H_{n,3} - \frac{1}{4}H_{n,4} + \cdots,$$

where we have summed terms by columns.

Thus,

$$\begin{aligned}
\ln(n+1) &= H_n - \frac{1}{2}H_{n,2} + \frac{1}{3}H_{n,3} - \frac{1}{4}H_{n,4} + \cdots \\
H_n - \ln(n+1) &= \frac{1}{2}H_{n,2} - \frac{1}{3}H_{n,3} + \frac{1}{4}H_{n,4} - \cdots \\
H_n - \ln n - (\ln(n+1) - \ln n) &= \frac{1}{2}H_{n,2} - \frac{1}{3}H_{n,3} + \frac{1}{4}H_{n,4} - \cdots.
\end{aligned}$$

Since

$$\lim_{n\to\infty} (\ln(n+1) - \ln n) = 0,$$

taking the limit as $n \to \infty$ we obtain

$$\lim_{n\to\infty} (H_n - \ln n) = \frac{1}{2}\zeta(2) - \frac{1}{3}\zeta(3) + \frac{1}{4}\zeta(4) - \cdots.$$

Since the sum on the right is an alternating series of strictly decreasing terms, it converges. The limit is defined to be Euler's constant γ. ∎

0.6 Exercises

Proofs by induction should clearly state the inductive basis, the inductive hypothesis, and the inductive step.

Exercise 0.6.1. *Prove by induction that for $n \geq 1$ and $r \neq 1$,*

$$\sum_{k=0}^{n} r^k = \frac{r^{n+1} - 1}{r - 1}.$$

Exercise 0.6.2. *Show that for $n \geq 1$, $r \neq 1$, and all $0 \leq j \leq n$,*

$$\sum_{k=j}^{n} r^k = \frac{r^{n+1} - r^j}{r - 1}.$$

Exercise 0.6.3. *Show that Proposition 0.2.1 holds when $r = 1$ in the sense that*

$$\lim_{r \to 1} \frac{r^{n+1} - 1}{r - 1} = n + 1.$$

Exercise 0.6.4. *Prove by induction that for all $n \geq 1$ we have*

$$\sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i \right)^2.$$

Exercise 0.6.5. *Suppose $n \geq 1$ and $k \geq 1$ are integers. Use Proposition 0.3.1to show that*

$$\lim_{n \to \infty} \frac{\sum_{i=1}^{n} i^k}{\frac{n^{k+1}}{(k+1)}} = 1.$$

Exercise 0.6.6. *Use the approach described in Section 0.3 to show that*

$$S_2(n) = \sum_{i=1}^{n} i^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} = \frac{n(n+1)(2n+1)}{6}.$$

Start by positing that

$$S_2(n) = an^3 + bn^2 + cn + d$$

and use the requirement that for all n we have $S_2(n+1) = S_2(n) + (n+1)^2$ to determine a, b, c, d.

Exercise 0.6.7. *Prove by induction that for every integer $n \geq 0$ and all $x \geq -1$,*

$$(1 + x)^n \geq 1 + nx.$$

Exercise 0.6.8. *Prove by induction that if $n \geq 1$ then*

$$\sum_{k=1}^{n} \frac{1}{k^2} \leq 2 - \frac{1}{n}.$$

Aside. Euler showed that the exact value of this sum is

$$\sum_{k=1}^{n} \frac{1}{k^2} = \frac{\pi^2}{6}.$$

Exercise 0.6.9. *Evaluate the proposed proof of the following result: For every positive integer n,*

$$1 + 3 + 5 + \cdots + (2n - 1) = n^2.$$

Proof. We proceed by induction.

Since $2 \times 1 - 1 = 1^2$ the identity holds for $n = 1$. Now suppose that $1 + 3 + 5 + \cdots + (2k - 1) = k^2$ for some positive integer k. We prove that $1 + 3 + 5 + \cdots + (2k + 1) = (k + 1)^2$. Observe that

$$1 + 3 + 5 + \cdots + (2k + 1) = (k + 1)^2 \tag{0.6.1}$$

$$1 + 3 + 5 + \cdots + (2k - 1) + (2k + 1) = (k + 1)^2 \tag{0.6.2}$$

$$k^2 + (2k + 1) = (k + 1)^2 \tag{0.6.3}$$

$$(k + 1)^2 = (k + 1)^2, \tag{0.6.4}$$

and the result follows.

Exercise 0.6.10. *Prove by induction that*

$$\sum_{i=1}^{n} i \cdot i! = (n + 1)! - 1$$

for all $n \geq 1$.

Exercise 0.6.11. *Consider the series*

$$1, 2, 3, 4, 5, 10, 20, 40, 80, \ldots \tag{0.6.5}$$

The series starts as an arithmetic series and becomes a geometric series starting with 10. Prove by induction that any positive integer can be written as a sum of distinct members of this series.

Exercise 0.6.12. *Consider the sequence a_n defined by*

$$a_0 = 0,$$
$$a_1 = 1,$$
$$a_n = a_{n-1} + a_{n-2}, \text{ if } n \geq 2.$$

1. *Use induction to show that $a_n \geq 2^{n/2}$ for all $n \geq 6$.*
2. *Find a constant $c < 1$ such that $a_n \leq 2^{cn}$ for all $n \geq 0$. Show that your answer is correct.*

Exercise 0.6.13. *Show that if n is a nonnegative integer, then*

$$\frac{d^n}{dx^n} e^{x^2/2} = p_n(x) e^{x^2/2},$$

where $p_n(x)$ is a polynomial of the form

$$p_n(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0.$$

Exercise 0.6.14. *Show that*

$$\lim_{n \to \infty} \frac{\lg n!}{n \lg n} = 1.$$

Exercise 0.6.15. *Show that for $n \geq 1$,*

$$\ln n! \leq n \ln n.$$

Exercise 0.6.16. *Use Stirling's approximation to estimate* 100! *and state your approximation in exponential format with 8 significant digits. What is the relative error, expressed in exponential format with 2 significant digits? You can use the* `factorial()` *function in Python's* `math` *module to compute the exact value.*

Chapter 1

Analysis of algorithms

In this chapter we lay out the fundamental questions and terminology of the analysis of algorithms.

1.1 What we can and cannot do

Ideally we would be able to look at an algorithm and its input and give an *a priori* analysis of how long it will take to run in terms of time. This, unfortunately, this is not possible except for the simplest snippets of code.

There are many factors affecting the real-life runtime of an algorithm, including

- **Algorithm**: Is the algorithm inherently efficient?
- **Data**: What is the size and nature of the input?
- **Compiler**: What code does the compiler generate?
- **Hardware**: What are the CPU and memory speeds? what instruction set? what memory hierarchy?
- **Implementation**: How efficient is the use of instructions and memory?
- **Environment**: What is the operating system? What is the system load?

Because of the complexity of these factors and their interactions, it is very difficult, if not impossible, to give an `a priori` analysis of how long an algorithm will take to run simply by looking at code.

For example, the dot product of two vectors $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$ is

$$x_1y_1 + x_2y_2 + \cdots + x_ny_n.$$

Here is an implementation in the C programming language:

```
double dot(double *x, double *y, int n)
{
  double sum = 0.0;
  for (int i = 0; i < n; i++) {
    sum += x[i] * y[i];
  }
  return sum;
}
```

Listing 1.1.1: dot.c

Here are some results obtained on 12/23/2023 on `bg1.cs.wm.edu` on an Intel E5-4627v2 CPU running at 3.30 GHz. For $n = 1,000,000,000$ the code compiled using

```
gcc -c -std=c11 dot.c
```

required 2.65 seconds to run, but when compiled using with aggressive code optimization,

```
gcc -c -std=c11 -O3 -march=native dot.c
```

it required only 1.06 seconds. Here is the assembly code for the two compilations. If you examine the assembly code you will see that the latter code includes AVX512 instructions and runs much faster as a consequence.

```
        .file   "dot.c"
        .text
        .globl  dot
        .type   dot, @function
dot:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movq    %rdi, -24(%rbp)
        movq    %rsi, -32(%rbp)
        movl    %edx, -36(%rbp)
        pxor    %xmm0, %xmm0
        movsd   %xmm0, -8(%rbp)
        movl    $0, -12(%rbp)
        jmp     .L2
.L3:
        movl    -12(%rbp), %eax
        cltq
        leaq    0(,%rax,8), %rdx
        movq    -24(%rbp), %rax
        addq    %rdx, %rax
        movsd   (%rax), %xmm1
        movl    -12(%rbp), %eax
        cltq
        leaq    0(,%rax,8), %rdx
        movq    -32(%rbp), %rax
        addq    %rdx, %rax
        movsd   (%rax), %xmm0
        mulsd   %xmm1, %xmm0
        movsd   -8(%rbp), %xmm1
        addsd   %xmm1, %xmm0
        movsd   %xmm0, -8(%rbp)
        addl    $1, -12(%rbp)
.L2:
        movl    -12(%rbp), %eax
        cmpl    -36(%rbp), %eax
        jl      .L3
        movsd   -8(%rbp), %xmm0
        movq    %xmm0, %rax
        movq    %rax, %xmm0
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   dot, .-dot
        .ident  "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
```

```
55          .align 8
56          .long   1f - 0f
57          .long   4f - 1f
58          .long   5
59  0:
60          .string "GNU"
61  1:
62          .align 8
63          .long   0xc0000002
64          .long   3f - 2f
65  2:
66          .long   0x3
67  3:
68          .align 8
69  4:
```

Listing 1.1.2: Assembler produced by `gcc -c -std=c11 dot.c`

```
1           .file   "dot.c"
2           .text
3           .p2align 4
4           .globl  dot
5           .type   dot, @function
6   dot:
7   .LFB0:
8           .cfi_startproc
9           endbr64
10          movq    %rsi, %rcx
11          testl   %edx, %edx
12          jle     .L9
13          leal    -1(%rdx), %eax
14          cmpl    $2, %eax
15          jbe     .L10
16          movl    %edx, %esi
17          shrl    $2, %esi
18          salq    $5, %rsi
19          xorl    %eax, %eax
20          vxorpd  %xmm0, %xmm0, %xmm0
21          .p2align 4,,10
22          .p2align 3
23  .L4:
24          vmovupd (%rdi,%rax), %ymm4
25          vmulpd  (%rcx,%rax), %ymm4, %ymm1
26          addq    $32, %rax
27          vaddsd  %xmm1, %xmm0, %xmm0
28          vunpckhpd       %xmm1, %xmm1, %xmm2
29          vextractf64x2   $0x1, %ymm1, %xmm1
30          vaddsd  %xmm2, %xmm0, %xmm0
31          vaddsd  %xmm1, %xmm0, %xmm0
32          vunpckhpd       %xmm1, %xmm1, %xmm1
33          vaddsd  %xmm1, %xmm0, %xmm0
34          cmpq    %rsi, %rax
35          jne     .L4
36          movl    %edx, %esi
37          andl    $-4, %esi
```

```
38          movl    %esi, %eax
39          cmpl    %esi, %edx
40          je      .L17
41          vzeroupper
42  .L3:
43          subl    %esi, %edx
44          cmpl    $1, %edx
45          je      .L7
46          vmovupd (%rdi,%rsi,8), %xmm5
47          vmulpd  (%rcx,%rsi,8), %xmm5, %xmm1
48          movl    %edx, %esi
49          andl    $-2, %esi
50          addl    %esi, %eax
51          vaddsd  %xmm1, %xmm0, %xmm0
52          vunpckhpd       %xmm1, %xmm1, %xmm1
53          vaddsd  %xmm0, %xmm1, %xmm0
54          cmpl    %esi, %edx
55          je      .L1
56  .L7:
57          cltq
58          vmovsd  (%rdi,%rax,8), %xmm1
59          vmulsd  (%rcx,%rax,8), %xmm1, %xmm1
60          vaddsd  %xmm1, %xmm0, %xmm0
61          ret
62          .p2align 4,,10
63          .p2align 3
64  .L9:
65          vxorpd  %xmm0, %xmm0, %xmm0
66  .L1:
67          ret
68          .p2align 4,,10
69          .p2align 3
70  .L17:
71          vzeroupper
72          ret
73  .L10:
74          xorl    %esi, %esi
75          xorl    %eax, %eax
76          vxorpd  %xmm0, %xmm0, %xmm0
77          jmp     .L3
78          .cfi_endproc
79  .LFE0:
80          .size   dot, .-dot
81          .ident  "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
82          .section        .note.GNU-stack,"",@progbits
83          .section        .note.gnu.property,"a"
84          .align 8
85          .long   1f - 0f
86          .long   4f - 1f
87          .long   5
88  0:
89          .string "GNU"
90  1:
91          .align 8
```

```
92          .long   0xc0000002
93          .long   3f - 2f
94  2:
95          .long   0x3
96  3:
97          .align 8
98  4:
```

Listing 1.1.3: Assembler produced by `gcc -c -std=c11 -O3 -march=native dot.c`

Let's ~~ignore reality~~ remove the distractions of

- programming,
- code generation,
- computing environment,
- &c.,

and focus instead on the abstract substance of the algorithm, in particular, how the runtime might grow with the size of the problem being solved. Understanding how runtime grows with problem size is a central question in the analysis of algorithms.

Several measures of performance are of interest in discussing algorithms, most commonly,

- **best case** performance—what is the best that can happen?
- **worst case** performance—what is the worst that can happen?
- **average case** performance—what can we expect to see in practice?
- **amoritized** performance—what is the cost of applying the algorithm repeatedly?

Performance typically refers to runtime, but it can also refer to resource needs such as memory consumption. In our model of algorithmic performance we will concentrate on operation counts and generally ignore the speed/efficiency of memory access, though this has an major impact on performance.

We will use the terms runtime and complexity interchangably.

Let $T(n)$ be the runtime of the dot product on vectors of length n. In the loop of `dot.c` there are n multiplications and n additions, and the loop operations are executed n times. Therefore, it is reasonable to predict that the running time is proportional to n: $T(n) \approx cn$, where c is an (unknown) constant, disirregardless of the actual instructions being executed. This model $T(n)$ tells us how the execution time grows: if n doubles, then $T(n)$ should double.

Here are observed execution times on an Intel E5-4627v2 CPU. Times are in seconds:

Without `-O3`	
n	$T(n)$ (sec)
10^3	3.5e-06
10^4	2.6e-05
10^5	3.5e-04
10^6	2.6e-03
10^7	2.6e-02
10^8	2.7e-01
10^9	2.8e+00

With `-O3 -march=native`	
n	$T(n)$ (sec)
10^3	1.6e-06
10^4	1.7e-05
10^5	1.6e-04
10^6	1.6e-03
10^7	1.5e-02
10^8	1.6e-01
10^9	2.0e+00

When n increases by a factor of 10, $T(n)$ increases (roughly) by a factor of 10.

1.2 Basic operations

When analyzing an algorithm we will concentrate on its complexity as measured by the number of some basic operations of interest, e.g.,

- Arithmetic operations: $+ \quad - \quad * \quad \div$.
- Comparison operations: $< \quad > \quad == \quad != \quad \cdots$.
- Assignment operations: $=$, swap.

For more complex operations such as $x^n, \log_2 x, \sqrt{x}, \cdots$ we count the number of basic operations required to execute the complex arithmetic operation. For instance, if we compute x^n as

```
double pow(double x, int n)
{
  double xn = x[0];
  for (int i = 1; i < n; i++) {
    xn *= x[i];
  }
  return xn;
}
```

there are $n-1$ multiplications and n assignments, a total of $2n-1$ basic operations. Control flow statements such as `for`, `while`, `if`, etc., are not counted, *per se*, but the evaluation of the statements that dictate the flow is counted, as is any work executed in the body of the flow statement.

Remark 1.2.1. *Memory access and assignments are not free and are often a major cost. We need to account for these operations in any realistic cost model. If we did not account for assignments, then we would be saying that*

```
    for i in range(n):
        y[i] = x[i]
```

takes no time to run.

Conditional execution complicates matters. Consider this C++ `if` statement:

```
for (int i = 0; i < n; i++) {}
    if (A[i] != 0.0) {
      A[i] = 1/A[i];
    }
}
```

How often does the statement `A[i] = 1/A[i]` execute? In the absence of any other information, how much work should we associate with this particular `if` statement? Or how about this more complex C++ snippet:

```
if (done) {
  return done;
}
else {
  for (int i = 0; i < n; i++) {
    y[i] = x[i];
  }
}
```

In the absence of any other information, how should we count operations?

To account for conditional execution,

- The test will always be evaluated, so we count the number of basic operations required for the test.
- In the absence of any other information, given an `if-else`, `switch`, &c., we assume the branch containing the most basic operations will always execute.

We generally consider worst-case behavior, so your accounting should take this perspective and overestimate the number of basic operations when the outcome of a test cannot be ascertained with certainty.

1.3 Asymptotic analysis

Asymptotic analysis deals with the behavior of functions for "sufficiently large" values of the input. The goal of such analysis is to model how the runtime of an algorithm grows as a function of the size of the problem.

We want to be able to compare the growth rate or order of magnitude of functions. We use asymptotic notation to compare the growth rate or order of magnitude of increasing functions. Central to this analysis is the use of functions whose behavior is well-understood. In Table 1.1 we list some of the commonly encountered growth rates along with programming patterns that give rise to them.

growth rate	name	pattern	example
1	constant	constant no. of operations	add two floats
$\log n$	logarithmic[1]	recursively halve	bisection
n	linear	loop	find the minimum
$n \log n$	$n \log n$ or linearithmic	recursively divide in two	mergesort
n^2	quadratic	doubly nested loops	matrix-vector multiplication
n^3	cubic	triply nested loops	matrix-matrix multiplication
2^n	exponential	exhaustive search	enumerate all n-bit integers

Table 1.1: Growth rates.

In Figure 1.3.1 we plot these functions (except for the exponential).

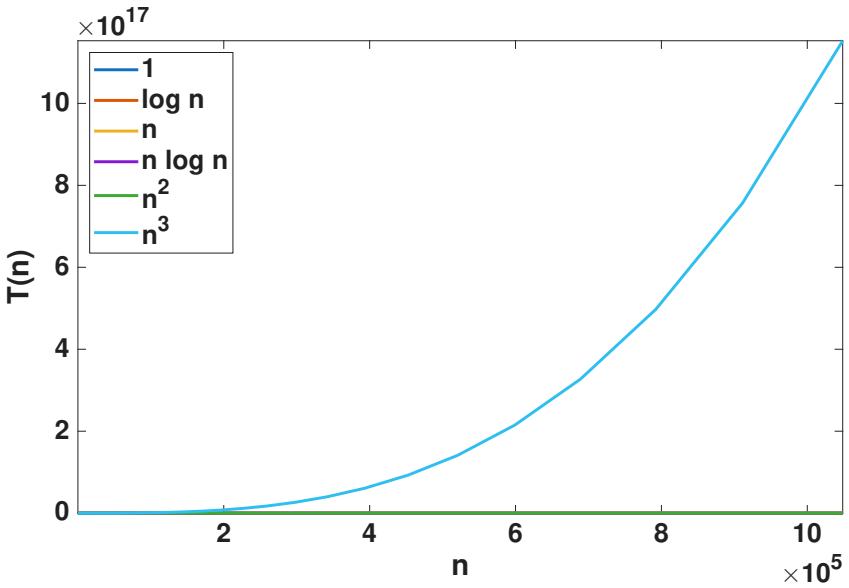


Figure 1.3.1: A linear-linear plot of growth rates.

You can see that the cubic function clearly dominates the others as n grows. The differences between the functions are more obvious if we plot $\log T(n)$ versus $\log n$, as illustrated in Figure 1.3.2.

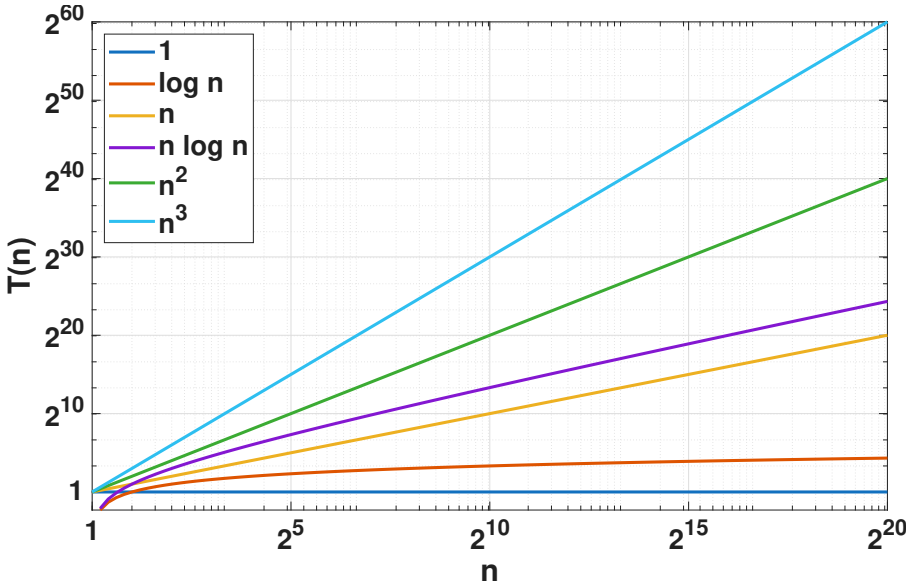


Figure 1.3.2: A log-log plot of growth rates.

1.4 Scaling tests

We can gain empirical insight into an algorithm's complexity by comparing the runtimes for different size problems. Let $T(n)$ be the runtime of an algorithm on a problem of size n. If $T(n) \approx Cn^3$ for large n, then

$$\frac{T(2n)}{T(n)} \approx \frac{C(2n)^3}{Cn^3} \approx 8.$$

Conversely, if the preceding relation holds, we may reasonably conclude $T(n) \approx Cn^3$.

More generally, if $T(n) \approx Cn^p$, we have

$$T(n_0) = Cn_0^p,$$
$$T(n_1) = Cn_1^p,$$

so

$$\frac{T(n_1)}{T(n_0)} = \frac{n_1^p}{n_0^p}. \tag{1.4.1}$$

This is an extremely useful relation since given any four of the five quantities n_0, n_1, $T(n_0)$, $T(n_1)$, and p in (1.4.1) we can compute the fifth.

Example 1.4.1. *Suppose an algorithm takes 1 ms*[2] *for an input of size $n = 1000$. Approximately how long will it take for an input of size 10000 if the running time is cubic?*

Here we have $T(n) \approx Cn^3$, or $p = 3$. We also have $n_0 = 1000$, $T(n_0) = 1ms$, and $n_1 = 10000$. We wish to find $T(n_1)$. From (1.4.1),

$$T(n_1) = T(n_0)\frac{n_1^p}{n_0^p} = 1\ ms \times \left(\frac{10000}{1000}\right)^3 = 1\ ms \times 1000 = 1\ s.$$

[2] ms = 1 millisecond.

Example 1.4.2. *Suppose an algorithm takes 2 seconds for an input of size $n = 1000$. How large a problem can we solve in 10 seconds if the running time is quadratic?*

Here we have $T(n) \approx Cn^2$, or $p = 2$. We also have $n_0 = 1000$, $T(n_0) = 2$, and $T(n_1) = 10$. We wish to find n_1. From (1.4.1),

$$n_1 = n_0 \left(\frac{T(n_1)}{T(n_0)}\right)^{1/p} = 1000 \left(\frac{10}{2}\right)^{1/2} = 1000\sqrt{5}.$$

This particular scaling test has trouble distinguishing the presence of logarithmic terms because the logarithm is slowly growing. For instance, if $T(n) = n \lg n$, then

$$\frac{T(2n)}{T(n)} = \frac{2n \lg 2n}{n \lg n} = \frac{2n(1 + \lg n)}{n \lg n} = \frac{2n + 2n \lg n}{n \lg n} = 2 + \frac{2}{\lg n}.$$

Depending on random variability in runtime, it may be hard to distinguish whether the ratio is 2 or $2 + \frac{2}{\lg n}$ if n is large. In the exercises you are asked to devise a test that will detect logarithmic terms.

1.5 Linear and logarithmic regression

The runtimes of many of the algorithms we will study have the form

$$T(n) \approx Cn^p.$$

Observe that

$$\log T(n) \approx p \log n + \log C.$$

As you can see in Figure 1.3.2, in a log-log plot of n vs runtime (i.e. plotting $\log T(n)$ vs $\log n$), the graph is a line of slope p,

Assuming $T(n) \approx Cn^p$, then for any base logarithm log,

$$\log \frac{T(n_1)}{T(n_0)} = p \log \frac{n_1}{n_0}$$

$$p = \frac{\log \frac{T(n_1)}{T(n_0)}}{\log \frac{n_1}{n_0}}$$

Thus if we observe the runtimes $T(n_0)$ and $T(n_1)$ for problem sizes n_0 and n_1 we can estimate p.

Linear regression finds the line that best fits a set of points in the following sense: given data $(x_1, y_1), \ldots, (x_n, y_n)$, find m and b such that the line $y = mx + b$ minimizes the sum of squares. Since we are minimizing a sum of squares, this approach is also known as the method of least squares. The use of squares rather than, say, the absolute value, makes the problem differentiable and easy to solve in closed-form. of the misfit between model prediction and the actual data:

$$\underset{m,b}{\text{minimize}} \sum_{i=1}^{n} ((mx_i + b) - y_i)^2 .$$

Let

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i,$$

$$\overline{y} = \frac{1}{n} \sum_{i=1}^{n} y_i,$$

denote the averages of the variables. Then the solution to our optimization problem is given by

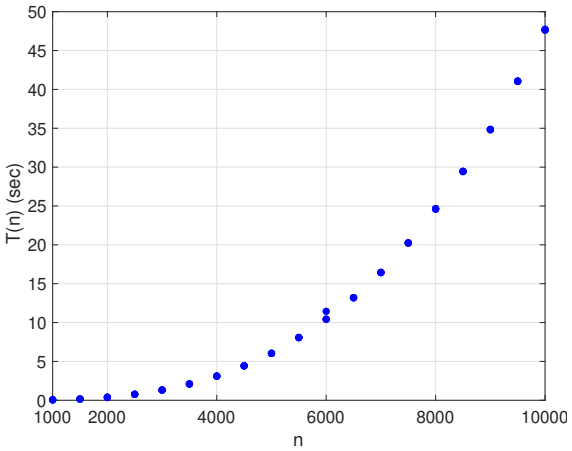
$$m = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})^2} = \frac{\text{covariance of } x \text{ and } y}{\text{variance of } x},$$
$$b = \overline{y} - m\overline{x},$$

If we expect x and y to be related by $y \approx Cx^p$ then we can estimate C and p by first applying a logarithmic transformation:

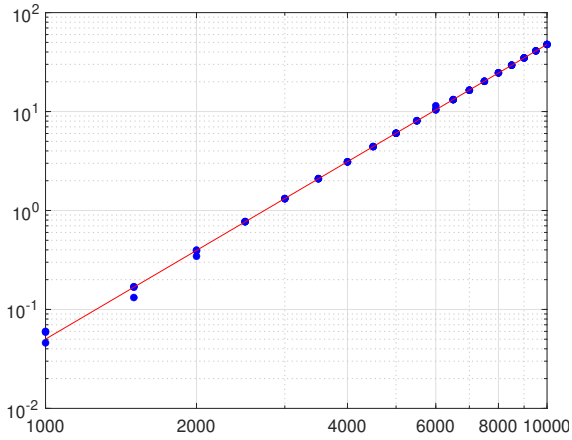
$$\log y = p \log x + \log C.$$

The response $\log y$ is linear in $\log x$. We can then apply linear regression to the transformed data $(\log x_i, \log y_i)$ to estimate p and $\log C$.[3] This is one version of logarithmic regression.

Example 1.5.1. *We illustrate logarithmic regression for the runtimes of the Fortran intrinsic subroutine* `matmul()`*, which performs matrix-matrix multiplication using n^3 multiplications and n^3 additions. Multiple runs were performed for each value of n.*



(a) Linear-linear plot.



(b) Log-log plot.

In the $(\log n, \log T(n)$ plot in Figure 1.5.1b, we see that the log-transformation makes the plot a straight line. The red line is the logarithmic regression fit; we compute the slope m and y-intercept b using the formulae above applied to the $(\log n, \log T(n)$ data. We then compute $y = mx + b$ for a number of values x in log-space (between 3 and 4 for this example), apply the transformation $(x, y) \mapsto (10^x, 10^y)$, and plot the resulting points.

1.6 Asymptotic notation

In this section we make the following assumption.

Assumption 1.6.1. *Until further notice, $f(n)$ and $g(n)$ are real-valued functions that are positive for $n \geq 1$.*

This is a reasonable assumption if f and g are quantities such as runtimes or operation counts.

A useful, if not entirely standard, bit of notation for comparing the growth of functions as $n \to \infty$ is tilde notation.

Definition 1.6.2. *We write $g(n) \sim f(n)$ if* $\lim_{n\to} \frac{g(n)}{f(n)} = 1$.

[3] The slope p will be of greater importance to us.

When using this notation to describe the growth of f we should compare against g that are as simple as possible. That is, if $f(n) = n^2 + 2n + 1$, then it would be true to say that $f \sim g$ for $g(n) = n^2 + 37n + 42$, but it would be in better taste to say $f \sim g$ for $g = n^2$.

Three concepts more commonly encountered in the analysis of algorithms are

$$f(n) = O(g(n)),$$
$$f(n) = \Omega(g(n)),$$
$$f(n) = \Theta(g(n)).$$

Although these relationships are written as equalities, they actually involve inequalities!

1.6.1 Big O

We begin with big O, which gives an upper bound on one function in terms of another.

Definition 1.6.3 (big-O). *We say $f = O(g)$ if there exist constants c and N such that*

$$f(n) \leq c \cdot g(n)$$

for all $n \geq N$.

Example 1.6.4. *If $f(n) = n$ and $g(n) = n \lg n$, then $f = O(g)$ since $f(n) \leq g(n)$ for all $n \geq 2$.*

Example 1.6.5. *If $f(n) = 54n^2 + 42n$ and $g(n) = n^2$, then $f = O(g)$ since*

$$f(n) \leq 55g(n)$$

for all $n \geq 42$. We are free to choose other values for n and c. For instance,

$$f(n) \leq 56g(n)$$

for all $n \geq 84$.

As the preceding example shows, there is latitude in finding c and N when applying Definition 1.6.3. The material point is that some such values exist. These values may be of purely theoretical interest. For instance, in [15] the authors present a $O(n \lg n)$ bound for

$$n \geq 2^{1729^{12}} = 2^{7137398073256634897664758526207831206 41} \tag{1.6.1}$$

and a comparably large value of c.[4]

The following result relates big O to Definition 1.6.2.

Proposition 1.6.6. *If*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = C$$

then $f = O(g)$.

Proof The fact that $f, g > 0$ and the definition of the limit mean that given any $\varepsilon > 0$, there exists N such that

$$-\varepsilon < \frac{f(n)}{g(n)} - C < \varepsilon$$

for all $n \geq N$. Since $g > 0$ we have

$$f(n) < (\varepsilon + C)g(n)$$

for all $n \geq N$. Choosing any such pair ε, N yields $f = O(g)$. ∎

[4] They also note that these values can be reduced.

There is also a little-o notation.

Definition 1.6.7 (little-o). *We say $f = o(g)$ as $n \to \infty$ if*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

We have the following corollary of Proposition 1.6.6.

Corollary 1.6.8. *If $f = o(g)$, then $f = O(g)$.*

1.6.2 Big Ω

Next we look at big Ω, which gives an lower bound on one function in terms of another.

Definition 1.6.9 (big-Ω). *We say $f(n) = \Omega(g(n))$ if there exist constants $c > 0$ and N such that*

$$f(n) \geq c \cdot g(n)$$

for all $n \geq N$.

Example 1.6.10. *If $f(n) = n \lg n$ and $g(n) = n$, then $f = \Omega(g)$ since $f(n) \geq g(n)$ for all $n \geq 2$.*

Big Ω is inverse to big O.

Proposition 1.6.11. *$f = O(g)$ if and only if $g = \Omega(f)$.*

Proof If $f = O(g)$, then there exist c and N such that

$$f(n) \leq cg(n)$$

for all $n \geq N$. Since $c > 0$ we can rearrange this as

$$g(n) \geq \frac{1}{c} f(n)$$

for all $n \geq N$. This means $g = \Omega(f)$.

A similar argument shows that if $g = \Omega(f)$, then $f = O(g)$ and is left as an exercise. ∎

1.6.3 Big Θ

Finally, we have big Θ, which says two functions grow at comparable rates.

Definition 1.6.12 (big-Θ). *We say $f(n) = \Theta(g(n))$ if there exist constants c_1, c_2, and N such that*

$$c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n) \quad \text{for all } n \geq N.$$

Example 1.6.13. *We have $2n^2 + n = \Theta(n^2)$ since*

$$2n^2 \leq 2n^2 + n \leq 3n^2$$

for all $n \geq 1$.

We have the following results on big Θ. The proofs are left as exercises.

Proposition 1.6.14. *If $f = \Theta(g)$, then $g = \Theta(f)$.*

Proposition 1.6.15. *$f = \Theta(g)$ if and only if $f = \Omega(g)$ and $f = O(g)$.*

Proposition 1.6.16. *If*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = C > 0,$$

then $f = \Theta(g)$.

As a corollary, if $f \sim g$ then $f = \Theta(g)$.

Example 1.6.17. *Another way to see that* $2n^2 + n = \Theta(n^2)$ *is to note that*

$$\lim_{n\to\infty} \frac{2n^2 + n}{n^2} = 2.$$

1.6.4 Putting it all together

Definition 1.6.18 (Sufficiently large). *We say that a statement S involving n is true when n is sufficiently large if there exists N such that S is true for all* $n \geq N$.

We can thus say that for n sufficiently large,

- $f(n) = O(g(n))$ gives an upper bound on f in terms of g,
- $f(n) = \Omega(g(n))$ gives a lower bound on f in terms of g, and
- $f(n) = \Theta(g(n))$ gives both an upper and lower bound on f in terms of g.

Example 1.6.19. *Suppose we know that* $f(n) = \Omega(n)$ *and* $g(n) = \Theta(n^3)$. *This means there exist* c_1, c_2, c_3 *and* N *such that*

$$c_1 n \leq f(n)$$
$$c_2 n^3 \leq g(n) \leq c_3 n^3$$

for all $n \geq N$.

The fact that we have no upper bound on f means we cannot say anything about an upper bound on $f + g$, *which rules out making any* Θ *or* O *statements about* $f + g$. *Thus, we need only consider the* Ω *statements. From the preceding set of inequalities we see that*

$$c_1 n + c_2 n^3 \leq f(n) + g(n)$$

so $f + g = \Omega(n^3)$.

In these definitions it is the existence of the bounds that matters for the purposes of asymptotic complexity analysis—not the exact values of the numbers involved.

Example 1.6.20. *All of the following imply* $n^2 + n = O(n^2)$*:*

$$n^2 + n \leq 4n^2 \quad \text{for all } n \geq 1$$
$$n^2 + n \leq 2n^2 \quad \text{for all } n \geq 1$$
$$n^2 + n \leq 1.01n^2 \quad \text{for all } n \geq 100$$
$$n^2 + n \leq 1.000001n^2 \quad \text{for all } n \geq 1000$$

We typically ignore constant factors and lower-order terms, since all we want to capture is the dominant growth trend.

Example 1.6.21. *Suppose* $f(n) = 1,000n^2 + 10,000n + 42$. *Then the first three of the following statements are true, but only the first statement is fit for public utterance.*

$$f(n) = O(n^2)$$
$$f(n) = O(1,000n^2) \qquad \textit{true, but bad form}$$
$$f(n) = O(n^2 + n) \qquad \textit{true, but bad form}$$
$$f(n) \leq O(n^2) \qquad \textit{huh?!}$$

A $f(n) = O(g(n))$ bound can be misleading: $n = O(n^2)$, since $n \leq n^2$ for all $n \geq 1$. However, for large n, $n \ll n^2$—the functions $f(n) = n$ and $g(n) = n^2$ are nothing alike for large n! We don't write something like $f(n) = O(n^2)$ if we know, say, that $f(n) = O(n)$. In general, upper bounds should be as small as possible.

Likewise, a $f(n) = \Omega(g(n))$ bound can be misleading: $n^2 = \Omega(n)$, since $n^2 \geq n$ for all $n \geq 1$. However, for large n, $n^2 \gg n$—the functions $f(n) = n$ and $g(n) = n^2 + n$ are nothing alike for large n! We don't write something like $f(n) = \Omega(n)$ if we know, say, that $f(n) = \Omega(n^2)$. Lower bounds should be as large as possible.

A $f(n) = \Theta(g(n))$ relationship is the most informative. It cannot conceal a gross underestimate or overestimate of growth rates.

1.7 Algebra of asymptotics

What can we say about asymptotic growth when we combine functions? This is the situation we encounter when we combine different units of computation into a larger algorithm. We will assume that all of the functions in this section satisfy Assumption 1.6.1.

Let's think about what we can say. If $f = \Omega(F)$ and $g = \Omega(G)$ we have lower bounds on f and g, so we have a lower bound on $f + g$. On the other hand, we cannot say anything about an upper bound on $f + g$.

Proposition 1.7.1. *If* $f = \Omega(F)$ *and* $g = \Omega(G)$, *then* $f + g = \Omega(F + G)$.

If $f = O(F)$ and $g = \Omega(G)$, then $f + g = \Omega(G)$ we have an upper bound on f but only a lower bound on g. Since $f + g \geq g = \Omega(G)$, we have a lower bound on $f + g$. On the other hand, without an upper bound on g we do not have an upper bound on $f + g$.

Proposition 1.7.2. *If* $f = O(F)$ *and* $g = \Omega(G)$, *then* $f + g = \Omega(G)$.

If $f = O(F)$ and $g = O(G)$, then we have upper bounds on f and g, and thus an upper bound on $f + g$. However, we know nothing about a lower bound for $f + g$.

Proposition 1.7.3. *If* $f = O(F)$ *and* $g = O(G)$, *then* $f + g = O(F + G)$.

Finally, if $f = \Theta(F)$ and $g = \Theta(G)$ we have lower and upper bounds on f and g, and thus on $f + g$.

Proposition 1.7.4. *If* $f = \Theta(F)$ *and* $g = \Theta(G)$, *then* $f + g = \Theta(F + G)$.

1.8 Exercises

Exercise 1.8.1. *Suppose you observe a program with the following run times for different sizes N of its input:*

n	T(n) = time (in sec)
10000	1.045514
15000	1.734443
20000	4.177824
25000	5.419267
30000	6.978952
35000	14.693412
40000	16.779861

What would you estimate the order of the time complexity to be? Explain your answer.

Exercise 1.8.2. *Plot the data in Exercise 1.8.1 in log-log scale and apply logarithmic regression to estimate the complexity.*

Exercise 1.8.3. *Consider the following Python fragment. Give an analysis of the complexity in terms of the number of times the statement* `sum += 1` *is executed. Begin with a summation for the number of times the statement is executed.*

```
sum = 0
for i in range(n):
    for j in range(i):
        sum += 1
```

Exercise 1.8.4. *Consider the following Python fragment. Give an analysis of the complexity in terms of the number of times the statement* `sum += 1` *is executed. Begin with a summation for the number of times the statement is executed.*

```
sum = 0
for i in range(n):
    for j in range(n**2):
        sum += 1
```

Exercise 1.8.5. *Consider the following Python fragment. Give an analysis of the complexity in terms of the number of times the statement* `sum += 1` *is executed. Begin with a summation for the number of times the statement is executed.*

```
sum = 0
for i in range(n):
    for j in range(i**2):
        for k in range(j):
            sum += 1
```

Exercise 1.8.6. *Consider the following Python fragment. Give an analysis of the complexity in terms of the number of times the statement* `sum += 1` *is executed. Begin with a summation for the number of times the statement is executed.*

```
sum = 0
for i in range(n):
    for j in range(i**2):
        if j % i == 0:
            for k in range(j):
                sum += 1
```

Exercise 1.8.7. *Devise a scaling test that will distinguish between n and $n \lg n$ complexity and show that it is correct.*

Exercise 1.8.8. *Give tilde approximations that are as simple as possible for the following, and show that your approximations are correct.*

(a) $n + 42$

(b) $(n + 1)/n)$

(c) $2n^2 + n \lg n + 7n$

(d) $(\lg 42n)/\lg n$

(e) $\lg(n^3 + n)/\lg n$

Exercise 1.8.9. *What is the significance of the number* 1729 *in (1.6.1)?*

Exercise 1.8.10. *Order the following functions by growth rate so that if f is to the left of g then $f = O(g)$.*

$n, \sqrt{n}, n^{1.5}, \log n, n^2, n^{0.000001}, n \log n, n \log \log n, n(\log n)^2, n^2 \log n, n^3$.

Exercise 1.8.11. *Order the following functions by growth rate so that if f is to the left of g then $f = O(g)$.*

n, $\sqrt{n}$, $n^{3/2}$, n^2, $n \log n$, $n \log \log n$, $n \log^2 n$, 2^n, $n!$, n^3, $n^2 \log n$.

Exercise 1.8.12. *For each of the following pairs of functions, indicate whether $f = O(g)$, $f = \Omega(g)$, or both (i.e., $f = \Theta(g)$.*

	$f(n)$	$g(n)$
(a)	$n - 100$	$n - 200$
(b)	$\sqrt{n}$	$n^{2/3}$
(c)	$100n + \log n$	$n + (\log n)^2$

Exercise 1.8.13. *Suppose $f(n) = n \log n$ and $g(n) = n^2$. For each of the following statements, indicate whether they are true or false.*

(a) $f(n) + g(n) = \Omega(n)$
(b) $f(n) + g(n) = \Theta(n)$
(c) $f(n) + g(n) = O(n)$
(d) $f(n) + g(n) = \Omega(n \log n)$
(e) $f(n) + g(n) = \Theta(n \log n)$
(f) $f(n) + g(n) = O(n \log n)$
(g) $f(n) + g(n) = \Omega(n^2)$
(h) $f(n) + g(n) = \Theta(n^2)$
(i) $f(n) + g(n) = O(n^2)$
(j) $f(n) + g(n) = \Omega(n^3)$
(k) $f(n) + g(n) = \Theta(n^3)$
(l) $f(n) + g(n) = O(n^3)$

Exercise 1.8.14. *Suppose $f(n) = \log n$ and $g(n) = n \log n$. For each of the following statements, indicate whether they are true or false.*

(a) $f(n) + g(n) = \Omega(n)$
(b) $f(n) + g(n) = \Theta(n)$
(c) $f(n) + g(n) = O(n)$
(d) $f(n) + g(n) = \Omega(n \log n)$
(e) $f(n) + g(n) = \Theta(n \log n)$
(f) $f(n) + g(n) = O(n \log n)$
(g) $f(n) + g(n) = \Omega(n^2)$
(h) $f(n) + g(n) = \Theta(n^2)$
(i) $f(n) + g(n) = O(n^2)$
(j) $f(n) + g(n) = \Omega(n^2 \log n)$
(k) $f(n) + g(n) = \Theta(n^2 \log n)$
(l) $f(n) + g(n) = O(n^2 \log n)$

Exercise 1.8.15. *Suppose $f(n) = \Omega(n \log n)$ and $g(n) = \Theta(n^2)$. For all of the following that are guaranteed to be true as a consequence, circle the letter that indicates the answer.*

(a) $f(n) + g(n) = \Omega(n)$
(b) $f(n) + g(n) = \Theta(n)$
(c) $f(n) + g(n) = O(n)$
(d) $f(n) + g(n) = \Omega(n \log n)$
(e) $f(n) + g(n) = \Theta(n \log n)$
(f) $f(n) + g(n) = O(n \log n)$
(g) $f(n) + g(n) = \Omega(n^2)$
(h) $f(n) + g(n) = \Theta(n^2)$
(i) $f(n) + g(n) = O(n^2)$
(j) $f(n) + g(n) = \Omega(n^3)$
(k) $f(n) + g(n) = \Theta(n^3)$
(l) $f(n) + g(n) = O(n^3)$

Exercise 1.8.16. *Prove Proposition 1.6.14.*

Exercise 1.8.17. *Prove Proposition 1.6.15.*

Exercise 1.8.18. *Prove Proposition 1.6.16*

Exercise 1.8.19. *Show that* $\log n = O(n^\alpha)$ *for all* $\alpha > 0$.

Exercise 1.8.20. *Show that* $(\log n)^\beta = O(n)$ *for all* $\beta > 0$.

Exercise 1.8.21. *Suppose* $\alpha > 0$*, and consider the sum*

$$S(k) = \sum_{i=0}^{k} \alpha^i.$$

Show that

$$S(k) = \begin{cases} \Theta(1) & \text{if } \alpha < 1, \\ \Theta(k) & \text{if } \alpha = 1, \\ \Theta(\alpha^k) & \text{if } \alpha > 1. \end{cases}$$

Exercise 1.8.22. *Complete the proof of Proposition 1.6.11.*

Exercise 1.8.23. *Show that* $f = O(g)$ *if and only if*

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} < +\infty.$$

Exercise 1.8.24. *Show that* $f = \Omega(g)$ *if and only if*

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0.$$

Exercise 1.8.25. *Show that* $f = \Theta(g)$ *if and only if*

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0.$$

and

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} < +\infty.$$

Chapter 2

Simple sorts

In this chapter we will give a detailed analysis of some simple sorting algorithms. By a simple sort we mean a sorting algorithm that proceeds by comparing or swapping adjacent elements. Humans tend to sort this way. Simple sorts are easy to explain and implement; however, we will see that they cannot be efficient in the worst case.

Our cost model for sorting will count the number of comparisons, swaps, and assignments. If there are memory accesses that are not associated with comparisons or swaps, we need to account for them, too.

In our discussion of sorting we will assume that we wish to sort an array (or other indexable object) of n items and place them in *ascending* order. We also assume our indexing begins with 0.

2.1 Selection sort

Our first sorting algorithm, selection sort, is simple to describe:

1. Find the smallest item and exchange it with the first entry.
2. Find the next smallest item and exchange it with the second entry.
3. Find the next smallest item and exchange it with the third entry.
4. &c.

At step p of this process the first p items are in sorted order. If we have n items to sort, then after n steps we know we have correctly sorted things.

Example 2.1.1. *Here is an example of selection sort:*

initial sequence	42	6	9	54	0
after $p = 0$	0	6	9	54	42
after $p = 1$	0	6	9	54	42
after $p = 2$	0	6	9	54	42
after $p = 3$	0	6	9	42	54

Here is a Python implementation of selection sort:

```
def selection_sort(a):
    '''Selection sort.'''
    for i in range(len(a)):
        # Locate the smallest value in a[i,:].
        j_min = i
        for j in range(i+1,len(a)):
            if a[j] < a[j_min]:
                j_min = j;
        # Swap this value with a[i].
```

```
10            a[i], a[j_min] = a[j_min], a[i]
11        return a
```

Listing 2.1.1: Selection sort.

2.1.1 Complexity of selection sort

We will measure the cost of selection sort in terms of the number of comparisons and swaps it performs. The number of comparisons is the number of times statement 7 is executed inside the doubly nested loop. When dealing with nested loops, work from the innermost loops outward. Each iteration of the innermost loop which begins at line 6 has 1 comparision, and j goes from $i+1$ to $n-1$, so each iteration of the outermost loop contains

$$\sum_{j=i+1}^{n-1} 1 = (n-1) - (i+1) + 1 = n - i + 1$$

comparisons. Each iteration of the outermost loop involves this much work, and i goes from 0 to $n-1$, so the total number of comparisons performed in the outermost loop is

$$\begin{aligned}\sum_{i=0}^{n-1}(n-i-1) &= \left(\sum_{i=0}^{n-1} n\right) - \left(\sum_{i=0}^{n-1} i\right) - \left(\sum_{i=0}^{n-1} 1\right) \\ &= n^2 - \frac{n(n-1)}{2} - n \\ &= \frac{n^2}{2} + \frac{n}{2} - n \\ &= \frac{n^2}{2} - \frac{n}{2} \\ &= \frac{n(n-1)}{2}.\end{aligned}$$

Selection sort performs this many comparisons no matter what the input! This means that

- the best-case,
- the worst-case, and
- the expected-case

number of comparisons grows quadratically with the length of the list being sorted. From the structure of the loops it is also clear that selection sort performs n swaps. We summarize these observations in the following proposition.

Proposition 2.1.2. *Selection sort performs $n(n-1)/2$ comparisons and n swaps when sorting n items.*

The amount of work is independent of the input—selection sort is no faster on sorted input than on random input. On the other hand, selection sort involves a smaller number of swaps than any of the other sorting algorithms we discuss.

Table 2.1 shows the runtimes for 10 repetitions of a Python implementation of selection sort on an Apple M1 laptop:

n	time (seconds)
64	0.001
128	0.003
256	0.009
512	0.043
1024	0.176
2048	0.714
4096	2.822
8192	11.957
16384	45.625

Table 2.1: Selection sort runtimes.

Observe that as we double n the runtime increases by roughly a factor of 4, which is consistent with our analysis that the runtime should grow quadratically.

2.2 Insertion sort

The next simple sort is insertion sort. When sorting n items insertion sort consists of $n-1$ passes over the data being sorted. In pass p, $p = 1, \ldots, n-1$, we move a_p to its correct location among $a_0, \ldots, a_p$. Thus at pass p insertion sort ensures the invariant that the elements in positions 0 to p are in sorted order.

Listing 2.2.1 shows an implementation of insertion sort.

```
1  def insertion_sort(a):
2      '''Insertion sort.'''
3      for p in range(1,len(a)):
4          tmp = a[p]
5          j = p
6          while (j > 0) and (a[j-1] > tmp):
7              a[j] = a[j-1]
8              j -= 1
9          a[j] = tmp
10     return a
```

Listing 2.2.1: Insertion sort.

In line 7 we move terms to the right so that ultimately there is a space to place a_p.

Here is insertion sort in action. We show the list `a[ ]` at the beginning of the p-loop and at the end of the j-loop.

p = 1		[42, 6, 1, 54, 0, 7]
p = 1	j = 0	[42, 42, 1, 54, 0, 7]
p = 2		[6, 42, 1, 54, 0, 7]
p = 2	j = 1	[6, 42, 42, 54, 0, 7]
p = 2	j = 0	[6, 6, 42, 54, 0, 7]
p = 3		[1, 6, 42, 54, 0, 7]
p = 4		[1, 6, 42, 54, 0, 7]
p = 4	j = 3	[1, 6, 42, 54, 54, 7]
p = 4	j = 2	[1, 6, 42, 42, 54, 7]
p = 4	j = 1	[1, 6, 6, 42, 54, 7]
p = 4	j = 0	[1, 1, 6, 42, 54, 7]
p = 5		[0, 1, 6, 42, 54, 7]
p = 5	j = 4	[0, 1, 6, 42, 54, 54]
p = 5	j = 3	[0, 1, 6, 42, 42, 54]
		[0, 1, 6, 7, 42, 54]

To analyze the work required we will count the number of comparisons of elements of `a[ ]` done at statement 6 and the number of data movements done at statement 7.[1]

As with selection sort, we start with the innermost loop. This time we have a more interesting situation than with selection sort, as the number of iterations in the innermost loop depends on the list being sorted:

- in the best-case, $a[i-1] > a[i]$ and we have only one comparison;
- in the worst-case, we have to go from $j = i$ to $j = 1$;
- the expected-case is not at all clear.

In the best-case situation we execute the innermost loop only once in every iteration of the outermost loop. This leads to

$$\sum_{i=0}^{n-1} 1 = n$$

comparisons. In this case there are no swaps. This situation obtains when the list is already sorted.

In the worst-case situation, we have to go from $j = i$ to $j = 1$ in the innermost loop for each iteration of the outermost loop. This occurs when the list is in the reverse of sorted order.[2] This leads to i comparisons in the innermost loop. Combined with the outermost loop we have

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

comparisons. In this case there are also $n(n-1)/2$ swaps.

Proposition 2.2.1. *Insertion sort performs between n and $n(n-1)/2$ comparisons and swaps to sort n items.*

Thus, insertion sort is

- linear in the best-case, when the keys are already sorted;
- quadratic in the worst-case, when the keys are in reverse sorted order;
- and still unclear as to the expected case

Insertion sort illustrates another complication in reckoning computational complexity—the behavior of the algorithm varies greatly depending on the data it is working on. There is a dramatic difference between the best-case and worst-case behavior. We will address the expected case in Section 2.4.

Actual runtimes for insertion sort are given in Table 2.2.[3]

n	time (seconds)
64	0.000
128	0.000
256	0.002
512	0.006
1024	0.029
2048	0.120
4096	0.471
8192	1.913
16384	7.655

Table 2.2: Insertion sort runtimes.

We see the algorithm exhibits quadratic complexity, as predicted.

[1] We will not count the cost of checking whether `j > 0`.

[2] I.e., when the original list has $a_0 > a_1 > \cdots > a_{n-1}$.

[3] As with selection sort, 10 repetitions are done for each n.

2.3 Bubble sort

There are many similar algorithms that go by the name of bubble sort. The name comes from the imagery of the terms being bubbling to their correct place during the sorting process..

In the version we discuss here, we scan the items from left to right. If two adjacent items are out of order, we swap them. We repeat this process until the data are sorted. A sweep with no swaps means we are done.

```
1 def bubblesort(a):
2     not_done = True
3     while (not_done):
4         not_done = False
5         for i in range(len(a)-1):
6             if a[i] > a[i+1]:
7                 a[i], a[i+1] = a[i+1], a[i]
8                 not_done = True
```

Listing 2.3.1: Bubble sort.

Here is an illustration of bubble sort. We show what goes on in each pass through the inner loop that starts at line 6.

initial sequence	42	6	9	54	0
inner loop	6	42	9	54	0
	6	9	42	54	0
	6	9	42	0	54
inner loop	6	9	0	42	54
inner loop	6	0	9	42	54
inner loop	0	7	9	42	54

It is not clear how to analyze bubble sort. If the data are already sorted we make only one sweep through the list, making $n-1$ comparisons and no swaps. Otherwise, the complexity depends on the number of times we execute the `while` loop, and it is not clear how the inner loop affects this. We will consider the worst case and expected complexities of bubblesort in Section 2.4.

2.4 Complexity of simple sorts

What can we say about the expected (average) case complexity of insertion sort and bubble sort? The answer lies in an abstraction of how simple sorts operate.

Given $a_0, \ldots, a_{n-1}$ that we wish to sort in ascending order, an inversion is any pair that are out of order relative to one another; i.e., a pair (a_i, a_j) for which $i < j$ but $a_i > a_j$.

Example 2.4.1. *For instance, the list*

$$42, 6, 9, 54, 0$$

contains the following inversions:

$$(42, 6), (42, 9), (42, 0), (6, 0), (9, 0), (54, 0).$$

The key observation is that swapping an adjacent pair of elements that are out of order removes exactly one inversion. Thus, any sort such as insertion sort or bubble sort that operates by swapping adjacent terms requires as many swaps as there are inversions. So, how many inversions can there be?

A list can have between 0 and $n(n-1)/2$ inversions. The former occurs when the list is already sorted; the latter occurs when it is sorted in reverse order. Thus, counting inversions shows that the worst-case behavior of insertion sort and bubble sort is quadratic.

What do inversions tell us about the expected behavior? Let P be the probability space of all permutations of n elements occuring with equal probability.

Assumption 2.4.2. *For simplicity, we will assume the elements are distinct. This is not essential.*

Theorem 2.4.3. *The expected number of inversions in a list taken from P is $n(n-1)/4$, and the standard deviation is approximately $n^{3/2}/6$.*

Proof Observe that any pair in a list that is an inversion is in the correct order if the list is reversed:

list	inversions	reversed list	inversions
1, 2, 3, 4	0	4, 3, 2, 1	6
2, 1, 3, 4	1	4, 3, 1, 2	5
3, 2, 1, 4	3	4, 1, 2, 3	3

This means that if we look at a list and its reverse and count the total number of inversions, then the combined number of inversions is $n(n-1)/2$.

Since there are $n!/2$ distinct pairs of lists and their reverses, there is a total of

$$\frac{n!}{2}\frac{n(n-1)}{2} = n!\frac{n(n-1)}{4}$$

inversions among the $n!$ possible lists of n distinct objects. Thus, the expected number of inversions in any given list is $n(n-1)/4$.

Computing the standard deviation for the number of inversions is much more involved. See [25] for a proof. ∎

This leads to the following corollary that applies to both insertion sort and bubble sort.

Corollary 2.4.4. *In the worst-case, any sorting algorithm that sorts by swapping adjacent elements requires $n(n-1)/2$ swaps in the worst case, and $\sim n^2/4$ swaps on average, to sort an array of length n with distinct keys.*

2.5 Shell sort

Shell sort [37], was the first sorting algorithm with sub-quadratic complexity. It is named for its inventor, Donald Shell. We include it for its insights and for historical interest; it does not seem to be much used at present.

We have seen that swapping only adjacent items dooms us to quadratic worst case behavior. To get around this, Shell sort swaps non-adjacent items. Shell sort starts with an increment sequence:

$$h_t > h_{t-1} > \cdots > h_2 > h_1 = 1.$$

It uses insertion sort to sort

1. every h_t-th term starting at a_0, then a_1, …, then a_{h_t-1},
2. every h_{t-1}-th term starting at a_0, then a_1, …, then $a_{h_{t-1}-1}$,
3. &c.,
4. every term ($h_1 = 1$) starting at a_0, after which the array is guaranteed to be sorted.

For example, suppose we use the increment sequence 15, 7, 5, 3, 1, and have finished the 15-sort and 7-sort. Then we know that

$$\begin{aligned} a_0 &\le a_{15} \le a_{30} \le \cdots \\ a_0 &\le a_7 \le a_{14} \le \cdots . \end{aligned}$$

We also know that

$$a_{15} \le a_{22} \le a_{29} \le a_{36} \le \cdots .$$

Putting these together we see that

$$a_0 \le a_7 \le a_{22} \le a_{29} \le a_{36}.$$

After we have performed the sort using increment h_k, the array is h_k-sorted: all elements that are h_k terms apart are in the correct order:

$$a_i \le a_{i+h_k}.$$

The key to Shell sort's efficiency is the following fact: an h_k-sorted array remains h_k-sorted after sorting with increment h_{k-1}.

A good increment sequence $h_t, h_{t-1}, \ldots, h_1 = 1$ has the property that for any element a_p at the time of the h_k-sort there are only a few elements to the left of p that are larger than a_p. Ironically, Shell's original increment sequence,

$$h_t = \lfloor \frac{N}{2} \rfloor, \quad h_k = \lfloor \frac{h_{k+1}}{2} \rfloor,$$

has n^2 worst-case behavior. On the other hand, the sequences

$$2^k - 1 = 1, 3, 7, 15, 31, \ldots, \quad \text{(Hibbard [17])},$$
$$(3^k - 1)/2 = 1, 4, 13, 40, 121, \ldots \quad \text{(Pratt [31])},$$

yield $O(n^{3/2})$ worst-case complexity. There are other sequences that yield $O(n^{4/3})$ worst-case complexity.

2.6 Two humorous sorts

We conclude with two sorting algorithms for your amusement.

2.6.1 Bogosort

Given n items to sort, there are $n!$ possible permutations of the items. Bogosort (from "bogus sort") sorts by enumerating all possible permutations until it finds a sorted order. Bogosort is obviously a tongue-in-cheek algorithm.[4]

Proposition 2.6.1. *Bogosort requires $O(n!)$ operations to sort n items.*

Note that this is a worst-case upper bound; there is no lower bound since we might get lucky and find the sorted order for the first permutation we try.

For example, if $n = 3$ and the keys a, b, c, there are $3! = 3 \cdot 2 \cdot 1 = 6$ possible permutations:

$$abc \quad acb \quad bac \quad bca \quad cab \quad cba.$$

In this case bogosort is not out of the question. However, brute force enumeration of permutations rapidly becomes computationally infeasible once $n > 10$, e.g.,

$$13! = 6.2270 \times 10^9$$
$$20! = 2.4329 \times 10^{18}.$$

2.6.2 I can't believe it sorts

The following sorting algorithm, named "I can't believe it sorts", appears to have been only recently discovered in 2021 [13]. It was discovered in the process of hacking up insertion sort in order to concoct incorrect sorting algorithms (I suspect for an algorithms class). Despite looking obviously wrong, it does, in fact, work.

```
def i_cant_believe_it_sorts(a):
    '''Fung's "I can't believe it sorts" sorting algorithm.'''
    for i in range(len(a)):
        for j in range(len(a)):
            if a[i]<a[j]:
                a[i], a[j]=a[j], a[i]
    return a
```

Listing 2.6.1: "I can't believe it sorts."

[4] That said, one of my graduate school officemates, Mike Fagan, actually encountered an instance of bogosort in the wild. He was once handed a convoluted, lengthy piece of code at his day job. After puzzling over it for a bit, he realized that the code explicitly enumerated all 216 permutations of 6 items in order to sort them.

2.7 Exercises

Exercise 2.7.1. *Give the best case, worst case, and expected case of the order of the number of comparisons used by insertion sort, selection sort, and bubble sort on an input of length n when the input is*

1. *sorted (the elements appear smallest to largest),*
2. *reversely sorted (the elements appear largest to smallest),*
3. *identical (all the elements have the same value).*

Provide your answers in a table with the following format.

	Sorted	*Reversely sorted*	*Identical*
insertion sort			
selection sort			
bubble sort			

Exercise 2.7.2. *Repeat the previous exercise, only for the number of swaps.*

Exercise 2.7.3. *List all of the inversions in the sequence*

$$9, 42, 6, 54, 2, 28, 1, 7, 4, 14.$$

Exercise 2.7.4. *Suppose we have an ordered array of n items, all of whose entries are less than 100. Now append k additional items, all of whose values are greater than 100. What is the sort case complexity of insertion sort applied to this new array?*

Exercise 2.7.5. *Show that the following two versions of insertion sort are the same, insofar as that at the end of the inner* `while` *loop the partially sorted arrays* `a[ ]` *are the same for both algorithms.*

```
def insertion_sort1(a):
    '''Insertion sort.'''
    for i in range(1,len(a)):
        j = i
        while j > 0 and (a[j-1] > a[j]):
            a[j-1], a[j] = a[j], a[j-1]
            j -= 1
    return a

def insertion_sort2(a):
    '''Insertion sort.'''
    for i in range(1,len(a)):
        tmp = a[i]
        j = i
        while j > 0 and (a[j-1] > tmp):
            a[j] = a[j-1]
            j -= 1
        a[j] = tmp
    return a
```

Exercise 2.7.6. *Implement both algorithms from Exercise 2.7.5 in C++ and perform timing trials to establish that the second version is more efficient than the first.*

Exercise 2.7.7. *Suppose we have an array $a_0, \ldots, a_n$ and we exchange a_i and a_{i+k}, where a_i and a_{i+k} were originally out of order. Prove that this removes at least 1 and at most $2k-1$ inversions.*

Exercise 2.7.8. *Answer the following questions about Shell sort:*

1. *Show the steps (passes) of running Shell sort on the input 9, 8, 7, 6, 5, 4, 3, 2, 1 using the sequence 7, 3, 1.*
2. *What is the worst-case time complexity of Shell Sort to sort any list of size n that consists of a repeated sequence of 2, 1? Justify your answer.*

Exercise 2.7.9. *Prove that in Shell sort an h_k-sorted array remains h_k-sorted after sorting with increment h_{k-1}.*

Exercise 2.7.10. *What is the complexity of "I can't believe it sorts" in Listing 2.6.1?*

Chapter 3

Recurrence relations

In this chapter we review techniques for solving recurrence relations. A recurrence relation defines the elements of a sequence recursively. In order for the recurrence to have a well-defined solution, one or more initial conditions need to be specified. We care about solving recurrences because recursively defined algorithms lead to recursive definitions of the computational complexity.

If you took CSCI 243, much of this material will be familiar to you. If you took MATH 214, you should definitely look over this chapter carefully, as it includes techniques that you likely did not see.

3.1 Motivation

Let's look at examples where the analysis of the algorithms leads to recurrence relations.

3.1.1 Fast exponentiation

The obvious way to compute a^n is iterated multiplication: $a * a * a * \cdots * a$. This requires $n - 1$ multiplications. However, we can compute a^n more cheaply than this. Consider the following Python algorithm that computes a^n given $a > 0$ and integer $n \geq 0$.

```
def pow(a, n):
    '''Compute␣a**n␣in␣log␣n␣time.'''
    if (n == 0):
        return 1
    elif (n == 1):
        return a
    if (n % 2 == 0):  # n is even.
        return pow(a*a, n//2)
    else:             # n is odd.
        return pow(a*a, n//2) * a
```

Listing 3.1.1: Fast exponentiation.

Let $T(n)$ be the cost of the call to `pow(a,n)`. The function recursively calls itself with an exponent half as large, to $T(n)$ satisfies the recurrence

$$T(n) = T(\frac{n}{2}) + c_1,$$

where c_1 reflects the cost of the logical tests and arithmetic in the function, which is independent of n. We also need an initial condition for our recurrence to have a well-defined solution; when $n = 1$ the cost is at most c_0, another quantity independent of n:

$$T(n) = T(\frac{n}{2}) + c_1,$$
$$T(1) = c_0.$$

This is the same recurrence as bisection! We will see at a number of places in this course that certain algorithmic patterns give rise to the same recurrence relations describing their complexity. Here the pattern is to reduce the computation to a single problem that is half as big.

3.1.2 Multiplication

How fast can we multiply two n-digit numbers? For base-10 numbers, it is clear that if we multiply two n-digit numbers in the obvious way, the time required is proportional to n^2:

$$\begin{array}{rr} & 123 \\ \times & 456 \\ \hline & 738 \\ + & 7150 \\ + & 49200 \\ \hline & 56088 \end{array}$$

The same holds for base-2 multiplication:

$$\begin{array}{rr} & 010 \\ \times & 101 \\ \hline & 010 \\ + & 0000 \\ + & 01000 \\ \hline & 01010 \end{array}$$

Let n be a power of 2. Given two n-bit numbers $u = (u_{n-1} \cdots u_1 u_0)_2$ and $v = (v_{n-1} \cdots v_1 v_0)_2$, we can write

$$u = 2^{n/2} U_1 + U_0$$
$$v = 2^{n/2} V_1 + V_0,$$

where $U_1 = (u_{n-1} \cdots u_{(n/2)+1} u_{n/2})_2$ consists of the $n/2$ most significant bits of u, while $U_0 = (u_{(n/2)-1} \cdots u_1 u_0)_2$ are the $n/2$ least significant bits, and similarly for V_1, V_0.

Standard multiplication can then be written as

$$uv = 2^n U_1 V_1 + 2^{n/2}(U_1 V_0 + U_0 V_1) + U_0 V_0.$$

Since the multiplication by powers of 2 is simply a matter of shifting the bits, their cost is proportional to n. Likewise, the cost of the one addition are also proportional to n. We do have the four $n/2$-bit multiplications involving the U_i and V_i which we can perform recursively. This leads to the following recursion for the cost of standard multiplication:

$$T(n) = 4T(\frac{n}{2}) + c\frac{n}{2}.$$

As we will see later, $T(n) = \Theta(n^2)$, as our previous examples suggest.

Can we do better than $\Theta(n^2)$? The answer is yes. The first subquadratic approach [23] is based on the observation that

$$(U_1 - U_0)(V_1 - V_0) = U_1 V_1 - U_1 V_0 - U_0 V_1 + U_0 V_0,$$

so

$$\begin{aligned} uv &= 2^n U_1 V_1 + 2^{n/2}(U_1 V_0 + U_0 V_1) + U_0 V_0 \\ &= (2^n + 2^{n/2}) U_1 V_1 + 2^{n/2}(-U_1 V_1 + U_1 V_0 + U_0 V_1 - U_0 V_0) + (2^{n/2} + 1) U_0 V_0 \\ &= (2^n + 2^{n/2}) U_1 V_1 - 2^{n/2}(U_1 - U_0)(V_1 - V_0) + (2^{n/2} + 1) U_0 V_0. \end{aligned}$$

Now there are three $n/2$-bit multiplications,[1], two additions, and three subtractions. Compared to the first approach we have traded one addition and two subtractions for one fewer multiplications. This leads to the recursion

$$T(n) = 3T(\frac{n}{2}) + cn$$
$$T(1) = 1$$

for the work needed to multiply two n-bit integers. The solution is $\Theta(n^{\lg 3} = n^{1.5850\ldots})$, which is better than the quadratic complexity of standard multiplication.

In 1971, Schönhage and Strassen [35] presented an $O(n \lg n \lg\lg n)$ algorithm for multiplication which has practical application for the multiplication of extremely large numbers (tens or more thousands of digits). They also conjectured that multiplication of two n-bit numbers is $\Omega(n \lg n)$. Currently the best algorithm known is the one due to Harvey and van der Hoeven [15] mentioned in Section 1.6.1. It is $O(n \lg n)$. In their approach problems less than size

$$2^{71373980732566348976647585262078312064 1}.$$

are solved with a less efficient algorithm such as Schönhage-Strassen or even the conventional $\Theta(n^2)$ method, leading to an astoundingly large constant in their big-O bound.

3.2 Linear homogeneous recurrences

A linear homogeneous recurrence of order k involving $a_1, a_2, \ldots$ is one that can be put in the form

$$\alpha_k a_{n+k} + \alpha_{k-1} a_{n+k-1} + \cdots + \alpha_1 a_{n+1} + \alpha_0 a_n = 0,$$

where the α_i are constants in dependent of n. There are also initial conditions given for $a_0, \ldots, a_{k-1}$. **Linear** refers to the fact that there are no nonlinear functions of any of the a_i appearing, and **homogeneous** refers to the fact that the right-hand side is zero.

For instance, consider the Fibonacci numbers $0, 1, 1, 2, 3, 5, \ldots$:

$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

We can write the general recurrence as

$$F_{n+2} - F_{n+1} - F_n = 0,$$

a linear homogeneous difference recurrence of order 2.

Linear homogeneous recurrence relations can be solved using the following approach. If you have seen the solution of linear homogeneous differential equations, the technique will look familiar.[2] We posit that there is a particular solution of the form

$$a_n = cr^n$$

for some c and r. If we substitute this into the recurrence relation we obtain a polynomial equation in r,

$$\alpha_k c r^{n+k} + \alpha_{k-1} c r^{n+k-1} + \cdots + \alpha_1 c r^{n+1} + \alpha_0 c r^n = 0,$$

and, after dividing by cr^n,

$$\alpha_k r^k + \alpha_{k-1} r^{k-1} + \cdots + \alpha_1 r + \alpha_0 = 0,$$

If $r_0, \ldots, r_k$ are the roots of this polynomial, then the general solution of the recurrence relation has the form

$$a_n = c_0 r_0^n + c_1 r_1^n + \cdots + c_k r_k^n.$$

[1] Remember that multiplications by powers of 2 are just one-bit shifts to the left.

[2] If you haven't seen the solution of linear homogeneous differential equations, there is still time.

We then use the initial conditions to solve for $c_0, \ldots, c_k$.

For example, consider the linear homogeneous recurrence relation

$$\begin{aligned} a_{n+1} &= a_{n-1} - 3a_n, \\ a_0 &= 0, \\ a_1 &= 1. \end{aligned}$$

Substituting r^n into the recurrence yields

$$r^{n+1} + 3r^n - r^{n-1} = 0$$

or

$$r^2 + 2r - 3 = (r-1)(r+3) = 0,$$

for which the $r_1 = 1$ and $r_2 = -3$.

The general solution of the recurrence thus has the form

$$a_n = c_1 1^n + c_2(-3)^n = c_1 + c_2(-3)^n.$$

We can determine c_1 and c_2 from the initial conditions:

$$a_0 = c_1 + c_2 = 0,$$

so $c_1 = -c_2$, from which it follows that

$$a_n = c_1 - c_1(-3)^n.$$

Since

$$a_1 = c_1 - c_1(-3) = 1,$$

we have $c_1 = 1/4$. The general solution is then

$$a_n = \frac{1}{4} - \frac{1}{4}(-3)^n.$$

3.3 Solving recurrences via forward iteration

Forward iteration involves

1. computing successive values of the recursively defined quantity starting from the initial condition,
2. power-reading the resulting sequence to guess the general form of the solution, and
3. proving that our guess at the solution is correct.[3]

This approach relies on a large element of luck, is not systematic, and is frequently unsuccessful. Do not rely on it!

[3] "One, two, three, yep!" is not a proof.

3.3.1 Example: $T(n) = 2T(n/2) + cn$

Consider

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(\frac{n}{2}) + cn & \text{if } n \geq 2. \end{cases}$$

We have

$$\begin{aligned} T(1) &= 1 \\ T(2) &= 2T(1) + c \cdot 2 = 2c + 2 \\ T(4) &= 2T(2) + c \cdot 4 = 2(2c + 2) + 4c = 8c + 4 \\ T(8) &= 2T(4) + c \cdot 8 = 2(8c + 4) + 8c = 24c + 8 \\ T(16) &= 2T(8) + c \cdot 16 = 2(24c + 8) + 16c = 64c + 16. \end{aligned}$$

If we are observant, we will notice that the pattern is

$$T(2^m) = cm2^m + 2^m,$$

or

$$T(n) = cn \lg n + n.$$

We can confirm this guess by induction. When $n = 1$ we have

$$c1 \lg 1 + 1 = 1 = T(1).$$

Now suppose that the result holds for some $n = 2^m \geq 1$:

$$T(2^m) = cm2^m + 2^m.$$

We wish to show that

$$T(2^{m+1}) = c(m + 1)2^{m+1} + 2^{m+1}.$$

By the recurrence and the inductive hypothesis we have

$$T(2^{m+1}) = 2T(2^m) + c2^{m+1} = 2(cm2^m + 2^m) + c2^{m+1} = cm2^{m+1} + 2^{m+1} + c2^{m+1} = c(m + 1)2^{m+1} + 2^{m+1},$$

as desired.

3.3.2 Example: $T(n) = 7T(n/2) + 14n^2$

Next we look at an example that illustrates the limitations of this approach. Consider

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 7T(\frac{n}{2}) + 14n^2 & \text{if } n \geq 2. \end{cases} \tag{3.3.1}$$

We have

$$\begin{aligned} T(1) &= 1 \\ T(2) &= 7 + 14 \times 2^2 = 63 \\ T(4) &= 7 \times T(2) + 14 \times 4^2 = 665 \\ T(8) &= 7 \times T(4) + 14 \cdot 8^2 = 4733 \\ T(16) &= 7 \times T(8) + 14 \cdot 16^2 = 36715. \end{aligned}$$

Is the general pattern clear to you?[4]

[4] It involves $n^{\lg 7}$.

3.4 Solving recurrences via backward iteration

In this section we review a general technique for solving recurrence relations that we will call backward iteration.[5] The idea of backward iteration is to apply the recurrence repeatedly until the expression is reduced to explicitly known quantities and there is no longer any recursive definition involved.

Here are some tips in connection with this technique.

- Simplifying expressions in backward iteration is usually a **bad** idea since it obscures how things depend on the level of iteration. For instance, seeing a geometric sum such as $7^3 + 7^2 + 7 + 1$ appear is more illuminating than the simplified expression 400.
- In recursions that arise in divide-and-conquer strategies, e.g.,

  $$T(n) = T(\frac{n}{2}) + cn,$$

  a common mistake when applying backward iteration is to write

  $$T(\frac{n}{2}) = T(\frac{n}{2^2}) + cn$$

  when it should be

  $$T(\frac{n}{2}) = T(\frac{n}{2^2}) + c\frac{n}{2}.$$

- In a proof by induction, you should make clear the inductive basis, the inductive hypothesis, and what you need to prove in the inductive step.

We illustrate the method with a number of examples.

3.4.1 Example: $T(n) = T(n/2) + c$

We begin with the recurrence relation we encountered in Section 3.1 when discussing bisection and fast exponentiation. Consider

$$T(n) = \begin{cases} c_0 & \text{if } n = 1 \quad \text{initial condition,} \\ T(n/2) + c_1 & \text{if } n \geq 2 \quad \text{recurrence relation,} \end{cases}$$

where we assume n is a power of 2: $n = 2^k$.

Iteration of the recurrence. Start with the general recurrence as Iteration 1:

$$T(n) = T(\frac{n}{2}) + c_1.$$

Iteration 2:

$$T(n) = \left(T(\frac{n}{2^2}) + c_1\right) + c_1 = T(\frac{n}{2^2}) + 2c_1.$$

Iteration 3:

$$T(n) = \left(T(\frac{n}{2^3}) + c_1\right) + 2c_1 = T(\frac{n}{2^3}) + 3c_1.$$

Iteration 4:

$$T(n) = \left(T(\frac{n}{2^4}) + c_1\right) + 3c_1 = T(\frac{n}{2^4}) + 4c_1.$$

At this point the pattern is clear:

[5] There does not seem to be a settled name for this method.

Conjecture 3.4.1. *At the end of the m-th iteration,*

$$T(n) = T(\frac{n}{2^m}) + mc_1.$$

Proof of the conjecture. *Inductive basis.* When $m = 1$,

$$T(n) = T(\frac{n}{2}) + c_1 = T(\frac{n}{2^1}) + c_1 \cdot 1 = T(\frac{n}{2^m}) + mc_1.$$

Inductive hypothesis. Now suppose that for some $m \geq 1$,

$$T(n) = T(\frac{n}{2^m}) + mc_1.$$

Inductive step. We wish to show that

$$T(n) = T(\frac{n}{2^{m+1}}) + (m+1)c_1.$$

From the general recurrence we know that

$$T(\frac{n}{2^m}) = T(\frac{n}{2^{m+1}}) + c_1.$$

Thus, from the inductive hypothesis we have

$$T(n) = \left(T\left(\frac{n}{2^{m+1}}\right) + c_1\right) + mc_1 = T\left(\frac{n}{2^{m+1}}\right) + (m+1)c_1,$$

which is what we wished to show.

Reducing the recursive expression. Choosing $m = k$, then, since $n = 2^k$ (and $k = \log_2 n$) we obtain

$$T(n) = T(\frac{n}{2^k}) + kc_1 = T(1) + kc_1 = c_0 + c_1 \log_2 n.$$

Conclusion. Thus, the solution of

$$T(n) = \begin{cases} c_0 & \text{if } n = 1 \quad \text{initial condition,} \\ T(n/2) + c_1 & \text{if } n \geq 2 \quad \text{recurrence relation,} \end{cases}$$

for $n = 2^k$ is

$$T(n) = c_1 \log_2 n + c_0.$$

3.4.2 Example: $T(n) = 3T(n/2) + c$

Suppose $n = 2^m$. From the recursion

$$T(n) = 3T(n/2) + c$$
$$T(1) = 1$$

we obtain

$$T(n/2) = 3T(n/4) + c$$

so

$$T(n) = 9T(n/4) + 3c + c.$$

However,

$$T(n/4) = 3T(n/8) + c,$$

so

$$T(n) = 27T(n/8) + 9c + 3c + c.$$

Since

$$T(n/8) = 3T(n/16) + c,$$

we have

$$T(n) = 81T(n/16) + 27c + 9c + 3c + c.$$

Now a pattern has emerged: we conjecture that after k steps of this process

$$T(n) = 3^k T(2^{-k}n) + c\sum_{i=0}^{k-1} 3^i.$$

To prove this is correct, the original recurrence

$$T(n) = 3T(n/2) + c$$

shows that the conjecture is true when $k = 1$. Now suppose it is true for some $k \geq 1$:

$$T(n) = 3^k T(2^{-k}n) + c\sum_{i=0}^{k-1} 3^i.$$

We wish to show that

$$T(n) = 3^{k+1} T(2^{-(k+1)}n) + c\sum_{i=0}^{k} 3^i.$$

From the original recurrence we know that

$$T(2^{-k}n) = 3T(2^{-(k+1)}n) + c.$$

From the inductive hypothesis it follows that

$$\begin{aligned} T(n) &= 3^k \left(3T(2^{-(k+1)}n) + c\right) + c\sum_{i=0}^{k-1} 3^i \\ &= 3^{k+1} T(2^{-(k+1)}n) + 3^k c + c\sum_{i=0}^{k-1} 3^i \\ &= 3^{k+1} T(2^{-(k+1)}n) + c\sum_{i=0}^{k} 3^i, \end{aligned}$$

which is what we wished to show.

If we take $k = m$, so $2^k = 2^m = n$, we obtain

$$\begin{aligned} T(n) &= 3^m T(1) + c\sum_{i=0}^{m-1} 3^i \\ &= 3^m + c\frac{1-3^m}{1-3} \\ &= 3^m + \frac{c}{2}(3^m - 1) \\ &= 3^{\lg n} + \frac{c}{2}(3^{\lg n} - 1) \end{aligned}$$

Now apply Proposition 0.1.2:

$$3^{\lg n} = n^{\lg 3}.$$

Thus,

$$\begin{aligned} T(n) &= n^{\lg 3} + \frac{c}{2}(n^{\lg 3} - 1) \\ &= (1 + \frac{c}{2})n^{\lg 3} - \frac{c}{2}. \end{aligned}$$

3.4.3 Example: $T(n) = T(n-1) + n$

Consider the following recurrence relation:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \quad \text{initial condition,} \\ T(n-1) + n & \text{if } n \geq 2 \quad \text{recurrence relation.} \end{cases}$$

Iteration of the recurrence. If we call the general recurrence Iteration 1 (it doesn't matter how you number your iterations as long as you are consistent),

$$T(n) = T(n-1) + n,$$

then applying the recurrence repeatedly yields the following.
Iteration 2:

$$\begin{aligned} T(n) &= [T(n-2) + (n-1)] + n \\ &= T(n-2) + (n-1) + n. \end{aligned}$$

Iteration 3:

$$\begin{aligned} T(n) &= [T(n-3) + (n-2)] + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n. \end{aligned}$$

Iteration 4:

$$\begin{aligned} T(n) &= [T(n-4) + (n-3)] + (n-2) + (n-1) + n \\ &= T(n-4) + (n-3) + (n-2) + (n-1) + n. \end{aligned}$$

By now, the pattern should be apparent:

Conjecture 3.4.2. *After the m-th iteration,*

$$T(n) = T(n-m) + \sum_{i=0}^{m-1} (n-i).$$

Proof of the conjecture. We prove the conjecture using induction on m.

Proof *Inductive basis.* When $m = 1$ (i.e., Iteration 1),

$$T(n) = T(n-1) + n = T(n-m) + \sum_{i=0}^{m-1} (n-i).$$

Inductive hypothesis. Now suppose that for some $m \geq 1$,

$$T(n) = T(n-m) + \sum_{i=0}^{m-1} (n-i).$$

Inductive step. You should state what it is you need to prove, so you will know what you need to prove, and when you've proved it. In this case we wish to show that

$$T(n) = T(n - (m + 1)) + \sum_{i=0}^{m} (n - i).$$

By the inductive hypothesis, after m iterations,

$$T(n) = T(n - m) + \sum_{i=0}^{m-1} (n - i).$$

Applying the recurrence $T(n) = T(n - 1) + n$ yields

$$T(n - m) = T((n - m) - 1) + (n - m) = T(n - (m + 1)) + (n - m).$$

Applying the inductive hypothesis yields

$$T(n) = [T(n - (m + 1)) + (n - m)] + \sum_{i=0}^{m-1} (n - i) = T(n - (m + 1)) + \sum_{i=0}^{m} (n - i),$$

which is what we wished to show. ∎

We now know that after m iterations,

$$T(n) = T(n - m) + \sum_{i=0}^{m-1} (n - i).$$

However, we're not done yet—we still have a recursive expression.

Reducing the recursive expression. For this recurrence, the initial condition is

$$T(1) = 1.$$

When we reach this case, we can eliminate the recurrence entirely and replace $T(1)$ with the value 1. This occurs when $n - m = 1$, or $m = n - 1$.

Thus, after $n - 1$ iterations,

$$T(n) = T(1) + \sum_{i=0}^{n-2} (n - i) = 1 + \sum_{i=0}^{n-2} (n - i) = \sum_{i=0}^{n-1} (n - i).$$

All that is left is to reduce this summation.

Reducing the summation. Observe that

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} (n - i) \\ &= (n - 0) + (n - 1) + \cdots + (n - (n - 2)) + (n - (n - 1)) \\ &= n + (n - 1) + (n - 2) + \cdots + 2 + 1 = \sum_{i=1}^{n} i. \end{aligned}$$

The formula for the sum of the first n integers then tells us that

$$T(n) = \frac{n(n + 1)}{2}.$$

Alternatively, we could use the linearity of sums:

$$T(n) = \sum_{i=0}^{n-1} (n - j) = \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} j = n \sum_{i=0}^{n-1} 1 - \sum_{j=1}^{n-1} j.$$

Using the first two identities for arithmetic sums yields

$$T(n) = n((n-1) - 0 + 1) - \frac{(n-1)n}{2}.$$

All that's left is to clean up:

$$T(n) = n^2 - \frac{n^2 - n}{2} = \frac{2n^2 - (n^2 - n)}{2} = \frac{n^2 + n}{2} = \frac{n(n+1)}{2}.$$

Conclusion. For $n > 1$, the solution of the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n & \text{if } n \geq 2, \end{cases}$$

is

$$T(n) = \frac{n(n+1)}{2}.$$

3.4.4 Example: $T(n) = \frac{n}{n+2} T(n-1)$

Consider

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ \frac{n}{n+2}\, T(n-1) & \text{if } n \geq 2. \end{cases}$$

Iteration of the recurrence. Start with the general recurrence as Iteration 1:

$$T(n) = \frac{n}{n+2}\, T(n-1).$$

Iteration 2:

$$T(n) = \frac{n}{n+2}\,\frac{n-1}{n+1}\, T(n-2).$$

Iteration 3:

$$T(n) = \frac{n}{n+2}\frac{n-1}{n+1}\,\frac{n-2}{n} T(n-3).$$

Iteration 4:

$$T(n) = \frac{n}{n+2}\frac{n-1}{n+1}\,\frac{n-2}{n}\frac{n-3}{n-1} T(n-4).$$

Now the pattern has emerged:

Conjecture 3.4.3. *After the m-th iteration,*

$$T(n) = \frac{n(n-1)(n-2)\cdots(n-m+1)}{(n+2)(n+1)n\cdots(n-m+3)}\, T(n-m) = \frac{\prod_{i=0}^{m-1}(n-i)}{\prod_{i=0}^{m-1}(n-i+2)}\, T(n-m).$$

We could have simplified the product at each step, but that would have made the pattern more difficult to discern.

Proof of the conjecture. We prove the conjecture using induction on m.

Proof *Inductive basis.* When $m = 1$,

$$T(n) = \frac{n}{n+2}\,T(n-1) = \frac{\prod_{i=0}^{0}(n-i)}{\prod_{i=0}^{0}(n-i+2)}\,T(n-1)$$
$$= \frac{n}{n+2}\,T(n-1) = \frac{\prod_{i=0}^{m-1}(n-i)}{\prod_{i=0}^{m-1}(n-i+2)}\,T(n-m).$$

Inductive hypothesis. Now suppose that for some $m \geq 1$,

$$T(n) = \frac{\prod_{i=0}^{m-1}(n-i)}{\prod_{i=0}^{m-1}(n-i+2)}\,T(n-m).$$

Inductive step. We wish to show that

$$T(n) = \frac{\prod_{i=0}^{m}(n-i)}{\prod_{i=0}^{m}(n-i+2)}\,T(n-(m+1).$$

From the recurrence we know that

$$T(n-m) = \frac{n-m}{n-m+2}\,T(n-m-1).$$

Applying the inductive hypothesis yields

$$T(n) = \frac{\prod_{i=0}^{m-1}(n-i)}{\prod_{i=0}^{m-1}(n-i+2)}\,T(n-m)$$
$$= \frac{\prod_{i=0}^{m-1}(n-i)}{\prod_{i=0}^{m-1}(n-i+2)}\,\frac{n-m}{n-m+2}\,T(n-m-1)$$
$$= \frac{\prod_{i=0}^{m}(n-i)}{\prod_{k=0}^{m}(n-i+2)}\,T(n-(m+1)),$$

which is what we wished to show. ∎

Reducing the recursive expression. If we choose $m = n-1$, then $n-m = 1$ and

$$T(n) = \frac{\prod_{i=0}^{n-2}(n-i)}{\prod_{i=0}^{n-2}(n-i+2)}\,T(1) = \frac{\prod_{i=0}^{n-2}(n-i)}{\prod_{i=0}^{n-2}(n-i+2)}.$$

Now it makes sense to simplify the quotient:

$$\prod_{i=0}^{n-2}(n-i) = n\cdot(n-1)\cdots 5\cdot 4\cdot 3\cdot 2$$
$$\prod_{i=0}^{n-2}(n-i+2) = (n+2)\cdot(n+1)\cdot n\cdot(n-1)\cdots 5\cdot 4.$$

All but the first two and last two terms in each product cancel, so

$$\frac{\prod_{i=0}^{n-2}(n-i)}{\prod_{i=0}^{n-2}(n-i+2)} = \frac{3\cdot 2}{(n+2)(n+1)}.$$

Conclusion. For $n > 1$, the solution of the recurrence relation

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \quad \text{initial condition,} \\ \dfrac{n}{n+2}\,T(n-1) & \text{if } n \geq 2 \quad \text{recurrence relation.} \end{cases}$$

is

$$T(n) = \frac{6}{(n+2)(n+1)}.$$

3.5 One theorem to rule them all: the Master Theorem

So far we have explicitly solved recurrence relations. If we are only interested in the asymptotic behavior then we can appeal to a set of general results. These results are generically called the Master Theorem,[6] and concern the solution of recursions of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

As we have seen, recursions of this form appear in the analysis of divide-and-conquer algorithms.

There are many forms of the Master Theorem; here is the one we will use.

Theorem 3.5.1 (Master Theorem). *Let $T(n)$ be a nondecreasing, nonnegative function that satisfies*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad \text{for } n = b^k,\ k = 1, 2, \ldots$$
$$T(1) = c.$$

where $a \geq 1$, $b \geq 2$, and $c > 0$. If $f(n) = \Theta(n^d)$ for $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \lg n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

This result is true whether one chooses $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$ as the meaning of n/b.

Example 3.5.2. *If we apply the theorem to the recurrence*

$$T(n) = 7T(\frac{n}{2}) + n^3,$$

we have $a = 7$, $b = 2$, and $f(n) = n^3$. Then $d = 3$ since $f(n) = \Theta(n^3)$. Since $a = 7 < b^d = 2^3 = 8$, we have the first case of the theorem:

$$T(n) = \Theta(n^3).$$

Example 3.5.3. *If we apply the theorem to the recurrence*

$$T(n) = 2T(\frac{n}{2}) + cn,$$

then $a = 2$, $b = 2$, and $f(n) = cn$. Then $d = 1$ since $f(n) = \Theta(n)$. Since $a = b^d$, then the second case of the theorem tells us that

$$T(n) = \Theta(n \log_2 n).$$

Example 3.5.4. *Finally, applying the theorem to the recurrence*

$$T(n) = 7T(\frac{n}{2}) + n^2,$$

we have $a = 7$, $b = 2$, and $f(n) = n^2$. Then $d = 2$ since $f(n) = \Theta(n^2)$. Since $a = 7 > b^d = 2^2 = 4$, the third case of the theorem tells us that

$$T(n) = \Theta(n^{\lg 7}).$$

[6] This name was popularized by the widely used text *Introduction to Algorithms* by T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. The result itself is first articulated in [6].

3.5.1 Derivation

The Master Theorem can be derived by applying backward iteration to the recurrence:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad \text{for } n = b^k,\ k = 1, 2, \ldots$$
$$T(1) = c,$$

Assume for now that $n = b^k$. Start with the general recurrence as Iteration 1:

$$T(n) = aT(\frac{n}{b}) + f(n).$$

Iteration 2:

$$T(n) = a\left(aT(\frac{n}{b^2}) + f(\frac{n}{b})\right) + f(n) = a^2T(\frac{n}{b^2}) + af(\frac{n}{b}) + f(n)$$

Iteration 3:

$$\begin{aligned} T(n) &= a^2\left(aT(\frac{n}{b^3}) + f(\frac{n}{b^2})\right) + af(\frac{n}{b}) + f(n) \\ &= a^3T(\frac{n}{b^3}) + a^2f(\frac{n}{b^2}) + af(\frac{n}{b}) + f(n). \end{aligned}$$

Iteration 4:

$$\begin{aligned} T(n) &= a^3\left(T(\frac{n}{b^4}) + f(\frac{n}{b^3})\right) + a^2f(\frac{n}{b^2}) + af(\frac{n}{b}) + f(n) \\ &= a^4T(\frac{n}{b^4}) + a^3f(\frac{n}{b^3}) + a^2f(\frac{n}{b^2}) + af(\frac{n}{b}) + f(n). \end{aligned}$$

We can now guess the pattern:

Conjecture 3.5.5. *At iteration m of this process,*

$$T(n) = a^mT\left(\frac{n}{b^m}\right) + \sum_{i=0}^{m-1} a^if\left(\frac{n}{b^i}\right).$$

3.5.2 Proof of the conjecture

Inductive basis. When $m = 0$ (i.e., no iterations),

$$T(n) = aT(\frac{n}{b}) + f(n) = a^mT\left(\frac{n}{b^m}\right) + \sum_{i=0}^{m-1} a^if\left(\frac{n}{b^i}\right).$$

Inductive hypothesis. Now suppose that for some $m \geq 0$,

$$T(n) = a^mT\left(\frac{n}{b^m}\right) + \sum_{i=0}^{m-1} a^if\left(\frac{n}{b^i}\right).$$

Inductive step. We wish to show that

$$T(n) = a^{m+1}T\left(\frac{n}{b^{m+1}}\right) + \sum_{i=0}^{m} a^if\left(\frac{n}{b^i}\right).$$

From the general recurrence we know that

$$T\left(\frac{n}{b^m}\right) = aT(\frac{n}{b^{m+1}}) + f\left(\frac{n}{b^m}\right).$$

The inductive hypothesis then yields

$$\begin{aligned} T(n) &= a^m \left(aT(\frac{n}{b^{m+1}} + f\left(\frac{n}{b^m}\right)\right) + \sum_{i=0}^{m=1} a^i f\left(\frac{n}{b^i}\right) \\ &= a^{m+1} T(\frac{n}{b^{m+1}}) + a^m + f\left(\frac{n}{b^m}\right) + \sum_{i=0}^{m=1} a^i f\left(\frac{n}{b^i}\right) \\ &= a^{m+1} T(\frac{n}{b^{m+1}}) + \sum_{i=0}^{m} a^i f\left(\frac{n}{b^i}\right), \end{aligned}$$

as threatened.

3.5.3 Reducing the recurrence

For simplicity we will only solve the case $f(n) = n^d$. The more general case $f(n) = \Theta(n^d)$ follows from a very similar argument using upper and lower bounds on $f(n)$ in terms of n^d.

Recall we are assuming $n = b^k$. If we choose $m = k$, then

$$T(n) = a^k T(\frac{n}{b^k}) + \sum_{i=0}^{k-1} a^i \left(\frac{n}{b^i}\right)^d = a^k T(\frac{n}{b^k}) + n^d \sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i.$$

For the first term we have

$$a^k T(1) = ca^k = ca^{\log_b n} = cb^{\log_b a \ \log_b n} = cb^{\log_b n \ \log_b a} = cn^{\log_b a}.$$

If $a/b^d = 1$, then

$$n^d \sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i = n^d k = n^d \log_b n.$$

We then have

$$T(n) = cn^{\log_b a} + n^d \log_b n.$$

Since $a = b^d$, we have

$$\log_b a = \log_b b^d = d,$$

so

$$T(n) = cn^d + n^d \log_b n = \Theta(n^d \log_b a).$$

This is case (2) in Theorem 3.5.1.

Meanwhile, if $a/b^d \neq 1$, then the formula for a geometric sum yields

$$n^d \sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i = n^d \frac{1 - \left(\frac{a}{b^d}\right)^k}{1 - \frac{a}{b^d}} = n^d \frac{\left(\frac{a}{b^d}\right)^k - 1}{\frac{a}{b^d} - 1}.$$

Reasoning as above, we see that

$$\left(\frac{a}{b^d}\right)^k = \frac{a^{\log_b n}}{b^{d \log_b n}} = \frac{b^{\log_b a \ \log_b n}}{b^{d \log_b n}} = \frac{n^{\log_b a}}{n^d}.$$

Thus, if $a/b^d \neq 1$,

$$T(n) = cn^{\log_b a} + n^d \frac{\frac{n^{\log_b a}}{n^d} - 1}{\frac{a}{b^d} - 1} = cn^{\log_b a} + \frac{n^{\log_b a} - n^d}{\frac{a}{b^d} - 1}.$$

If $a < b^d$, then $\log_b a < d$, so $n^{\log_b a} < n^d$ and

$$T(n) = \Theta(n^d).$$

This is case (1) in Theorem 3.5.1.

On the other hand, if $a > b^d$, then $\log_b a > d$, so $n^{\log_b a} > n^d$ and

$$T(n) = \Theta(n^{\log_b a}).$$

This is case (3) in Theorem 3.5.1.

3.5.4 Some technicalities

We proved the Master Theorem under the assumption that n was a power of b. What if it isn't? Let $T(n)$ be a nondecreasing, nonnegative function. In this context we say $T(n)$ **grows sanely** if

$$T(2n) = \Theta(T(n)).$$

That is, there exist m, M and N such that

$$mT(n) \leq T(2n) \leq MT(n)$$

for all $n \geq N$. Functions like n^k and $\lg n$ satisfy this condition. On the other hand, rapidly growing functions like 2^n and $n!$ do not.

We now present a series of propositions that assure us that it is OK to assume n is a power of b in our complexity analysis.

Proposition 3.5.6. *Let $T(n)$ be a sanely growing function, with*

$$mT(n) \leq T(2n) \leq MT(n).$$

Then for any integer $k \geq 1$,

$$m^k T(n) \leq T(2n) \leq M^k T(n).$$

Proof Use induction on k. ∎

Proposition 3.5.7. *Let $T(n)$ be a sanely growing function. Then for any integer $b \geq 2$,*

$$T(bn) = \Theta(T(n)).$$

Proof Let k be such that $2^{k-1} \leq b \leq 2^k$. Because T is nondecreasing,

$$T(bn) \leq T(2^k n) \leq M^k T(n).$$

Similarly,

$$m^{k-1} T(n) \leq T(2^{k-1} n) \leq T(bn).$$

∎

Proposition 3.5.8. *Let $T(n)$ be a sanely growing function. If $T(n) = \Theta(f(n))$ for values of n that are powers of b, where $b \geq 2$, then $T(n) = \Theta(f(n))$ for all n.*

PROOF There exists M such that for all k sufficiently large we have

$$T(b^k) \le Mf(b^k).$$

Given n, choose k so that $b^k \le n \le b^{k+1}$. Since T is nondecreasing,

$$T(b^k) \le T(n) \le T(b^{k+1}) \le Mf(b^{k+1}).$$

Since T is sanely growing, there exists B such that

$$f(b^{k+1}) = f(b \cdot b^k) \le Bf(b^k).$$

Then

$$T(n) \le T(b^{k+1}) \le MBf(b^k) \le MBf(n).$$

The lower bound on $T(n)$ in terms of $f(n)$ can be derived in a similar manner. ∎

3.6 The Akra–Bazzi theorem

Despite the generality of Theorem 3.5.1, it has limitations. For instance, it does not apply to recurrences with uneven partitions such as

$$T(n) = T(n/2) + T(n/4) + \lg n.$$

Theorem 3.5.1 also does not apply cleanly to a one-term recurrence such as

$$T(n) = 2T(n/2) + \lg n.$$

In terms of Theorem 3.5.1, $f(n) = \lg n$, which is $O(n^d)$ for all $d > 0$. From this Theorem 3.5.1 tells us only that $T(n) = O(n^d)$ for all $d > 0$. But is $T(n)$, say, logarithmic?

An even more general theorem is due to Akra and Bazzi [2]. For an elegant and surprisingly short proof, see [27]. The version of the theorem we give here is due to Leighton.

Theorem 3.6.1. *Suppose*

$$T(x) = \begin{cases} \text{is defined} & \text{for } 0 \le x \le x_0, \\ \displaystyle\sum_{i=1}^{k} a_i T(b_i x + h_i(x)) + f(x) & \text{for } x > x_0, \end{cases}$$

where

1. $a_1, \ldots, a_k$ *are positive constants,*
2. $b_1, \ldots, b_k$ *are constants between* 0 *and* 1*,*
3. x_0 *is sufficiently large,*[7]
4. $|f'(x)| = O(x^c)$ *for some* $c \in \mathbb{N}$*,*
5. $|h_i(x)| = O(x/\lg^2 x)$*.*

Let p *satisfy*

$$\sum_{i=1}^{k} a_i b_i^p = 1.$$

Then

$$T(x) = \Theta\left(x^p \left(1 + \int_1^x \frac{f(u)}{u^{p+1}}\, du\right)\right).$$

[7] The actual description of what "sufficiently large" means here is rather technical and unedifying. See [27] for details.

The role of the h_i in Theorem 3.6.1 is interesting. Among other things, h_i can represent the presence of floor and ceiling functions, e.g.,

$$h(x) = \lfloor x \rfloor - \frac{x}{2}$$
$$T(\lfloor \frac{x}{2} \rfloor) = T(\frac{x}{2} + h(x)).$$

In Section 3.5.4 we had to specially address terms such as this in connection with Theorem 3.5.1, but for Theorem 3.6.1 the $h_i(x)$ terms take care of the technicalities in a straightforward manner. Moreover, the h_i do not figure in the final Θ estimate. The theorem says that if the perturbations $h_i(x)$ are not too large ($|h_i(x)| = O(x/\lg^2 x)$) then they do not affect the complexity bound.

In the following examples we use ln rather than lg in the recurrence relations as it simplifies the integration. If you want to use lg, see Proposition 0.1.1.

3.6.1 Example: $T(n) = 2T(n/2) + n$

We apply Theorem 3.6.1 with $a_1 = 2$, $b_1 = 1/2$, $h_1(x) = 0$, and $f(n) = n$. We have

$$a_1 b_1^p = 2^{1-p},$$

so to obtain $2^{1-p} = 1$ we choose $p = 1$. Since

$$\int_1^x \frac{u}{u^2}\, du = \ln u \Big|_1^x = \ln x,$$

we have

$$T(x) = \Theta\left(x\left(1 + \ln x\right)\right) = \Theta\left(x \ln x + x\right),$$

so from Proposition 0.1.1, $T(x) = \Theta(x \lg x)$.

3.6.2 Example: $T(n) = T(n/2) + \ln n$

We apply Theorem 3.6.1 with $a_1 = 1$, $b_1 = 1/2$, $h_1(x) = h_2(x) = 0$, and $f(n) = \ln n$. We have

$$a_1 b_1^p = 2^{-p}$$

so we have $p = 0$. Then

$$T(x) = \Theta\left(1 + \int_1^x \frac{\ln u}{u}\, du\right) = \Theta\left(1 + \frac{1}{2}(\ln u)^2 \Big|_1^x\right) = \Theta\left((\ln x)^2\right).$$

3.7 Solving recurrences via generating functions

Generating functions are another approach to solving recurrences. Given a sequence $a_0, a_1, a_2, \ldots$, the associated (ordinary) generating function is

$$G(z) = \sum_{k=0}^{\infty} a_k z^k. \tag{3.7.1}$$

For now, this is purely a formal power series—at this point we are not worried about whether or not the series converges.[8]

Here is the strategery for solving recurrences via generating functions:

1. Start with a recurrence relation involving the a_k.

[8] In mathematical terminology, we are working in the ring of formal power series over the real numbers.

2. Turn this into a relation involving the generating function $G(z)$.
3. Use the new relation to identify $G(z)$.
4. Find the series expansion (3.7.1) for $G(z)$.
5. The coefficients of the series expansion of $G(z)$ are the a_k we seek.

3.7.1 **Example**: $T(n) = T(n/2) + c$

Consider

$$T(n) = \begin{cases} c_0 & \text{if } n = 1 \quad \text{base case} \\ T\left(\frac{n}{2}\right) + c_1 & \text{if } n \geq 2 \quad \text{general case.} \end{cases}$$

Let $a_k = T(2^k)$. Then

$$a_0 = T(1) = c_0$$
$$a_k = a_{k-1} + c_1.$$

Multiply $a_k = a_{k-1} + c_1$ by z^k and sum over k:

$$\begin{aligned} a_1 z &= a_0 z + c_1 z \\ a_2 z^2 &= a_1 z^2 + c_1 z^2 \\ a_3 z^3 &= a_2 z^3 + c_1 z^3 \\ \vdots &= \vdots \end{aligned}$$

so

$$\begin{aligned} a_1 z + a_2 z^2 + \cdots &= (a_0 z + a_1 z^2 + \cdots) + c_1(z + z^2 + \cdots) \\ (a_0 + a_1 z + a_2 z^2 + \cdots) - a_0 &= z(a_0 + a_1 z + a_2 z^2 + \cdots) + c_1 z(1 + z + z^2 + \cdots) \\ G(z) - a_0 &= zG(z) + c_1 z \sum_{k=0}^{\infty} z^k, \end{aligned}$$

and, finally,

$$(1 - z)G(z) = c_0 + c_1 z \sum_{k=0}^{\infty} z^k.$$

Now we use the convergence of the series: if $|z| < 1$,

$$\sum_{k=0}^{\infty} z^k = \frac{1}{1 - z},$$

so

$$\begin{aligned} (1 - z)G(z) &= c_0 + c_1 z \frac{1}{1 - z}, \\ G(z) &= c_0 \frac{1}{1 - z} + c_1 z \frac{1}{(1 - z)^2}. \end{aligned}$$

However,[9]

$$\frac{z}{(1 - z)^2} = z \frac{d}{dz} \frac{1}{(1 - z)} = z \frac{d}{dz} \sum_{k=0}^{\infty} z^k = z \sum_{k=1}^{\infty} k z^{k-1} = \sum_{k=1}^{\infty} k z^k,$$

[9] Technically, we need to justify the interchange of differentiation and infinite summation. This interchange is valid because of the uniform convergence of the series that is the putative derivative for $|a| < 1$. The uniform convergence of this series can be confirmed using the Weierstrass M-test.

so

$$G(z) = \sum_{k=0}^{\infty} c_0 z^k + \sum_{k=1}^{\infty} c_1 k z^k$$
$$\sum_{k=0}^{\infty} a_k z^k = c_0 + \sum_{k=1}^{\infty} (c_0 + c_1 k) z^k.$$

Equating coefficients, we see that

$$a_k = \begin{cases} c_0 & \text{if } k = 0, \\ c_0 + c_1 k & \text{if } k > 0. \end{cases}$$

Since $a_k = T(2^k)$, we have

$$T(2^k) = c_0 + c_1 k,$$

or, if $n = 2^k$

$$T(n) = c_0 + c_1 \lg n.$$

3.7.2 Example: the Fibonacci numbers

We can apply the method of generating functions to the Fibonacci numbers,

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots,$$

where the n-th Fibonacci number F_n is given by

$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

Multiply $a_n = a_{n-1} + a_{n-2}$ by z^n and sum over n:

$$\begin{aligned} a_2 z^2 &= a_1 z^2 + a_0 z^2 \\ a_3 z^3 &= a_2 z^3 + a_1 z^3 \\ a_4 z^4 &= a_3 z^4 + a_2 z^4 \\ \vdots &= \vdots \end{aligned}$$

so

$$a_2 z^2 + a_3 z^3 + \cdots = z(a_1 z + a_2 z^2 + \cdots) + z^2(a_0 + a_1 z + a_2 z^2 + \cdots)$$
$$(a_0 + a_1 z + a_2 z^2 + \cdots) - (a_1 z + a_0) = z(a_0 + a_1 z + a_2 z^2 + \cdots) - a_0 z + z^2 G(z)$$
$$G(z) - (a_1 z + a_0) = zG(z) - a_0 z + z^2 G(z).$$

Since $a_0 = 0$ and a_1, this reduces to

$$G(z) - z = zG(z) + z^2 G(z),$$

whence

$$G(z) = \frac{z}{1 - z - z^2}.$$

We can simplify the quotient on the right using partial fractions. Before doing so, consider the factorization of $z^2 + z - 1$. If the roots of this polynomial are r_1, r_2, then

$$z^2 + z - 1 = (z - r_1)(z - r_2) = z^2 - (r_1 + r_2)z + r_1 r_2.$$

From this we see that $r_1 r_2 = 1$, so

$$z^2 + z - 1 = (z - r_1)(z - r_2) = r_1 r_2 (\frac{z}{r_1} - 1)(\frac{z}{r_2} - 1)$$
$$= (\frac{z}{r_1} - 1)(\frac{z}{r_2} - 1) = (r_2 z - 1)(r_1 z - 1).$$

Now we apply the method of partial fractions: we want to find c_1, c_2 such that

$$\frac{z}{1 - z - z^2} = -\frac{z}{z^2 + z - 1} = \frac{c_1}{r_1 z - 1} + \frac{c_2}{r_2 z - 1}.$$

Since

$$\frac{c_1}{r_1 z - 1} + \frac{c_2}{r_2 z - 1} = \frac{c_1(r_2 - 1) + c_2(r_1 - 1)}{z^2 + z - 1},$$

we want

$$c_1 r_2 + c_2 r_1 = -1$$
$$c_1 + c_2 = 0.$$

Thus, $c_2 = -c_1$, and

$$c_1 r_2 - c_1 r_1 = -1$$
$$c_1 = \frac{1}{r_1 - r_2}.$$

It follows that

$$G(z) = \frac{z}{1 - z - z^2} = c_1 \left(\frac{1}{r_1 z - 1} - \frac{1}{r_2 z - 1} \right) = c_1 \left(\sum_{n=0}^{\infty} r_1^n z^n - \sum_{n=0}^{\infty} r_2^n z^n \right).$$

Since

$$G(z) = \sum_{n=0}^{\infty} a_n z^n,$$

equating like terms yields

$$a_n = c_1 (r_1^n - r_2^n).$$

The roots of $z^2 + z - 1$ are

$$r_1 = \frac{1 + \sqrt{5}}{2}$$
$$r_2 = \frac{1 - \sqrt{5}}{2}.$$

Since

$$c_1 = \frac{1}{r_1 - r_2} = \frac{1}{\sqrt{5}},$$

we obtain the formula

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right]$$

for the n-th Fibonacci number.

3.7.3 Example: the Catalan numbers

The Catalan numbers C_n satisfy the recurrence

$$C_0 = 1 \tag{3.7.2}$$

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}, \; n \geq 0. \tag{3.7.3}$$

The Catalan numbers appear in connection with counting binary trees and groupings of associative binary operations.

Consider the generating function

$$G(z) = \sum_{n=0}^{\infty} C_n z^n.$$

It is not difficult to verify that

$$G^2(z) = \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} C_k C_{n-k} \right) z^n,$$

so from the recurrence we see that

$$G^2(z) = \sum_{n=0}^{\infty} C_{n+1} z^n.$$

Then

$$\begin{aligned} zG^2(z) &= \sum_{n=0}^{\infty} C_{n+1} z^{n+1}, \\ 1 + zG^2(z) &= G(z) \\ zG^2(z) - G(z) + 1 &= 0. \end{aligned}$$

Solving for $G(z)$ yields two possible solutions:

$$G(z) = \frac{1 \pm \sqrt{1 - 4z}}{2z}$$

Since $C_0 = 1$, we must have $G(0) = 1$, so we choose the solution that does not blow up as $z \to 0$:

$$G(z) = \frac{1 - \sqrt{1 - 4z}}{2z}.$$

You can verify that $\lim_{z \to 0^+} G(z) = 1$ using L'Hôpital's rule.

Recall the Taylor's series expansion for $\sqrt{1 + x}$:

$$\sqrt{1 + x} = \sum_{n=0}^{\infty} \frac{(-1)^n}{4^n (1 - 2n)} \binom{2n}{n} x^n,$$

where

$$\binom{2n}{n} = \frac{2n!}{n!(2n - n)!}$$

is the binomial coefficient. From this we obtain

$$\begin{aligned}
G(z) &= \frac{1}{2z}\left(1 - \sum_{n=0}^{\infty} \frac{(-1)^n}{4^n(1-2n)}\binom{2n}{n}(-4z)^n\right) \\
&= \frac{1}{2z}\sum_{n=1}^{\infty} \frac{1}{(2n-1)}\binom{2n}{n}z^n \\
&= \frac{1}{2}\sum_{n=1}^{\infty} \frac{1}{(2n-1)}\binom{2n}{n}z^{n-1} \\
&= \frac{1}{2}\sum_{n=0}^{\infty} \frac{1}{2(n+1)-1}\binom{2(n+1)}{(n+1)}z^n \\
&= \frac{1}{2}\sum_{n=0}^{\infty} \frac{1}{2n+1}\frac{(2n+2)(2n+1)}{(n+1)(n+1)}\binom{2n}{n}z^n \\
&= \frac{1}{2}\sum_{n=0}^{\infty} \frac{2n+2}{(n+1)(n+1)}\binom{2n}{n}z^n \\
&= \sum_{n=0}^{\infty} \frac{n+1}{(n+1)(n+1)}\binom{2n}{n}z^n \\
&= \sum_{n=0}^{\infty} \frac{1}{n+1}\binom{2n}{n}z^n.
\end{aligned}$$

From this we conclude that the n-th Catalan number is

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

From this we can determine the asymptotic growth of the Catalan numbers. Applying Stirling's approximation, we obtain

$$\binom{2n}{n} = \frac{2n!}{(n!)^2} \approx \frac{\sqrt{2\pi 2n}\ e^{-2n}(2n)^{2n}}{(\sqrt{2\pi n}\ e^{-n}n^n)^2} = \frac{\sqrt{2}\sqrt{2\pi n}\ e^{-2n}4^n n^{2n}}{(\sqrt{2\pi n})^2\ e^{-2n}n^{2n}} = \frac{4^n}{\sqrt{\pi n}},$$

so for large n we have

$$C_n \approx \frac{1}{n+1}\frac{4^n}{\sqrt{\pi n}}.$$

The Catalan numbers grow very quickly.

3.8 Exercises

To receive full credit for the exercises involving the backward iteration technique from Section 3.4, the solution must clearly and completely lay out all of the following.

1. First, iterate to discern the general pattern to derive an equation that shows what $T(n)$ looks like after i iterations of the substitution process.
2. Second, prove by induction that the equation derived is, in fact, correct. Proofs by induction should clearly state the inductive basis, the inductive hypothesis, and the inductive step.
3. Third, reduce the equation to arrive at a final solution explicitly in terms of n and not containing a recursive definition.
4. Any summations that you obtain as a consequence of backward iteration must be reduced to receive full credit for the problem.

Exercise 3.8.1. *The Fibonacci numbers* $0, 1, 1, 2, 3, 5, \ldots$ *are given by the linear homogenous recurrence relation*

$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

Use the technique of Section 3.2 to find an explicit formula for the Fibonacci numbers.

Exercise 3.8.2. *Solve the recurrence*

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(n-1) + (-1)^n n & \text{if } n > 1 \end{cases}$$

using backward iteration.

Exercise 3.8.3. *Solve the recurrence*

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 9T(\frac{n}{2}) + cn & \text{for } n > 1 \text{ and a power of 2} \end{cases}$$

using backward iteration. Check your result using Theorem 3.5.1.

Exercise 3.8.4. *Solve the recurrence*

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 3T(\frac{n}{2}) + n & \text{for } n > 1 \text{ and a power of 2.} \end{cases}$$

using backward iteration. Check your result using Theorem 3.5.1.

Exercise 3.8.5. *Solve the recurrence*

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 3T(\frac{n}{2}) + n^2 & \text{for } n > 1 \text{ and a power of 2.} \end{cases}$$

using backward iteration. Check your result using Theorem 3.5.1.

Exercise 3.8.6. *Show that if*

$$\begin{aligned} f(1) &= 1 \\ f(n+1) &= f(n) + (n+1)^k, \quad n \geq 1, \end{aligned}$$

then $f(n)$ *must be a polynomial of degree* $k+1$ *in* n*.*

Exercise 3.8.7. *Solve the recurrence (3.3.1) using backward iteration. Check your result using Theorem 3.5.1.*

Exercise 3.8.8. *Solve the recurrence*

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(\frac{n}{2}) + \lg n & \text{for } n > 1 \text{ a power of 2} \end{cases}$$

using backward iteration.

Exercise 3.8.9. *Solve the recurrence*

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 4\,T(\frac{n}{2}) + n^2 & \text{if } n > 1. \end{cases}$$

using backward iteration. Check your answer using Theorem 3.5.1.

Exercise 3.8.10. *Solve the recurrence*

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(\frac{n}{4}) + n & \text{if } n \geq 2. \end{cases}$$

using backward iteration. Check your answer using Theorem 3.5.1.

Exercise 3.8.11. *Solve the recurrence*

$$T(n) = 2T(\sqrt{n}) + \lg n$$
$$T(2) = 1.$$

using backward iteration. You may assume that $n = 2^m$ for some $m \geq 1$ such that all the square roots you encounter are integers. The master theorem does not apply to this recurrence.

Exercise 3.8.12. *Use Theorem 3.6.1 to obtain a Θ bound for $T(n)$, where*

$$T(n) = 2T(n/2) + \ln n.$$

Exercise 3.8.13. *Use Theorem 3.6.1 to obtain a Θ bound for $T(n)$, where*

$$T(n) = T(b_1 n) + T(b_2 n) + cn.$$

and

$$b_1 + b_2 = 1.$$

Exercise 3.8.14. *Use Theorem 3.6.1 to prove Theorem 3.5.1.*

Exercise 3.8.15. *Show that under the assumptions of Theorem 3.6.1, if there exists $\varepsilon > 0$ such that $f(x) = O(x^{p-\varepsilon})$, then $T(n) = \Theta(n^p)$.*

Exercise 3.8.16. *Show that under the assumptions of Theorem 3.6.1, if there exists $\varepsilon > 0$ such that $f(x) = \Omega(x^{p-\varepsilon})$ and $f(x)/x^{p+\varepsilon}$ is nondecreasing, then $T(n) = \Theta(f(n))$.*

Exercise 3.8.17. *Show that under the assumptions of Theorem 3.6.1, if $f(x) = \Theta(x^p)$, then $T(n) = \Theta(n^p \log n)$.*

Exercise 3.8.18. *Show that under the assumptions of Theorem 3.6.1, if $f(x) = x$, $a_i = 1$ for all i, and $\sum_{i=1}^{k} b_i = 1$, then $T(n) = \Theta(n \lg n)$.*

Exercise 3.8.19. *Show that under the assumptions of Theorem 3.6.1, if $f(x) = x$, $a_i = 1$ for all i, and $\sum_{i=1}^{k} b_i < 1$, then $T(n) = \Theta(n)$.*

Exercise 3.8.20. *Consider the following recursive algorithm for finding the maximum and minimum of an array of numbers.*

- *If there is only one item, it is the maximum and minimum, and no comparisons are needed.*
- *If there are two items, with one comparison we can determine the maximum and minimum.*
- *If there are three items, with three comparisons we can determine the maximum and minimum.*
- *If there are N items, $N > 3$, divide them as evenly as possible, find the maximum and minimum of each half, and with two comparisons determine the maximum and minimum of all N items.*

Questions:

(a) Suppose $N = 2^k$ for some $k \geq 0$. What is the exact number of comparisons done by this algorithm?

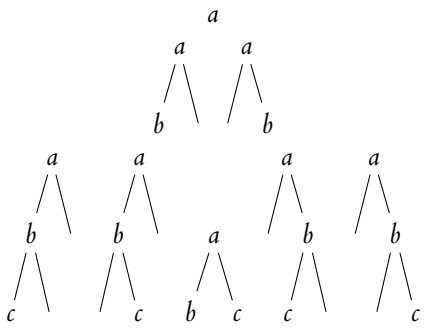
(b) Suppose $N = 3 \times 2^k$ for some $k \geq 0$. What is the exact number of comparisons done by this algorithm?

(c) *Suppose we modify the algorithm so that when $N > 2$ is even, but not divisible by 4, we split the input into size $N/2+1$ and $N/2-1$. What is the exact number of comparisons done by this modified algorithm?*

Hints.

- *Implement the algorithm and have it count the number of comparisons it uses.*
- *In parts (a) and (b), look at the binary tree that represents all the recursive calls.*
- *Use your implementation to check your solutions to (a) and (b).*
- *For part (c), use your implementation to first develop a conjecture of the correct answer.*

Exercise 3.8.21. *Show that C_n, the number of structurally distinct binary trees with n nodes, satisfies (3.7.2)–(3.7.3). Here are the cases for $n = 1, 2, 3$.*



Exercise 3.8.22. *Show that if A_n is the number of properly matched sequences of parentheses of length $2n$ then*

$$A_n = \sum_{i=1}^{n} A_i A_{n-i-1}.$$

Use an argument similar to that used to number of binary trees: Given all the sequences of shorter length, explain how to combine them to produce the sequences of length $2n$ in such a way that the sum clearly counts the number of sequences.

Hint: Prove the following result: If s is a properly matched sequence of parentheses of length $2n$, then s can be written uniquely in the form $(s_1)s_2$, where s_1 and s_2 are properly matched sequences of parentheses whose lengths add to $2n-2$. For example, (())()=([()])[()] and ()(())=([])[(())], with s_1 and s_2 indicated by [ ]. Note that s_1 and s_2 are allowed to be empty sequences with length 0, in the same way we allows for empty subtrees when counting binary trees.

Exercise 3.8.23. *Consider the following function:*

```
def tortoise_sort(A, i, j):
    '''This procedure sorts the slice A[i],A[i+1],...,A[j].}'''
    if i >= j:
        return
    m = (i + j)//2
    tortoise_sort(A, i, m)
    tortoise_sort(A, m+1, j)
    if A[m] > A[j]:
        A[m], A[j] = A[j], A[m]
    tortoise_sort(A, i, j-1)
    return A
```

[language=python] Let a be an array of n items starting with `a[0]`*. Let $T(n)$ be the time complexity of the call* `tortoise_sort(a, 0,n-1)`*. Derive a recurrence for $T(n)$.*

Chapter 4

Mergesort

4.1 Heapsort

We will begin with heapsort [40] since it relies on binary heaps and priority queues which you saw in CSCI 241. Heapsort is particularly useful if we only want to sort the largest or smallest k elements and $k \ll n$. The idea is simple:

- Build a maximum binary heap. This requires $\Theta(n)$ operations if we use Floyd's linear time heapification [12].
- Repeat until all items have been deleted:
  - Pop the maximum from the front of the queue and reheapify. This is at most $\lg n$ per operation, so at most $n \lg n$ overall.

As we repeatedly pop off the maximum, we obtain the sorted array, largest to smallest. As we do this, the heap shrinks in size, so we can place the sorted elements at the end of the array, so no extra space is needed. We summarize this as follows:

Proposition 4.1.1. *Heapsort requires $O(n \lg n)$ operations and no extra space. In the worst case heapsort is $\Theta(n \lg n)$. In the best case it is $\Theta(n)$.*

Proof In the worst case, n distinct keys, we must perform n heap operations at an average cost of $\lg n$ each. In the best case, n equal keys, we perform n heap operations each of $O(1)$ cost. ∎

A priority queue tailored for sorting is given in Listing 4.1.1 and the heapsort algorithm is given in Listing 4.1.2.

```
from copy import deepcopy
class max_pq:
    '''
    A max-priority queue / max-heap class for in-place heapsort.
    A parent and child are out-of-order if parent < child.
    '''
    def __init__(self, x):  # Build the heap from x for in-place heapsort.
        self.heap = [None for _ in range(len(x)+1)]
        self.current_size = len(x);
        self.heap[1:] = deepcopy(x)
        self.heapify()

    def heapify(self):  # Linear time heapification.
        for i in range(self.current_size//2, 0, -1):
            self.bubble_down(i)

    def pop_max(self):  # Pop the maximum value.
```

```
        if self.current_size == 0:
            raise RuntimeError("Attempting to retrieve the max of an empty heap.")
        top = self.heap[1];                          # Nab the first element.
        self.heap[1] = self.heap[self.current_size]; # The last shall be first...
        self.current_size -= 1
        self.bubble_down(1)  # Let the newly moved value find its place.
        return top

    def bubble_down(self, k):  # Move value[k] down the tree to its correct place.
        while (2*k <= self.current_size):
            j = 2*k
            if (j < self.current_size) and (self.heap[j] < self.heap[j+1]):
                j += 1
            if not (self.heap[k] < self.heap[j]):
                break
            self.heap[k], self.heap[j] = self.heap[j], self.heap[k]
            k = j;
```

Listing 4.1.1: A maximum binary heap for heapsort.

```
from pq import max_pq
def heapsort(a):
    '''Sort a in ascending order using heapsort.'''
    pq = max_pq(a)
    for k in range(len(a)-1, -1, -1):
        a[k] = pq.pop_max();
    return a
```

Listing 4.1.2: Heapsort.

4.2 Mergesort

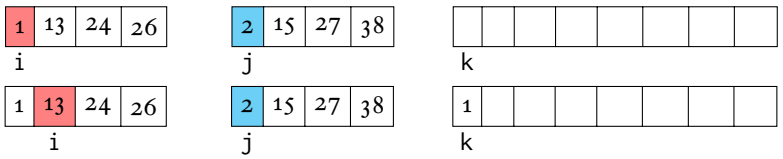
The idea of mergesort is simple:

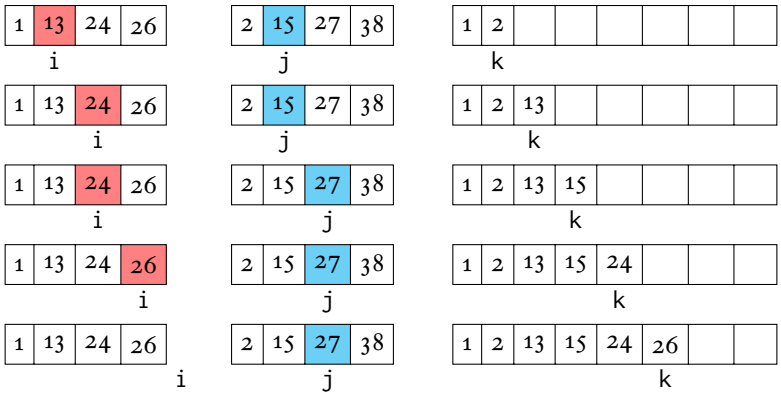
1. Divide the array in two, into a left and right half.
2. Sort each the left half and the right half.
3. Merge the two sorted left and right halves.

How do we sort each half? We use mergesort!

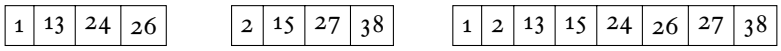
4.2.1 The merge operation

First we describe the merge operation, which merges two sorted arrays into a single sorted array. Let's merge `b = [1,13,24,26]` and `c = [2, 15, 27, 38]` into the array `a`. The index `i` keeps track of our location in `b`, the index `j` keeps track of our location in `c`, and the index `k` keeps track of our position in `a`. Initially they all point to the beginning of their respective array. We advance through `b` and `c`, choosing the smaller of the two terms `b[i]` and `c[j]`. The following figures show the situation after we have copied a term to `a` but before we have advanced the counters.





At this point there are no more elements of b to process, so we can simply copy all the remaining elements in c to a:



Listing 4.2.1 merges the sorted arrays b and c of length n into the array a.

```
def merge(b,c):
  '''Merge two sorted lists and return the result.'''
  n = len(b) + len(c)
  a = ["" for _ in range(n)]
  i = j = k = 0
  while (k < n):
    if i >= len(b):   # Done with b! copy the rest of c.
      a[k:] = c[j:]
      break
    elif j >= len(c): # Done with c! copy the rest of b.
      a[k:] = b[i:]
      break
    elif b[i] < c[j]:
      a[k] = b[i]
      i += 1
    else:
      a[k] = c[j]
      j += 1
    k += 1
  return a
```

Listing 4.2.1: Merging two arrays.

Now that we have the merge operation we can describe mergesort. Mergesort can be implemented both recursively and iteratively. We present a recursive version, since it is clearer.

```
from merge_arrays import merge
def mergesort(a):
    '''Textbook␣mergesort␣applied␣to␣a[␣].'''
    if len(a) <= 1:
        return a
    mid = len(a) // 2
    a[:mid] = mergesort(a[:mid])  # Sort the left half.
    a[mid:] = mergesort(a[mid:])  # Sort the right half.
    a = merge(a[:mid], a[mid:])   # Merge the results.
    return a
```

Listing 4.2.2: A toy mergesort.

When you first see mergesort you might ask: "Where is the sorting?!" The sorting is taking place in the merge operations.

Typically mergesort uses a second array to store the merged results, though it is possible to implement it as an in-place algorithm. Because of the overhead of recursive calls, it is not efficient to follow the mergesort recursion all the way down to problems of size 1 (see Exercise 4.5.3). Instead, it is more efficient to switch to insertion sort once the subproblems reach a certain size. The cutoff is hardware dependent; my most recent tests put the optimal cutoff to be in the range from 40 to 50.

4.2.2 Complexity analysis of mergesort

Merging two sorted arrays totaling n elements requires time proportional to n. For simplicity, we will assume n is a power of 2. If $T(n)$ is the time mergesort takes to sort an array of length n, then

$$T(n) = \begin{cases} 2T(n/2) + c_1 n & \text{if } n > 1, \\ c_0 & \text{if } n = 1. \end{cases}$$

Proposition 4.2.1. *The number of comparisons in mergesort is* $\Theta(n \lg n)$.

We will solve this recurrence via backwards iteration. Recall the steps required:

- work backwards to the initial condition;
- substitute repeatedly until a pattern emerges;
- conjecture the general form of the recurrence after m iterations;
- prove the conjecture is correct using induction on m;
- use the general form of the recurrence to determine how many iterations are required to reach the base case for the recurrence;
- substitute this value into the general form;
- clean up the expression.

Start with the general recurrence as iteration 1:

$$T(n) = 2T(n/2) + c_1 n.$$

Iteration 2: since $T(n/2) = 2T(n/2^2) + c_1(n/2)$, we have

$$T(n) = 2(2T(n/2^2) + c_1(n/2)) + c_1 n = 2^2 T(n/2^2) + 2c_1 n.$$

Iteration 3: since $T(n/2^2) = 2T(n/2^3) + c_1(n/2^2)$, we have

$$T(n) = 2^2(2T(n/2^3) + c_1(n/2^2)) + 2c_1 n = 2^3 T(n/2^3) + 3c_1 n.$$

Iteration 4: since $T(n/2^3) = 2T(n/2^4) + c_1(n/2^3)$, we have

$$T(n) = 2^3(2T(n/2^4) + c_1(n/2^3)) + 3c_1 n = 2^4 T(n/2^4) + 4c_1 n.$$

Conjecture 4.2.2. *At the m-th iteration,*

$$T(n) = 2^m T(n/2^m) + mc_1 n.$$

PROOF Inductive basis. When $m = 1$,

$$T(n) = T(n/2) + c_1 n = T(n/2^m) + mc_1 n.$$

Inductive hypothesis. Now suppose that for some $m \geq 1$,

$$T(n) = 2^m T(n/2^m) + mc_1 n.$$

Inductive step. We wish to show that

$$T(n) = 2^{m+1} T(n/2^{m+1}) + (m+1)c_1 n.$$

From the general recurrence we know that

$$T(n/2^m) = 2T(n/2^{m+1}) + c_1(n/2^m).$$

Thus, from the inductive hypothesis we have

$$\begin{aligned} T(n) &= 2^m(2T(n/2^{m+1}) + c_1(n/2^m) + mc_1 n \\ &= 2^{m+1}T(n/2^{m+1}) + (m+1)c_1 n. \end{aligned}$$

■

Now we can finish the reduction. Choosing $m = k = \lg n$, we obtain

$$T(n) = 2^k T(n/2^k) + kc_1 n = nT(1) + c_1 n \lg n = c_0 n + c_1 n \lg n.$$

as before.

4.3 A lower bound for sorting

So we have seen that both heapsort and mergesort require $\Omega(n \lg n)$ comparisons. A natural question is whether we can do better.

It turns out there is a limit to how well we can do when sorting by comparing pairs of terms.

Theorem 4.3.1. *No algorithm based on pairwise comparisons can guarantee sorting n items with fewer than $\lceil \lg n! \rceil \sim n \lg n$ comparisons.*

In this sense heapsort and mergesort are optimal.

This is a strong result—it is a limit on a large class of algorithms, and does not take into account any feature of the algorithms other than that they use pairwise comparisons. To prove this result, we abstract the behavior of such algorithms using a decision tree.

A decision tree for sorting is a binary tree in which each node represents a set of possible orderings. The root consists of the $n!$ possible orderings of the items to be sorted. The edges represent the results of comparisons, and a node comprises the orderings consistent with the comparisons made on the path from the root to the node. Each leaf consists of a single sorted ordering. The decision tree for sorting three items is shown in Figure 4.3.1.

Remarkably, the proof of Theorem 4.3.1 reduces to some properties of binary trees.

Lemma 4.3.2. *Let T be a binary tree of depth d. Then T has at most 2^d leaves.*

PROOF If $d = 0$, there is one leaf (the root). Now suppose $d > 0$, and assume the result is true for trees of depth $d - 1$ and less. Consider a tree of depth d. The root cannot be a leaf, and it has two subtrees, each of depth at most $d - 1$, so there are at most $2^{d-1} + 2^{d-1} = 2^d$ leaves. ■

Corollary 4.3.3. *A binary tree with L leaves must have depth at least $\lceil \lg L \rceil$.*

Now we can prove our main result. It is perhaps surprising that this result on sorting is really a simple consequence of the structure of binary trees.

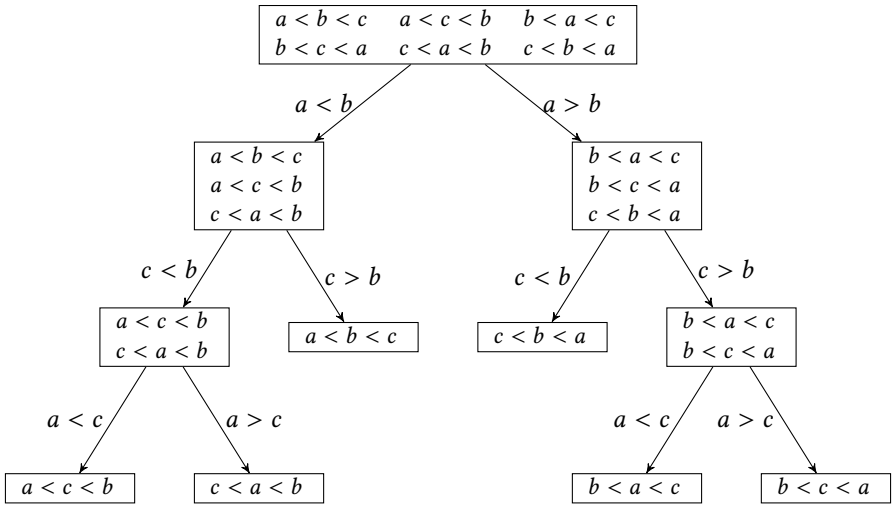


Figure 4.3.1: The decision tree for sorting three items.

Theorem 4.3.4. *No algorithm based on pairwise comparisons can guarantee sorting n items with fewer than $\lceil \lg n! \rceil = \Omega(n \lg n)$ comparisons.*

This $\lceil \lg n! \rceil$ lower bound is tight for $2 \leq n \leq 11$. Starting with $n = 12$ the number of comparisons can be more than this bound. For instance, when $n = 12$ it can be shown that 30 comparisons are required in the worst case, while $\lceil \lg 12! \rceil = 29$.

Proof A decision tree to sort n items must have $n!$ leaves. This requires a tree of depth $\lceil \lg n! \rceil$. Stirling's approximation (Theorem 0.4.2) tells us that

$$\lim_{n\to\infty} \frac{\lg\left(\sqrt{2\pi n} n^n e^{-n}\right)}{\lg n!} = 1,$$

so

$$\lim_{n\to\infty} \frac{\lg n!}{n \lg n} = \lim_{n\to\infty} \frac{\lg n!}{n \lg n} \frac{\lg(\sqrt{2\pi n} n^n e^{-n})}{\lg n!} = \lim_{n\to\infty} \frac{n \lg n - n \lg e + \frac{1}{2} \lg 2\pi n}{n \lg n} = 1,$$

which yields the $\Omega(n \lg n)$ bound. ∎

4.4 Stability

A sorting algorithm is stable if it preserves the relative order of equal keys. To understand why stability is a desirable property, suppose we wish to sort a set of titles first by title and then by author. We first sort by title:

Austen	Emma
Shakespeare	Hamlet
Shakespeare	King Lear
Shakespeare	Macbeth
Austen	Pride and Prejudice
Shakespeare	Romeo and Juliet
Austen	Sense and Sensibility

Table 4.1: Original list (sorted by title).

Now suppose we sort by author. A stable sort will preserve the order of the titles for all titles by the same author:

Austen	Emma
Austen	Pride and Prejudice
Austen	Sense and Sensibility
Shakespeare	Hamlet
Shakespeare	King Lear
Shakespeare	Macbeth
Shakespeare	Romeo and Juliet

Table 4.2: Stably sorted by author.

On the other hand, an unstable sort may change the relative positions of titles by the same author:

Austen	Pride and Prejudice
Austen	Emma
Austen	Sense and Sensibility
Shakespeare	Romeo and Juliet
Shakespeare	Hamlet
Shakespeare	Macbeth
Shakespeare	King Lear

Table 4.3: Unstably sorted by author.

The stability of some of the sorting algorithms that we are studying is summarized in Table 4.4.

Algorithm	Stable?
insertion sort	yes
selection sort	no
bubble sort	yes
Shell sort	no
heapsort	no
mergesort	yes
quicksort	no

Table 4.4: Stability of sorting algorithms.

4.5 Exercises

Exercise 4.5.1. *What would be a sufficient condition for heapsort to run in linear time?*

Exercise 4.5.2. *Trace the execution of mergesort on the array by showing the contents of the array* `a[]` *printed by the statement at line 29 (after the return from* `merge()`*), as well as the values of* `left`*,* `center`*, and* `right`*, when* **`left < right`**. *Execution begins with* `mergesort(a, 0, 7)`. *You will need to add more rows to the table.*

	a[0]	*a[1]*	*a[2]*	*a[3]*	*a[4]*	*a[5]*	*a[6]*	*a[7]*
left, center, right	*3*	*1*	*4*	*1*	*5*	*9*	*2*	*6*

Example (with letters):

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
left, center, right	*m*	*e*	*r*	*g*	*e*	*s*	*o*	*r*
0, 0, 1	*e*	*m*	*r*	*g*	*e*	*s*	*o*	*r*
2, 2, 3	*e*	*m*	*g*	*r*	*e*	*s*	*o*	*r*
0, 1, 3	*e*	*g*	*m*	*r*	*e*	*s*	*o*	*r*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

You will need to be careful in tracing the recursive calls in mergesort. I suggest you draw yourself a tree tracing the recursive calls, so that you see the correct ordering of the returns from `merge()`.

Also, note that the conditional `if (left < right)` *means that you will not indicate the results when mergesort returns from having "sorted" a single item (i.e., when* `right <= left`.

Exercise 4.5.3. *Suppose you apply mergesort to an array of length 2^{32}. How many recursive calls are made sorting arrays of length 64 and smaller?*

This is why we switch to a simple sort such as insertion sort once the arrays are sufficiently small—it is more efficient then this huge number of recursive calls.

Exercise 4.5.4. *Explain why insertion sort is stable.*

Exercise 4.5.5. *What must we do in the merge operation to ensure mergesort is stable?*

Exercise 4.5.6. *Give a non-recursive implementation of mergesort.*

Exercise 4.5.7. *Consider the following modification of mergesort. At each recursive call we divide the array into k arrays of the same size (so we will assume $n = k^m$ for some m). We then sort each subarray recursively in the same way, and then merge the k sorted subarrays by merging the first two, then merging the result with the third, and so on.*

(a) What is the recurrence satisified for the number of comparisons $T(n)$?

(b) How does the complexity compare to that of the original mergesort?

Exercise 4.5.8. *Use Stirling's approximation (Theorem 0.4.2) to obtain a more precise estimate of $\lg n!$. How does your more precise estimate compare with the true value in terms of relative error for small values of n?*

Exercise 4.5.9. *Draw the decision tree for sorting 4 items.*

Exercise 4.5.10. *Show that any algorithm based on pairwise comparisons to sort 4 items requires 5 comparisons in the worst case.*

Exercise 4.5.11. *If someone told you they had an algorithm that could sort 7 items in at most 12 comparisons, would you believe them? What if they said 13 comparisons? Explain your answer.*

Exercise 4.5.12. *Consider the following algorithm for sorting 6 numbers:*

- *Sort the first three numbers using Algorithm A.*
- *Sort the second three numbers using Algorithm B.*
- *Merge the sorted lists using Algorithm C.*

Show that this approach is suboptimal, disirregardless of the nature of Algorithms A, B, and C.

Exercise 4.5.13. *Suppose you are given a sorted list of n items followed by $f(n)$ randomly ordered items. How would you sort the entire set of items efficiently if*

(a) $f(n) = O(1)$?

(b) $f(n) = O(\lg n)$*?*

(c) $f(n) = O(\sqrt{n})$*?*

Exercise 4.5.14. *In Exercise 4.5.13, how large can* $f(n)$ *be for the entire list to be sortable in* $O(n)$ *time?*

Exercise 4.5.15. *Given an array of* n *elements, show how to remove all duplicate entries in the array in time* $O(n \lg n)$*.*

Exercise 4.5.16. *Given an array of* n *positive integers between* 1 *and* m*, show how to remove all duplicate entries in the array in time* $O(n)$*.*

Exercise 4.5.17. *Give an algorithm that finds the* k *largest elements of an array of length* n *in time* $O(n + k \lg n)$*.*

Exercise 4.5.18. *Suppose you are given* k *sorted arrays, each with* n *elements, and you wish to combine them into a single sorted array of* kn *elements.*

(a) Consider the following strategy. Use the merge operation from mergesort to merge the first two arrays, then merge the result with the third array, then merge the result with the fourth array, and so on. What is the time complexity of this approach in terms of k *and* n*?*

(b) Give a more efficient algorithm for this problem, using divide and conquer. What is the time complexity of your approach in terms of k *and* n*?*

Chapter 5

Quicksort

Quicksort [18, 19] is the most efficient general purpose sorting algorithm. It is an interesting algorithm insofar that its worst-case performance is quadratic, but it is efficient with very high probability.

5.1 Quicksort

Quicksort uses recursion to sort a set S by recursively sorting subsets of S:

1. If there are either 0 or 1 elements in S, then return.
2. Choose an element $v \in S$ to serve as the pivot.
3. Partition $S - \{v\}$ into two disjoint subsets S_1 and S_2 such that
   a) if $x \in S_1$ then $x \leq v$;
   b) if $x \in S_2$ then $x \geq v$.
4. Apply quicksort recursively to S_1 and S_2.

We have flexibility in the choice of pivot and how we partition the array.

Figure 5.1.1 gives an illustration of quicksort. Pivots are indicated in red.

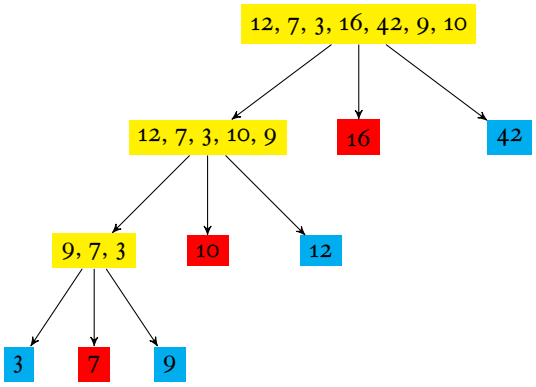


Figure 5.1.1: Quicksort.

5.1.1 A first attempt

Listing 5.1.1 presents a toy quicksort, adapted from [7], which the authors describe as "unfit for general use". It chooses the first element as the pivot; this is known as Lomuto pivoting. At each iteration of the for-loop, if $j > 0$ then elements a[1] to a[j] are strictly less than the pivot a[0], while a[j+1] is greater than or equal to a[0]. This implementation is quadratic if the input is already sorted.

```
void toy_quicksort (int *a, long int n)
{
  long int i, j;
  if (n <= 1) return;
  for (i = 1, j = 0; i < n; i++) {
    if (a[i] < a[0]) {
      std::swap(a[++j], a[i]);
    }
  }
  std::swap(a[0], a[j]);   // Move the pivot to where it belongs.
  qsort(a, j);             // Sort the left partition.
  qsort(a+j+1, n-j-1);     // Sort the right partition.
}
```

Listing 5.1.1: A toy quicksort. The array a has n elements.

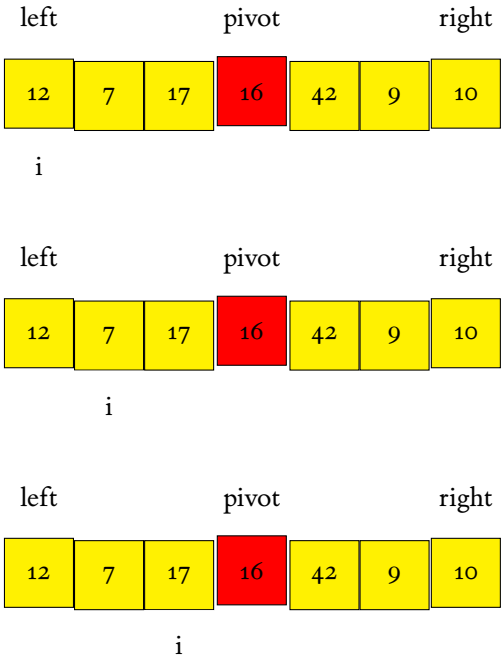
Listing 5.1.2 gives an implementation of quicksort in Python as a function named q(). It also uses the Lomuto pivot and is worst-case quadratic. It has the attraction of fitting on one line.

```
q = lambda a: q([x for x in a[1:] if x <= a[0]]) + [a[0]] + q([x for x in a[1:] if x > a[0]]) if a else []
```

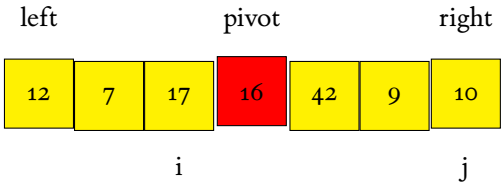
Listing 5.1.2: Quicksort based on code from Python One-Liners.

5.1.2 The partition

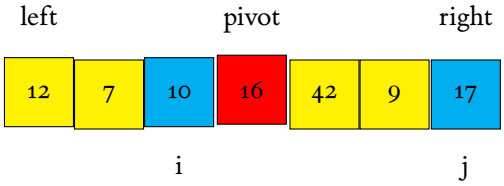
The partition is the trickiest part of quicksort, and easy to get wrong in an implementation. We will look at one efficient partition scheme. Assume for the moment that we have selected a pivot. We start scanning from left to right until we encounter a term as large as the pivot:



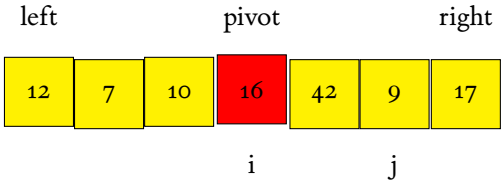
Stop the scan from left to right ane being a scan from right to left until we encounter a term as large as the pivot:



Now swap the i-th and j-th terms:



Now resume the advance from the left and right:



We are done when the pointers meet.

Listing 5.1.3 (Program 3 from [7]) illustrates this partitioning scheme. This version uses `a[0]` as the pivot (Lomuto pivoting).

```
void quicksort(int *a, int n)
{
  int i, j;
  if (n <= 1) return;
  i = 0;
  j = n;
  for (;;) {
    do {
      i++;
    } while (i < n && a[i] < a[0]);
    do {
      j--;
    } while (a[j] > a[0]);
    if (j < i) break;
    std::swap(a[i], a[j]);
  }
  std::swap(a[0], a[j]);
  quicksort(a, j);
```

```
19   quicksort(a+j+1, n-j-1);
20 }
```

Listing 5.1.3: Another quicksort with Lomuto pivoting.

5.1.3 Median of 3 partitioning

Ideally our pivot would be the median of the items we are sorting as this would make each partition half the size of the original data, which is the same as mergesort. This will guarantee $n \lg n$ behavior, which we know is the best we can do.

However, finding the median is a nontrivial problem in itself. A practical trade-off between the cost of finding the pivot and the quality of the pivot is given by median of three partitioning: choose the pivot by finding the median of the first, last, and middle elements of the array. This can be done with 3 comparisons.

We give an example of an implementation in Listing 5.1.4, following [39]. In this version the function that computes the median also rearranges the array slightly to avoid some duplicate comparisons. We switch to insertion sort once the arrays are sufficiently small.

```
#ifndef __CSCI303_QUICKSORT_HPP__
#define __CSCI303_QUICKSORT_HPP__
#include <vector>
using std::vector;
namespace csci303 {
  template <typename T>
  T median3(vector<T> &a, const long int left, const long int right)
  {
    // Cuidado!  this function may rearrange a[].
    long int center = (left + right) / 2;

    // Rearrange a[] so that a[left] <= a[center] <= a[right].
    if (a[center] < a[left]) std::swap(a[left], a[center]);
    if (a[right] < a[left])  std::swap(a[left], a[right]);
    if (a[right] < a[center]) std::swap(a[center], a[right]);

    // At this point a[left] <= a[center] <= a[right], so a[center] is the median.
    // Now swap a[center] and a[right-1]; a[right] is already in the correct partition
    // relative to the pivot in a[right-1].
    std::swap(a[center], a[right-1]);
    return a[right-1];
  }

  template <typename T>  // Internal version called recursively.
  void quicksort(vector<T> &a, const long int left, const long int right,
                 const int threshold=42)
  {
    if (right <= left) return;

    if (left + threshold <= right) {
      auto pivot = median3(a, left, right);

      // Begin partitioning
      long int i = left, j = right-1;  // Partitioning starts at a[right-2].
      for (;;) {
        while (a[++i] < pivot) {}
        while (pivot < a[--j]) {}
        if (i < j) {
```

```
39            std::swap(a[i], a[j]);
40          }
41          else {
42            break;
43          }
44        }
45        std::swap(a[i], a[right-1]); // Restore the pivot to the correct place.
46        quicksort(a, left, i-1);  // Sort the elements to the left of the pivot.
47        quicksort(a, i+1, right); // Sort the elements to the right of the pivot.
48      }
49      else {  // Switch to insertion sort.
50        for (auto i = left+1; i <= right; i++) {
51          auto tmp = a[i];
52          auto j = i;
53          while ((j > left) and (a[j-1] > tmp)) {
54            a[j] = a[j-1];
55            j--;
56          }
57          a[j] = tmp;
58        }
59      }
60    }
61
62    template <typename T>  // Public interface.
63    void quicksort(vector<T> &a, const int threshold=42)
64    {
65      quicksort(a, 0, a.size()-1, threshold);
66    }
67  }
68  #endif
```

Listing 5.1.4: Quicksort with median of 3 paritioning.

5.1.4 A deluxe implementation

Let's look at practical implementation of quicksort, modeled on Program 6 in the classic discussion given in [7]. This version uses a slightly more expensive pivot selection rule, median of median of three partitioning. We choose the pivot by finding the medians of the three triplets

$$(a[0], a[n/8], a[2n/8]) \quad (a[3n/8], a[4n/8], a[5n/8]) \quad (a[6n/8], a[7n/8], a[n-1]),$$

and taking the median of the three medians. This can be done with 12 comparisons.

Our implementation will also switch to insertion sort for short arrays, which improves performance. As with mergesort, we avoid a large number of recursive calls for short arrays.

Finally, we partition into three groups: those elements strictly less than, equal to, and strictly greater than the pivot. This is more efficient if there are large numbers of one or more repeated elements. This is sometimes called the Dutch national flag problem in view of the tripartite color scheme of the flag of Holland.

We begin with the median of median of three code in Listing 5.1.5.

```
1  #ifndef __CSCI303_MEDIAN3_HPP__
2  #define __CSCI303_MEDIAN3_HPP__
3  namespace csci303 {
4    template <typename T>
5    T median3 (const T &a, const T &b, const T &c)
6    {
7      // Find the median of a, b, c using at most 3 comparisons.
```

```
    T m;
    if (a < b) {
      if (b < c)      m = b;
      else if (a < c) m = c;
      else            m = a;
    }
    else {
      if (b > c)      m = b;
      else if (a > c) m = c;
      else            m = a;
    }
    return m;
  }
}
#endif
```

Listing 5.1.5: Finding the median of three items.

We then have the main procedure in Listing 5.1.6.

```
#ifndef __CSCI303_QUICKSORT3_HPP__
#define __CSCI303_QUICKSORT3_HPP__
#include <vector>
namespace csci303 {
  template <typename T>
  T median3(T a, T b, T c)
  {
    if (a > b) std::swap(a, b);
    if (b > c) std::swap(b, c);
    if (a > b) std::swap(a, b);
    return b;
  }

  template <typename T>  // Internal version called recursively.
  void quicksort3 (vector<T> &x, const long int left, const long int right,
                   const int threshold=54)
  {
    // Quicksort based on Program 6 in Bentley and McIlroy, featuring
    // tripartite partitioning:
    // * This subroutine sorts the entries from x[left] to x[right], inclusive.
    // * The switch to insertion sort is controlled by threshold.
    // * Values equal to the pivot are placed in the middle.

    if (right <= left) return;

    if (left + threshold <= right) {
      // Compute the pivot using median-of-median-of-three pivot selection.
      auto stride = (right - left)/8;
      auto med1  = median3(x[left],          x[left+stride],   x[left+2*stride]);
      auto med2  = median3(x[left+3*stride], x[left+4*stride], x[left+5*stride]);
      auto med3  = median3(x[left+6*stride], x[left+7*stride], x[right]);
      auto pivot = median3(med1, med2, med3);

      // Use a three-way partition that moves values equal to the pivot to
      // the left and right ends of the block being partitioned.
      auto a = left;  auto b = left;
      auto c = right; auto d = right;

      for (;;) {
```

```
40          while ((b <= c) && (x[b] <= pivot)) {
41            if (x[b] == pivot) std::swap(x[a++], x[b]);
42            b++;
43          }
44          while ((c >= b) && (x[c] >= pivot)) {
45            if (x[c] == pivot) std::swap(x[c], x[d--]);
46            c--;
47          }
48          if (b > c) {
49            break;
50          }
51          else {
52            std::swap(x[b++], x[c--]);
53          }
54        }
55
56        // Now move values equal to the pivot to the middle of the block.
57        auto s = std::min(a-left, b-a);
58        auto l = left;
59        auto h = b-s;
60        while (s > 0) {
61          std::swap(x[l++], x[h++]); s--;
62        }
63        s = std::min(d-c, right-d);
64        l = b;
65        h = right+1-s;
66        while (s > 0) {
67          std::swap(x[l++], x[h++]); s--;
68        }
69
70        // Apply quicksort recursively to the left and right ends of the block.
71        quicksort3(x, left, left+(b-a), threshold);
72        quicksort3(x, (right+1)-(d-c), right, threshold);
73      }
74      else { // Switch to insertion sort for small sorts.
75        for (auto i = left+1; i <= right; i++) {
76          auto tmp = x[i];
77          auto j = i;
78          while ((j > left) and (x[j-1] > tmp)) {
79            x[j] = x[j-1];
80            j--;
81          }
82          x[j] = tmp;
83        }
84      }
85    }
86
87    template <typename T>
88    void quicksort3(vector<T> &a, const int threshold=54)  // Public interface.
89    {
90      quicksort3(a, 0, a.size()-1, threshold);
91    }
92  }
93  #endif
```

Listing 5.1.6: Quicksort, version 1.

5.2 Introsort

Introsort [28] is the default sorting algorithm in the C++ standard library. It is a hybrid of quicksort and heapsort. Introsort starts with quicksort, but switches to heapsort whenever the number of levels of recursion exceed a specified threshold (e.g., $2\lfloor \lg n \rfloor$).

If introsort switches to heapsort, the subarrays that remain to be sorted will likely be much smaller than the original array and will fit in fast cache memory. This approach gives an $n \lg n$ guaranteed complexity, while making the most of the efficiency of quicksort.

5.3 Practical matters

As a practical matter,

- if the objects being sorted are large, we should swap pointers or references to the objects, rather than the objects themselves;
- in implementing a sorting algorithm in C++, we should consider whether to implement a general-purpose sorting algorithm and take advantage of an overloaded comparison operator or whether we should implement type-specific templated algorithms.

The C++ standard template library has the following templated sorting algorithms:

- **std::sort()**: introsort, guaranteed to be $n \lg n$
- **std::stable_sort()**: mergesort, guaranteed to be stable and
  - $n \lg n$ time, if a $\Theta(n)$-sized auxiliary array can be allocated;
  - $n \lg^2 n$ time for an in-place sort, otherwise
- **std::partial_sort()**: sort the k largest (or smallest) items

5.4 Summary of sorting algorithms

Let's add quicksort and introsort to our summary of sorting algorithms.

algorithm	stable?	in-place?	worst-case	best-case	expected case	extra space
insertion sort	yes	yes	n^2	n	n^2	$O(1)$
selection sort	no	yes	n^2	n^2	n^2	$O(1)$
bubble sort	yes	yes	n^2	n	n^2	$O(1)$
heapsort	no	yes	$n \lg n$	n	$n \lg n$	$O(1)$
mergesort	yes[1]	no[2]	$n \lg n$	$n \lg n$	$n \lg n$	$O(n)$
quicksort	no	yes	n^2	$n \lg n$	$n \lg n$	$O(\lg n)$[3]
introsort	no	yes	$n \lg n$	$n \lg n$	$n \lg n$	$O(\lg n)$[4]

[1] Provided our merge preserves order.
[2] The usual implementation is not in-place. There exist in-place versions.
[3] For the recursive calls.
[4] For the recursive calls.

5.5 Quickselect

We can use the partitioning idea in quicksort to solve the problem of finding k-th smallest element of an array a. For instance, we might wish to find the median of the elements. If we first sort the entire array, then the the k-th smallest element is $a[k]$. This gives us an $n \lg n$ algorithm for solving the problem.

Can we solve the problem more quickly? At first blush, it seems necessary to sort the array. However, this is not the case. The quickselect algorithm adapts the approach in quicksort to solve the problem in expected $O(n)$ time.

Quickselect finds the k largest element in a set S as follows. Given a set S, let $|S|$ be its cardinality.

- If there is 1 element in S, return $k = 1$.
- Choose an element v in S to serve as the pivot.
- Partition $S - \{v\}$ into two disjoint subsets S_1 and S_2 with the properties that
  - $x \leq v$ if $x \in S_1$, and
  - $x \geq v$ if $x \in S_2$.
- Now the search proceeds on S_1 and S_2:
  - If $k \leq |S_1|$, then the k-th smallest element must be in S_1, so call Quickselect(S_1,k).
  - If $k = 1 + |S_1|$, then the pivot is the k-th smallest, so return v.
  - Otherwise, the k-th smallest element must be in S_2, and it is the $(k - |S_1| - 1)$-th element of S_2, so return Quickselect(S_2, $k - |S_1| - 1$).

5.5.1 Examples

In the examples that follow, red denotes pivots, yellow denote the partition that we continue to search, and green denotes the partition that is ignored.

We first look for the median of 12, 7, 3, 16, 42, 9, 10. Since there are 7 items, $k = 4$. The median is 10.

1. Call Quickselect(S, 4).
2. Partition, then call Quickselect(S_1, 4).
3. Once again, partition. At this point $|S_1| = 3$, so the pivot is the 4-th element, and thus the answer.

This process is illustrated in Figure 5.5.1.

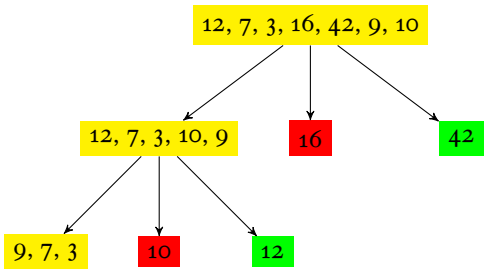


Figure 5.5.1: Finding the median.

Next we repeat this example but with different choices of pivots.

1. Call Quickselect(S, 4).
2. Partition. Since $|S_1| = 2$, we want the $k - |S_1| - 1 = 4 - 1 - 1 = 1$-st smallest element of S_2, so call Quickselect(S_2, 1).
3. Partition. Since we are inside the call Quickselect(S_2, 1), we want the 1-st smallest element, so we call Quickselect(S_1, 1), which immediately exits, returning 10.

These steps are illustrated in Figure 5.5.2.

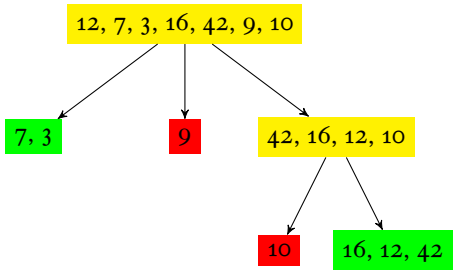


Figure 5.5.2: Finding the median.

5.5.2 Probabilistic analysis of quickselect

What is the time complexity of quickselect? At each recursive step quickselect ignores one partition—will this make it faster than quicksort?

In the worst case, quickselect behaves like quicksort, and has n^2 complexity. This occurs if one of the partitions is empty at each partitioning, and we have to look at all the terms in the other partition.

The best case behavior is linear. This occurs if at each partitioning the partitions are equal in size. Since quickselect ignores one partition at each step, its runtime $T(n)$ satisfies the recurrence

$$T(n) = T(\frac{n}{2}) + cn.$$

This leads to $T(n)$ being linear.

What is the expected behavior of quickselect? We will choose our pivot v randomly from the terms we are searching. Suppose v lies between the 25th and 75th percentiles of the terms (i.e., v is larger than 1/4 and smaller than 1/4 of the terms). This means that neither partition can contain more than 3/4 of the terms, so the partitions cannot be too imbalanced. Call such a pivot "good".

On average, how many v do we need to choose before we get a good one? A randomly chosen v is good with probability 1/2—a good pivot lies in the middle 50% of the terms. Choosing a good pivot is like tossing a coin and seeing heads. The expected number of tosses to see a heads is two. To see this, let E be the expected number of tosses before seeing a heads. Toss the coin. If it is heads, we are done; if it is tails (which occurs with probability 1/2) we have to toss it again, so

$$E = 1 + \frac{1}{2}E,$$

whence $E = 2$. Thus, on average, quickselect will take two partitions to reduce the array to at most 3/4 of the original size.

In terms of $T(n)$,

$$\text{expected value of } T(n) \leq T(3n/4) + \text{expected time to reduce the array to size} \leq 3n/4.$$

Since each partitioning step requires cn work, and we expect to need 2 of them to reduce the array size to $3n/4$ or less, we have

$$T(n) \leq T(3n/4) + cn.$$

Consider the more general recurrence

$$T(n) \leq T(\alpha n) + cn,$$

where $\alpha < 1$. At the k-th level of the recursion, starting with $k = 1$, there is a single problem of size at most $\alpha^k n$. The amount of work done at each level is thus at most $c\alpha^k n$. The recursion continues until

$$\alpha^m n \leq 1,$$

so $m \lg \alpha + \lg n \leq 0$, or

$$m \leq -\frac{\lg n}{\lg \alpha}.$$

Thus, the total amount of work is bounded above by

$$cn + c\alpha n + c\alpha^2 n + \cdots + c\alpha^{\lceil m \rceil} n = cn\frac{1-\alpha^{\lceil m+1 \rceil}}{1-\alpha} \leq cn\frac{1}{1-\alpha} = O(n).$$

Thus, the best case and expected case behavior of quickselect with a randomly chosen pivot is $O(n)$.

5.6 Exercises

Exercise 5.6.1. *Suppose the vector* a *is initially* 3, 1, 4, 1, 5, 9, 2, 6, 5, 3. *What does the vector look like after returning from the call*

```
partition(a, 0, a.size()-1);
```

where partition() is the partitioning algorithm below with median of 3 pivot selection?

```
void partition (std::vector<int> &a, int lo, int hi)
{
  if (hi <= lo) {
    return;
  }

  // Partition.
  const int &pivot = {median3(a, lo, hi)}; // Pivot selection.

  int i = {lo}, j = {hi-1};

  for (;;) {
    while (a[++i] < pivot) {}; // Advance scan from left.
    while (pivot < a[--j]) {}; // Advance scan from right.
    if (i < j) {
      std::swap(a[i], a[j]);
    }
    else {
      break;
    }
  }

  std::swap(a[i], a[hi-1]); // Restore the pivot.
}
```

Listing 5.6.1: Simple quicksort partition

```
const int& median3(std::vector<int> &a, int lo, int hi)
{
  int mid = {(lo + hi)/2};
  if (a[mid] < a[lo]) std::swap(a[lo], a[mid]);
  if (a[hi]  < a[lo]) std::swap(a[lo], a[hi]);
  if (a[hi] < a[mid]) std::swap(a[mid], a[hi]);
  std::swap(a[mid], a[hi-1]); // Place the pivot at a[hi-1];
  return a[hi-1];
}
```

Listing 5.6.2: Median o' 3 pivot selection

Exercise 5.6.2. *Give an example that shows that quicksort is not stable. Try to make your example as small as possible. You are free to choose your pivot however you wish, but you should clearly indicate which element is the pivot at each step.*

Exercise 5.6.3. *Let $k > 1$ be an integer. Suppose quicksort were to always pivot on the $\lceil n/k \rceil$-th smallest value. How many comparisons would be made then in the worst case? Rather than an exact count, give an asymptotic bound.*

Exercise 5.6.4. *(Based on Exercise 2.3.10 [36].)*

Chebyshev's inequality says that the probability that a random variable is more than k standard deviations away from its mean is no more than $1/k^2$. In mathematical notation, if x is a random variable, μ is its expected value (mean), and σ its the standard deviation, then

$$\text{Prob}\left(|x - \mu| > k\sigma\right) \leq \frac{1}{k^2}.$$

For instance, suppose $\mu = 10$ and $\sigma = 0.25$. What is the probability that $x \geq 20$? We have

$$k = \frac{|20 - 10|}{0.25} = 40$$
$$1/k^2 = 1/40^2 = 1/1600.$$

This tells us the probability that $x \geq 20$ is no greater than $1/1600$.

If $n = 1,000,000$, use Chebyshev's inequality to bound the probability that the number of comparisons used by quicksort is more than $500,000,000$.

Exercise 5.6.5. *Suppose we take advantage of the fact that insertion sort is more efficient than quicksort on small arrays by we switching to insertion sort once the subarray (partition) we are sorting has k or fewer elements. Show that this algorithm runs in $O(nk + n\lg(n/k))$ expected time. For simplicity assume n is a power of 2.*

Exercise 5.6.6. *Some authors suggest the following variant of quicksort. It takes advantage of the fact that insertion sort is efficient on nearly sorted data. When quicksort is called on a subarray (partition) with few than k elements, simply return without sorting. After the entire quicksort recursion is done, apply insertion sort to the entire array. Show that this algorithm runs in $O(nk + n\lg(n/k))$ expected time.*

Exercise 5.6.7. *Assume that in the quickselect algorithm* `quickselect(L, k)` *we choose the first number in L as the pivot.*

1. *Describe an input of length n that makes the algorithm exhibit a quadratic run-time Explain your answer.*
2. *Describe an input of length n that makes the algorithm exhibit a linear run-time. Explain your answer.*

Exercise 5.6.8 (*n*-derella). *Suppose you have a set of n potential princesses, all with different shoe sizes, and n corresponding glass slippers. That is, there is a unique 1-1 correspondence between slippers and possible princesses. Your aim is to find the appropriate slipper for each of the princesses.*

You are allowed to try glass slippers on a potential princess and determine whether the slipper is too small, too big, or fits perfectly. However, you cannot directly compare slippers (since they are glass and aren't labeled with their size), nor can you directly compare feet (since that would be weird).

Find an algorithm with expected case $\Theta(n\lg n)$ runtime that solves this problem, and explain why your algorithm has the desired complexity.

Exercise 5.6.9. *Give an algorithm that sorts an array of integers $a[0], \ldots, a[n-1]$ in $O(n+m)$ time, where*

$$m = \max_i a[i] - \min_i a[i].$$

For small $m \leq n$ this is linear time. Why does this not violate the $\Omega(n\lg n)$ bound from Theorem 4.3.1?

Chapter 6

Union-find

Union-find can be used to answer questions like the following:

- In a computer network, we know that certain pairs of computers are connected. How do we use that information to determine whether we can get traffic from one arbitrary computer to another?
- In a model of a porous medium like sand or gravel or clay, how much of the medium must be void before we can expect a fluid to percolate through it?
- In a social network, we know that certain people are friends. How do we use that information to determine whether we are a friend of a friend of a friend . . . of a friend of Momofoku Ando?

6.1 Equivalence classes

We will abstract these notions of connectedness in terms of equivalence relations. Recall that a relation R on a set S is a subset of $S \times S$, i.e., the set of ordered pairs (p, q) with $p, q \in S$. We say that p is related to q if $(p, q) \in R$. An equivalence relation is a relation R with these properties:

1. Reflexivity: p is related to p.
2. Symmetry: if p is related to q, then q is related to p.
3. Transitivity: if p is related to q, and q is related to r, then p is related to r.

Given an equivalence relation R, the equivalence class of p is the set of q related to p.

For example, we can say two nodes in an undirected graph are equivalent if they are connected by a path.

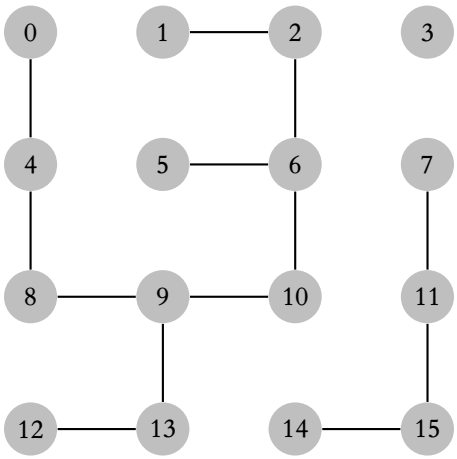


Figure 6.1.1: Connected components in a graph as equivalence classes.

In this graph there are three equivalence classes.

6.2 The dynamic equivalence problem

An equivalence relation on a set partitions the set into disjoint equivalence classes. We will write $p \sim q$ if p and q are in the same equivalence class. Given p and q, how can we determine if $p \sim q$?

The difficulty is that the equivalence classes may be defined indirectly. For example, in Figure 6.1.1 two nodes are in the same equivalence class if and only if they are connected by a path. However, the graph is specified by pairwise connections:

$$0 \sim 4, 4 \sim 8, 8 \sim 9, 1 \sim 2, 2 \sim 6, 9 \sim 13, 11 \sim 15, 14 \sim 15, 12 \sim 13, 7 \sim 11, 5 \sim 6, 6 \sim 10.$$

How can we decide whether $0 \sim 1$?

In the general version of the problem, we begin with a collection of disjoint sets $S_1, \ldots, S_n$, each with a single distinct element. Two operations exist on these sets:

1. FIND (p), which returns the identity of the equivalance class containing p;
2. UNION (p,q), which merges the equivalence classes of p and q.

In building up the connected components of a graph, when we are told of a connection between p and q we call UNION (p,q), which in turn calls FIND (p) and FIND (q). These operations are dynamic:

- the sets may change because of the UNION operation, and
- FIND must return an answer before the equivalence classes have been entirely constructed.

Denote the items by $0, 1, 2, \cdots, n-1$. We are given pairs of items (p, q), $0 \le p, q \le n-1$, which is interpreted as meaning $p \sim q$. In keeping with the graph example, we will refer to the items as vertices and say that p and q are connected if $p \sim q$. We will also refer to the equivalence classes as connected components or just components, thinking Figure 6.1.1.

We need a data structure that will represent known connections and allow us to answer

- Given arbitrary vertices p and q, are they connected?
- How many connected components are there?

We will use a vertex-indexed array `id[]` to represent the components. The value `id[p]` either directly or indirectly tells us which component p belongs to.

6.3 Quick-find

In the analysis of algorithms an invariant is a condition that is guaranteed to be true at specified points in the algorithm. We can use invariants and their preservation by an algorithm to prove that the algorithm is correct.

Our first attempt at solving the dynamic equivalence problem is called quick-find. A Python implementation is given in Listing 6.3.1. Initially, we do not know that any vertices are connected, so we initialize `id[p] = p` for all p (i.e., each vertex is initially in its own component).

```
class Quick_Find:
    # Quick-find: id[p] is the component of p.

    def __init__(self, n):
        self.id = list(range(n))

    def find(self, p):
        return self.id[p]

    def union(self, p, q):
```

```
        '''Union␣p's component into q's.'''
        p_id = self.find(p)
        q_id = self.find(q)
        if p_id == q_id:
            return
        for i in range(self.n):
            if self.id[i] == p_id:
                self.id[i] = q_id
```

Listing 6.3.1: Quick-find.

Quick-find maintains the invariant that p and q are connected if and only if `id[p] == id[q]`. Quick-find is so called because the FIND operation is a trivial, $O(1)$ operation. Since the FIND is trivial, all of the work is in the UNION operation and is, in the worst case, proportional to n.

The quick-find UNION preserves the invariant. However, in the worst case, the number of operations for a single call to UNION is $\propto N$. If there is only a single component, then we will need at least $N-1$ calls to UNION. In this situation each call to UNION requires work $\propto N$. This means that in this case, the work is at least $\propto N^2$, so in the worst case, quick-find is a quadratic algorithm.

6.4 Quick-union

Quick-union almost but not quite avoids the quadratic worst-case behavior of quick-find. In quick-union, given a vertex p, the value `id[p]` is the number of another vertex that is in the same component. We call such a connection a link.

In quick-union, UNION is $O(1)$ and the bulk of the work is in FIND. To determine which component p lies in, we start at p, follow the link from p to `id[p]`, follow the link from there (`id[id[p] ]`), and so on, until we come to a vertex that has a link to itself (i.e., `p == id[p] `). We call such a vertex a root. We use the roots as the identifiers of the components. Recall that initially, `id[p]` = p, so all vertices start off as roots. An implementation is given in Listing 6.4.1.

```
class Quick_Union:
    def __init__(self, n):
        self.id = list(range(n))

    def find(self, p):
        while p != id[p]:
            p = id[p]
        return p

    def union(self, p, q):
        i = self.find(p)
        j = self.find(q)
        if i == j:
            return
        id[i] = j
```

Listing 6.4.1: Quick-union.

Let's look at an example of FIND. In the following figure the top row are the index values for the array `id[.]` The bottom row are the values in `id[.]`

0	1	2	3	4	5	6	7	8	9
4	8	2	8	1	1	3	2	8	8

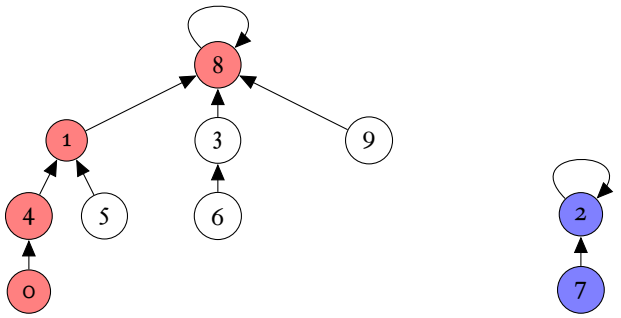
To evaluate FIND (0), we chase through the locations in the chain that begins at `id[0]`:

FIND (0) = `id[0]` = `id[4]` = `id[1]` = `id[8]` = 8.

To evaluate FIND (7), we chase through the locations in the chain that begins at `id[7]`:

$$\text{FIND } (7) = \texttt{id[7]} = \texttt{id[2]} = 2.$$

The information in `id[]` encodes a tree in which each node stores the number of its parent. This is unlike the typical tree, in which parents store the numbers of their children.



The UNION operation in quick-union acts as follows. In a call FIND (p,q) we first find the roots of the components containing p and q, and then attach the root of p to the the root of q. This has the effect of relabling all the notes rooted at p as now being rooted at q, hence the name quick-union. Note that the operation of UNION ensures that we eventually arrive at a root, so calls to FIND terminate.

Let's look at an example of UNION. We show the array `id[]` and the associated tree after each call to UNION.

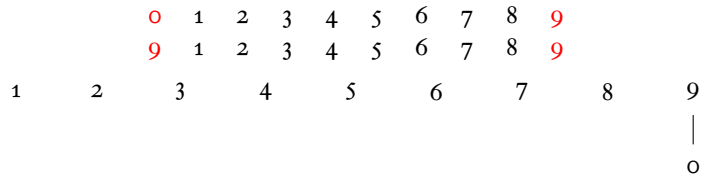


Figure 6.4.1: union(0,9)

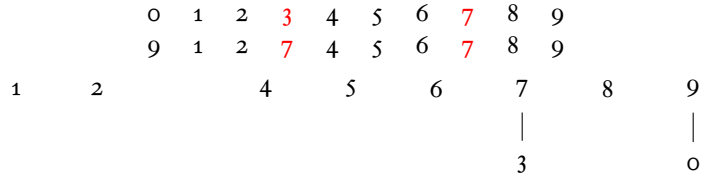


Figure 6.4.2: union(3,7)

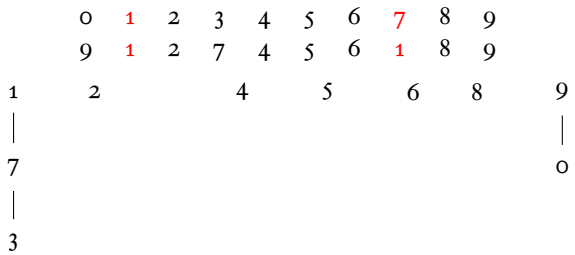


Figure 6.4.3: union(7,1)

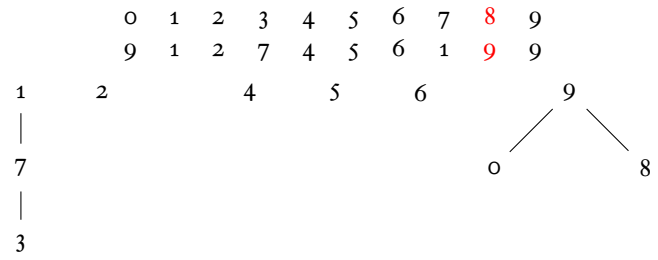


Figure 6.4.4: union(8,9)

The main computational cost of quick-union is the cost of FIND. The cost of a call of FIND depends on how many links we must follow to find a root, which, in turn, depends on UNION.

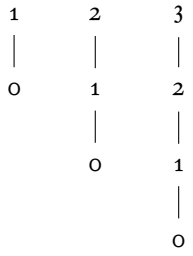
Proposition 6.4.1 (Proposition G in §1.5). *The number of accesses of* `id[]` *used by the call* FIND *(p) in quick-union is proportional to the depth of* `p` *in its tree. The number of accesses used by* UNION *and connected() is proportional to the cost of* FIND.

So, how tall can the trees be in the worst-case?

Suppose there is only a single component, and the connections are specified as follows:

0-1, 0-2, . . . , 0-(n-1)

Then the trees that result are



In the worst case, the height is proportional to n, so applying UNION to all n nodes is quadratic. We do not yet have the algorithm we seek.

6.5 Weighted quick-union

Weighted quick-union is more clever: in UNION, it connects the smaller tree to the larger to avoid growth in the height of the trees. We call this variant union-by-size.

Proposition 6.5.1 (Proposition H in §1.5). *Suppose we attach the smaller tree to the larger tree whenever we merge trees in a* UNION *operation. Then the depth of any node in a forest built by union-by-size for n vertices is at most* $\lg n$.

PROOF We will prove that the height of every tree of with n nodes in the forest is at most $\lg n$. The proof is by induction on n. If $n = 1$, such a tree has height $0 = \lg n$.

Now assume that the tree height of a tree of size i is at most $\lg i$ for all $i < n$. When we combine a tree of size i with a tree of size j, with $i \leq j$ and $i + j = n$, we increase the depth of each nodes in the smaller tree by 1 (this is true even if the two trees are the same size).

However, they are now in a tree of size $i + j = n$, and

$$1 + \lg i = \lg 2 + \lg i = \lg(2 * i) \leq \lg(i + j) = \lg n,$$

and the result follows. ∎

Since no tree in the forest is larger than $\lg n$ in height we have the following.

Corollary 6.5.2. *For union-by-size with n sites, the cost of* FIND *or* UNION *is $O(\lg n)$.*

Union-by-size requires us to store the number of elements in each tree. We can do so compactly by storing the size of a tree in its root and updating as needed. In order to indicate that we have arrived at a root, we will make the values in the roots the negatives of the size of the tree. Listing 6.5.1 gives an implementation.

```
class Union_by_Size:
    def __init__(self, n):
        # We indicate roots with negative numbers.
        self.id = [-1 for _ in range(n)]

    def find(self, p):
        while id[p] >= 0:  # Follow links to a root.
            p = id[p]
        return p

    def union(self, p, q):
        i = find(p)  # Get the components containing p and q.
        j = find(q)
        if (i == j):
            return
        # Since s[i], s[j] < 0, s[i] < s[j] means that i is the larger tree.
        if id[i] < id[j]:
            id[i] += id[j]  # Update the size of component i.
            id[j] = i       # Merge component j into component i.
        else:
            id[j] += id[i]  # Update the size of component j.
            id[i] = j       # Merge component i into component j.
```

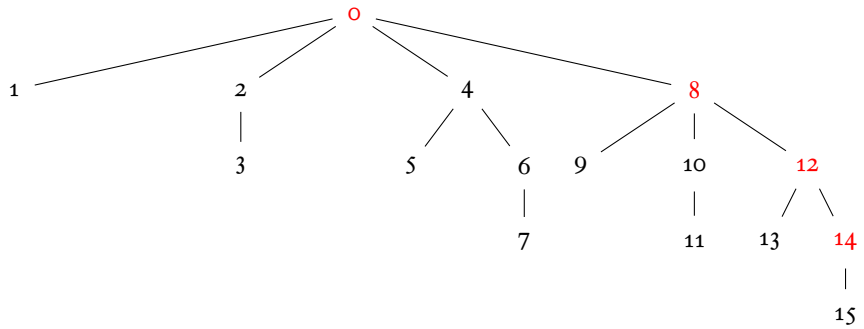
Listing 6.5.1: Union-by-size.

Union-by-height is another variant of weighted quick-union. In this version, we store and update the heights of the trees, attaching the shorter tree to the taller tree.

6.6 Path compression

Ideally, we would like every node in a tree to link to its root, so FIND would be $O(1)$ time. We can almost achieve this using path compression, in which we set the entries in `id[]` that we visit along the way to a root to point directly to the root.

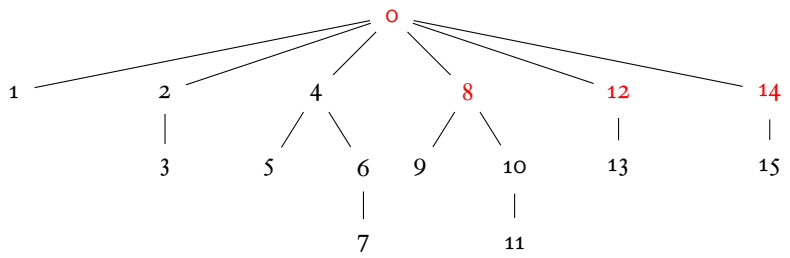
Consider the following tree, which is the worst-case for weighted quick-union when $n = 16$:



Here is the the array `id[]` at this point:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
id[i]	-16	0	0	2	0	4	4	6	0	8	8	10	8	12	12	14

Suppose we perform the call FIND (14). We can directly attach each node we visit on the way from 14 to the root directly to the root. This results in a shorter tree:



Here is the the array `id[ ]` at the call for FIND (14):

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
id[i]	-16	0	0	2	0	4	4	6	0	8	8	10	0	12	0	14

Subsequent calls to FIND will further flatten the tree. Since a call to UNION calls FIND, calls to UNION will typically become faster as we make more calls to UNION and FIND. As we will see, this makes union-find with path compression very efficient.

Adding path compression to FIND is simple—we modify the FIND method in Listing 6.5.1 as follows:

```
def find (self, p):
    if self.id[p] < 0:  # A negative value indicates a root.
        return p
    else:
        self.id[p] = self.find(self.id[p])
        return self.id[p]
```

Listing 6.6.1: FIND with path compression.

Here is the call stack in the call FIND (14), showing the recursion as we visit 14, 12, 8, and 0:

FIND (14): return FIND (`id[14]`) = FIND (12)
  FIND (12): return FIND (`id[12]`) = FIND (8)
    FIND (8): return FIND (`id[8]`) = FIND (0)
      FIND (0): return FIND (`id[0]`) = 0
    FIND (8): `id[8]` = 0
  FIND (12): `id[12]` = 0
FIND (14): `id[14]` = 0

Path compression does not change the number of elements in a tree, so no modification of union-by-size other than FIND is necessary to incorporate it.

On the other hand, path compression does change the height of the tree, so if we strictly adhered to union-by-height we would need to update the height. How do we do this? The answer is—we don't! The heights stored in the trees are left alone; in this context they are called ranks. The rank overestimates the height of the tree. In union-by-rank we use the ranks just as we would heights in union-by-height.

6.7 Complexity analysis

By itself, union-by-size yields trees with worst-case height $\lg n$ and path compression, by itself, yields trees with worst-case height $\lg n$. However, if used together, union by size + path compression does better: the worst-case complexity of a sequence of m calls to FIND (where $m \geq n$) is almost, but not quite, $\Theta(m)$. More exactly, it is $\Theta(m\alpha(n))$, where $\alpha(n)$ is an unbelievably slowly growing function of n.

The analysis of union-find is the most subtle analysis we will see in CSCI 303. The best bound known is due to Tarjan [38] and involves a rather complicated function usually called the inverse Ackermann function.[1] We will look at a slightly weaker upper bound, due to Hopcroft and Ulllman [20], whose proof is easier to understand.

The analysis of union-find is an example of **amoritized analysis**. Amortized analysis looks at the worst-case cost of a sequence of operations. In the case of union-find, this makes sense since we have seen that with path compression, the more FIND operations we perform, the faster FIND operations will be in the future. In amortized analysis the time required to perform a sequence of operations is averaged over all the operations performed. Amoritized analysis differs from expected case analysis in that the average is taken over time, not over the set of possible inputs.

The iterated logarithm, denoted by $\lg^*$, is the number of times the logarithm function must be applied before the result is ≤ 1:

$$\lg^* n = \min\{i \geq 0 \mid \overbrace{\lg\lg\lg\cdots\lg}^{i \text{ times}} n \leq 1\}.$$

Observe that $\lg^* n$ is very, very slowly growing:

$1 = 1$	$\lg^* 1 = 0$
$\lg 2 = 1$	$\lg^* 2 = 1$
$\lg\lg 2^2 = 1$	$\lg^* 4 = 2$
$\lg\lg\lg 2^4 = 1$	$\lg^* 16 = 3$
$\lg\lg\lg\lg 2^{16} = 1$	$\lg^* 65536 = 4$
$\lg\lg\lg\lg\lg 2^{65536} = 1$	$\lg^* 2^{65536} = 5.$

For any value of n that is meaningful in our universe, $\lg^* n \leq 5$, as 2^{65536} is greater than the estimated number of atoms in the cosmos.

We will analyze union-by-size with path compression, following the original analysis given in [20]. We will prove the following bound.

Theorem 6.7.1. *Given $n \geq 2$ objects, the time necessary to execute any sequence of $m > n$ UNION and FIND operations on these objects by union-by-size is $O(m \lg^* n)$.*

Given how slowly $\lg^* n$ grows, this result says that we would expect the runtime to grow linearly with m and be more or less constant as n grows.

In Figures 6.7.1–6.7.2 we plot the runtime of the code in Listing 6.5.1 with path compression. In Figure 6.7.1 we hold n constant at $n = 10,000$ and increase m. As m increases the runtime grows linearly with m. In Figure 6.7.2 we hold m constant at $m = 100,000$ and increase n. As n increases we see random variation in the runtime but no perceptible growth. This behavior comports with Theorem 6.7.1.

6.7.1 Proof of Theorem 6.7.1

In the discussion that follows,

$$\lg^i n = \overbrace{\lg\lg\lg\cdots\lg}^{i \text{ times}} n.$$

Consider a sequence of UNION and FIND operations.

Lemma 6.7.2 (Lemma 3, [20]). *If the rank of v is r, then at some time during the sequence of operations v was the root of a tree with at least 2^r vertices.*

PROOF The proof is by induction on the value of r. The case $r = 0$ is obvious; there is only one node in the tree.

Now assume the result is true for all values up to $r - 1$, where $r \geq 1$. If v is of rank r, then at some point v must become the parent of some vertex u of rank $r - 1$. This cannot occur during a FIND, for if it did, that would mean

[1] It is more properly called the inverse Ackermann-Péter-Robinson function [30, 33] as it is not Ackermann's original function [1].

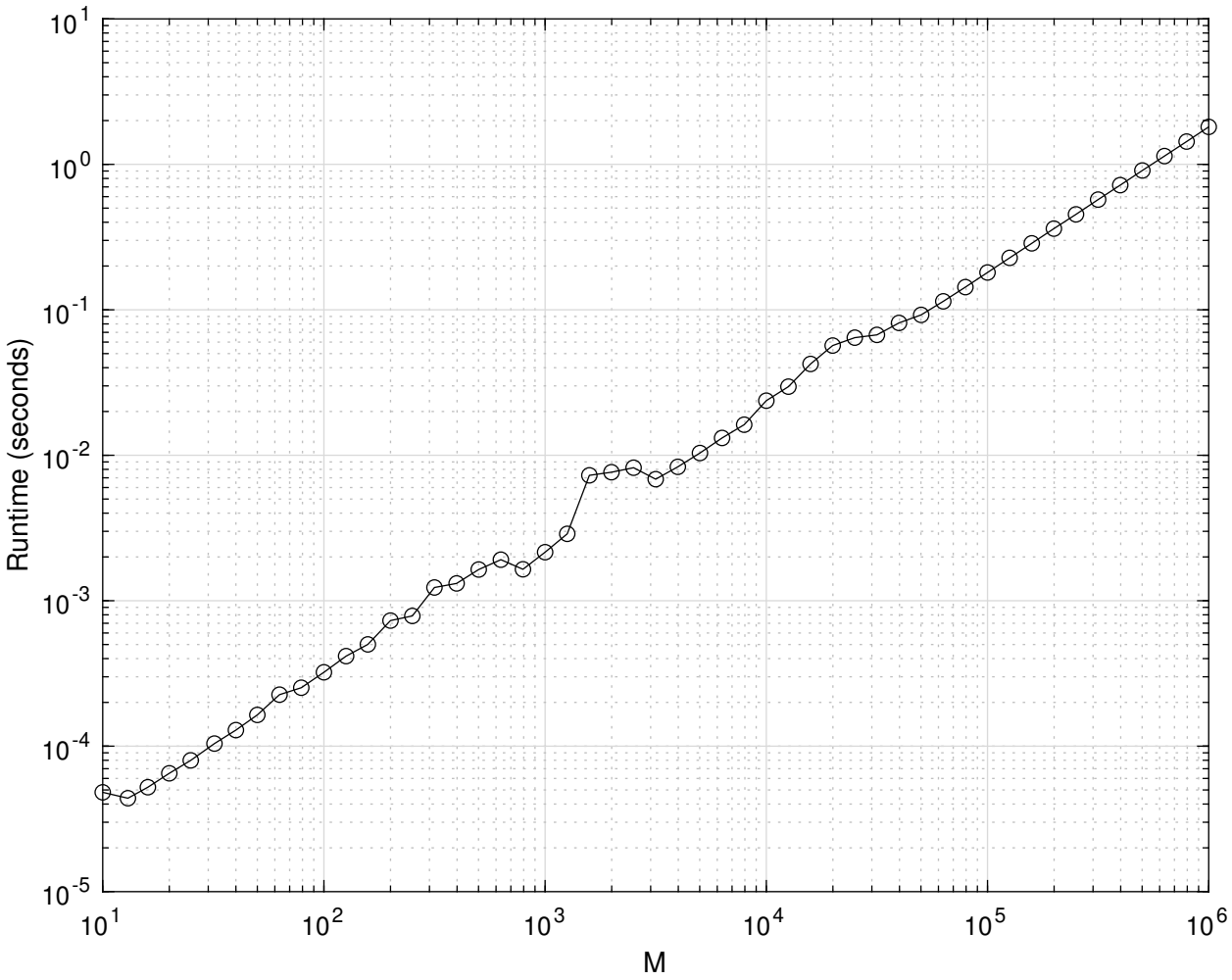


Figure 6.7.1: Runtime growth as a function of m.

that u would previously have been a descendant of v with some vertex w between them. But then w has rank at least r and v has rank at least $r + 1$ by the definition of rank, a contradiction.

Thus, v becomes the parent of u in a UNION. This means u must be the root of a tree T_u which is merged with a tree T_v whose root is v. By the inductive hypothesis, T_u has at least 2^{r-1} vertices, since a root cannot lose descendants and a non-root cannot gain descendants, and hereafter u will no longer be a root. Because we union by size, T_v has at least as many vertices as T_u. Therefore the merged tree has at least $2^{r-1} + 2^{r-1} = 2^r$ vertices and v is its root. ∎

Lemma 6.7.3 (Lemma 4, [20]). *Each time a vertex v gets a new parent w during the execution of union-by-size, that parent has a rank higher than any previous parent $u \neq w$ of v.*

PROOF If v, formerly a child of u, becomes a child of w, it must be during a FIND. Then w is an ancestor of u and hence of higher rank by definition. ∎

Lemma 6.7.4 (Lemma 5, [20]). *For each vertex v and rank r, there is at most one vertex u of rank r that is ever an ancestor of v.*

PROOF Suppose there were two such u, say u_1 and u_2. Assume, without loss of generality, that v first becomes a descendant of u_1. Then u_2 is of higher rank than u by Lemma 6.7.3 and the fact that at all times, the ranks of vertices on paths up trees are monotonically increasing. This contradicts the assumption that u_1 and u_2 were both of rank r. ∎

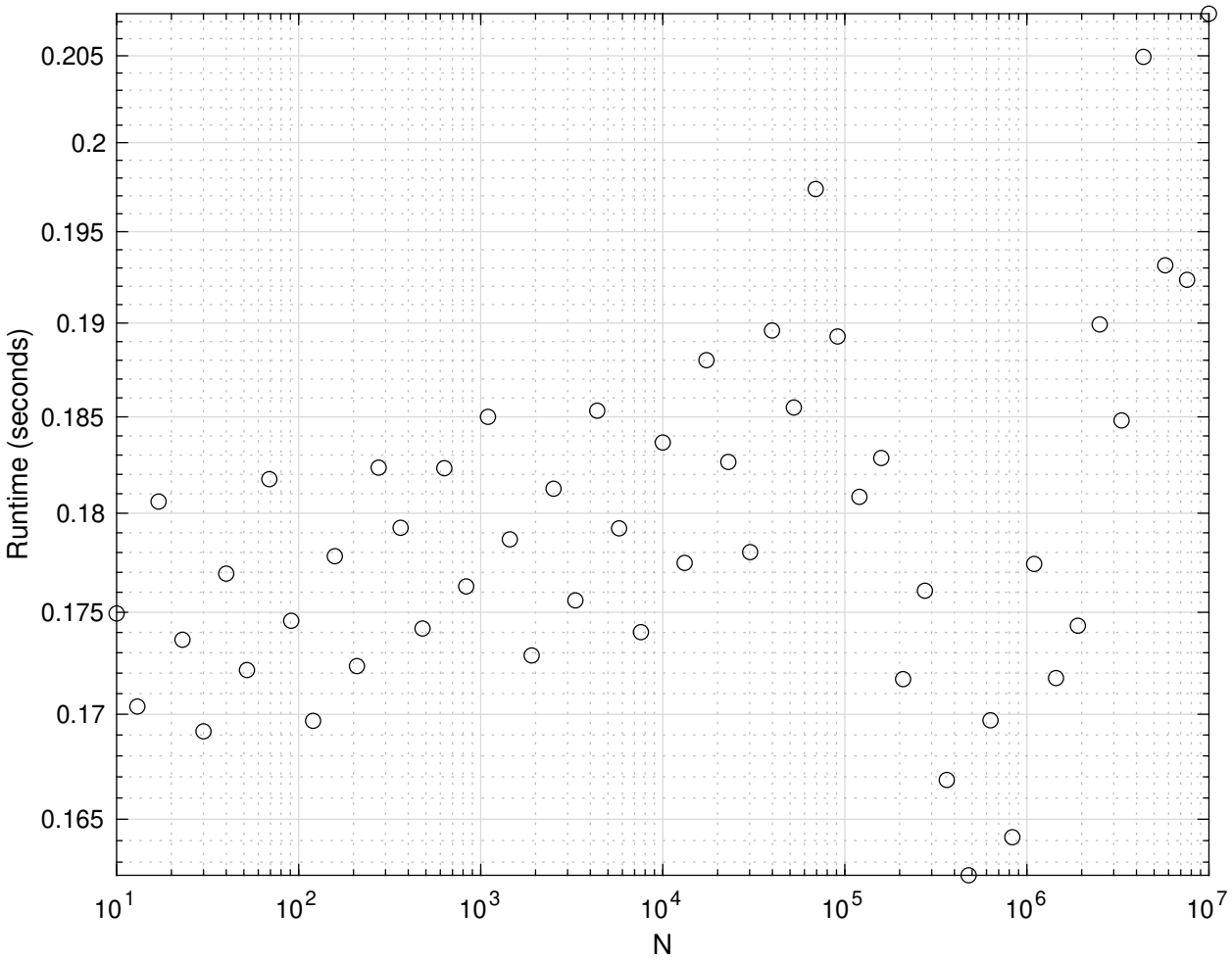


Figure 6.7.2: Runtime growth as a function of n.

Lemma 6.7.5 (Lemma 6, [20]). *There are at most $n/2^r$ vertices of rank r.*

Proof Lemma 6.7.2 says that every vertex of rank r is, at some point, the root of a tree with at least 2^r vertices. Meanwhile, Lemma 6.7.4 says that no vertex can be the descendant of two vertices of rank r. This implies that there are at most $n/2^r$ vertices of rank r. ∎

For $j \geq 1$, define $\mathcal{S}_j$, the j-th rank group, as follows:

$$\mathcal{S}_j = \left\{ v \mid \lg^{j+1} n < \text{rank } v \leq \lg^j n \right\}.$$

Note that the *higher* the value of j, the *lower* the ranks of the vertices it contains.

Lemma 6.7.6 (Lemma 7, [20]). $|\mathcal{S}_j| \leq 2n/\lg^j n$.

Proof Since there are at most $n/2^r$ vertices of rank r, we have

$$|\mathcal{S}_j| \leq \sum_{\lg^{j+1} n}^{\lg^j n} \frac{n}{2^i} \leq \frac{n}{\lg^j n}\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) \leq \frac{2n}{\lg^j n}.$$

∎

Lemma 6.7.7 (Lemma 8, [20]). *Each vertex is in some $\mathcal{S}_j$ for $1 \leq j \leq \lg^* n + 1$.*

PROOF Lemma 6.7.5 tells us that there is a most one vertex of rank $\lg n$ and that no vertex has rank greater than $\lg n$. Therefore we know that $j \geq 1$ since

$$\mathcal{S}_0 = \{v \mid \lg n < \text{rank } v \leq n\}$$

is empty.

Meanwhile, by definition we have

$$\lg^{\lg^* n} n \leq 1,$$

so

$$\lg \lg^{\lg^* n} n = \lg^{\lg^* n + 1} n \leq 0$$

and $\lg^{\lg^* n + 2} n$ does not exist. Since ranks are non-negative, this means that for all $j \geq \lg^* n + 2$ the set

$$\mathcal{S}_j = \left\{v \mid \lg^{j+1} n < \text{rank } v \leq \lg^j n\right\}$$

is empty, and we conclude that $j \leq \lg^* n + 1$. ∎

Now we can prove Theorem 6.7.1.

PROOF The cost of the UNION operations, not including the calls to FIND, is clearly $O(m)$. The cost of all the FIND operations is proportional to the sum, over all FIND operations, of the number of vertices visited and moved by FIND. We now show that this sum is $O(m \lg^* n)$.

Let v be a vertex in $\mathcal{S}_j$. If before a move of v the parent of v is in a lower rank group (a smaller value of j), assign the cost to the FIND operation. Otherwise, assign the cost to v itself. For a call FIND (i), consider the path from vertex i to the root of its tree. The ranks of vertices along the path to the root are monotonically increasing, and hence there can be on the path at most $\lg^* n$ vertices whose parents are in a lower rank group. Hence no FIND operation is assigned a cost more than $\lg^* n$.

By Lemma 6.7.6, there are at most $n/\lg^j n$ vertices in $\mathcal{S}_j$, and by Lemma 6.7.3, each vertex in a can be moved at most $\lg^j n$ times before its new parent is in $\mathcal{S}_{j-1}$ or a lower rank group. Thus, the total cost of moving vertices in $\mathcal{S}_j$, not counting moves of a vertex whose parent is in a lower rank group, is at most n. Since we know from Lemma 6.7.7 that there are at most $\lg^* n + 1$ rank groups, the total cost excluding that assigned to FIND is $O(n \lg^* n)$. Hence, the total cost of executing the sequence of UNION and FIND instructions is $O(m \lg^* n)$. ∎

6.8 Exercises

Exercise 6.8.1. *What type of induction is used in the proof of Proposition 6.5.1?*

Exercise 6.8.2. *Show that in union-by-height the height of any tree is $O(\lg n)$.*

Exercise 6.8.3. *Show that in union-by-rank with path compression the height of any tree is $O(\lg n)$.*

Exercise 6.8.4. *Consider union-find by size as in Listing 6.5.1 with path compression as in Listing 6.6.1. Recall that*

- *This implementation uses a negative value to indicate that you have arrived at the root of a tree.*
- *The absolute value of this number is the size of the tree.*

For the sequence of operations

union(1,2)
union(3,4)
union(3,5)
union(1,7)
union(3,6)
union(8,9)
union(1,8)

show `id[]` *after each* UNION *operation. Present your results in the following format:*

node	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
initial `id[]`	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*
union(1,2)	*-1*	*2*	*-2*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*

and so on.

Exercise 6.8.5. *Draw the collection of trees that results after each* UNION *operation in the preceding exercise.*

Exercise 6.8.6. *Consider the rapidly growing function F defined for integer $k \geq 0$ by the recurrence*

$$F(0) = 1$$
$$F(k) = 2^{F(k-1)}.$$

The first few values of F are

$$F(0) = 1$$
$$F(1) = 2^1 = 2$$
$$F(2) = 2^2 = 4$$
$$F(3) = 2^{2^2} = 2^4 = 16$$
$$F(4) = 2^{2^{2^2}} = 2^{16} = 65536$$
$$F(5) = 2^{2^{2^{2^2}}} = 2^{65536}.$$

Show that for all $k \geq 1$,

$$F(\lg^* k) = k.$$

6.9 Appendix: More union-find traces

Example 6.9.1. *union(0,1), union(3,4), union(2,3), union(1,8), union(1,6), union(0,9), union(1,3), union(3,8)*

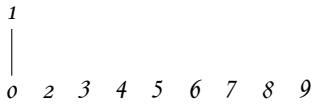
Initial values:

node	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
`id[]`	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*

0 1 2 3 4 5 6 7 8 9

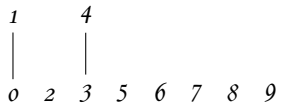
After union(0,1):

node	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
`id[]`	*1*	*-2*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*



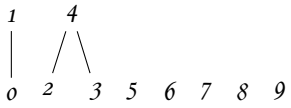
After union(3,4):

node	0	1	2	3	4	5	6	7	8	9
id[]	1	-2	-1	4	-2	-1	-1	-1	-1	-1



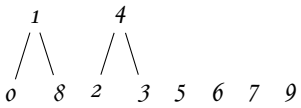
After union(2,3):

node	0	1	2	3	4	5	6	7	8	9
id[]	1	-2	4	4	-3	-1	-1	-1	-1	-1



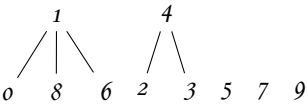
After union(1,8):

node	0	1	2	3	4	5	6	7	8	9
id[]	1	-3	4	4	-3	-1	-1	-1	1	-1



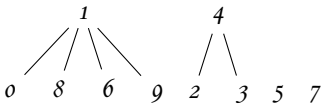
After union(1,6):

node	0	1	2	3	4	5	6	7	8	9
id[]	1	-4	4	4	-3	-1	1	-1	1	-1



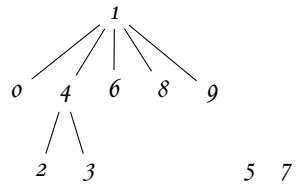
After union(0,9):

node	0	1	2	3	4	5	6	7	8	9
id[]	1	-5	4	4	-3	-1	1	-1	1	1



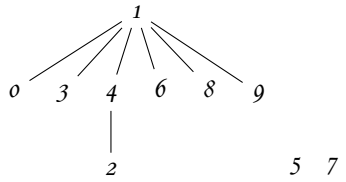
After union(1,3):

node	0	1	2	3	4	5	6	7	8	9
id[]	1	-8	4	4	1	-1	1	-1	1	1



After union(3,8):

node	0	1	2	3	4	5	6	7	8	9
id[]	1	-8	4	1	1	-1	1	-1	1	1



Example 6.9.2. *union(2,7), union(4,8), union(7,9), union(3,6), union(5,7), union(6,9), union(3,9), union(0,7), union(7,8), union(1,9)*

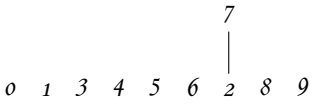
Initial values:

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

0 1 2 3 4 5 6 7 8 9

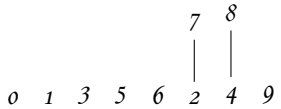
After union(2,7):

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	-1	7	-1	-1	-1	-1	-2	-1	-1



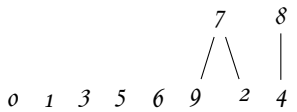
After union(4,8):

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	-1	7	-1	8	-1	-1	-2	-2	-1



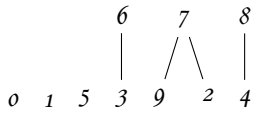
After union(7,9):

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	-1	7	-1	8	-1	-1	-3	-2	7



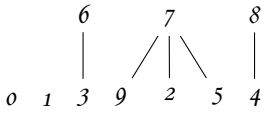
After union(3,6):

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	-1	7	6	8	-1	-2	-3	-2	7



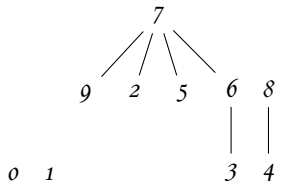
After union(5,7):

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	-1	7	6	8	7	-2	-4	-2	7



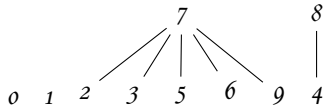
After union(6,9):

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	-1	7	6	8	7	7	-6	-2	7



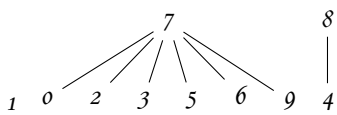
After union(3,9):

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	-1	7	7	8	7	7	-6	-2	7



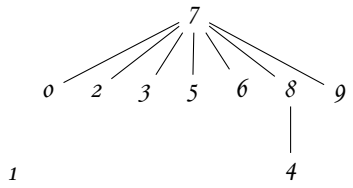
After union(0,7):

node	0	1	2	3	4	5	6	7	8	9
id[]	7	-1	7	7	8	7	7	-7	-2	7



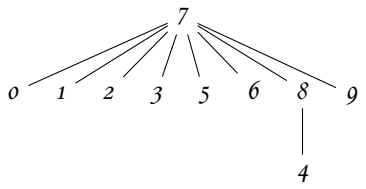
After union(7,8):

node	0	1	2	3	4	5	6	7	8	9
id[]	7	-1	7	7	8	7	7	-9	7	7



After union(1,9):

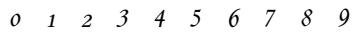
node	0	1	2	3	4	5	6	7	8	9
id[]	7	7	7	7	8	7	7	-10	7	7



Example 6.9.3. *union(5,7), union(2,4), union(0,2), union(5,8), union(1,7), union(5,8), union(0,9), union(2,6)*

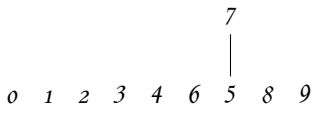
Initial values:

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1



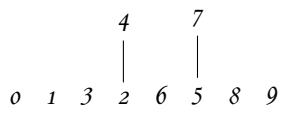
After union(5,7):

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	-1	-1	-1	-1	7	-1	-2	-1	-1



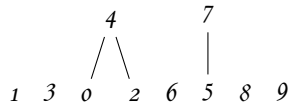
After union(2,4):

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	-1	4	-1	-2	7	-1	-2	-1	-1



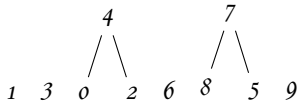
After union(0,2):

node	0	1	2	3	4	5	6	7	8	9
id[]	4	-1	4	-1	-3	7	-1	-2	-1	-1



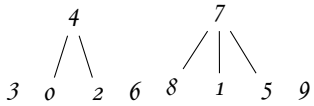
After union(5,8):

node	0	1	2	3	4	5	6	7	8	9
id[]	4	-1	4	-1	-3	7	-1	-3	7	-1



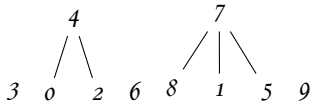
After union(1,7):

node	0	1	2	3	4	5	6	7	8	9
id[]	4	7	4	-1	-3	7	-1	-4	7	-1



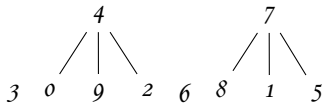
After union(5,8):

node	0	1	2	3	4	5	6	7	8	9
id[]	4	7	4	-1	-3	7	-1	-4	7	-1



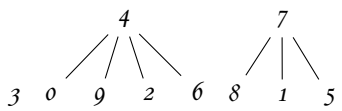
After union(0,9):

node	0	1	2	3	4	5	6	7	8	9
id[]	4	7	4	-1	-4	7	-1	-4	7	4



After union(2,6):

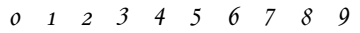
node	0	1	2	3	4	5	6	7	8	9
id[]	4	7	4	-1	-5	7	4	-4	7	4



Example 6.9.4. *union(1,9), union(4,7), union(0,9), union(2,7), union(5,9), union(0,4), union(3,7), union(6,8)*

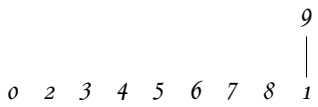
Initial values:

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1



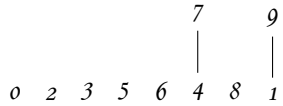
After union(1,9):

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	9	-1	-1	-1	-1	-1	-1	-1	-2



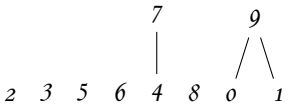
After union(4,7):

node	0	1	2	3	4	5	6	7	8	9
id[]	-1	9	-1	-1	7	-1	-1	-2	-1	-2



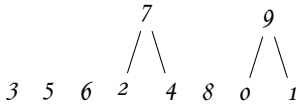
After union(0,9):

node	0	1	2	3	4	5	6	7	8	9
id[]	9	9	-1	-1	7	-1	-1	-2	-1	-3



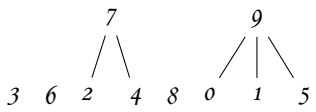
After union(2,7):

node	0	1	2	3	4	5	6	7	8	9
id[]	9	9	7	-1	7	-1	-1	-3	-1	-3



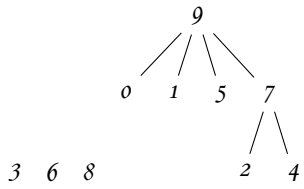
After union(5,9):

node	0	1	2	3	4	5	6	7	8	9
id[]	9	9	7	-1	7	9	-1	-3	-1	-4



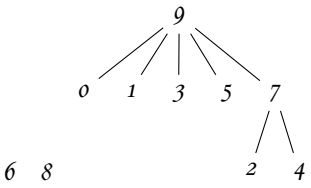
After union(0,4):

node	0	1	2	3	4	5	6	7	8	9
id[]	9	9	7	-1	7	9	-1	9	-1	-7



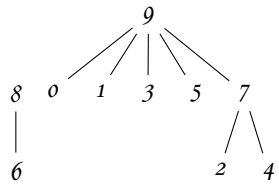
After union(3,7):

node	0	1	2	3	4	5	6	7	8	9
id[]	9	9	7	9	7	9	-1	9	-1	-8



After union(6,8):

node	0	1	2	3	4	5	6	7	8	9
id[]	9	9	7	9	7	9	8	9	-2	-8



Chapter 7

Greedy algorithms

Greedy algorithms build solutions bit by bit, choosing the next part of the solution by selecting a step that is view to have the the greatest immediate benefit. This is in contrast to strategies in games like chess, go, and bridge, which require thinking ahead in order to win.

Greedy algorithms are frequently optimization problems: find the best solution that satisfies specified requirements.

We have already seen some greedy algorithms, namely, Prim's and Kruskal's minimum weight spanning tree algorithms and Dijkstra's algorithm for shortest paths. In this section we will look at a few more:

- Optimal interval scheduling.
- Bin packing.

7.1 Interval scheduling

Suppose we are scheduling events in a single classroom. We have a set of requests $i = \{1, 2, 3, \ldots, n\}$ for different time slots. The time slot for request i begins at $s(i)$ and finishes at $f(i)$.

A subset of the requests is called compatible if no two requested time slots overlap in time. The problem is to find as large a compatible subset as possible. That is, try to maximize the number of different requests we can satisfy.

Here is a greedy approach:

- Use a simple selection rule to select a first request i_1.
- Drop all requests that are not compatible with i_1.
- Use a simple selection rule to select the second request i_2 from those that remain.
- Drop all requests that are not compatible with i_2.
- Repeat the previous two steps until there are no requests left.

What should be our selection rule?

7.1.1 Selection rule: earliest start

What if we choose a request that begins earliest? Imagine three short jobs versus one long job that begin earlier:

Oops! a long job that starts early can squeeze out multiple compatible jobs.

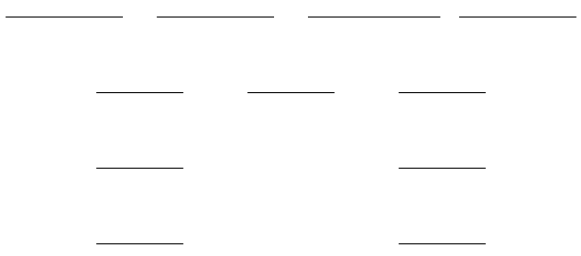
7.1.2 Selection rule: shortest first

What if we choose a request that is of shortest duration?

Oops! a short job in the middle can preempt multiple compatible jobs.

7.1.3 Selection rule: fewest conflicts

What if we choose a request with the fewest number of incompatible requests?



Oops! this rule chooses the middle request in the second row as part of the solution, but the unique optimal solution is the first row.

7.1.4 Selection rule: earliest finish

What if we select the request that finishes first? This turns out to work, as it frees the resource as soon as possible.

```
from math import inf
def earliest_finish(R):
    # R is a list of tuples representing the intervals.
    # Sort the requests by increasing finish time.
    R.sort(key=lambda interval: interval[1])
    A = set()
    latest_end_time = -inf
    for interval in R:
        if interval[0] >= latest_end_time:
            A.add(interval)
            latest_end_time = interval[1]
    return A
```

Listing 7.1.1: Earliest finish greedy algorithm

Proposition 7.1.1. *The earliest finish greedy algorithm returns an optimal solution.*

Proof Suppose the greedy solution is not optimal.

Let

$$\{i_1, i_2, \ldots, i_k\}$$

denote the jobs selected by the algorithm, in order of selection and

$$\{j_1, j_2, \ldots, j_m\}$$

denote the set of jobs in the optimal solution, also in order of selection.

Because the greedy algorithm starts by choosing the interval that finishes earliest we know that

$$f(i_1) \leq f(j_1)$$

Claim 7.1.2. *For all* $1 \leq r \leq k$ *we have* $f(i_r) \leq f(j_r)$.

Proof of the claim. The proof of the claim is by induction on r. We know that $f(i_1) \leq f(j_1)$, so we have established the inductive basis. Now assume the statement is true for $r-1$: $f(i_{r-1}) \leq f(j_{r-1})$. We wish to show that $f(i_r) \leq f(j_r)$.

Because the intervals in the optimal solution are compatible, we know that $f(j_{r-1}) \leq s(j_r)$. From the inductive hypothesis we have $f(i_{r-1}) \leq f(j_{r-1})$. It follows that $f(j_{r-1}) \leq s(j_r)$, which means that when the greedy algorithm chose i_r, j_r was in the set of available intervals. This means $f(i_r) \leq f(j_r)$, since otherwise the greedy algorithm would not have selected interval i_r. ∎

We next show that $m \leq k$. Suppose $m > k$. We know that $f(i_k) \leq f(j_k)$. Since $m > k$, there is an interval j_{k+1} in the optimal solution, and $f(i_k) \leq f(j_k) \leq s(j_{k+1})$. But if this were true, the greedy algorithm would have continued and chosen j_{k+1}, a contradiction.

Thus $m \leq k$, which means the greedy solution has at least as many intervals as the optimal solution. Since the optimal solution has the maximum number of intervals, the greedy solution must also be optimal. ∎

7.2 Huffman encoding

Huffman encoding [21] is a lossless compression technique that uses a greedy approach to attain optimality with regards to brevity.

Consider the sequence AAAABBCD. We can encode this using 16 bits if we encode each character using two bits:

$$\begin{array}{llll} A = 00, & & C = 10 \\ B = 01, & & D = 11, \end{array}$$

so AAAABBCD = (00)(00)(00)(00)(01)(01)(10)(11).

Huffman encoding allows us to obtain a 14 bit encoding by encoding the least common symbols with the longest codes:

$$\begin{array}{llll} A = 0, & & C = 110 \\ B = 10, & & D = 111. \end{array}$$

Then

$$\begin{aligned} \text{AAAABBCD} &= (0)(0)(0)(0)(10)(10)(110)(111) \\ &= 00001010110111. \end{aligned}$$

However, how do we delimit the codes so we can distinguish, say, the code for B from the end of the code for C? If we do this using a delimiting character, our compression will suffer.

Huffman compression solves this using an encoding that ensures that no character code is the prefix (i.e., first characters) of another. This makes the codes unambiguously decodable.

7.2.1 Tries

Huffman encoding uses a data structure called a trie. A trie is yet another type of search tree (the name "trie" comes from "retrieval").

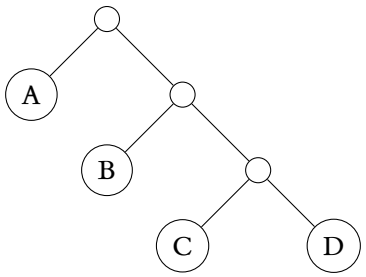
- A trie consists of nodes that contain links that are either null or links to other nodes.
- Each node other than the root is pointed to by exactly one other node, its parent.
- We assume we have an alphabet of R characters, and each node has R links.
- Each node also has an associated value, which is either null or is a value we wish to store.
- The value associated with a key is stored in the node corresponding to its last character.

To decoding a stream encoded by Huffman encoding, for each "word" in the stream we start at the root of the trie.

1. A 0 in the compressed bitstream means follow the left link.

2. A 1 in the compressed bitstream means follow the right link.

3. When we arrive at a leaf we have the decoded character and return to the root and the next bit in the compressed bitstream.

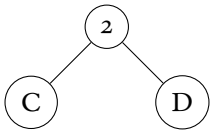


7.2.2 Building the encoding trie

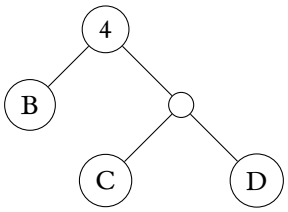
We begin the construction of a Huffman encoder by first counting the frequencies of each character.

	1	1	2	4
AAAABBCD	C	D	B	A

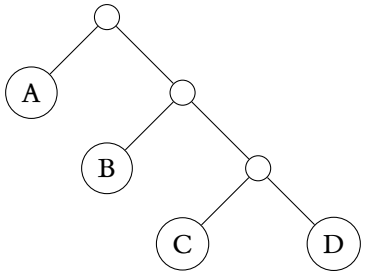
Build a tree from two nodes with the smallest frequency count. Place the combined frequency count in the root of the tree. This is the greedy step in the algorithm: we assign the least frequent characters the longest codes.



Add to the tree using two nodes with the smallest frequency count. Place the combined frequency count in the root of the tree.



and again…



7.2.3 Optimality of Huffman encoding

The weighted external path length of a tree is the sum over all the leaves of their frequency count times their depth. For our example trie we have

symbol	depth	frequency
A	1	4
B	2	2
C	3	1
D	3	1

The weighted external path length is

$$1 \times 4 + 2 \times 2 + 3 \times 1 + 3 \times 1 = 14.$$

Note that this is the number of bits we needed to represent AAAABBCD. In general, the depth of a leaf is the number of bits needed to encode the character in the leaf. This means that the weighted external path length is the number of bits in the compressed bitstream.

Proposition 7.2.1. *For any prefix-free code, the length of the encoded bitstream is equal to the weighted external path length of the corresponding trie.*

Proposition 7.2.2. *Give a set of r symbols and frequencies, Huffman encoding builds a trie of minimal weighted external path length.*

This means Huffman encoding uses the minimal number of bits among prefix-free codes.

Proof We use induction on r.

Suppose that Huffman encoding is optimal for any set of fewer than r symbols. Let T_H be the Huffman code for the set of symbols and associated frequencies $(s_1, f_1), \ldots, (s_r, f_r)$. Let the associated weighted external path be $W(T_H)$.

Suppose that (s_i, f_i) and (s_j, f_j) are the first two symbols chosen when generating the Huffman code. The algorithm then computes the code $T_H^\star$ for the set of $r-1$ symbols with (s_i, f_i) and (s_j, f_j) replaced by a new symbol $(s^\star, f_i + f_j)$ in a leaf. Let d be the depth of this leaf.

Note that

$$\begin{aligned} W(T_H) &= W(T_H^\star) - d(f_i + f_j) + (d+1)(f_i + f_j) \\ &= W(T_H^\star) + (f_i + f_j). \end{aligned}$$

Now consider an optimal trie T for $(s_1, f_1), \ldots, (s_r, f_r)$ of height h. We know that (s_i, f_i) and (s_j, f_j) must be at depth h since otherwise we could make a trie with smaller lower external path length by swapping them with notes at depth h.

We may also assume that (s_i, f_i) and (s_j, f_j) are siblings by rearranging nodes at depth h. Consider the tree $T^\star$ obtained by replacing the parent of (s_i, f_i) and (s_j, f_j) with a new symbol $(s^\star, f_i + f_j)$. By the argument above,

$$W(T) = W(T^\star) + (f_i + f_j).$$

By the inductive hypothesis, $T_H^\star$ is optimal, so $W(T_H^\star) \leq W(T^\star)$. Thus,

$$W(T_H) = W(T_H^\star) + (f_i + f_j) \leq W(T^\star + (f_i + f_j) = W(T).$$

Since T is optimal, T_H must also be optimal. ∎

7.2.4 Shannon entropy

Again consider the sequence AAAABBCD. Let p_i be the fraction of the time symbol i appears, and define the Shannon entropy to be

$$-\sum_i p_i \log_2 p_i.$$

The entropy is a measure of the average number of bits needed to encode each symbol. For our sequence, the entropy is

$$-\left[\frac{1}{2}\log\frac{1}{2}+\frac{1}{4}\log\frac{1}{4}+\frac{1}{8}\log\frac{1}{8}+\frac{1}{8}\log\frac{1}{8}\right]=\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{3}{8}=\frac{7}{4}.$$

The entropy predicts that we can encode our sequence using $8\times\frac{7}{4}=14$ bits. This is achieved by Huffman encoding.

7.3 Exercises

Exercise 7.3.1. *Apply the greedy algorithm of Section 7.1.4 to solve the interval scheduling problem if the intervals are*

$$(0,1),(3,5),(2,3),(1,2),(3,4),(0,2),(2,4),(1,5)$$

Exercise 7.3.2. *We are given items of value $v_1, v_2, \ldots, v_n > 0$ and corresponding weight $w_1, w_2, \ldots, w_n > 0$. We wish to maximize the value of the items we select subject to a limit $W > 0$ on the total weight.*

Consider the following greedy strategy. Choose the highest value item that will fit. Then choose the next most valuable item that will fit, and so on. Give a counterexample that shows this approach is not guaranteed to always find an optimal solution.

Exercise 7.3.3. *Suppose we have coins of integer value $1 \le v_1 < v_2 < \cdots < v_n$. We define the coin-changing problem to be: Given an integer $C \ge 1$, find the smallest number of coins that add up to C (assuming an unlimited number of coins of each type are available).*

We can define a greedy algorithm to make change by starting with the larger coins first, using as many coins of each type as possible before moving on to the next lower denomination.

1. *Show that this algorithm does not necessarily find a solution with the minimum number of coins.*
2. *Let $c \ge 2$ be an integer, and suppose the coins have value $1, c, c^2, \ldots, c^{n-1}$. Show that greedy algorithm always finds a minimal solution to the coin-changing problem.*

Chapter 8

Greedy algorithms and matroids

In a greedy maximization problem we hope that the greedy choices at each step lead to a global maximizer. If a global maximizer is always reached, we call the algorithm correct.

When can we be assured that a greedy algorithm is correct? It turns out we have such an assurance if our problem is posed on a matroid.

8.1 Matroids

We begin by defining independence systems and matroids.

Definition 8.1.1. *Let S be a finite set. An independence system over S is a family F of subsets of S with the following properties.*

1. *F is nonempty.*
2. *If $A \in F$, then so is every subset of A (the hereditary property).*

The sets in F are called the independent sets.

Definition 8.1.2. *Suppose F is an independence system on S. We say (S, F) is a matroid if the following is satisfied.*

1. *If $A, B \in F$ and $|A| < |B|$, then there exists $x \in B - A$ such that $A \cup \{x\} \in F$ (the exchange property).*

8.1.1 Example: linearly independent vectors

Let M be a nonzero matrix, S be the set of rows (or columns) of M, and

$$F = \{A \mid A \subseteq M \text{ and } A \text{ is linearly independent}\}.$$

Claim: (S, F) is a matroid.

This connection is why the terminology connected with matroids comes from the terminology of matrices.

To prove the claim,

1. Note that F is not empty since it contains every individual nonzero row of M.
2. If B is a set of linearly independent rows of M, then any subset A of B is linearly independent, so F is hereditary.
3. If A, B are sets of linearly independent rows of M, and $|A| < |B|$, then

   $$\dim \operatorname{span} A < \dim \operatorname{span} B.$$

   Choose a row x in B that is not contained in $\operatorname{span} A$. Then $A \cup \{x\}$ is a linearly independent subset of rows of M, so F has the exchange property.

8.1.2 Maximal independent sets

A set $A \in F$ is called maximal independent set if A is not a proper subset of any other independent set. The exchange property says if we have an independent set that is not maximal independent, we can extend the set by adding some suitably chosen element from a larger set in F.

Proposition 8.1.3. *All maximal independent sets are of the same cardinality m. Conversely, if m is the cardinality of a maximal independent set and $A \in F$ is of cardinality m, then A is maximal.*

For this reason the maximal independent sets of a matroid are called bases, and the size of any basis is the rank of the matroid.

Proof Suppose you have you have two maximal sets A, B with $|A| > |B|$. Take any subset of C of A whose size is $|B| + 1$. By the exchange property we can find $x \in C$ such that $A \cup \{x\} \in F$. However, $A \subsetneq A \cup \{x\}$, contradicting the maximality of A.

For the converse, let B be a maximal independent set and suppose $|A| = |B|$. If A is not maximal, then there exists a maximal C for which $A \subsetneq C$. This means $|A| < |C| = |B|$, a contradiction. ∎

8.2 A greedy algorithm for weighted matroids

A weight function on a set S is a map $w : S \to (0, +\infty)$. If A is a subset of S, then we define

$$w(A) = \sum_{a \in A} w(a).$$

Suppose that F is an independence system on the weighted set S and consider the following greedy algorithm.

Greedy algorithm for maximization:

1. Start with $A = \emptyset$.
2. Sort S into decreasing order by weight w.
3. For $s \in S$, if $A \cup \{s\} \in F$ then add x to A: $A = A \cup \{s\}$.

The complexity depends on how much work it is the chack that $A \cup \{s\} \in F$ in step 3. Let $n = |S|$. Sorting $w(a)$ for all $a \in S$ can be done in $\Theta(n \lg n)$ time.

The for-loop iterates n times. In the body of the loop one needs to check whether $A \cup \{X\} \in F$. If each check takes $f(n)$ time, then the loop takes $\Theta(nf(n))$ time. Thus, Greedy is $\Theta(n \lg n + nf(n))$.

What about correctness? We have the following result.

Theorem 8.2.1. *Greedy produces a maximally independent set of S of maximal weight for every weight function on S if and only if (S, F) is a matroid.*

Proof First we prove sufficiency. Clearly the algorithm produces a maximal independent set since it keeps adding elements until it can no longer maintain independence.

Suppose the algorithm produces $A = \{a_1, \ldots, a_m\}$, in that order. Let $B = \{b_1, \ldots, b_m\}$ be any other maximal independent set. We claim that $w(a_k) \geq w(b_k)$ for all k. It follows that $w(A) \geq w(B)$.

To prove the claim, suppose to the contrary that it were false. Then for some k we have $w(a_k) < w(b_k)$. Choose the smallest such k. We know that $k > 1$ because the greedy algorithm is guaranteed to have $w(a_1) \geq w(b_1)$.

Consider the sets

$$A' = \{a_1, \ldots, a_{k-1}\}$$
$$B' = \{b_1, \ldots, b_{k-1}, b_k\}.$$

Since $|B'| = |A'| + 1$, the exchange property tells us that there is some $j \in \{1, 2, \ldots, k\}$ such that $b_j \notin A'$ and $A' \cup \{b_j\}$ is an independent set.

This would mean $w(b_j) \geq w(b_k) > w(a_k)$, so $b_j \neq a_k$ and the greedy algorithm would have chosen b_j before choosing a_k. This contradicts the greedy nature of the algorithm.

Next we turn to necessity. Suppose (S, F) is not a matroid. This means the exchange property does not hold, so there exist $A, B \in F$ with $|B| > |A|$ and $A \cup \{x\} \notin F$ for all $x \in B - A$.

Define a weight on S as follows:

$$w(s) = \begin{cases} 0.1 & \text{if } s \in S - (B \cup A), \\ 1 + \frac{1}{|B|} & \text{if } s \in A, \\ 1 & \text{if } s \in B - A. \end{cases}$$

Then the greedy algorithm will choose A. However, choosing B is strictly better, so A cannot be optimal::

$$w(A) = |A|\left(1 - \frac{1}{|B|}\right) = |A| - \frac{|A|}{|B|} < |A| + 1 \leq |B| \leq w(B).$$

∎

8.3 A scheduling problem

We are given a set S of n tasks $a_1, \ldots, a_n$ to do sequentially.

- Each task takes one hour.
- The tasks have deadlines $d_1, \ldots, d_n$ and non-negative penalties $w_1, \ldots, w_n$ for missing deadlines.

Our goal is to find a schedule S where the total penalty of the tasks that miss their deadline is minimized.

Key observation: this is the same as asking that the total penalty of the tasks that meet their deadline is maximized.

8.3.1 Matroid structure

Call a set A of tasks feasible if there is a schedule in which every task in A is on time. Let $N_t(A)$ be the number of tasks in A whose deadline is t or earlier. Note that $N_0(A) = 0$ and $N_n(A) = |A|$.

Proposition 8.3.1. *Let A be any subset of tasks. Then A is feasible if and only if $N_t(A) \leq t$ for every integer t.*

Proof Consider a schedule in which every task in A is on time. Let i_t be the t-th task in A to be completed. Since the task was completed on time, we must have $t \leq d_i$, which implies $N_t(A) \leq t$.

Conversely, suppose $N_t(A) \leq t$ for every integer t. If we perform the tasks in A in increasing order of deadline, then we complete all tasks in A with deadlines t or earlier by time t. In particular, for any $i \in A$ we perform task i on or before its deadline d_i, so A is feasible. ∎

The following assures us that a greedy algorithm will always solve this scheduling problem.

Proposition 8.3.2. *The collection of feasible sets forms a matroid.*

Proof The empty set is trivially feasible, and any subset of a feasible set is clearly feasible. It remains to show that the exchange property holds.

Let A and B be feasible sets of jobs with $|B| > |A|$. Let k be the largest integer such that $N_k(B) \leq N_k(A)$. Such a k must exist because $N_0(B) = N_0(A) = 0$ and $N_n(B) = |B| > |A| = N_n(A)$.

By the definition of k there are more tasks with deadline $k + 1$ in B than in A. Thus, we can choose a task $j \in B - A$ with deadline $k + 1$. Let $C = A \cup \{j\}$.

Let ℓ be an arbitrary integer. If $\ell \leq k$ then $N_\ell(C) = N_\ell(A) \leq \ell$, because A is feasible.

On the other hand, if $\ell > k$, then $N_t(C) = |A| + 1 \leq |B| \leq t$ by the definition of k and because B is feasible. Our earlier claim implies C is feasible. ∎

8.4 Exercises

Exercise 8.4.1. *Design an algorithm that, given a weighted matroid, computes a maximal independent set of minimum weight.*

Chapter 9

Balanced trees: 2-3-4 and red-black trees

A balanced tree is one in which all leaves are roughly the same distance from the root. A perfectly balanced tree is one in which all leaves are the same distance from the root. A balanced binary tree containing n nodes has height roughly $\lg n$, guaranteeing $\lg n$ complexity for searches in the worst-case. The challenge is building a balanced tree and maintaining its balance in the face of insertions and deletions.

In CSCI 241 you saw AVL trees, which are a type of balanced tree. In this chapter we look at 2-3-4 trees and red-black trees, two related types of balanced trees. They have the property that we can insert new items while maintaining perfect balance. Moreover, 2-3-4 trees do not need rotations as in AVL trees to maintain balance, which makes them simpler to understand.

9.1 2-3-4 trees

A 2-3-4 tree consists of three types of nodes:

- 2-nodes: one key, two children;
- 3-nodes: two keys, three children;
- 4-nodes: three keys, four children.

These nodes are depicted in Figure 9.1.1.

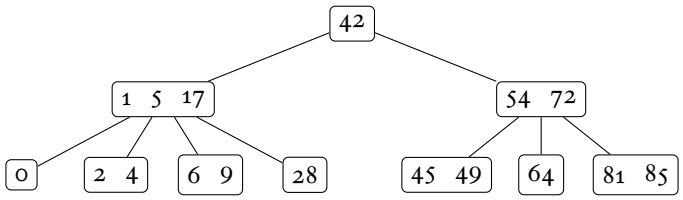


Figure 9.1.1: A 2-3-4 tree.

By construction the height of an n-node 2-3-4 tree is between $\log_4 n = \frac{1}{2}\lg n$ and $\lg n$. Thus, searching and insertion in a 2-3-4 tree are both $O(\lg n)$ operations.

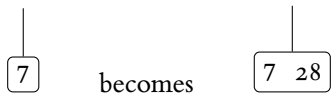
In practice 2-3-4 trees are not much used for reasons of efficiency:

- If we use different types of nodes with different numbers of keys, then we will incur the overhead of transforming them into other types in the course of insertions and deletions.
- If we make all of our nodes large enough to hold three keys, then this space will be wasted in 2-nodes and 3-nodes.

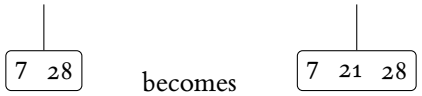
However, 2-3-4 trees are closely related to red-black trees, which are frequently encountered, and understanding 2-3-4 trees makes red-black trees less mysterious.

9.1.1 Insertion into a 2–3–4 tree

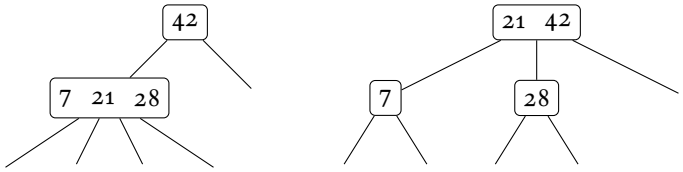
We insert the new key into the lowest existing node reached in the search. If it is a a 2–node it becomes a 3–node:



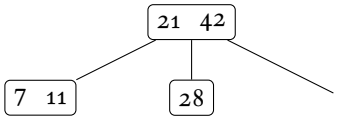
A 3–node becomes a 4–node:



What about a 4–node? In top–down insertion, as we move down the tree, we split all 4–nodes we encounter in the following manner: move the middle key up to the parent and split the remaining keys into two 2–nodes. This guarantees that the parent of any 4–node we encounter is a 2–node or 3–node, so it will always have room to accept the middle element of the 4–node if we need to split it.

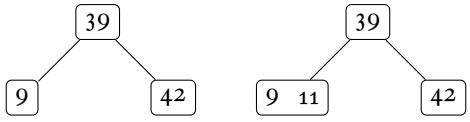


Insertion, if done here, now reduces to the case of a 2–node or 3–node.



Let's building a 2–3–4 tree with the following insertion order: 42, 9, 39, 11, 27, 13, 33, 16, 28. The first three insertions are straightforward.

Now we encounter a 4–node, which we first split, and then insert 11:

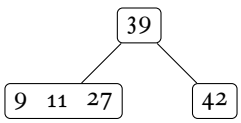


In this case we first promoted 39 into a new root node. This illustrates an important point: 2–3–4 trees only grow at the root.

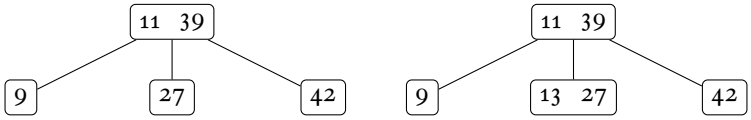
Proposition 9.1.1. *Perfect balance is maintained in 2-3-4 trees.*

PROOF This is an immediate consequence of the fact that 2–3–4 trees only grow at the root. ∎

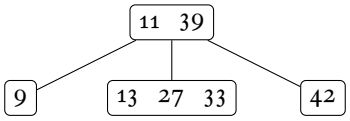
Now insert 27.



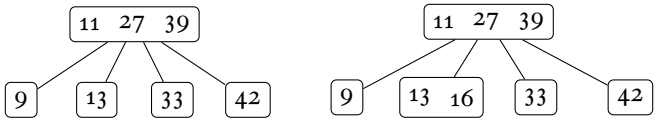
When inserting 13, we first split a 4-node and then insert 13.



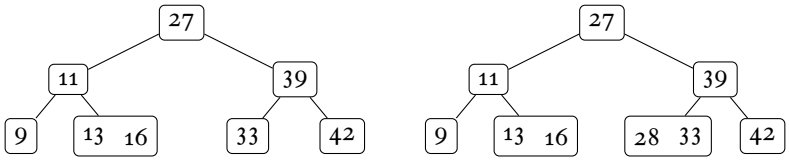
The insertion of 33 is straightforward ...



...but the insertion of 16 causes a split ...



On the way to inserting 28 we split the 4-node at the root.



Once again, growth at the root maintains perfect balance.

Rather than splitting 4-nodes on the way down, we can also perform bottom-up insertion, starting at the insertion node and moving upwards.

9.1.2 Deletion

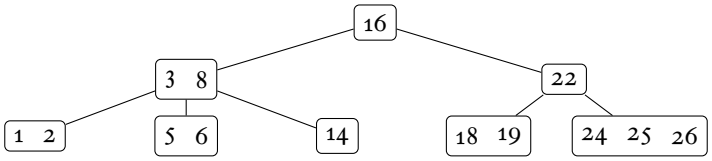
Deletion involves fusing nodes and is also $O(\lg n)$). All true deletions are done at leaves, as with binary search trees. We may have to push elements down to a leaf before deleting them.

Let k be the key we wish to delete. There are a number of cases to consider.

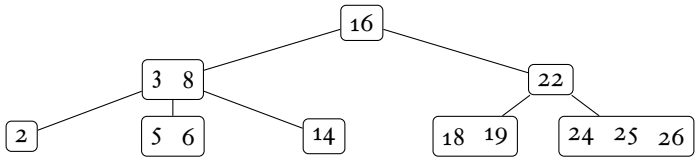
0. If k is in a leaf containing two or more keys, remove k from the leaf.
1. If k is in an internal node, we proceed as follows.
   - 1.0 If the left child of k has at least 2 keys, replace k with its predecessor p and recursively delete p.
   - 1.1 If the right child of k has at least 2 keys, replace the element with its successor s and recursively delete s.
   - 1.2 If both children have only 1 key, merge the right child into the left child and include k in the left child. Delete the empty right child and recursively delete k from the left child.
2. If k is in a leaf with only one key, follow the links to k. We ensure that all nodes we traverse contain at least 2 keys as follows.
   - 2.0 If the node we are about to enter has only one key and has an immediate sibling with at least two keys, move an element down from the parent into the node and move an element from the sibling into the parent.
   - 2.1 If both the node we are about to enter and its immediate siblings have only one key each, merge the node with one of the siblings and move an element down from the parent into the merged node. This element will be the middle element in the node. Free the empty node whose elements were merged into the other node.

This ensures that we always descend into a node with two or more keys, and eventually arrive at case 0 or 1.

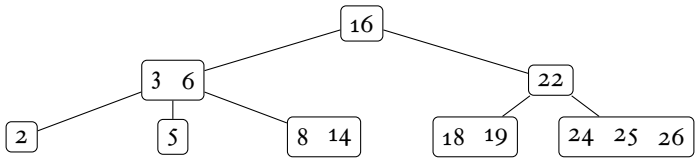
Here is an example of deletion. The initial tree is



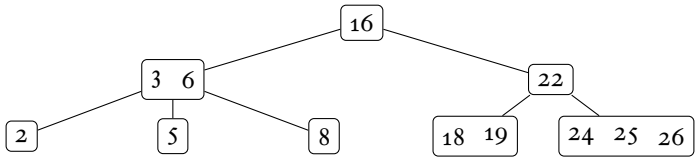
Let's delete 1. This is Case 0: 1 is in a leaf containing two or three keys, so we remove 1 from the leaf.



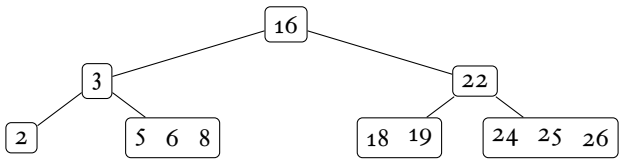
Next, delete 14. This is Case 2: 14 is the only key in a leaf. In particular, we have Case 2.1: The node we are about to enter has only one key and its immediate sibling [5 6] has at least two keys, so we move an element from the parent into the node and move an element from the sibling into the parent.



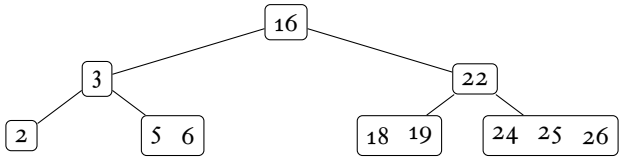
Now there are two keys in the leaf containing 14, so we are back to Case 0: delete 14 from the leaf.



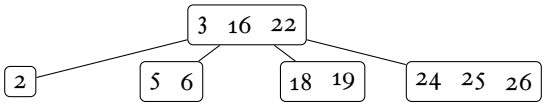
Deleting 8 is Case 2.1: 8 is in a leaf, and its siblings have only one key each. We merge the node with one of the siblings and move an element down from the parent into the merged node. This element will be the middle element in the node. Free the empty node whose elements were merged into the other node.



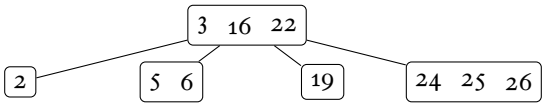
Now we're back to Case 0, and simply delete 8.



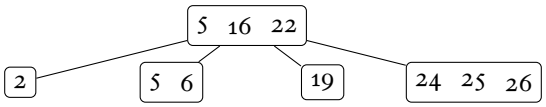
Deleting 18 is Case 2. We will need to merge the root with its children on the way down.



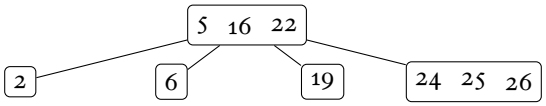
Now we're back to Case 0: simply delete 18.



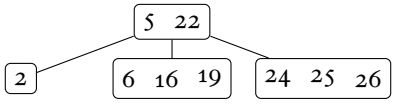
Next delete 3. This is Case 1.1: 3 is in an internal node, and 3's right child [5 6] has at least 2 keys, so we replace the 3 with its successor 5 and recursively delete 5.



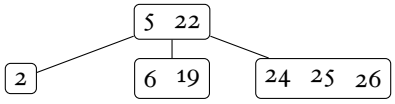
Since 5 is in a leaf with two keys, we're back to Case 0.



Finally, delete 16. This is Case 1.2: 16 is in an internal node, and 16 left and right children have only one key each, so we merge the right child into the left child and include 16 in the left child. Delete the empty right child and recursively delete 16 from the left child.



Since 16 is now in a leaf with two or more keys, we're back to Case 0.



9.2 Red-black trees

Red-black trees[1] [14] are a class of balanced binary search trees. In a red-black tree, we add the attribute of color (red or black) to the nodes. Red-black trees are used in the C++ Standard Library and in Java to implement maps (the equivalent of Python's dictionaries).

9.2.1 Definition of a red-black tree

A red-black tree is a binary tree in search order with the following properties.

1. Every node is either red or black.
2. The root is black.
3. If a node is red, its children must be black.
4. Every path from the root to a null link contains the same number of black nodes.

The last condition we will call perfect black balance. The height of an N-node red-black BST is at most $2\lg(N+1)$, so search, insertion, and deletion are $\lg N$ operations.

If we add the requirement that if a (black) node has only one red child, then it is the right child, then we have what is called a right-leaning red-black tree. One could alternatively speak of left-leaning red-black trees.

[1] The name derives from the fact that the original authors decided red and black were the best colors for the early laser printer available to them at Xerox Palo Alto Research Center (PARC).

9.2.2 Red-black trees as an encoding of 2-3-4 trees

Red-black trees are most easily understood as a way of representing 2-3-4 trees as binary trees.[2]

- A 2-node in a 2-3-4 tree is a black node in a red-black tree.
- A 3-node in a 2-3-4 tree is a black node with a single red child in a red-black tree.
- A 4-node in a 2-3-4 tree is a black node with two red children in a red-black tree.

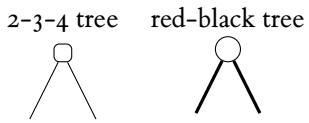


Figure 9.2.1: A 2-node in a 2-3-4 tree and its red-black representation.

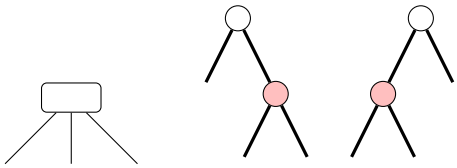


Figure 9.2.2: A 3-node in a 2-3-4 tree and its red-black representation.

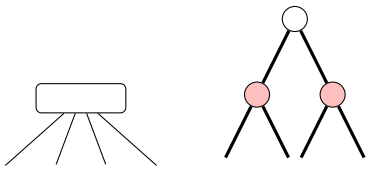


Figure 9.2.3: A 4-node in a 2-3-4 tree and its red-black representation.

In Figure 9.2.4 we give a 2-3-4 tree and in Figure 9.2.5 we give its red-black equivalent.

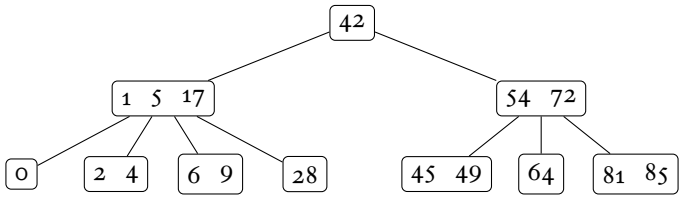


Figure 9.2.4: A 2-3-4 tree.

[2] The original motivation in [14] was a related tree called a 2-3 tree.

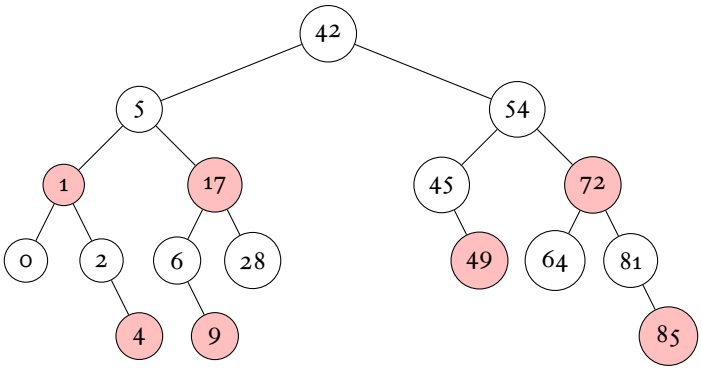


Figure 9.2.5: An equivalent right-leaning red-black tree.

Red-black trees and 2-3-4 trees are isomorphic in the sense that there is a one-to-one correspondence between 2-3-4 trees and right-leaning red-black trees.

9.2.3 The red-black equivalent of splitting a 3-node

Top-down insertion into a red-black tree follows top-down insertion into a red-black tree. As such, a red-black tree needs to mimic the splitting of nodes as in a 2-3-4 tree. First we look at the equivalent of splitting a 3-node.

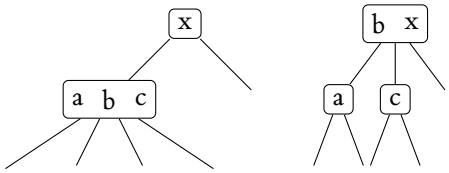


Figure 9.2.6: Splitting a 3-node in a 2-3-4 tree.

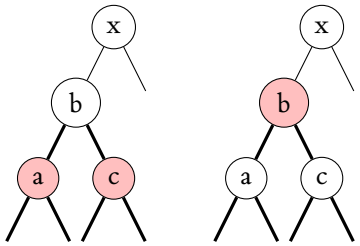


Figure 9.2.7: The red-black equivalent of splitting a 3-node.

Observe that in the red-black equivalent there is a color swap between parent and child nodes.

9.2.4 The red-black equivalent of splitting a 4-node

Next we lok at the red-black equivalent of spliting a 4-node.

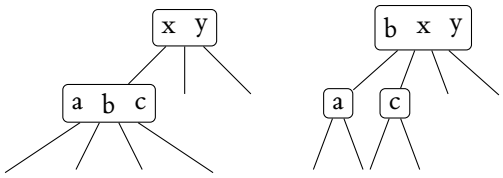


Figure 9.2.8: Splitting a 4-node in a 2-3-4 tree.

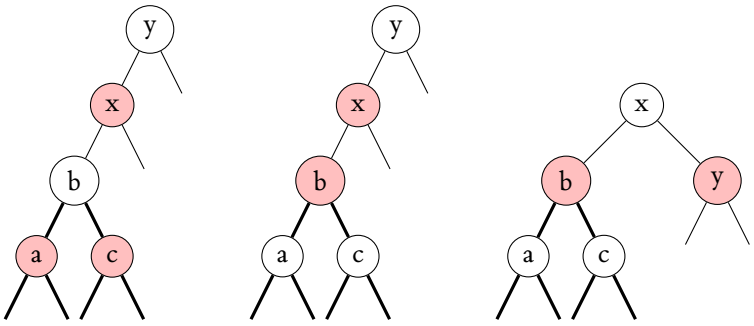


Figure 9.2.9: The equivalent of splitting a 4-node.

The red-black equivalent involves a double rotation as in an AVL tree and a color swap.

9.2.5 Building a red-black tree: top-down insertion

In this section we will map top-down insertion into a 2-3-4 trees to top-down insertion into a red-black tree using the equivalences in the previous section, expressed in terms of red-black trees.

In order to maintain perfect black balance, any new node added to the tree must be red:

- If the parent of the new node is black, all is well.
- If the parent of the new node is red, this violates the condition that red nodes have only black children.

In the latter case we can fix things using rotations similar to those for AVL trees. We will show that these rotations can be used to maintain the red-black structure at any point in the tree, not just at insertion of a new node.

To preserve the perfect black balance, a newly inserted node must be red. In top-down insertion, we change the tree as we move down the tree to the point of insertion. The changes we make insure that when we insert the new node, the parent is black.

As we move down the tree to insert a new node, if we encounter a node X with two red children, we make X red and its children black. (If X is the root, we then change the color back to black.) A color flip can cause a red-black violation (a red child with a red parent) only if X's parent P is red.

First suppose the parent is red and the parent's sibling is black (or missing). We have two cases.

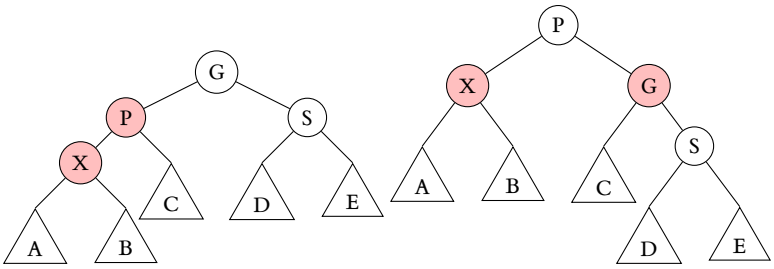


Figure 9.2.10: This is a single rotation and a color swap for P and G.

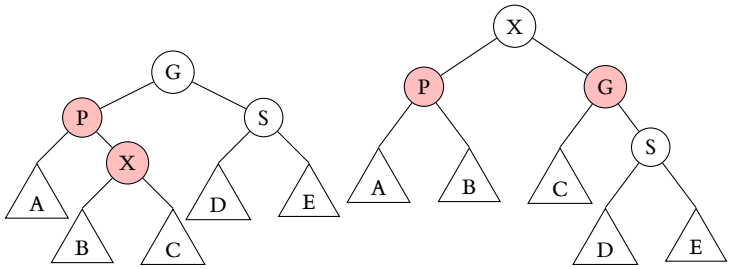


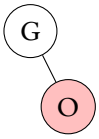
Figure 9.2.11: This is a double rotation and a color swap for X and G.

On the other hand, suppose the parent is red and the parent's sibling is red. This can't happen—it would mean that the parent and its sibling are both red, and we changed all such pairs to black on the way down.

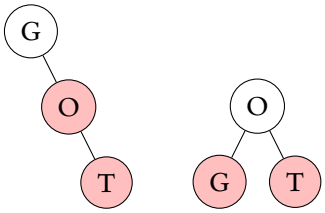
9.2.6 Example: GOTCHA

Insert G:

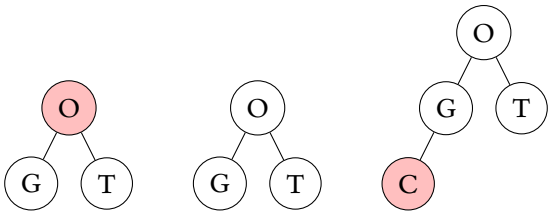
Insert O:



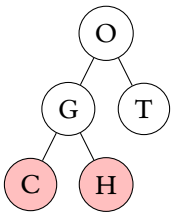
Insert T:



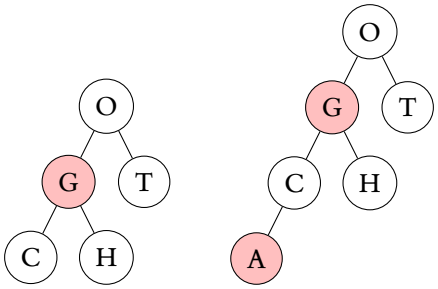
Insert C:



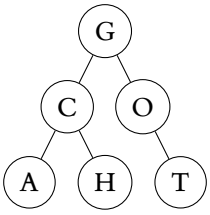
Insert H:



Insert A:



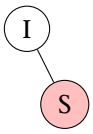
The standard BST built from GOTCHA is slightly shorter, showing that a red-black tree for a series of insertions need not be shorter than the corresponding standard BST.



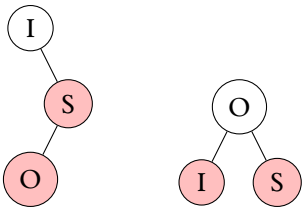
9.2.7 Example: ISOGRAM

Insert I:

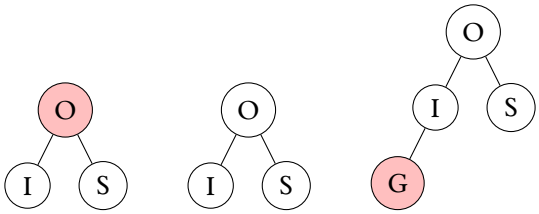
Insert S:



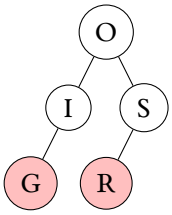
Insert O:



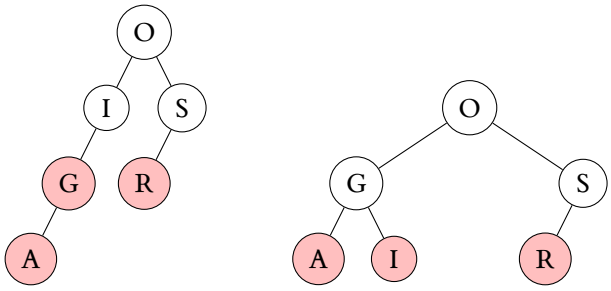
Insert G:



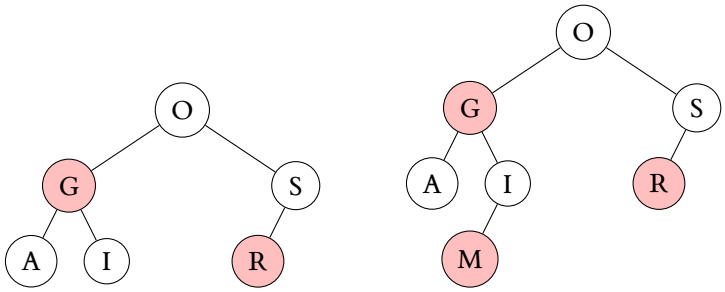
Insert R:



Insert A:



Insert M:



9.3 Exercises

Exercise 9.3.1. *Insert 3,1,4,5,9,2,6,8,7,0 one-by-one into an empty 2-3-4 tree using top-down insertion. Delete 0 and then 9 from the 2-3 tree. You must show the tree after each insertion and each deletion.*

Exercise 9.3.2. *Draw the 2-3-4 tree that results if we insert 22 into the following 2-3-4 tree using top-down insertion.*

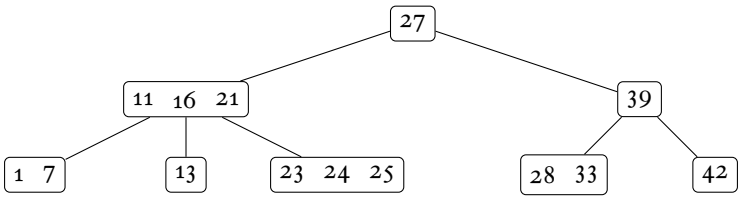


Figure 9.3.1: Insert 22 into this tree.

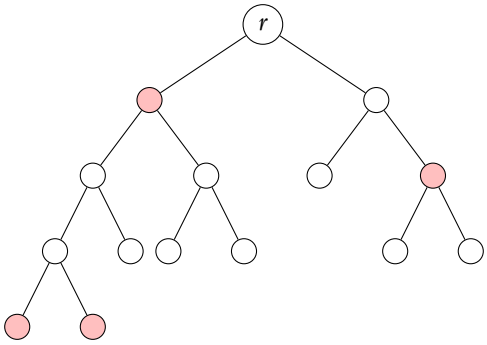
Exercise 9.3.3. *Draw the right-leaning red-black tree that is equivalent to the tree in Figure 9.3.1.*

Exercise 9.3.4. *Show the result of building a red-black tree using top-down insertion from 2, 1, 4, 5, 9, 3, 6, 7. No partial credit is possible unless you also show the tree at the end of each insertion.*

Exercise 9.3.5. *Given a node a in a red-black tree, define*

$\text{bhops}(a)$ = *the number of black nodes (including a) on a path from a to a leaf.*

For example, $\text{bhops}(r) = 3$ *in the following graph:*



Let T be a red-black tree containing n nodes. Let r be its root. In this exercise we are using the usual definitions of the height of a tree and the depths of nodes.

1. *Show that the height of T is no more than* $2\,\text{bhops}(r)$.
2. *Explain why there are no leaves at depths d satisfying* $0 \le d \le \text{bhops}(r) - 1$.
3. *Use the preceding fact to show that*

$$2^{\text{bhops}(r)-1} - 1 \le n.$$

4. *Show that the height of T is no larger than* $2\lg(n+1)$.

Chapter 10

B-trees

B-trees [3] were originally motivated by applications in information storage and retrieval, particularly databases. We will focus on a particular variant of B-tree known as the B^+tree (see [9] for an overview of B-trees and B^+trees). Besides what we normally consider databases, filesystems are just big databases, and B^+trees are used to store meta-data in many filesystems (e.g., HFS+, ext4, BTRFS).

10.0.1 Motivation

We have frequently treated the key as if it were the data being stored, but that is rarely the case. For example, consider student records in Banner. The most effective search key is your W&M id number (e.g., 93...), since it is unique. However, the record associated with each key contains more details:

- Student Information
- Student Address and Phones
- Student E-mail Address
- Student Schedule
- Student Active Registrations
- Student Academic Transcript
- and so on...

B^+trees are particularly useful when we cannot fit all of our data in memory, but have to perform reads and writes from secondary storage (e.g., electromechanical disk drives). Consider a disk drive that rotates at 7200 rpm. The rotational speed plays a role in retrieval time; for a 7200 rpm disk, each revolution takes

$$\frac{60}{7200} = \frac{1}{120} \text{ second,}$$

or about 8.3 milliseconds. A typical seek time—the time for the disk head to move to the location where data will be read or written—for 7200 rpm disks is around 9 ms. This means we can perform 100–120 random disk accesses per second. Meanwhile, our CPU can perform more than 1,000,000,000 operations per second. Relative to the speed of the CPU disk access is incredibly slow.

Now suppose we have a database with $N = 10,000,000$ entries that we organize in a tree. In a standard BST, searches in a randomly build tree take, on average, approximately $1.39 \lg N \approx 32$ disk accesses. We would expect 1000 random searches of a BST to take

$$1000 \times 32 \times 9 = 290,000 \text{ milliseconds,}$$

or about 5 minutes.

In an AVL tree, a worst-case search requires $1.44 \lg N \approx 33$ disk accesses. At 9 ms per access, this requires about 300 ms, so on average we can perform less than 4 searches per second. We would expect 1000 worst-case searches to take

$$1000 \times 33 \times 9 \approx 300,000 \text{ milliseconds},$$

again, about 5 minutes. What we see is that in this application, search trees with height $\propto \lg N$ are still too high!

We can reduce the height if we allow more branching. Binary search trees only allow 2-way branching (i.e., each node has only two descendents). We can reduce the height of our search trees if we allow multiway branching. We can also make things more efficient if we separate the search index from the data being stored. This allows us to keep the search index in memory.

We distinguish between comparing keys to direct the search to the data and fetching the values (records). While the individual records could be huge and are thus stored on disk, the keys themselves should be small. So, we wait to fetch any data from the disk until we know exactly which item is needed.

10.0.2 B$^+$trees

Given M (typically even), a B$^+$tree of order M is an M-ary search tree with the following properties:

1. The data values are stored in the leaves, also called external nodes.
2. All nonleaf nodes, also called internal nodes, other than the root have between $\lceil M/2 \rceil$ and M children.
   - This means each nonleaf has up to $M - 1$ keys to guide searches.
   - It also means that key i is the smallest key in subtree $i + 1$.
3. The root is either a leaf or it has between 2 and M children.
4. All leaves are at the same depth and store between $\lceil L/2 \rceil$ and L data values (where we choose L).

The last property says that a B$^+$tree is perfectly balanced.

For example,

- For $M = 2$, there are between $\lceil 2/2 \rceil = 1$ and 2 children.
- For $M = 3$, there are between $\lceil 3/2 \rceil = 2$ and 3 children.
- For $M = 4$, there are between $\lceil 4/2 \rceil = 2$ and 4 children.
- For $M = 5$, there are between $\lceil 5/2 \rceil = 3$ and 5 children.
- For $M = 42$, there are between $\lceil 42/2 \rceil = 21$ and 42 children.

Figure 10.0.1 shows a B$^+$tree with $M = 5$.

10.0.3 Choosing M and L

The choice of M and L reflects the properties of our storage medium. For efficiency, each node will occupy a disk block, say, 8192 bytes. Suppose each key uses 32 bytes and a link to another node uses 8 bytes.

A node in a B$^+$tree of order M has $M - 1$ keys and M links, so a node requires

$$32(M - 1) + 8M = 40M - 32 \text{ bytes}.$$

We choose as our M the largest M that will allow a node to fit in a block:

$$M = \lfloor \frac{8192 + 32}{40} \rfloor = 205.$$

This means each internal node branches in at least 103 ways. If the values are each 256 bytes, then we can fit

$$L = \lfloor \frac{8192}{256} \rfloor = 32$$

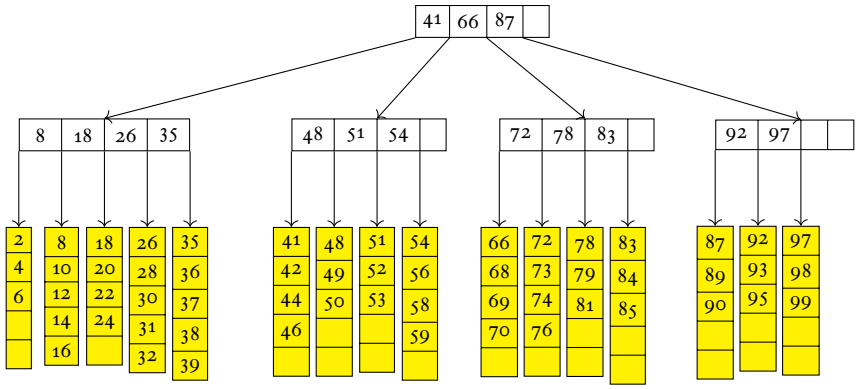


Figure 10.0.1: A B$^+$tree with $M = 5$.

in a single block. This means each leaf has between 16 and 32 values,

If there are 1,000,000,000 values to store, there are at most 31,250,000 leaves. The leaves would be, in the worst case, on level

$$1 + \log_{103} 31,250,000 = 5,$$

so we can find data in at most 5 disk access. By contrast, a BST would have at least $1 + \lg 31,250,000 = 26$ levels!

10.0.4 Insertion into a B$^+$tree

Insertion: easy case

First, follow the search tree to the correct leaf (external node). If there are fewer than $L - 1$ items in the leaf, insert the new key in the correct location. This costs one disk access.

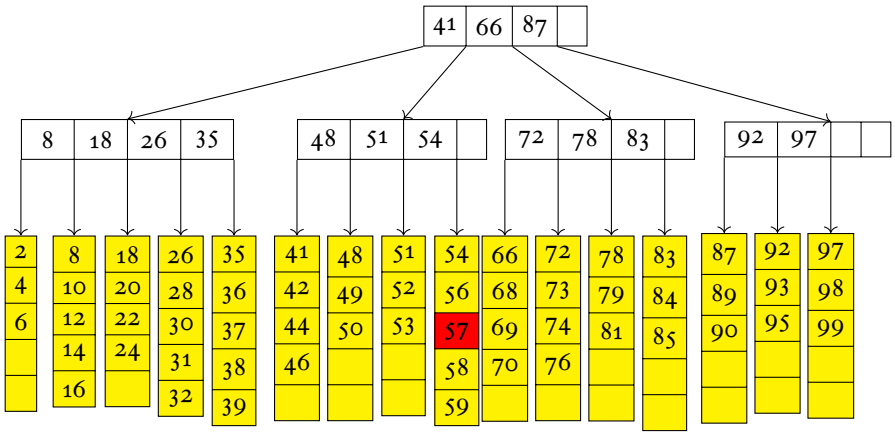


Figure 10.0.2: The easy case.

Insertion: splitting a leaf

Otherwise, there are already $L - 1$ items in the leaf. Add the new item, split the node in two, and update the links in the parent node. This costs three disk accesses (one for each new node and one for the update of the parent node).

The splitting rule ensures we still have a B$^+$tree: each new node has at least $\lceil L/2 \rceil - 1$ values (e.g., if $L = 3$, there are 2 values in one node and 1 in the other, and if $L = 4$, each new node has 2 keys).

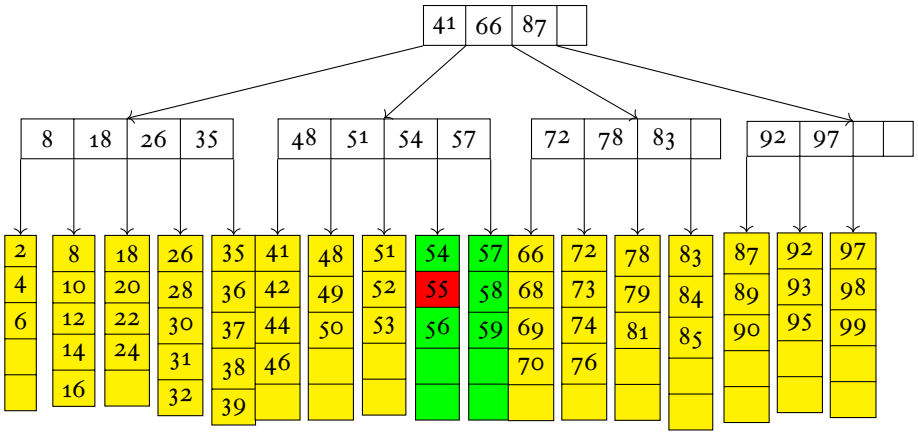


Figure 10.0.3: Splitting a leaf.

Insertion: splitting a parent

What if the parent node already has all the child nodes it can have? We must split the parent node, and update its parent. We repeat this process until we arrive at the root. If necessary, we split the root into two nodes and create a new root with the two nodes as children.[1] This means that a B$^+$tree grows upward at the root.

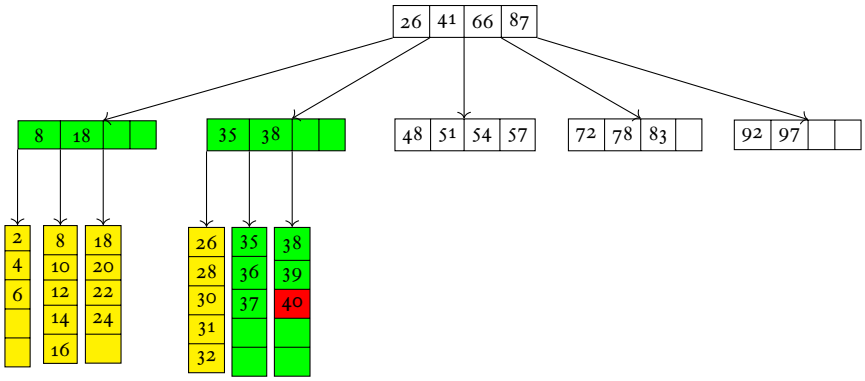


Figure 10.0.4: Splitting a parent.

10.0.5 Complexity of searching

We have the following result on the height of M-ary trees.

Proposition 10.0.1. *A B$^+$tree of order M with N items requires between $\log_M N$ and $\log_{M/2} N$ search key comparisons.*

Before we give the proof, let us look at what Proposition 10.0.1 says. Suppose $N = 1,000,000,000$ and $M = 500$. Then

$$3 \leq \log_M N, \log_{M/2} N < 4.$$

If $M = 500$, then $\log_{M/2} N \leq 4$ for $N \leq 3,900,000,000$. M-ary trees are short and squat, which is what we want for search trees.

Proof All the internal nodes have between $M/2$ and $M-1$ links (children).

In the best case, we have a complete tree with branching factor $M-1$, which leads to a height of $\log_M N$.

In the worst case, we have a complete tree with branching factor $M/2$, which leads to a height of $\log_{M/2} N$. ∎

[1] This is why the root is allowed as few as 2 children.

As a consequence of Proposition 10.0.1 we have the following.

Proposition 10.0.2. *Searching in a B^+tree of order M with N items is $\Theta(\lg N)$, worst-case.*

The complexity for seaching a B^+tree is comparable to that of a binary tree, but the B^+tree can be read more efficiently.

Proof Suppose that each node has $M-1$ keys and M children. Proposition 10.0.1 tells us that the height of the tree is proportional to $\log_M N$. At each node we can use binary search to find which branch to take, which takes work proportional to $\log_2 M$, worst-case. Thus, in the worst case, the total amount of work for a search is proportional to

$$\log_2 M \times \log_M N = \log_2 N.$$

■

10.1 Exercises

Chapter 11

Hashing

A hash table is a generalization of an ordinary array. Hashing converts search keys into locations in a hash table. This turns searching on the key into something like array lookup. Python uses hash tables to implement dictionaries.

Hashing is typically a many-to-one map: multiple keys are mapped to the same location. If multiple keys hash to the same value, we have a collision that must be resolved.

There are two parts to hashing:

1. A hash function, which transforms keys into array indices.
2. A collision resolution procedure, which handles collisions.

Let K be the set of search keys. A hash functions map K into the set of M slots in the hash table:

$$h : K \to \{0, 1, \ldots, M-1\}.$$

Ideally, h distributes K uniformly over the slots of the hash table, to minimize collisions. That is, if we are hashing N items, we want the number of items hashed to each location to be close to N/M.

The Library of Congress Classification system is a hash function if we look at the first part of the call numbers (e.g., E470, PN1995). Collision resolution then involves going to the stacks and looking through the books. Almost all computer science titles are hashed to QA75 and QA76, which is doubleplus ungood!

Python has a built-in hash function named `hash()`.

11.1 Examples of hash functions

There seem to be as many hash functions in the world as there are programmers. We will look only at some of the more common approaches to hashing.

Suppose we are storing a set of nonnegative integers. Given M, we can obtain hash values between 0 and $M-1$ with the hash function

$$h(k) = k \mod M,$$

i.e., the remainder when k is divided by M. This is called modular hashing. Some care is needed inchoosing M. For instance, if $M = 2^p$, then $h(k)$ is the just the p lowest-order bits of k. If the structure of the data means that keys have the same low-order bits there may be many collisions. Choosing M to be a prime not too close to a power of 2 works well in practice.

In multiplicative hashing we choose a constant A in the range $0 < A < 1$. We then perform the following:

1. Multiply k by A.
2. Multiply the fractional part of kA by M and take the floor of the result.

Mathematically,

$$h(k) = \lfloor M(kA \mod 1) \rfloor.$$

We can implement multiplicative hashing efficiently as follows. Suppose the word size on our machine is w and k fits in a single word.

1. Choose $M = 2^p$.
2. Choose A to have the form $A = 2/2^w$.
3. Multiply k by $2^w A$, giving us a $2w$-bit integer $2^w r + q$.
4. The p-bit hash value is then the p most significant bits of q.

11.2 Hashing non-integers

So far we have only hashed integers. How do we handle other types?

11.2.1 Hashing floating point numbers

Return to simple multiplicative hashing for nonnegative integers:

$k \mapsto k \% M$.

We can hash floating point numbers using the same rule as for integers if we interpret the bits of the floating point number as if they were the bits of an integer.

Here are two ways to this in C, assuming `unsigned long int` and `double` have the same length. The first uses some pointer hackery:

```
unsigned long *k; double x;
k = (unsigned long*) &x;
long int hash = *k % M;
```

The second uses a union, which is a variable that can hold objects of different types and sizes:

```
union {
  long int k;
  double x;
} u;
u.x = 3.1416;
long int hash = u.k % M;
```

11.2.2 Hashing strings

We can hash strings by combining a hash of each character.

```
char *s = "hello!";
unsigned long hash = 0;
for (int i = 0; i < strlen(s); i++) {
  unsigned char w = s[i];
  hash = (R * hash + w) % M;
}
```

R is an additional parameter we must choose. If R is larger than any character value, then this approach is what you would obtain if you treated the string as a base-R integer.

Kernighan and Ritchie [24] suggest a slightly simpler hash function corresponding to $R = 31$:

```
char *s;
unsigned hash;
for (hash = 0; *s != '\0'; s++) {
  hash = 31 * hash + *s;
}
hash = hash % M;
```

The value $R = 37$ also works well.

11.2.3 Hashing compound keys and vectors

We can use the idea for strings if our search key has multiple parts, say, `street`, `city`, `state`:

```
hash = ((street * R + city) % M) * R + state) % M;
```

The same idea applies to hashing vectors.

11.3 Quality of hashing

The choice of parameters can have a dramatic effect on the results of hashing. We compare the text's string hashing algorithm for different pairs of R and M. We plot histograms of the number of words hashed to each hash table location. We use the American dictionary from the `aspell` program as data (305,089 words). We display plots for the following choices:

1. $R = 31, M = 1024$,
2. $R = 32, M = 1024$,
3. $R = 31, M = 1000$,
4. $R = 32, M = 1000$.

Figure 11.3.1, $R = 31, M = 1024$, and Figure 11.3.3, $R = 31, M = 1000$, are good hashes—the hash values of the words are evenly distributed, with about 300 words hashing to the same value. On the other hand, Figure 11.3.2, $R = 32, M = 1024$, is terrible—just a few hash values account for the vast bulk of the words. Figure 11.3.4, $R = 32, M = 1000$, is also not a good hash, as it is also unevenly distributed.

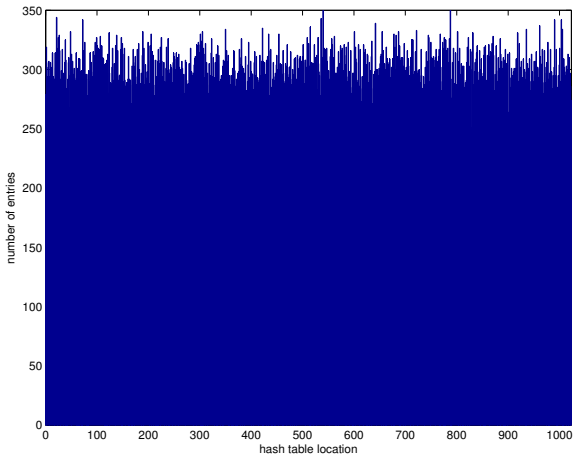


Figure 11.3.1: $R = 31, M = 1024$

11.4 Collision resolution via separate chaining

In separate chaining, we create a linked list for each entry in the hash table. Items that hash to the same entry are then added the the list for that hash table entry. Adding, retrieving, and deletion are then just those operations for linked-lists. The cost of each is proportional to the length of the linked list.

So, how long are the linked lists in the hash table? The expected value (average) is N/M. However, it is likely that we will be near the expected value? or is it reasonably likely that we will be $\gg N/M$? This is a question about the variance.

Proposition 11.4.1. *Suppose our hash function uniformly and independently distributes keys over* $0, 1, 2, \ldots, M-1$. *In a separate-chaining hash table with M lists and N keys, then is extremely likely that the number of keys in a list is close to* N/M.

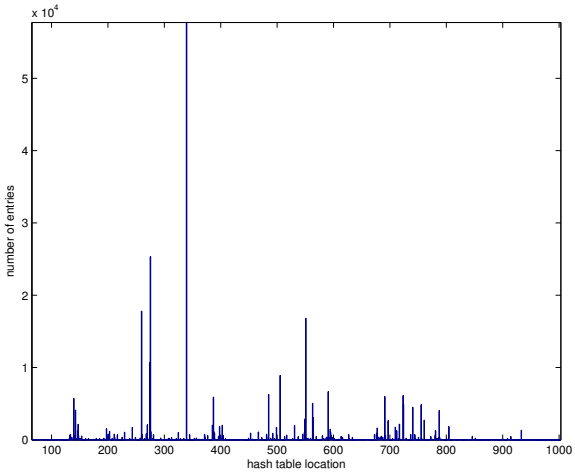


Figure 11.3.2: $R = 32, M = 1024$

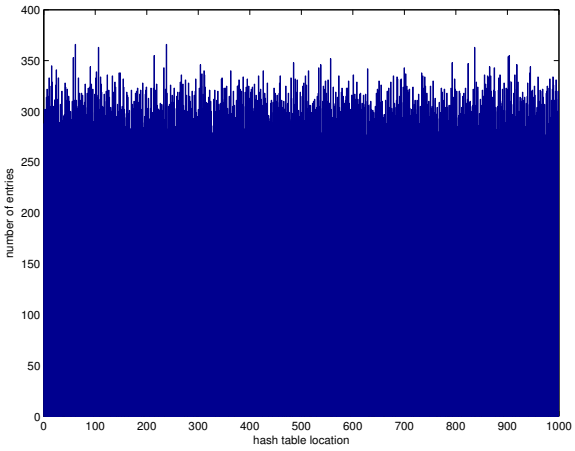


Figure 11.3.3: $R = 31, M = 1000$

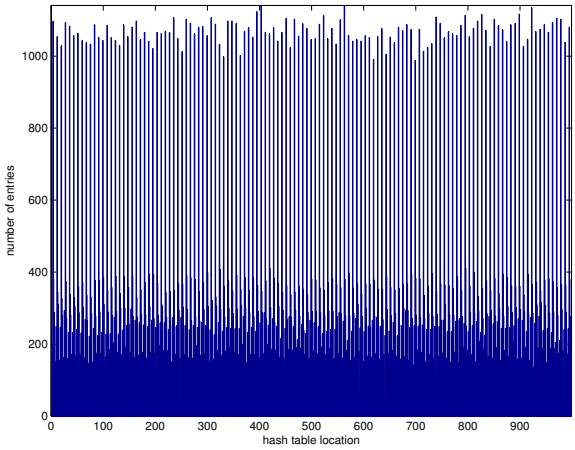


Figure 11.3.4: $R = 32, M = 1000$

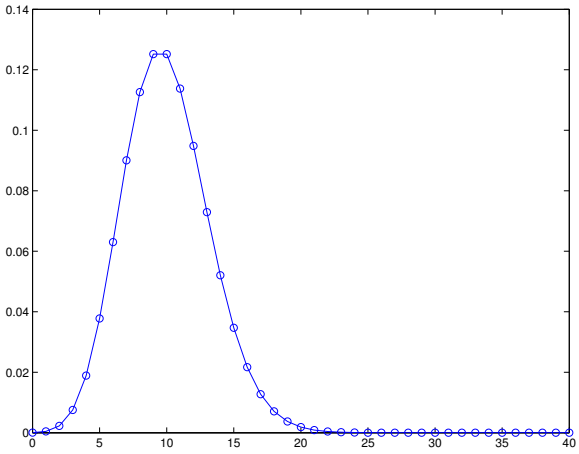


Figure 11.4.1: Binomial pdf: $N = 10^4, M = 10^3, \alpha = 10$.

PROOF We consider the probability that a particular list will contain exactly k keys. Choose k of the N keys. The probability that these k keys, and only these k keys, hash to the list is

$$\left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k}$$

Since there are $\binom{N}{k}$ ways to choose a subset of k keys from the set of N, the probability that a given list contains exactly k keys is

$$\binom{N}{k} \left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k}.$$

This is the binomial distribution. An example of the probability density function for the distribution is given in Figure 11.4.1; you can see that it is fairly concentrated about the expected value $N/M = 10$. It is very unlikely we will see any list with more than 20 items in it.

If we define $\alpha = N/M$, we can rewrite this as

$$\binom{N}{k} \left(\frac{\alpha}{N}\right)^k \left(1 - \frac{\alpha}{N}\right)^{N-k}.$$

This shows the dependence on N and $\alpha = N/M$. ∎

11.5 Collision resolution via open addressing

An alternative to separate chaining is open addressing. Here the idea is to use a single table of size $M > N$ (rather than the M linked lists of separate chaining). The trick is figuring out where to put things. A simple approach is linear probing. The operation of checking whether a table entry is vacant (or is one we seek) is called a probe. In linear probing, insertion works like this:

- If, when we hash k, the slot $h(k)$ is open, then we put k there.
- If there is a collision, then we start looking for an empty slot starting with location $h(k) + 1$ in the hash table, and proceed linearly through $h(k) + 2, \ldots, m - 1, 0, 1, 2, \ldots, h(k) - 1$, wrapping around the hash table, looking for an empty slot.

The search operation is similar.

Linear probing has some drawbacks. First of all, deletion is not straightforward, and potentially requires moving other entries in a cluster of entries. As clusters form, it takes longer to find an empty spot (or search for an item).

Proposition 11.5.1. *In a linear-probing hash table with M entries and $\alpha = N/M$ keys, $\alpha < 1$, the average number of probes required is approximately*

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

for search hits and approximately

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

for insertions and search misses.

For $\alpha = 1/2$, these values are $3/2$ and $5/2$, respectively.

In general open addressing, there will be a probe sequence $h(k,0), h(k,1), \ldots, h(k, M-1)$ that is a permutation of $0, 1, \ldots, M-1$. In this way every hash table position is eventually considered.

In linear probing,

$$h(k,i) = (h'(k) + i) \mod M, \quad i = 0, 1, \ldots, M-1,$$

where $h'(k)$ is our initial entry into the table. Quadratic probing uses a hash function of the form

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod M,$$

whre $i = 0, 1, \ldots, M-1$ and $c_1, c_2 \neq 0$. This approach suffers less from clustering, at the cost of a slightly more expensive hash function.

11.6 Universal hashing

In hashing, it is conceivable we could end up with all of our keys are hashed to the same location in the hash table. More realistically, we could choose a hash function that does not evenly distribute the keys. We saw examples of this in Section 11.3.

Among other properties, a good hash function should

1. be computable in constant time, independent of the size of the hash table;
2. distribute items uniformly across slots in the hash table.

To avoid imbalanced hashing, we can choose the hash function randomly from a family of hash functions. This approach, called universal hashing, yields provably good performance on average, disirregardless of the keys.

Definition 11.6.1. *Let $\mathcal{H}$ be a finite collection of hash functions mapping our set of keys K to the range $\{0, 1, \ldots, m-1\}$. $\mathcal{H}$ is called a universal collection if for each pair of distinct keys $x, y \in K$, the probability of randomly selecting a hash function $h \in \mathcal{H}$ for which $h(x) = h(y)$ is at most $1/m$.*

Otherwise said, given a pair of distinct keys x, y, the number of hash functions in $\mathcal{H}$ for which $h(x) = h(y)$ is at most $|\mathcal{H}|/m$.

For a randomly selected hash function h from a universal collection, the chance of a collision between distinct x and y is not more than the probability of a collision if $h(k)$ and $h(y)$ were chosen randomly and independently from $\{0, 1, \ldots, m-1\}$. In this way the results of universal hashing resemble random assignment of keys to hashes.

11.6.1 The Carter-Wegman universal collection

In the paper where universal hashing first appears [8] the authors propose the family of hash functions described in Proposition 11.6.2.

Proposition 11.6.2. *Assume the keys are nonnegative integers and let p be a prime larger than all the keys. Then the family*

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m, 1 \leq a \leq p - 1, 0 \leq b \leq p - 1$$

is a universal class of hash functions.

There are $p(p-1)$ hash functions in this family. The range of any $h_{a,b}$ is $\{0, 1, \ldots, p-1\}$.

Proof Suppose $x > y$ are two keys such that $h_{a,b}(x) = h_{a,b}(y)$. We first rule out one possibility that could lead to a collision. We have a collision if and only if

$$(ax + b) \equiv (ay + b) \bmod p.$$

However, this cannot occur, since it would mean

$$a(x - y) \equiv 0 \bmod p.$$

Since p is prime, this means p divides one of a or $x - y$, neither of which can be true since $a, x - y < p$. This shows that if

$$r = (ax + b) \bmod p$$
$$s = (ay + b) \bmod p,$$

then $r \neq s$. There are p possible values for r and $p - 1$ possible values for s, so there are $p(p-1)$ possible pairs (r, s).

We claim there is a one-to-one correspondence of (a, b) pairs and (r, s) pairs. To see why, observe that we can solve the relation

$$a(x - y) \equiv (r - s) \bmod p$$

for a by multiplying by the multiplicative inverse of $x - y \bmod p$. Such an inverse exists because p is prime and $x - y \neq 0$. Once we have a we can solve for b.

The claim means that the probability that x and y collide is the probability that

$$r \equiv s \bmod m,$$

and choosing (a, b) randomly is equivalent to choosing (r, s) randomly. For a given r, the number of values of s for which $r \equiv s \bmod m$ is at most

$$\lceil \frac{p}{m} \rceil - 1.$$

The -1 term appears because we know $r \neq s$. Since

$$\lceil \frac{p}{m} \rceil - 1 < \frac{p-1}{m},$$

and there are $p - 1$ possible values of s, the probability r and s collide is $1/m$. ∎

Ideally we would have no collisions. This will be the case if our hash function is one-to-one. The following proposition addresses the probability this will happen.

Proposition 11.6.3. *Let $S \subset \{0, 1, \ldots, p-1\}$ be a set of n integers and choose $h_{a,b}$ at random. Then the probability that $h_{a,b}$ is one-to-one on S is at least $1 - n^2/p$.*

PROOF We will first compute the probability that $h_{a,b}$ is **not** one-to-one on S. We want to compute the probability P that $h_{a,b}(x) = h_{a,b}(y)$ for some pair $x, y \in S$, $x \neq y$. There are at most $n(n-1)/2$ possible pairs x, y. By Proposition 11.6.2,

$$P \leq \frac{n(n-1)}{2}\frac{1}{p} < \frac{n^2}{p}.$$

Therefore the probability $h_{a,b}$ is one-to-one on S is $1 - n^2/p$ ∎

How do we efficiently compute a Carter-Wegman hash function? Recall that

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m.$$

Carter and Wegman suggested choosing p to be a Mersenne prime, which is a prime number of the form $2^k - 1$. One such prime is $2^{31} - 1$.[1] Multiplication by a Mersenne prime can be implemented by shifting bits to the left k locations and then subtracting 1.

Similarly, a modulus operation for a Mersenne prime can be done by a bit shift and an addition. Suppose

$$r \equiv y \bmod p.$$

Divide y by $p + 1$, so we can write

$$y \equiv q'(p + 1) + r' \bmod p,$$

where q' is the result of shifting the bits of y to the right k places and r' are the k lower bits of y. Then

$$r \equiv q'(p + 1) + r' \bmod p.$$

Since $p + 1 \equiv 1 \bmod p$, it follows that $r \equiv q' + r' \bmod p$.

11.6.2 Universal hashing for strings and vectors

Now assume we wish to hash a vector of integers

$$x = (x_1, \ldots, x_k),$$

where $0 \leq x_i \leq M - 1$ for all i. For instance, x could represent an ASCII string and each x_i would be a single character (unsigned int) in the range 0 to 127.

Proposition 11.6.4. *Suppose we choose M, the table size, to be prime. Choose random numbers $r_1, \ldots, r_k$ in the range 0 to $M - 1$ and define*

$$h(x) = (r_1x_1 + r_2x_2 + \cdots + r_kx_k) \bmod M. \tag{11.6.1}$$

Then (11.6.1) defines a universal hashing function.

PROOF Let x and y be distinct keys. We want to show that the probability $h(x) = h(y)$ is no more than $1/M$.

Since $x \neq y$, there must be some component i for which $x_i \neq y_i$. Let's look at all the random numbers for r_j for $j \neq i$, and define

$$h'(x) = \sum_{j \neq i} r_j x_j.$$

Then

$$h(x) = h'(x) + r_i x_i.$$

This means we can have a collision between x and y if and only if

$$h'(x) + r_i x_i = h'(y) + r_i y_i \bmod M,$$

[1] This is number so important it has its own Wikipedia page.

or, equivalently, when

$$r_i(x_i - y_i) = h'(y) - h'(x) \bmod M.$$

Since M is prime, every integer between 1 and $M-1$ has a multiplicative inverse modulo M. Thus, since $x_i - y_i \neq 0$, there is a multiplicative inverse $(x_i - y_i)^{-1}$ module M:

$$(x_i - y_i)(x_i - y_i)^{-1} \bmod M.$$

Thus there is exactly one value of r_i modulo M that leads to a collision:

$$r_i = (h'(y) - h'(x))(x_i - y_i)^{-1} \bmod M.$$

The probability of this occurring is $1/M$. ∎

11.6.3 The DHKP universal collection

As an alternative to Carter-Wegman we will look at another universal collection, first proposed in [10]. Suppose we wish to hash of w-bit keys to u-bit indices. Choose a random, odd, w-bit number b. The associated hash function is

$$h_b(k) = ((bk) \mod 2^w)//2^{w-u}$$

where // denotes integer division, as in Python. There are 2^{w-1} such functions, as this is the number of odd numbers between 1 and 2^w.

These hash functions are particularly easy to compute. Both the remainder calculation and the integer division are a matter of right shifts—no arithmetic needs to be done. In C or Python the hash function is quite simple:

```
hb(k) = (b*k) » (w-u);
```

In the proof, k is w and ℓ is u

Proof Consider distinct integers $x, y \in 0, ..., 2^w - 1$ with $x > y$, and let $z = x - y$. Let $A = \{a \mid 0 < a < 2^w\}$ and a is odd. By the definition of h_a, every $a \in A$ with $h_a(x) = h_a(y)$ satisfies

$$|ax \mod 2^w - ay \mod 2^w| < 2^{w-u}.$$

Since $z \neq 0 \mod 2^w$ and a is odd, we have $az \neq 0 \mod 2^w$. Therefore all such a satisfy

$$az \mod 2^w \in \{1, ..., 2^{w-u} - 1\} \cup \{2^w - 2^{w-u+1}, ..., 2^w - 1\}. \tag{11.6.2}$$

In order to estimate the number of $a \in A$ that satisfy (11.6.2), write $z = z'2^s$ with z' odd and $0 \leq s < w$. Since the odd numbers $1, 3, \ldots, 2^w - 1$ form a group with respect to multiplication modulo 2^w, the mapping

$$a2^s \mapsto az' \mod 2^w$$

is a permutation of A. Consequently, the mapping

$$a2^s \mapsto az'2^s \mod 2^{w+s} = az \mod 2^{w+s}$$

is a permutation of the set $\{a2^s \mid a \in A\}$. Thus, the number of $a \in A$ that satisfy (11.6.2) is the same as the number of $a \in A$ that satisfy

$$a2^s \mod 2^w \in \{1, \ldots, 2^{w-u} - 1\} \cup \{2^w - 2^{w-u} + 1, u, 2^w - 1\}. \tag{11.6.3}$$

Now, $a2^s \mod 2^w$ is just the number whose binary representation is given by the $w - s$ least significant bits of a, followed by s zeroes. This easily yields the following: If $s \geq w - u$, then no $a \in A$ satisfies (11.6.3). For smaller s, the number of $a \in A$ satisfying (11.6.3) is at most $2^w - u$. Hence the probability that a randomly chosen $a \in A$ satisfies (11.6.2) is at most $2^{w-u}/2^w - 1 = 1/2^{u-1}$. ∎

Proposition 11.6.5. *Let n, w and u be positive integers with $u \leq w$ and let S be a set of n integers in the range $\{0, \ldots, 2^w - 1\}$. Choose $h \in H_{w,u}$ at random. Then the probability that h is one-to-one on S is at least $1 - n^2/2^u$.*

11.7 Exercises

Exercise 11.7.1. *Show that if we wish to hash of w-bit keys to u-bit indices, we can do so by choosing a $p > 2^w$ and a random value $0 < a < p$ and use the hash function*

$$h_a(k) = ((ak) \mod p) \mod 2^u.$$

Exercise 11.7.2 (Exercise 3.4.6 in [36]). *Suppose that keys are t-bit integers. For a modular hash function with prime $M > 2$,*

$$h(k) = k \mod M,$$

show that each key bit has the property that there exist two keys differing only in that bit that have different hash values.

Exercise 11.7.3. *Prove Proposition 11.6.5.*

Chapter 12

Dynamic programming

Dynamic programming was introduced by Richard Bellman in the early 1950s [4, 5]. In his autobiography he describes the origins of the name [11]:

> I spent the Fall quarter (of 1950) at RAND. My rst task was to nd a name for multistage decision processes.
>
> An interesting question is, "Where did the name,dynamic programming, come from?" The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse,he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the rst place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, lets kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. Its impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

A formal mathematical definintion of the approach is possible but Bellman's Principle of Optimality [5] expresses the idea more clearly

> An optimal policy[1] has the property that whatever the initial state and decisions are, the remaining decisions must constitute an optimal policy with regard to state resulting from the first decision.

In practice, dynamic programming works by embedding the problem we wish to solve into a sequence of related problems and solving them beginning at some initial state we know.

12.1 The knapsack problem

In the knapsack problem,

- we have a knapsack that can hold a weight of at most W;
- we can choose from n different types of items to pack;

[1] A policy means a sequence of decisions.

- each item of type i has integer weight w_i and integer value of v_i.

Our goal is to load a knapsack to maximize the total value of the articles included, subject to the capacity constraint. A mathematical formulation of the optimization problem is

$$\begin{aligned} \text{maximize} \quad & \text{value} = \textstyle\sum_i v_i x_i \\ \text{subject to} \quad & \text{weight} = \textstyle\sum_i w_i x_i \le W, \\ & x_i \ge 0 \text{ and integer for all } i. \end{aligned}$$

12.1.1 Greedy approaches

Let's first see if some greedy algorithms work. Suppose $W = 10$ and

item type	weight	value
1	4	11
2	3	7
3	5	12

The optimal solution is one1 of item 1, two of item 2, and zero of item 3, for a total value of 25.

Our first greedy strategy will be to start with adding as many items of the highest value that will fit, then the items of the 2nd highest value, the 3rd highest, and so on. This algorithm leads us to choose two of item 3, with a value of 24, which is suboptimal.

Let's try a different greedy strategy. This time we start with adding as many items with the highest value per weight ratio as will fit, then items with the 2nd highest aratio, the 3rd highest, and so on. Ranking the items by value to weigh ratio, we have

$$\text{item 1} < \text{item 3} < \text{item 2}.$$

This yields the solution three of item 2 and none of the others, value 21, which is also suboptimal.

12.1.2 Dynamic programming

Next we turn to a dynamic programming approach. The approach is

1. first determine how to fill a smaller knapsack optimally, then
2. use this knowledge to fill a larger knapsack optimally.

Let $\textsc{Opt}(w)$ denote the maximum value of a w-lb knapsack. To fill a w-lb knapsack optimally, we must begin by packing an item. If we choose an item of type i then the best value we can achieve is

$$v_i + \text{the best we can do with a } (w - w_i)\text{-lb knapsack.}$$

This leads to the recurrence

$$\begin{aligned} \textsc{Opt}(0) &= 0 \\ \textsc{Opt}(w) &= \max_i \{v_i + \textsc{Opt}(w - w_i) \mid w_i \le w\}. \end{aligned}$$

The answer we seek is $\textsc{Opt}(W)$.

We illustrate this approach with $W = 10$ and

item type	weight	value
1	4	11
2	3	7
3	5	12

Clearly,

$$\textsc{Opt}(0) = \textsc{Opt}(1) = \textsc{Opt}(2) = 0,$$

since no item weighs 2 pounds or less, and

$$\textsc{Opt}(3) = 7$$

since only a single item of type 2 will fit in a 3 lb knapsack.

Now we use the recursion to fill out the values of V. For each value of w the optimal strategy is marked with a ☞.

$$\textsc{Opt}(4) = \max \begin{cases} 11 + \textsc{Opt}(0) = 11 & \text{type 1} \ ☞ \\ 7 + \textsc{Opt}(1) = 7 & \text{type 2} \end{cases}$$

$$\textsc{Opt}(5) = \max \begin{cases} 11 + \textsc{Opt}(1) = 11 & \text{type 1} \\ 7 + \textsc{Opt}(2) = 7 & \text{type 2} \\ 12 + \textsc{Opt}(0) = 12 & \text{type 3} \ ☞ \end{cases}$$

$$\textsc{Opt}(6) = \max \begin{cases} 11 + \textsc{Opt}(2) = 11 & \text{type 1} \\ 7 + \textsc{Opt}(3) = 14 & \text{type 2} \ ☞ \\ 12 + \textsc{Opt}(1) = 12 & \text{type 3} \end{cases}$$

$$\textsc{Opt}(7) = \max \begin{cases} 11 + \textsc{Opt}(3) = 18 & \text{type 1} \ ☞ \\ 7 + \textsc{Opt}(4) = 18 & \text{type 2} \ ☞ \\ 12 + \textsc{Opt}(2) = 12 & \text{type 3} \end{cases}$$

$$\textsc{Opt}(8) = \max \begin{cases} 11 + \textsc{Opt}(4) = 22 & \text{type 1} \ ☞ \\ 7 + \textsc{Opt}(5) = 19 & \text{type 2} \\ 12 + \textsc{Opt}(3) = 19 & \text{type 3} \end{cases}$$

$$\textsc{Opt}(9) = \max \begin{cases} 11 + \textsc{Opt}(5) = 23 & \text{type 1} \ ☞ \\ 7 + \textsc{Opt}(6) = 21 & \text{type 2} \\ 12 + \textsc{Opt}(4) = 23 & \text{type 3} \ ☞ \end{cases}$$

$$\textsc{Opt}(10) = \max \begin{cases} 11 + \textsc{Opt}(6) = 25 & \text{type 1} \ ☞ \\ 7 + \textsc{Opt}(7) = 25 & \text{type 2} \ ☞ \\ 12 + \textsc{Opt}(5) = 24 & \text{type 3} \end{cases}$$

We see that for a 10 lb knapsack, one optimal selection is given by

1. an item of type 1, leaving 10 - 4 = 6 lb;
2. an item of type 2, leaving 6 - 3 = 3 lb;
3. another item of type 2, leaving 3 - 3 = 0 lb.

This DP approach requires we compute $\textsc{Opt}(0), \ldots, \textsc{Opt}(W)$, and each $\textsc{Opt}(w)$ requires we look at (at most) n sums. Thus, $O(nW)$ operations are required. Rather than a truly recursive algorithm, we store values of $\textsc{Opt}(w)$ for later use; this is called memoization.

This approach is called bottom-up dynamic programming, since we solve all instances of the problem until we reach the solution we seek.

12.2 Optimal ordering of matrix multiplication

Consider the problem of multiplying three or more matrices. Suppose

- A is a 50×10 matrix;
- B is a 10×40 matrix;
- C is a 40×30 matrix;

- D is a 30×5 matrix.

When we compute the product $ABCD$ we must associate matrices since matrix multiplication is a binary operation. Recall that multiplying an $m \times p$ matrix with a $p \times n$ matrix requires $m \times p \times n$ multiplications and approximately the same number of additions. What is the best way to organize the product $ABCD$ to minimize the number of multiplications?

For instance, the cost of $A((BC)D)$ is:

- BC requires $10 \times 40 \times 30 = 12000$ multiplications;
- $(BC)D$ requires $10 \times 30 \times 5 = 1500$ multiplications;
- $A((BC)D)$ requires $50 \times 10 \times 5 = 2500$ multiplications;
- total: 16,000 multiplications.

On the other hand, the cost of $A(B(CD))$ is:

- CD requires $40 \times 30 \times 5 = 6000$ multiplications;
- $B(CD)$ requires $10 \times 40 \times 5 = 2000$ multiplications;
- $A(B(CD))$ requires $50 \times 10 \times 5 = 2500$ multiplications;
- total: 10,500 multiplications.

On the third hand, the cost of $((AB)C)D$ is:

- AB requires $50 \times 10 \times 40 = 20000$ multiplications;
- $(AB)C$ requires $50 \times 40 \times 30 = 60000$ multiplications;
- $((AB)C)D$ requires $50 \times 30 \times 5 = 7500$ multiplications;
- total: 87,500 multiplications.

12.2.1 Dynamic programming solution

The number of ways we can group n matrices for pairwise multiplication is given by the Catalan numbers:

$$C_n = \frac{(2n)!)}{(n+1)!n!}.$$

In the example of the previous section, we have

$$C_4 = \frac{8!}{5!\,4!} = 14.$$

The Catalan numbers grow very quickly. For instance $C_{20} = 6,564,120,420$.

We determine the optimal way to compute $A_1A_2 \cdots A_n$ quite efficiently via dynamic programming. Let A_i have dimensions $r_i \times c_i$. Let $m_{\ell,r}$ be the number of multiplications needed to multiply

$$A_\ell A_{\ell+1} \cdots A_{r-1} A_r.$$

We set $m_{\ell,\ell} = 0$. We also know that $m_{\ell,\ell+1} = r_\ell c_\ell c_{\ell+1}$.

Suppose the last product in the optimal grouping is

$$(A_\ell \cdots A_i)(A_{i+1} \cdots A_r).$$

The product $A_\ell \cdots A_i$ is an $r_\ell \times c_i$ matrix, while $A_{i+1} \cdots A_r$ is a $c_i \times c_r$ matrix. This means the total number of multiplications required is to compute the last product in the optimal grouping is

$$m_{\ell,i} + m_{i+1,r} + r_\ell c_i c_r,$$

where we use the dynamic programming technique of solving the subproblems optimally.

If $M_{\ell,r}$ is the optimal number of multiplications needed to multiply

$$A_\ell A_{\ell+1} \cdots A_{r-1} A_r,$$

then

$$M_{\ell,r} = \min_{\ell \le i < r} [M_{\ell,i} + M_{i+1,r} + c_\ell c_i c_r].$$

We can use this recursion to compute $M_{1,N}$, which is the optimal number of multiplications for the entire product.

12.2.2 Illustration

Let's apply the dynamic programming solution to the problem in the beginning of this section. We have

$$\begin{aligned} r_1 = 50, c_1 = 10; \\ r_2 = 10; c_2 = 40; \\ r_3 = 40; c_3 = 30; r_4 = 30; c_4 = 5. \end{aligned}$$

We wish to compute

$$M_{1,4} = \min_{1 \le i < 4} [M_{1,i} + M_{i+1,4} + r_1 c_i c_4].$$

This requires $M_{1,1}$ and $M_{2,4}$; $M_{1,2}$ and $M_{3,4}$; and $M_{1,3}$ and $M_{4,4}$. We have

$$\begin{aligned} M_{1,1} &= 0 \\ M_{4,4} &= 0 \end{aligned}$$

and

$$\begin{aligned} M_{1,2} &= 50 \times 10 \times 40 = 20000 \\ M_{3,4} &= 40 \times 30 \times 5 = 6000. \end{aligned}$$

Finally,

$$\begin{aligned} M_{2,4} &= \min_{2 \le i < 4} [M_{2,i} + M_{i+1,4} + r_2 c_i c_4] \\ &= \min \begin{cases} M_{2,2} + M_{3,4} + r_2 c_2 c_4 = 6000 + (10 \times 40 \times 5) = 8000, \\ M_{2,3} + M_{4,4} + r_2 c_3 c_4 = (10 \times 40 \times 30) + (10 \times 30 \times 5) = 12000 + 1500 = 13500, \end{cases} \end{aligned}$$

so

$$M_{2,4} = 8000.$$

Meanwhile,

$$\begin{aligned} M_{1,3} &= \min_{1 \le i < 3} [M_{1,i} + M_{i+1,3} + r_1 c_i c_3] \\ &= \min \begin{cases} M_{1,1} + M_{2,3} + r_1 c_1 c_3 = (10 \times 40 \times 30) + (50 \times 50 \times 30) = 12000 + 75000 = 87000 \\ M_{1,2} + M_{3,3} + r_1 c_2 c_3 = 20000 + (50 \times 40 \times 30) = 20000 + 60000 = 80000, \end{cases} \end{aligned}$$

so

$$M_{1,3} = 80000.$$

Finally,

$$M_{1,4} = \min_{1 \le i < 4} [M_{1,i} + M_{i+1,4} + r_1 c_i c_4]$$
$$= \min \begin{cases} M_{1,1} + M_{2,4} + r_1 c_1 c_4 = 8000 + (50 \times 10 \times 5) = 8000 + 2500 = 10500 \\ M_{1,2} + M_{3,4} + r_1 c_2 c_4 = 20000 + 6000 + (50 \times 40 \times 5) = 26000 + 10000 = 36000 \\ M_{1,3} + M_{4,4} + r_1 c_3 c_4 = 80000 + (50 \times 30 \times 5) = 80000 + 7500 = 87500 \end{cases} = 10500.$$

Note that the optimal answer is one of the groupings we considered earlier.

This is an example of top-down dynamic programming. Here we work backwards from the solution we seek, solving only the smaller instances of the problem that we need.

12.3 Sequence alignment

A basic task in biomolecular sequence analysis is that of determining whether two sequences of amino acid or nucleic acid residues are related. A standard approach to this question is to align the sequences and then decide whether the alignment is likely to have occurred because the sequences are related, or whether the alignment is likely due to chance. The Needleman–Wunsch algorithm [29] is a dynamic programming approach to pairwise sequence alignment.

Here is a chart of the amino acids, with their abbreviations.

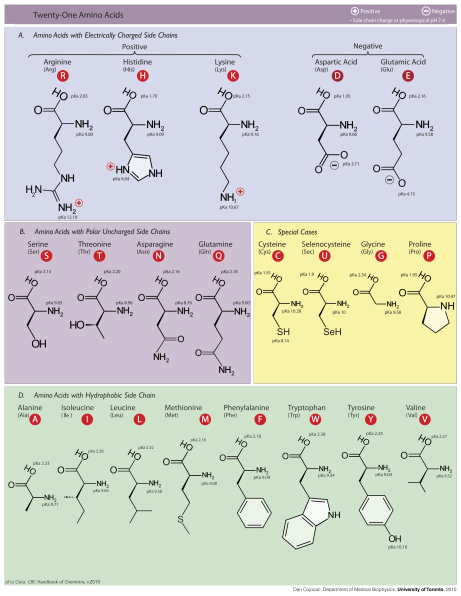


Figure 12.3.1: The amino acids

Consider the alignment of the top and bottom sequences in the following diagram:

```
GSAQVKGHGKKVADA
G+ +VK+HGKKV  A
GNPKVKAHGKKVLGA
```

	A	N	D	E	S
A	4	-2	-2	-1	1
N	-2	6	1	0	1
D	-2	1	6	2	0
E	-1	0	2	5	0
S	1	1	0	0	4
-	-4	-4	-4	-4	-4

Table 12.1: A subset of the BLOSUM62 substitution matrix.

In the line in the middle we indicate matching amino acids with letters, and similar amino acids with +. For instance, serine (S) and asparagine (N) are both in the same group in Figure 12.3.1.

Basic mutational processes lead to

- substitutions, which change residues, and
- gaps, which correspond to insertions and deletions of residues.

The quality of the alignment is computed by giving a positive score to matching terms and a negative score for gaps.

Suppose we wish to align the sequences

$$\begin{aligned} x &= x_1\ x_2\ \cdots\ x_m \\ y &= y_1\ y_2\ \cdots\ y_n \end{aligned}$$

Let $s(x_i, y_j)$ be the score for aligning the terms x_i and y_j, and let g be the penalty for aligning a symbol against a gap. We do not allow gaps to match gaps as this is meaningless. Our goal will be to align the sequences to maximize the sum of the scores of aligned symbols minus the gap penalties.

In practice the values $s(x_i, y_j)$ come from evolutionary substitution matrices. These are based on the probablities of a character in a protein or nucleotide chain transforming to another character over time. For instance, Table 12.1 shows a subset of the BLOSUM62 substitution matrix. The - denotes a negative values correspond to gaps.

Let $F(i, j)$ be the score of the best alignment between the subsequences $x_1\ x_2\ \cdots\ x_i$ and $y_1\ y_2\ \cdots\ y_j$. The solution to our original problem is $F(m, n)$.

We once again turn to a dynamic programming recursion. We compute $F(m, n)$ by constructing the matrix $F(i, j)$ recursively. We set

$$F(0, 0) = 0.$$

In addition, the value $F(i, 0)$, $i = 1, \ldots, m$ correspond to alignments of x to all gaps in y, so $F(i, 0) = -id$. Similarly, $F(0, j) = -jd$.

We claim that for $i, j > 0$, if we know $F(i-1, j-1)$, $F(i-1, j)$, and $F(i, j-1)$ we can compute $F(i, j)$. Why is this? When we process x_i and y_j, one of the following must hold:

1. x_i and y_j are aligned: $F(i, j) = F(i-1, j-1) + s(x_i, y_j)$,

I	G	A	x_i
L	G	V	y_j

2. or x_i is aligned with a gap: $F(i, j) = F(i-1, j) - g$,

A	I	G	A	x_i
G	V	y_j	–	–

3. or y_j is aligned with a gap: $F(i, j) = F(i, j-1) - g$.

$$\begin{array}{cccccc} \mathrm{G} & \mathrm{A} & x_i & - & - \\ \mathrm{S} & \mathrm{L} & \mathrm{G} & \mathrm{V} & y_j \end{array}$$

It follows that

$$F(i,j) = \max \begin{cases} F(i-1,j-1) + s(x_i, y_j), \\ F(i-1,j) - g, \\ F(i,j-1) - g. \end{cases}, \tag{12.3.1}$$

which proves the claim. This means we can bootstrap our way to the solution.

Let's look at an example, using the values in Table 12.1 with a gap penalty $g = 10$.

	S	E	N	D
	$F(1,1)$	$F(1,2)$	$F(1,3)$	$F(1,4)$
A	$F(2,1)$	$F(2,2)$	$F(2,3)$	$F(2,4)$
N	$F(3,1)$	$F(3,2)$	$F(3,3)$	$F(3,4)$
D	$F(4,1)$	$F(4,2)$	$F(4,3)$	$F(4,4)$

		S	E	N	D
	0	-10	-20	-30	-40
A	-10				
N	-20				
D	-30				

Then

$$F(2,2) = \max \begin{cases} F(1,1) + s(x_1, y_2), \\ F(1,2) - g, \\ F(2,1) - g \end{cases} = \max \begin{cases} 0 + 1, \\ -10 - 10, \\ -10 - 10 \end{cases}$$

so $F(2,2) = 1$, which comes from its northwest diagonal neighbor. We note both pieces of information in the following tables.

		S	E	N	D
	0	-10	-20	-30	-40
A	-10	1			
N	-20				
D	-30				

		S	E	N	D
	done	left	left	left	left
A	up	diag			
N	up				
D	up				

Now compute $F(2,3)$:

		S	E	N	D
	0	-10	-20	-30	-40
A	-10	1	-9		
N	-20				
D	-30				

		S	E	N	D
	done	left	left	left	left
A	up	diag	left		
N	up				
D	up				

The final score and traceback tables are

		S	E	N	D
	0	-10	-20	-30	-40
A	-10	1	-9	-19	-29
N	-20	-9	-1	-3	-13
D	-30	-19	-11	2	3

		S	E	N	D
	done	left	left	left	left
A	up	diag	left	left	left
N	up	diag	diag	diag	left
D	up	up	diag	diag	diag

We determine the alignment by tracing the route back from the solution in the lower right-hand corner:

		S	E	N	D
	done	left	left	left	left
A	up	diag	left	left	left
N	up	diag	diag	diag	left
D	up	up	diag	diag	diag

The letters are aligned in reverse order, starting at the end. In light of (12.3.1), the move are interpreted as follows:

- **diag**: the letters from two sequences are aligned;
- **left**: a gap is introduced in the bottom sequence;
- **up**: a gap is introduced in the top sequence.

The first cell in the traceback is a "diag", meaning the corresponding letters are aligned:

```
D
D
```

The second cell in the traceback is also a "diag", meaning the corresponding letters are aligned:

```
ND
ND
```

The third cell is "left", meaning a gap in the bottom sequence:

```
END
-ND
```

The fourth cell is another "diag", meaning the corresponding letters are aligned:

```
SEND
A-ND
```

12.4 Exercises

Exercise 12.4.1. *Use bottom-up dynamic programming to solve the following instance of the knapsack problem. The weight limit is* $W = 10$*, and the items we can pack are as follows:*

item type	*weight*	*value*
1	*6*	*30*
2	*3*	*14*
3	*4*	*16*
4	*2*	*9*

Show the details of your work.

Exercise 12.4.2. *Use dynamic programming to find the minimal number of multiplications needed to compute the product ABCDE and the optimal grouping, where*

- *A is* 10×1*;*
- *B is* 1×5*;*
- *C is* 5×1*;*
- *D is* 1×100*;*
- *E is* 100×1

Show the details of your work.

Chapter 13

Undirected graphs

This chapter begins our investigation of graph algorithms. Much of the first part of the chapter lays out definitions and terminology. Later we look at two important algorithms, breadth-first search and depth-first search.

13.1 Definitions galore

An undirected graph $G = (V, E)$ is a finite set V of vertices together with a set E of edges. An edge is a pair (v, w), where v and w are vertices. This definition allows

- self-loops, edges that connect vertices to themselves, and
- parallel edges, multiple edges that connect the same pair of vertices.

An undirected graph without self-loops is a simple graph, while a graph with parallel edges is sometimes called a multigraph. Our undirected graphs will mostly be simple graphs without parallel edges.

Two vertices are adjacent if there is an edge between them, and the edge is said to be incident to the two vertices. If there are no parallel edges, the degree of a vertex is the number of edges incident to it, where self-loops add only 1 to the degree.

A path in an undirected graph is a sequence of vertices connected by edges. A simple path is a path with no repeated vertices, except possibly the first and last. The length of a path is the number of edges in the path.

A cycle is a path of at least one edge whose first and last vertices are the same. A simple cycle is a cycle with no repeated edges of vertices other than the first and last. A graph without cycles is called acyclic.

A graph is connected if every vertex is connected to every other vertex by a path through the graph. A connected component G' of a graph G is a maximal connected subgraph of G: if $G' \subseteq F$ and F is a connected subgraph of G, then $F = G'$. A graph that is not connected consists of a set of connected components.

A subgraph of a graph G is a subset of G's edges together with the incident vertices. A tree is a connected, acyclic undirected graph. A forest is a disjoint set of trees. A spanning tree of a connected graph is a subgraph that is a tree and also contains all of the graph's vertices. A spanning forest of a graph is the union of spanning trees of its connected components.

If $|V|$ is the number of vertices and $|E|$ the number of edges, then, in a graph without self-loops and parallel edges, there are $|V|(|V|-1)/2$ possible edges. A graph is complete if there is an edge between every pair of vertices.

The density of a graph refers to the proportion of possible pairs of vertices that are connected. A sparse graph is one for which $|E| \ll |V|(|V|-1)/2$. Most large graphs encountered in practice are sparse. A dense graph is a graph that is not sparse. Sparcity and density are subjective.

A bipartite graph is one whose vertices can be divided into two sets so that every vertex in one set is connected to at least one vertex in the other set.

A weighted undirected graph is an undirected graph with weights or costs associated with each edge. A road map with mileage is the prototypical example of a weighted graph, assuming the roads are all two-way. For a weighted undirected graph the length of a path is the sum of the edge weights in the path.

13.2 Representations of graphs

When thinking of how to represent a graph there are two concerns: memory and speed. With regard to speed, we are thinking of the basic operations on graphs:

- Return the number of edges and vertices.
- `add_edge(v, w)`: Add an edge connecting v and w.
- `adj(v)`: Find the vertices adjacent to a v.
- `edges(v)`: Find the edges incident to v.

Here are some possible graph representations:

- A list of edges: we use a list of length $|E|$, each of whose entries are the vertices the edge connects. However, `adj(v)` is slow, since we have to look at all edges.
- A dense adjacency matrix: use a $|V| \times |V|$ array A whose (i, j) entry is 1 if vertices i and j are adjacent, and 0 otherwise. If there are n vertices, a dense adjacency matrix has $n \times n$ entries. This is not practical if we have a large number of vertices.
- Adjacency lists or a sparse adjacency matrix: use a vertex-indexed array of lists (or vectors) of the vertices adjacent to each vertex. This squeezes out the zeros in the dense adjacency matrix, and only stores the nonzeros in each row. In C++ we can implement adjacency lists as a vector of vectors of index labels.

We summarize the time and space complexities in the following table.

		cost		
data structure	space	add $(v - w)$	check for $(v - w)$	get adj. vertices
array of edges	$\|E\|$	$O(1)$	$\|E\|$	$\|E\|$
full adjacency matrix	$\|V\|^2$	$O(1)$	$O(1)$	$\|V\|$
adjacency lists	$\|E\| + \|V\|$	$O(1)$	degree(v)	degree(v)

13.3 Breadth-first search (BFS)

The single-source shortest path problem for unweighted, undirected graphs is:

> Given an unweighted, undirected graph and a source vertex s, what is a shortest path (if any) from s to a given vertex v?

By "shortest path" we mean a path containing the smallest possible number of edges. Note that we say "a shortest path" rather than "the shortest path" as it might not be unique.

We can use breadth-first search to find shortest paths between a specified source s and other vertices, or prove none exists.

```
boolean marked[];
vertex previous[];
bfs(Graph G, vertex s) {
  visited[s] = true; // Mark v as visited.
  enqueue s;
  while (queue is not empty) {
    v = the next vertex on the queue;
    for (w adjacent to v) {
      if (!visited[w]) {
        previous[w] = v;
        visited[w] = true;
```

```
12          enqueue w;
13        }
14      }
15    }
16  }
```

The use of a queue ensures that we process all of the nodes that are distance d from s before we proceed to the nodes that are distance $d + 1$ from s.

The trick to storing paths is to store the path from the destination vertex v back to s, rather than from the starting vertex s foward to v. This we do with the array `previous`; `previous[w]` is the vertex that precedes `w` along the shortest path to `w`. We can recover the path from the array `visited` as follows:

```
stack path;
w = v;
push w onto the stack;
while (previous[w] != s) {
  w = previous[w];
  push w onto the stack;
}
```

When we end, we can pop the stack to get the vertices on a shortest path from s to v

Proposition 13.3.1. *For any vertex v reachable from s, BFS computes a shorest path from s to v.*

PROOF At any moment, the queue consists of zero or more vertices of distance d from s, followed by zero or more vertices of distance $d+1$, starting with $d = 0$. This means vertices enter and exit the queue in order of distance from s. When vertex v enters the queue, no shorter path to v will be found before it exists the queue, and no shorter path to v can be found after it exits the queue. ∎

Proposition 13.3.2. *BFS takes time proportional to $|V| + |E|$ in the worst case.*

PROOF BFS marks all vertices connected to s in time proportional to the sum of their degrees. ∎

13.4 Depth-first search (DFS)

Depth-first search (DFS) plumbs a graph recursively, descending as far as it can before returning and then recursively visiting other nodes. DFS would be useful when traversing a labyrinth.

```
boolean visited[];
dfs(Graph G, vertex v) {
  visited[v] = true; // Mark v.
  for (w adjacent to v) { // Visit neighbors.
    if (!visited[w]) {
      dfs(G, w);
    }
  }
}
```

If we call `dfs(G,s)`, upon exit the array `marked[]` will contain the vertices we can reach from the source vertex `s`.

Proposition 13.4.1. *DFS marks all the vertices connected to a given source s in time proportional to the sum of their degrees.*

PROOF First we prove correctedness of DFS: DFS marks all vertices connected to s, and only vertices connected to s:

- Since DFS visits vertices by following paths, every marked vertex is connected to s.

- If w is connected to s but unmarked, any path from s to w must contain an edge from the set of marked vertices to the set of unmarked vertices (since s is marked). Call this edge (v, x)). DFS would have found x after marking v, so no such edge can exist.

The time bound follows since marking ensures each marked vertex is visited only once, and visiting a node results in work $\propto$ the number of incident edges. ∎

With a slight tweak to DFS, we can store the paths.

```
boolean marked[];
private edge_to[];
dfs(Graph G, vertex v) {
  marked[v] = true; // Mark v.
  for (w adjacent to v) { // Visit neighbors.
    if (!marked[w]) {
      edge_to[w] = v;
      dfs(G, w);
    }
  }
}
```

If we call `dfs(G,s)`, we can find a path from v to s by tracing through `edge_to[]`.

If we save the vertices passed recursively to DFS in some data structure, then when we are done we can iterate over the data structure and look at the saved vertices. Their order is determined by the data structure and when we save the vertices:

1. preorder: put vertex v on a queue before calling dfs(G,v);
2. postorder: put v on a queue after dfs(G,v) returns;
3. reverse postorder: put v on a stack after dfs(G,v) returns.

13.5 Exercises

Exercise 13.5.1. *What is the maximum number of edges in an undirected graph with V vertices if there are no parallel edges? What is the minimum number of edges if the graph is connected?*

Chapter 14

Minimum spanning trees

A minimum spanning tree (MST) or minimum-weight spanning tree of a weighted graph is a spanning tree whose weight (the sum of the weights of the edges in the tree) is the smallest among all spanning trees. MSTs originally arose in the design of minimal cost electrical networks.

We make the following assumptions:

1. The graph is weighted and undirected.
2. The graph is connected.
3. The edge weights are not necessarily Euclidean distances.
4. The edge weights need not be all the same.
5. The edge weights may be zero or negative.

The two algorithms we will study for finding MSTs are Kruskal's algorithm [26] and Prim's algorithm [32]. As Prim notes in his paper, he actually rediscovered the algorithm, as it was first published in 1930 [22]. So far as known, it was originally formulated in 1926 by Otakar Boruvka of the Electrical Power Company of Western Moravia in Czechoslovakia.

14.1 Key idea

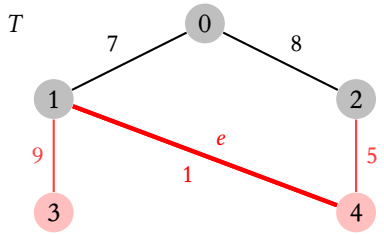
A cut in a graph is a partition of the vertices into two nonempty, disjoint sets. A crossing edge of a cut is an edge that connects a vertex in one partition to a vertex in the other.

14.1.1 The cut property

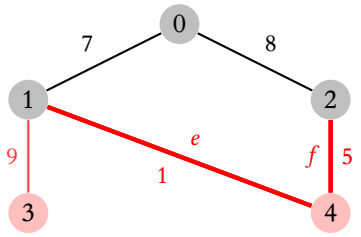
Our approach to finding MSTs is rooted in the following result.

Proposition 14.1.1 (The cut property). *Suppose the weights are all distinct. Then, given any cut, the crossing edge of minimum weight is in the MST of the graph.*

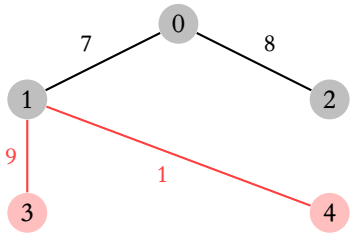
Proof Let e be the crossing edge of minimum weight and let T be the MST. We use proof by contradiction: suppose $e \notin T$. In the following figure, the MST consists of the grey nodes and black edges.



Add e to T. The resulting graph has a cycle, since T was a spanning tree, and that cycle must contain e since otherwise T would already have had a cycle. This cycle must include at least one other crossing edge, since the cut partitions are connected by edges in T. Let f be such an edge as depicted below.



Since the weights are all different and the weight of e is minimal, we know f has a higher weight than e. But if we delete f from T and replace it with e, we obtain a spanning tree of strictly lower weight:



But this contradicts the assumption that T is already minimal. ∎

14.1.2 A greedy approach

Greedy algorithms make locally optimal choices to try to find globally optimal solutions. Here is how a greedy MST algorithm might work:

1. Color all edges black.
2. Find a cut with no red crossing edges and color the minimum weight crossing edge red.
3. Continue until $|V| - 1$ edges have been colored red.

The red edges then form an MST. The greediness in this algorithm is in adding currently minimum weight black edges. This algorithm also works if edge weights are not all distinct.

Why does the greedy approach work? By the cut property, any crossing edge that is colored red is in the MST. If fewer than $|V| - 1$ edges are red, a cut with no red crossing edges exists, so we can continue. Once $|V| - 1$ edges are red, the red edges form an MST.

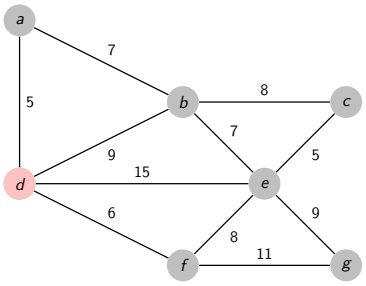
Prim's algorithm and Kruskal's algorithm are both greedy algorithms that differ in how they choose the cuts.

14.2 Prim's algorithm

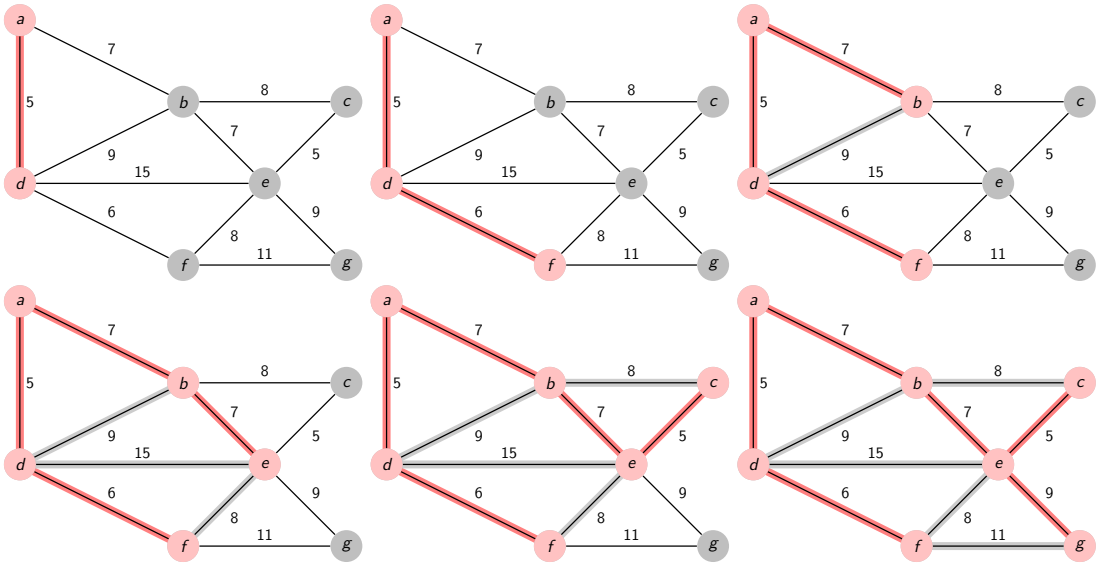
Prim's algorithm is as follows:

1. Initialization: $S \leftarrow \{s\}$ and $D \leftarrow V - \{s\}$.
2. While there remains a $v \in D$:
   a) Find an edge with minimum weight (u, v) such that $u \in S$ and $v \in D$.
   b) $S \leftarrow S \cup \{v\}$ and $D \leftarrow D - \{v\}$.

Let's look at Prim's algorithm applied to the following graph:



Our starting vertex is d. The red edges are part of the MST, while grey edges are edges that go between vertices already in the MST and are ignored.



Let's look at a naive implementation of Prim's algorithm:

1. Initialization: $S \leftarrow \{s\}$ and $D \leftarrow V - \{s\}$.
2. While there remains a $v \in D$:
   a) Find an edge with minimum weight (u, v) such that $u \in S$ and $v \in D$.
   b) $S \leftarrow S \cup \{v\}$ and $D \leftarrow D - \{v\}$.

In step 1, suppose we look for the minimum weight edge by looking at all edges from S to D. This yields a complexity $\propto |V||E|$, since as we add each vertex to the MST we look at all the edges. In the worst case, $|E| \propto |V|^2$, leading to a complexity proportional to $|V|^3$!

In order to obtain an efficient implementation, we use priority queues. Recall that a priority queue is a type of queue that allows for returning and deleting the minimum (or maximum) value. In a priority queue of size N, the insertion and removal of the maximum (minimum) can both be done in $O(\lg N)$ work. Moreover, priority queues are easy to implement using binary heaps.

Using priority queues the space requirements are $3|V|$:

1. Two vertex-indexed boolean arrays edge_to[] and weight[]. If v is not in the MST but has an edge connecting it to the MST,

- edge_to[v] is the least weight edge connecting v to the tree, and
- weight[v] is the cost of that edge.

2. All such vertices v are kept on a priority queue, as an index associated with the edge_to[v] and weight[v].

We have the following invariant: At each step, the minimum key in the priority queue is the weight of the minimum weight crossing edge from the (growing) MST to its complement, and the associated vertex v is the next added to the tree.

Proposition 14.2.1. *Prim's algorithm uses space $\propto |V|$ and time $\propto |E| \lg |V|$ (worst case) to compute the MST of a connected weighted graph.*

Proof The number of edges in the priority queue as at most $|V|$. The algorithm uses

1. $|V|$ insertions into the queue;
2. $|V|$ delete the minimum operations;
3. and $|E|$ change priority operations (worst case).

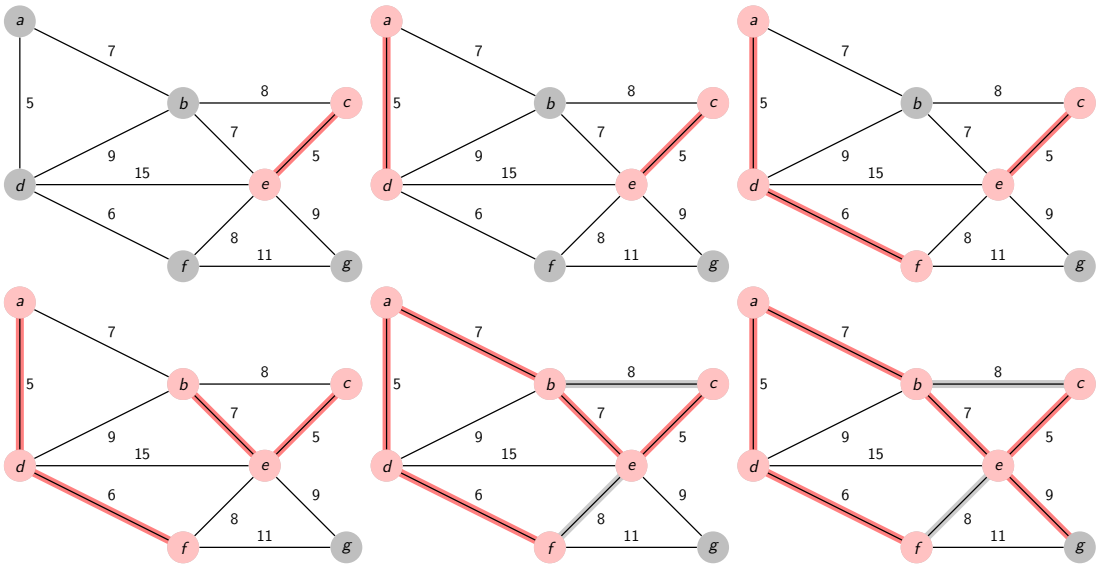
Since the queue has at most size $|V|$, each operation requires time that is $O(\lg |V|)$. ∎

14.3 Kruskal's algorithm

Kruskal's algorithm is

1. Initialization: $T \leftarrow \emptyset$.
2. While $|T| < |V| - 1$:
   a) Find an edge e in E with minimum weight such that adding e to T will not create a cycle.
   b) $T \leftarrow T \cup \{e\}$ and $E \leftarrow E - \{e\}$.

Let's apply Kruskal's algorithm to the example of the previous section. The red edges are part of the MST, while grey edges are edges that would create cycles and are ignored.



14.3.1 Kruskal's algorithm: complexity

Kruskal's algorithm requires us to test whether adding an edge to our growing MST would produce a cycle in the MST. We can perform such a test efficiently using union-find:

- Initially, each vertex is in its own component.
- When we add an edge to the MST, we union its endpoints.
- Adding the edge to the MSTR will produce a cycle if and only if the endpoints are in the same component, so we need only perform a find operation to test the edge.

Recall that with path compression union-find will become increasingly efficient as we add vertices, so the amortized cost will be roughly linear in the number of vertices.

In terms of the data structures in Kruskal's algorithm,

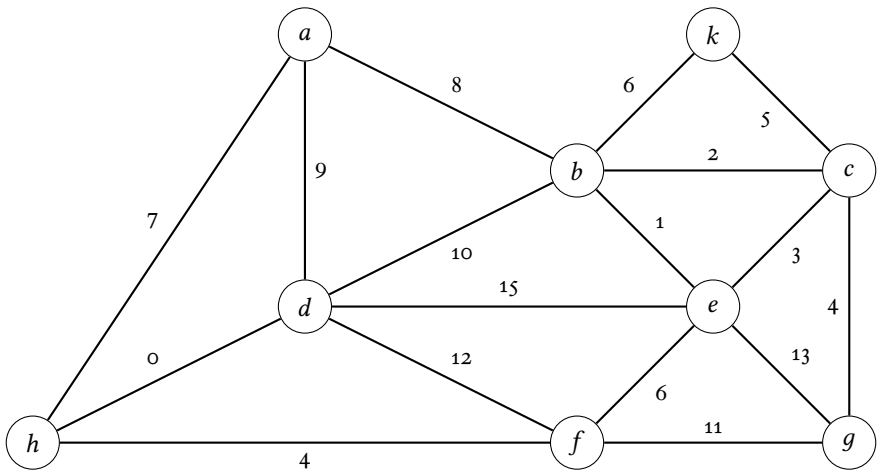
1. Use a priority queue to consider edges in order of weight.
   a) The priority queue has size $|E|$ and requires work $\propto |E|$ to build.
   b) Operations on the queue require $\lg|E|$ time.
2. Use union-find to check for cycles. Union-find requires $|E|$ space and each check requires time $\propto |E|$ at most.
3. A queue to collect the MST edges.

Proposition 14.3.1. *Kruskal's algorithm finds an MST using space $\propto |E|$ and time $\propto |E| \lg|E|$ (in the worst case).*

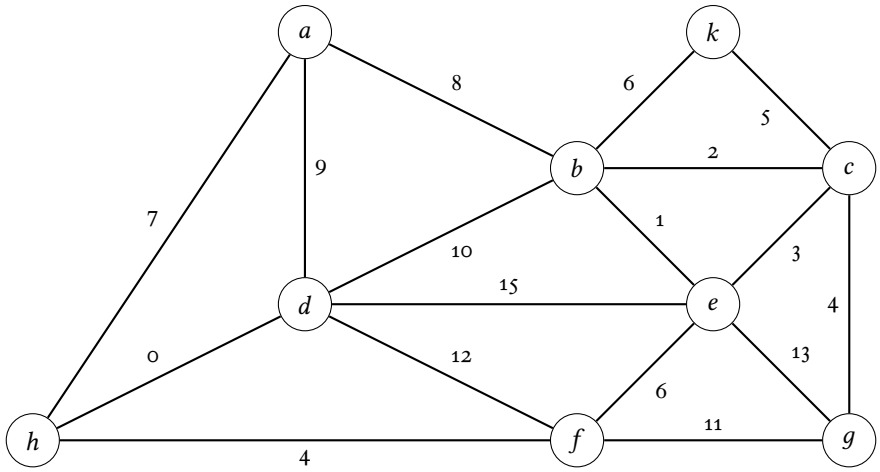
PROOF In the worst case, we check all the edges and have to run union-find $|E|$ times. This yields an upper bound of $|E| \lg|E|$ on work. ∎

14.4 Exercises

Exercise 14.4.1. *Apply Prim's algorithm starting with node a to find a minimum weight spanning tree for the graph below. Indicate the sequence in which edges are added; i.e., if an edge from b to a is added at step 3, then enter (a, b) beside 3 below. To make grading easier, edges should be specified so that the endpoints appear in alphabetical order (i.e., write (a, b) rather than (b, a).*



Exercise 14.4.2. *Apply Kruskal's algorithm to find a minimum weight spanning tree for the graph below. Indicate the sequence in which edges are added; i.e., if an edge from b to a is added at step 3, then enter (a, b) beside 3 below. To make grading easier, edges should be specified so that the endpoints appear in alphabetical order (i.e., write (a, b) rather than (b, a).*



Exercise 14.4.3. *Show that if the edge weights in a weighted graph G are all distinct then the minimum weight spanning tree is unique.*

Exercise 14.4.4. *Prove or give a counterexample: If a graph has a unique minimum weight spanning tree then the edge weights are distinct.*

Exercise 14.4.5. *Let $G = (V, E)$ be a connected, undirected graph and let $S = (V, T)$ be a spanning tree of G.*

1. *Show that for all $u, v \in V$, the path between u and v in S is unique.*
2. *Show that if any edge in $E - T$ is added to S, a unique cycle results.*

Exercise 14.4.6. *Let $G = (V, E)$ be a connected, undirected, weighted graph. For any edge $e \in E$ let $w(e)$ denote the weight of e. Let $(V_1, T_1), \ldots, (V_k, T_k)$ be any spanning forest of G with $k > 1$. By a spanning forest we mean a collection of trees (V_i, T_i) with*

$$V_1 \cup V_2 \cup \cdots \cup V_k = V,$$
$$V_i \cap V_j = \emptyset \quad \text{if } i \neq j.$$

Let $T = \cup_{k=1}^{k} T_i$. Suppose $e = (v, w)$ is an edge of lower cost in $E - T$ such that $v \in V_1$ and $w \notin V_1$.

Show that there is a spanning tree which includes $T \cup \{e\}$ and has a total weight as low as any spanning tree of G that includes T.

Exercise 14.4.7. *Show that Prim's algorithm still works if the edge weights are not distinct.*

Exercise 14.4.8. *Show that Kruskal's algorithm still works if the edge weights are not distinct.*

Exercise 14.4.9. *How could you find a spanning tree of maximum weight for a weighted graph?*

Exercise 14.4.10. *Suppose the edge weights in a weighted graph G are all distinct. Show that for any cycle in G the edge of maximum weight in the cycle is not in the MST of the graph.*

Exercise 14.4.11. *Suppose the edge weights in a weighted graph G are all distinct.*

1. *Does the edge of minimum weight in G belong to the MST of G?*
2. *Can the edge with maximum weight in G belong to the MST of G?*
3. *Does the minimum weight edge of every cycle belong to the MST of G?*

For each statement prove that the statement is true or give a counterexample.

Chapter 15

Directed graphs

15.1 Directed graphs

In a directed graph or digraph the pairs (v, w) indicating edges are ordered. The edge (v, w) goes from v (the tail) to w (the head). We will refer to edges in digraphs as arcs.

The in degree of a vertex w is the number the number of arcs coming into w (i.e., the number of arcs for which w is the head). The out degree of v is the number of arcs exiting v (i.e., the number of arcs for which v is the tail). We will call w a source if its indegree is 0. A directed acyclic graph (DAG) is a digraph with no directed cycles.

Proposition 15.1.1. *A DAG always has at least one source vertex.*

In a directed graph, two vertices v and w are strongly connected if there is a directed path from v to w and a directed path from w to v. A digraph is strongly connected if all its vertices are strongly connected to one another. If a digraph is not strongly connected but the underlying undirected graph is connected, then the digraph is called weakly connected.

15.2 Topological sort

A topological sort is an ordering of the vertices in a directed graph such that if there a path from v to w, then v appears before w in the ordering.

Proposition 15.2.1. *A digraph has a topological ordering if and only if it is a DAG.*

Proof The necessity is clear since a digraph with a directed cycle cannot have a topological ordering. The sufficiency will be proved by presenting an algorithm that finds a topological ordering.

Here is one such algorithm:

1. Determine the indegree for every $v \in V$. Place all source vertices in a queue.
2. While there remains a $v \in V$:
    a) find a source vertex;
    b) append the source vertex to the topological sort;
    c) delete the source and its adjacent arcs from G;
    d) update the indegrees of the remaining vertices in G;
    e) place any new source vertices in the queue.

■

When no vertices remain, we have our ordering. If we are missing vertices from the output list, the graph has no topological sort.

```
    for each vertex {
      compute indegree;
      enqueue sources;
    }

    while the queue is nonempty {
      dequeue v, the head of the queue;
      foreach vertex w adjacent to v {
        decrease indegree of w by 1;
        enqueue w if it is a source;
      }
    }
```

The complexity of this approach is $O(|E| + |V|)$.

15.3 Depth-first search

Depth-first search in a directed graph is similar to that in undirected graphs:

```
boolean marked[];
dfs(Graph G, vertex v) {
  marked[v] = true; // Mark v.
  for w adjacent to v { // Visit neighbors.
    if (!marked[w]) {
      dfs(G, w);
    }
  }
}
```

If we call `dfs(G,s)`, upon exit the array `marked[]` will contain the vertices we can reach from the source vertex `s`.

Proposition 15.3.1. *DFS marks all the vertices connected to a given source s in time proportional to the sum of their degrees.*

Proof First, we prove correctedness of DFS: DFS marks all vertices connected to s, and only vertices connected to s:

- Since DFS visits vertices by following paths, every marked vertex is connected to s.
- If w is connected to s but unmarked, any path from s to w must contain an edge from the set of marked vertices to the set of unmarked vertices (since s is marked). Call this edge (v, x). DFS would have found x after marking v, so no such edge can exist.

The time bound follows since marking ensures each marked vertex is visited only once, and visiting a node results in work $\propto$ the number of incident edges. ∎

With a slight tweak to DFS, we can store the paths.

```
boolean marked[];
private edge_to[];
dfs(Graph G, vertex v) {
  marked[v] = true; // Mark v.
  for (w adjacent to v) { // Visit neighbors.
    if (!marked[w]) {
      edge_to[w] = v;
      dfs(G, w);
    }
  }
}
```

If we call `dfs(G,s)`, we can find a path from v to s by tracing through `edge_to[]`.

15.3.1 Detecting cycles using DFS

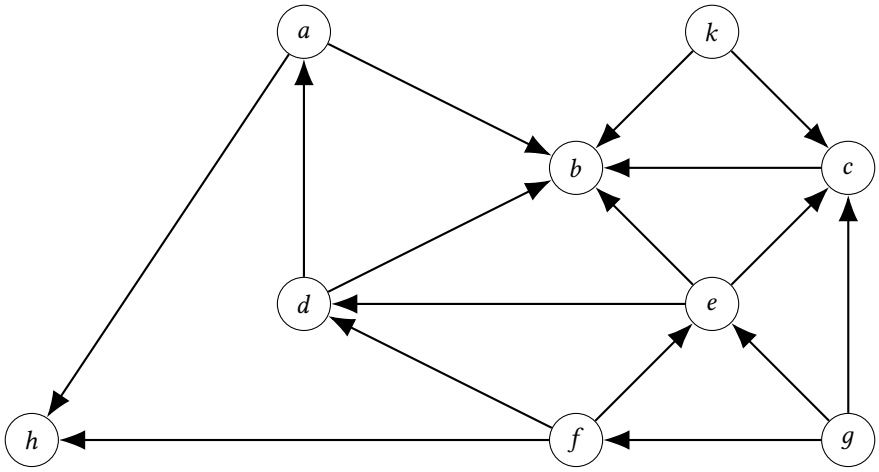
We can also use DFS to detect cycles in a digraph. Since DFS is recursive, at any given moment the list of vertices in the call stack represent a directed path that is being traversed.

If we ever encounter a directed edge (v, w) to a vertex w on the sack, we have found a cycle! Conversely, the absence of any such edges means that graph is acyclic. This approach examines all edges and vertices, so the time is proportional to $|V| + |E|$.

15.4 Exercises

Exercise 15.4.1. *Two graphs G and H are said to be isomorphic if there exists is a bijection between the set of vertices of G and H such that u is adjacent to v if and only if $f(u)$ is adjacent to $f(v)$. Draw all the nonisomorphic DAGs with two, three, four, and five vertices.*

Exercise 15.4.2. *Give a topological sort of the following graph, or explain why one does not exist.*



Exercise 15.4.3. *Show that a finite directed acylic graph is guaranteed to have a node with in-degree* 0.

Exercise 15.4.4. *Show that reverse postorder DFS gives a topological sort of a DAG.*

Exercise 15.4.5. *What is the time complexity of the algorithm in Exercise 15.4.4?*

Bibliography

[1] W. Ackermann, Zum Hilbertschen Aufbau der reellen Zahlen, Mathematische Annalen, 99 (1928), pp. 118–133.

[2] M. Akra and L. Bazzi, On the solution of linear recurrence equations, Computational Optimization and Applications, 10 (1998), pp. 195–210.

[3] R. Bayer and E. M. McCeight, Organization and maintenance of large ordered indexes, Acta Informatica, 1 (1972), pp. 173–189.

[4] R. Bellman, On the theory of dynamic programming, Proceedings of the National Academy of Sciences, 38 (1952), pp. 716–719.

[5] ——, Dynamic programing, Princeton University Press, 1957.

[6] J. L. Bentley, D. Haken, and J. B. Saxe, A general method for solving divide-and-conquer recurrences, SIGACT News, 12 (1980), pp. 36–44.

[7] J. L. Bentley and M. D. McIlroy, Engineering a sort function, Software—Practice and Experience, 23 (1993), pp. 1249–1265.

[8] J. L. Carter and M. N. Wegman, Universal classes of hash functions, Journal of Computer and System Sciences, 18 (1979), pp. 143–154.

[9] D. Comer, The ubiquitous B-tree, ACM Computing Surveys, 11 (1979), pp. 121–137.

[10] M. Dietzfelbinger, T. Hagerup, J. Katajaine, and M. Penttonen, A reliable randomized algorithm for the closest-pair problem, Journal of algorithms, 25 (1997), pp. 19–51.

[11] S. Dreyfus, Richard Bellman on the birth of dynamic programming, Operations Research, 50 (2002), pp. 48–51.

[12] R. W. Floyd, Algorithm 113: Treesort, Communications of the ACM, 5 (1962), p. 434.

[13] S. P. Y. Fung, Is this the simplest (and most surprising) sorting algorithm ever?, October 2021.

[14] L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, in SFCS '78]: Proceedings of the 19th Annual Symposium on Foundations of Computer Science, October 1978, pp. 8–21.

[15] D. Harvey and J. van der Hoeven, Integer multiplication in time $O(n \log n)$, Annals of Mathematics, 193 (2021), pp. 563–617.

[16] T. Hastie, R. Tibshirani, and J. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Springer, 2nd ed., 2009.

[17] T. N. Hibbard, Some combinatorial properties of certain trees with applications to searching and sorting, Journal of the ACM, 9 (1962), pp. 13–28.

[18] C. A. R. Hoare, Quicksort: Algorithm 64, Communications of the ACM, 4 (1961), p. 321.

[19] ——, Quicksort, The Computer Journal, 5 (1962), pp. 10–16.

[20] J. E. Hopcroft and J. D. Ullman, Set merging algorithms, SIAM Journal on Computing, 2 (1973), pp. 294–303.

[21] D. A. Huffman, A method for the construction of minimum-redundancy codes, Proceedings of the IRE, 40 (1952), pp. 1098–1011.

[22] V. Jarník, O jistém problému minimálním, Práce Moravské Přírodovědecké Společnosti, 6 (1930), pp. 57–63.

[23] A. Karatsuba and Y. Ofman, Multiplication of many-digital numbers by automatic computers, Proceedings of the USSR Academy of Sciences, 145 (1962), pp. 293–294. English translation in *Soviet Physics-Doklady*, 7 (1963), pp. 595–596.

[24] B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice–Hall, second ed., 1988.

[25] D. E. Knuth, The Art of Computer Programming, vol. 3, Sorting and Searching, Addison-Wesley, 2nd ed., 1998.

[26] J. B. Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem, Proceedings of the American Mathematical Society, 7 (1956), pp. 48–50.

[27] T. Leighton, Notes on better master theorems for divide-and-conquer recurrences. (class notes), October 1996.

[28] D. R. Musser, Introspective sorting and selection algorithms, Software: Practice and Experience, 27 (1997), pp. 983–993.

[29] S. B. Needleman and C. D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, Journal of Molecular Biology, 48 (1970), pp. 443–453.

[30] R. Péter, Konstruktion nichtrekursiver Funktionen, Mathematische Annalen, 111 (1935), pp. 42–60.

[31] V. R. Pratt, Shellsort and sorting networks, PhD thesis, Stanford University, 1971.

[32] R. C. Prim, Shortest connection networks and some generalizations, Bell System Technical Journal, 36 (1957), pp. 1389–1401.

[33] R. M. Robinson, Recursion and double recursion, Bulletin of the American Mathematical Society, 54 (1948), pp. 987–993.

[34] D. Romik, Stirling's approximation for $n!$: the ultimate short proof?, The American Mathematical Monthly, 107 (2000), pp. 556–557.

[35] A. Schönhage and V. Strassen, Schnelle Multiplikation großer Zahlen, Computing, 7 (1971), pp. 281–292.

[36] R. Sedgewick and K. Wayne, Algorithms, Addison-Wesley Professional, 4th ed., 2011.

[37] D. L. Shell, A high-speed sorting procedure, Communications of the ACM, 2 (1959), pp. 30–32.

[38] R. E. Tarjan, Efficiency of a good but not linear set union algorithm, Journal of the ACM, 22 (1975), pp. 215–225.

[39] M. A. Weiss, Data Structures and Algorithm Analysis in C++, Pearson, 4th ed., 2014.

[40] J. W. J. Williams, Algorithm 232: Heapsort, Communications of the ACM, 7 (1964), pp. 347–348.

Nomenclature

lg	the base-2 logarithm
$\lg^*$	iterated logarithm
ln	the natural logarithm
lb	the ISO 80000 abbreviation for the base-2 logarithm
$\propto$	proportional to
$\sim$	tilde notation
😱	anxious reader
	danger sign

Index