Chapter 1
Programming: A General Overview

Introduction

This class is an introduction to the design, implementation, and analysis of algorithms.

- examples:
  - sorting large amounts of data
  - organizing information for efficient <u>search</u> and <u>retrieval</u>
  - finding the <u>shortest path</u> between nodes in a graph
  - looking for <u>matching</u> substrings

Analysis of Algorithms

- to be able to assess the <u>efficiency</u> of an algorithm, we need to be able to characterize (quantify) the amount of time it requires to execute

- standard approach: construct a mathematical model of the amount of work as a function of the <u>size</u> of the problem
  - allows us to study how much <u>work</u> is required by an algorithm as the size of the problem grows
  - gives us a basis for <u>comparing</u> different algorithms to solve the same problem

Run Time Example

suppose

- our computer can perform 10^9 operations per second
- we have two algorithms that perform operation X on a set of n numbers
  - Algorithm A requires $\underline{n^2}$ operations
  - Algorithm B requires $\underline{n \log_2 n}$ operations
- $n = 10^6$
- we need to compute X in less than a second

Run Time Example

Algorithm A requires 10^{12} operations and runs in $10^{12}/10^{9}$ = 10^{3} seconds

- we would need a computer that runs 1000 times faster!
- if processor speeds double roughly every 18 months, the hardware will catch up in

$18 \times \log_2 1000 \approx \underline{179}$ months, or about $\underline{15}$ years

Run Time Example

meanwhile, Algorithm B requires $10^6 \log_2 10^6 \approx \underline{10^6\ }$ or

$(20 \times 10^6)\ /\ 10^9\ =\ 20 \times 10^{-3}$ seconds, or 20 ms

we can get by with a machine that runs at $1/50^{th}$ the speed!

Factors Affecting the Running Time of an Algorithm

Hardware

- what is the clock speed of the chip? what is the memory hierarchy? how many processors?

Compiler

- what code does the compiler generate?

Program

- how efficient is the use of instructions and memory?

Data

- what is the size of the input? how will it affect execution time?

Environment

- what is the operating system? what is the system load? how do they interact with the code?

Example: dot product

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

$$\underline{x_1 y_1 + x_2 y_2 + \ldots + x_n y_n}$$

C code:

```
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;
}
```

Let $T(n)$ be the runtime of the dot product on vectors of length n.

Example: dot product

We cannot give a good prediction of the total runtime of the algorithm by simply looking at the code.

Let’s remove the distractions of

- programming
- code generation
- computing environment
- etc.

We will focus instead on operation counts and generally ignore the speed/efficiency of memory access, though this has a major impact on performance.

Example: dot product

We will frequently use pseudocode to express algorithms in a way that is independent of any specific programming language:

human language + math + programming language constructs

dot product:

```
sum = 0
for i = 1 to n
    sum += x[i] * y[i];
end
```

```
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;
}
```

Pseudocode

pseudocode allows us to ignore the peculiarities of any given programming language and concentrate on the essential operations

it should be complete enough that it could be given to any (competent) programmer for implementation in any reasonable programming language

- pseudocode should specify all initializations and the termination conditions

Example: dot product

Inside the loop there is 1 multiplication and 1 addition; and these loop operations are executed n times:

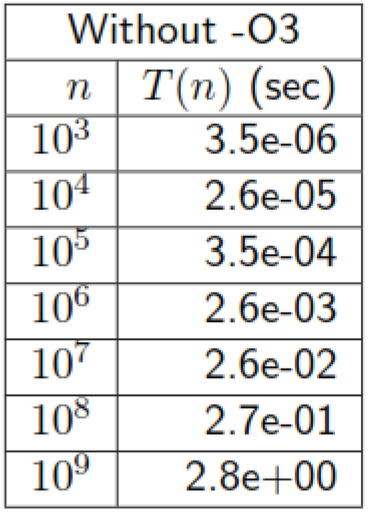
```
sum = 0
for i = 1 to n
   sum += x[i] * y[i];
end
```

It is reasonable to predict that the running time is proportional to n: $T(n) \approx cn$, where c is an (unknown) constant, regardless of the actual instructions being executed!

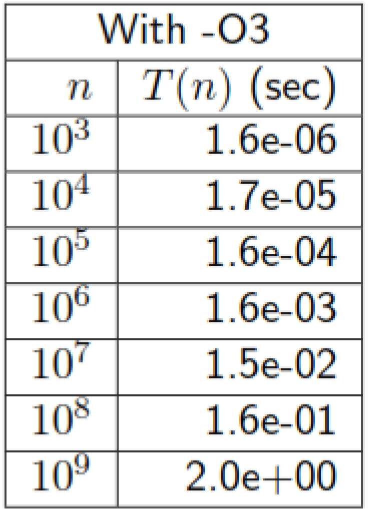
This tells us how the execution time grows: if n doubles, then $T(n)$ should double.

Example: dot product observed execution times

On an Intel E5-4627v2 CPU:



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	
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, $\underline{T(n)}$ increases (roughly) by a factor of 10.

Selection Problem

Selection problem: suppose you have N numbers and would like to determine the k^{th} largest

- how would you try to solve this problem?

Selection Problem

one straightforward solution:

- read the N numbers into an array and sort them in decreasing order
- return the element in position k

better algorithm:

- read the first $\underline{k}$ elements into an array and sort them in decreasing order
- as a new element is considered, <u>ignore</u> it if it is smaller than the k^{th} element in the array
- otherwise, place it in its correct spot, bumping one element out of the array
- when the algorithm ends, the element is in the $\underline{k^{th}}$ position

Selection Problem

for $N = \underline{30{,}000{,}000}$ and $k = \underline{15{,}000{,}000}$:

- both algorithms would take several days to run
- later, we'll discuss an algorithm that gives the solution in about a second!

Just because an algorithm works, therefore, does not necessarily make it a <u>good</u> algorithm.

Mathematics Review: Exponents

$$X^A X^B = X^{A+B}$$

$$\frac{X^A}{X^B} = X^{A-B}$$

$$(X^A)^B = X^{AB}$$

$$X^N + X^N = 2X^N \neq \underline{X^{2N}}$$

$$2^N + 2^N = 2^{N+1}$$

Mathematics Review: Logarithms

All logarithms are base <u>2</u> unless specified otherwise.

$$X^A = B \textit{ if and only if } \log_X B = A$$

$$\log_A B = \frac{\log_C B}{\log_C A} \quad A, B, C > 0,\ A \neq 1$$

$$\log AB = \log A + \log B \quad A, B > 0$$

$$\log \frac{A}{B} = \log A - \log B$$

$$\log(A^B) = B \log A$$

Mathematics Review: Logarithms

$\log X < X \; for \; all \; X > 0$

use lg to represent $\log_2$

$\lg 1 = \underline{0}$

$\lg 2 = \underline{1}$

$\lg 1024 = \underline{10}$

$\lg 1{,}048{,}576 = \underline{20}$

Mathematics Review: Series

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$$

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$$

$$\sum_{i=0}^{N} A^i \leq \frac{1}{1 - A} \quad if\ 0 < A < 1$$

Mathematics Review: Series

$$\sum_{i=0}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

$$\sum_{i=0}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$$

Mathematics Review: Modular Arithmetic

A is congruent to B modulo N

$$A \equiv B(mod\ N)\ \ if\ \ N\ divides\ A - B$$

i.e., the remainder is the same when A or B is divided by N

if $A \equiv B(mod\ N)\ \ then\ A + C \equiv B + C(mod\ N)$

if $A \equiv B(mod\ N)\ \ then\ AD \equiv BD(mod\ N)$

Proofs

most common proofs in algorithm analysis:

- proof by <u>counterexample</u>
- proof by <u>contradiction</u>
- proof by <u>induction</u>

Proof by Counterexample

Prove or disprove: All prime numbers are odd.

Proof: Since 2 is prime and is not odd, then the original premise is false.

Proofs

Proof by Contradiction

Prove or disprove: The sum of two odd numbers is odd.

Proof: Let x and y both be odd numbers

therefore, $\underline{x = 2i + 1}$ and $\underline{y = 2j + 1}$

if $z = x + y$ then

$z = 2i + 1 + 2j + 1 = 2i + 2j + 2 = 2(i + j + 1)$,

which is even

this contradicts the original statement, which is therefore false

Proofs

Proof by Mathematical Induction

- very useful in studying algorithms
- must include three parts:
  - base case (almost always trivial)
    - state the value of *n*
    - compute lhs and rhs independently
  - inductive hypothesis
    - state in terms of *k*
    - assume true for some *k*
  - inductive step (show true for *k*+1)
    - must list explicitly what you're trying to show!
    - must use the <u>inductive hypothesis</u>

Proofs

Inductive Proof exercises

$$\sum_{i=0}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \quad for\ N \geq 1$$

Base: N = 1

lhs/ $\sum_{i=0}^{1} i^2 = 0^2 + 1^2 = 1$ rhs/ $\frac{1(1+1)(2\cdot 1+1)}{6} = \frac{1\cdot 2\cdot 3}{6} = 1$ ✓

I.H.: Assume: $\sum_{i=0}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$ for some k

I.S.: Show: $\sum_{i=0}^{k+1} i^2 = \frac{(k+1)(k+2)(2k+3)}{6}$

$$\sum_{i=0}^{k+1} i^2 = \sum_{i=0}^{k} i^2 + (k+1)^2$$

$$= \frac{k(k+1)(2k+1)}{6} + (k+1)^2$$ by I.H.

$$= \frac{k(k+1)(2k+1) + 6(k+1)^2}{6}$$

$$= \frac{(k+1)(k(2k+1) + 6(k+1))}{6}$$

$$= \frac{(k+1)(2k^2 + k + 6k + 6)}{6}$$

$$= \frac{(k+1)(2k^2 + 7k + 6)}{6}$$

$$= \frac{(k+1)(k+2)(2k+3)}{6}$$ ✓

Proofs

Inductive Proof exercises

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$$

Fibonacci numbers:

$$F_i = F_{i-1} + F_{i-2} \text{ where } F_1 = 1, F_2 = 2$$

Show: $F_i < \left(\frac{5}{3}\right)^i$ for $i \geq 1$

Recursion

We will study both recursive functions and recurrence relations in this course. We often use inductive proofs to show various properties about these recursive problems.

recursive function:

```
int f(int x)
{
   if (x == 0)
     return 0;
   else
      return 2 * f(x-1) + x*x;
}
```

Recursion

```
int f(int x)
{
   if (x == 0)
     return 0;
   else
     return 2 * f(x-1) + x*x;
}
```

Note that the first part of the function establishes the <u>base case</u>, that is, the value for which the function is directly known without resorting to recursion.

This is not circular logic since we use terms that progressively get to the base case.

Recursion

Four rules of recursion:

- Base cases – a base case must always exist, which can be solved without recursion
- Making progress – for cases solved recursively, the recursive call must always be to a case that makes progress toward the base case
- Design rule – assume that all recursive calls work
- Compound interest rule – never duplicate work by solving the same instance of a problem in separate recursive calls

example: looking up words in a dictionary

Recursion

```
void printOut (int n)
{
   if (n >= 10)
     printOut (n / 10);
   printDigit (n % 10);
}
```

Prove the above program works using induction for $n \geq 0$.