Analysis of Algorithms

CSci 653, TTh 9:30-10:50, ISC 0248

Professor Mao, wm@cs.wm.edu, 1-3472, McGl 114

General Information

- Office Hours: TTh 11:00 – 12:00 and W 3:00 – 4:00
- Textbook: Intro to Algorithms (any edition), CLRS, McGraw Hill or MIT Press.
- Prerequisites/background: Linear algebra, Data structures and algorithms, and Discrete math

Use of Blackboard

- Announcements
- Problem sets (aka assignments or homework)
- Lecture notes
- Grades
- Check at least weekly

Lecture Basics

- Lectures: Slides + board (mostly)
- Lecture slides $\subset$ lecture notes posted on BB
- Not all taught in class are in the lecture notes/slides, e.g., some example problems. So take your own notes in class

Course Organization

- Mathematical foundation
- Methods of analyzing algorithms
- Methods of designing algorithms
- Additional topics chosen from the lower bound theory, amortization, probabilistic analysis, competitive analysis, NP-completeness, and approximation algorithms

Grading

- About 12 problem sets: 60%
- Final: 40% (in-class)

Grading Policy (may be curved)

- $[90, 100]$: A or A-
- $[80, 90)$: B+, B, or B-
- $[70, 80)$: C+, C, or C-
- $[60, 70)$: D+, D, or D-
- $[0, 60)$: F

Homework Submission Policy

- Hardcopy (stapled and typeset in LaTex) to be submitted at the beginning of class on the due date
- Extensions may be permitted for illness, family emergency, and travel to interviews/conferences. Requests must be made prior to the due date

Homework Policy

- Homework must be typeset in LaTex. Nothing should be handwritten, including diagrams and tables
- Empty-hand policy when you discuss homework problems with your classmates
- List your collaborators for every homework problem
- Cite all references that help you solve a problem
- In no case you should copy verbatim from other people's work without proper attribution, as this is considered plagiarism

Honor Code

- ”As a member of the William and Mary community, I pledge on my honor not to lie, or steal, either in my academic or personal life. I understand that such acts violates the Honor Code and undermine the community of trust, of which we are all stewards.”
- Academic honesty, the cornerstone of teaching and learning, lays the foundation for lifelong integrity. The Honor Code is, as always, in effect in this course. Please go to the ”Honor System” page in the wm.edu site to read the complete honor code document.

Common functions

- Monotonicity: Definitions of monotonically increasing/decreasing or strictly increasing/decreasing functions.
  Important note: In this course, since functions are used to represent time complexity, we restrict our attention to only increasing functions that map positive number(s) to positive number.
- Ceilings and floors: $\lceil x \rceil$ and $\lfloor x \rfloor$, where x can be any real number.
  $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$, $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$.
- Modular arithmetic: $a \bmod n = a - \lfloor a/n \rfloor n$. $a \equiv b \bmod n$ iff $a \bmod n = b \bmod n$.

- Polynomials: $p(n) = \sum_{i=0}^{d} a_i n^i$. (Note: Coefficients a_i and degree d are constants.)
- Exponentials: $a^0 = 1$, $a^{-1} = \frac{1}{a}$, $a^m \cdot a^n = a^{m+n}$, $a^m / a^n = a^{m-n}$.
  $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}$. (Note: $e = 2.71828\ldots$.)
  $\lim_{n\to\infty} (1 + \frac{x}{n})^n = e^x$.

- Logarithms: $\log n = \log_2 n$ or $\log_c n$ for some c we don't care about.
  $\log(ab) = \log a + \log b$, $\log(\frac{a}{b}) = \log a - \log b$.
  $\log_a b = \frac{\log_c b}{\log_c a}$.
  $\log_a b^n = n \log_a b \neq (\log_a b)^n$, $a^{\log_a n} = n$, $a^{\log_c b} = b^{\log_c a}$.
  $ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots$.

- Factorials: $n! = n \cdot (n-1) \cdots 2 \cdot 1$.
  $n! = n \cdot (n-1)!$, $0! = 1$. (Recursive definition)
  Sterling's approximation: $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \Theta(\frac{1}{n}))$. (Note: Θ means having the same order of magnitude.)
  The following approximation also holds: $n! = \sqrt{2\pi n}(\frac{n}{e})^n e^{\alpha_n}$, where $\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}$.
  $\log n! = \Theta(n \log n)$.

- Functional iteration: A function f applied iteratively i times to an initial argument n. Defined recursively, $f^{(0)}(n) = n$ and $f^{(i)}(n) = f(f^{(i-1)}(n))$ for $i > 0$. (Note: The distinction between $f^{(i)}(n)$ and $f^{i}(n)$.)
  For example, if $f(n) = 2n$ then $f^{(i)}(n) = 2^i n$.
- The log star function: $\log^* n = \min\{i \geq 0 : \log^{(i)} n \leq 1\}$, which is a very slowly growing function. $\log^* 2 = 1$, $\log^* 4 = 2$, $\log^* 16 = 3$, $\log^* 65536 = 4$, $\log^* 2^{65536} = 5$.

- Fibonacci numbers: $F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$.
  $F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$, where $\phi = \frac{1+\sqrt{5}}{2} = 1.61803\ldots$ is called the golden ratio, $\hat{\phi} = \frac{1-\sqrt{5}}{2} = -0.61803\ldots$ is the conjugate of ϕ, and both are roots of equation $x^2 = x + 1$.

Asymptotic notation

- Used to compare the growth rate or order of magnitude of increasing functions. “Asymptotic” describes the behavior of functions in the limit, for sufficiently large values of variables.
- $f(n) = O(g(n))$ if $\exists c, n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$.
- $f(n) = \Omega(g(n))$ if $\exists c, n_0$ such that $f(n) \geq cg(n)$ for $n \geq n_0$.
- $f(n) = \Theta(g(n))$ if $\exists c_1, c_2, n_0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$.
- $f(n) = o(g(n))$ if $\forall c\ \exists n_0$ such that $f(n) < cg(n)$ for $n \geq n_0$.
- $f(n) = \omega(g(n))$ if $\forall c\ \exists n_0$ such that $f(n) > cg(n)$ for $n \geq n_0$.

- Remarks:
  - In CLRS, the above notation is defined as sets of functions. For example, $f(n) \in O(g(n))$.
  - Comparison of growth rates of two functions: $O(\leq)$, $\Omega(\geq)$, $\Theta(=)$, $o(<)$, $\omega(>)$.
  - $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$, and $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$.
  - $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
  - $f(n) = O(g(n))$ if $f(n) = o(g(n))$, and $f(n) = \Omega(g(n))$ if $f(n) = \omega(g(n))$.
  - An alternative definition for $f(n) = o(g(n))$ is $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. Likewise, an alternative definition for $f(n) = \omega(g(n))$ is $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

- More remarks:
  - Asymptotic notation ignores constant factors and lower-order terms.
  - *Rule of thumb:* constant $\leq$ polylogarithmic $\leq$ polynomial $\leq$ exponential $\leq$ superexponential.

Example: 1, $\sqrt{\log n}$, $\ln n$, $(\log n)^2$, $\sqrt{n}$, $\sqrt{n}\log n$, n, $n\log n$, n^2, $n^{\log\log n}$, 2^n, $n2^n$, $n!$, 2^{2^n}.

Summations/Series

- Property of linearity: $\sum_{i=1}^{n}(ca_i + b_i) = c\sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$ and $\sum_{i=1}^{n} \Theta(f(i)) = \Theta(\sum_{i=1}^{n} f(k))$.
- Arithmetic sum/series: $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{1}{2}n(n+1)$.
- Geometric sum/series:
  $\sum_{i=0}^{n} r^i = 1 + r + r^2 + \cdots + r^n = \frac{r^{n+1}-1}{r-1}$ for $r \neq 1$.
  $1 + r + r^2 + \cdots = \frac{1}{1-r}$ for $|r| < 1$.
- Harmonic series: $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \ln n + \gamma + \frac{\varepsilon}{2n}$ for $\gamma = 0.5772156649\ldots$ (Euler's constant) and $0 < \varepsilon < 1$.
  Example: Prove that $ln(n+1) < H_n < \ln n + 1$.
  (Approximation by integrals)

- Binomial series: $\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = 2^n$.
- Other useful sums:
  $\sum_{i=1}^{n} i^2 = \frac{1}{6}n(n+1)(2n+1)$. (A direct proof starting with $\sum_{j=1}^{i}(2j-1) = i^2$)
  $\sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$. (Proved by induction)
  $\sum_{i=1}^{n} ix^{i-1} = \frac{nx^{n+1}-(n+1)x^n+1}{(x-1)^2}$. (Proved by using derivatives)

Proof techniques

- Proving by contradiction:
  The following three statements are logically equivalent:

  1. If A then B.
  2. If not B then not A.
  3. If A and not B then not C, where C is a proved fact or axiom.

  Example: Use contradiction to prove that
  (a) There are infinitely many prime numbers and
  (b) $\sqrt{2}$ is irrational.

- Proving by induction:
  The following statements are mathematically equivalent:

  1. $P(n)$ for integers $n \geq c$.
  2. Simple integer induction: $P(c)$ and $P(n-1) \rightarrow P(n)$. (What are inductive basis, inductive hypothesis, and inductive step?)
  3. General integer induction: $P(c)$ and $(\forall i : c \leq i \leq n-1) P(i) \rightarrow P(n)$.

  Example: Use induction to prove that
  (a) $\sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$ and
  (b) Every positive composite integer can be expressed as a product of prime numbers.

Solving recurrences

- Recurrence is an equation or inequality that defines a function in terms of the function's values on smaller inputs. For example, $T(1) = \Theta(1)$ (boundary condition) and $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ for $n \geq 2$ (recurrence) or almost equivalently, $T(1) = 1$ and $T(n) = 2T(\frac{n}{2}) + n$ for $n \geq 2$.
- *Remark:* We may neglect some technical details due to our interest in asymptotic solutions:
  - Relax the integer argument requirement on functions. For example, use $T(n/2)$ instead of $T(\lfloor n/2 \rfloor)$ or $T(\lceil n/2 \rceil)$.
  - Assume boundary condition $T(n) = \Theta(1)$ for small n if not given explicitly. Asymptotically, $\Theta(1)$ is the same as any constant c no matter how large it is.
  - Use $\Theta(f(n))$ or $f(n)$ at will in the recursive definition since this will have no affect on the final answer when expressed in Θ..

- (1) The iteration method: Apply recurrence until a summation pattern can be figured out.
  Example: $T(n) = 3T(\frac{n}{4}) + n$. (Assume $n = 4^k$.)
  Example: Solve $T(n) = \sqrt{n}T(\sqrt{n}) + n$ by iteration.

- (2) The recursion-tree method: Similar to the iteration method, use a tree for bookkeeping. Suitable for solving recurrence in big-O, where the function appears more than once on the right-hand-side of the recursive equation
  Example: $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + n$.
  Example: Solve $T(n) = T(\alpha n) + T((1-\alpha)n) + n$, where $0 < \alpha < 1$, by recursion tree.

- (3) The master method:
  Theorem: If $T(n) = aT(\frac{n}{b}) + f(n)$ for $a \geq 1$ and $b > 1$, then
  (a) if $f(n) = O(n^{(\log_b a)-\varepsilon})$ for some $\varepsilon > 0$, then
  $T(n) = \Theta(n^{\log_b a})$;
  (b) if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$;
  (c) if $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$ for $\varepsilon > 0$ and if $af(\frac{n}{b}) \leq cf(n)$ for $c < 1$ and all large n, then $T(n) = \Theta(f(n))$.
  Remark: The master method does not cover all cases.
  Example: $T(n) = 3T(\frac{n}{4}) + n \log n$. ($a = 3$, $b = 4$, and $f(n) = n \log n$. Case (c) applies.)
  Example: Solve $T(n) = 4T(\frac{n}{2}) + f(n)$ by the master theorem for $f(n) = n, n^2, n^3$.
  Example: $T(n) = 2T(\frac{n}{2}) + n \log n$. (The master theorem does not work.)

- (4) The substitution method: Guess and verify.
  Example: Let $T(n) \leq cn$ for $n \leq 49$ and $T(n) \leq T(\frac{n}{5}) + T(\frac{3n}{4}) + cn$ for $n \geq 50$. (Guess $T(n) \leq 20cn$ and then prove by induction. Can the recursion tree method be used?)

Remarks for the substitution method:

- Making a good guess.
- To prove $T(n) = O(f(n))$, sometimes we use an inequality stronger than $T(n) \leq cf(n)$ in the induction, such as $T(n) \leq 20cf(n)$ in the earlier example or $T(n) \leq cf(n) - d$ which can be used for solving $T(n) = 2T(\frac{n}{2}) + 1$.
- Avoid using asymptotic notation in the inductive proof.
  Example: $T(n) = T(n-1) + n$. What is wrong with the following proof?
  First guess $T(n) = O(n)$.
  Inductive basis: For $n = 1$, $T(1) = 1 = O(1)$.
  Inductive step: Assume $T(n-1) = O(n-1)$

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

- (5) Changing variables.
  Example: $T(n) = 2T(\sqrt{n}) + \log n$.

Analysis of Algorithms
An overview

- An algorithm is any well-defined computational procedure that takes some value or set of values as input and produces some value or set of values as output.
- Design of algorithms: Design techniques such as divide-and-conquer, greedy method, and dynamic programming plus use of appropriate data structures
- Description of algorithms: Shows the ideas of algorithms in a clear and clean way.
- Proof of correctness of algorithms: Requires special proof techniques and can be tricky sometimes.
- Analysis of the time complexity of algorithms: Must be independent of (1) makes/models of computers, (2) programming languages, (3) input data, and (4) skills of individual programmers.

- Our solution is
  1. Use any abstract computing model such as Turing Machine or its high-level equivalence Random Access Machine and count number of steps of the algorithm within the chosen model to measure time. Note all such computing models are polynomially-related so it really does not matter which model to choose.
  2. Use pseudocode to describe algorithms, which is the combination of natural language, math language, and programming language. Remember the description of an algorithm should be concise but also easy to follow.
  3. Represent time complexity (or number of basic steps) of an algorithm as a function of input size (not value).
  4. Express the time complexity function in asymptotic notation. This shows the performance of an algorithm for large-size input and makes it easy to compare the time complexity of different algorithms and to classify problems according to the time complexity of their best algorithms.

Worst-case analysis

- Definition:
  I_n: Any instance of size n;
  $t(I_n)$: Time (# of basic steps) spent on I_n by the algorithm;
  $T(n)$: Worst-case time complexity of any instance of size n,
  $T(n) = \max_{\forall I_n}\{t(I_n)\}$.

- Insertion sort: $A \Rightarrow A$ sorted in increasing order.

```
Insertion_Sort(A)
  for j = 2 to A.length
    key = A[j]
    //Insert A[j] into sorted A[1..j-1]
    i = j - 1
    while i > 0 and A[i] > key
      A[i+1] = A[i]
      i = i - 1
    A[i+1] = key
```

To insert $A[j]$ into the sorted $A[1 \ldots j-1]$, the algorithm will make at most $j-1$ comparisons and $j-1$ shifts. So the overall time complexity, which is dominated by the number of comparisons and the number of shifts, is at most $2\sum_{j=2}^{n}(j-1) = \Theta(n^2)$, where $n = A.length$.

- Euclid's algorithm (300 B.C.): Greatest common divisor.
  Recursively defined: $gcd(m, n) = m$ for $n = 0$:
  $gcd(m, n) = gcd(n, m \bmod n)$ for $n > 0$;
  Euclid(m, n) //Assume $m \geq n$
  while $n > 0$
  $t = m \bmod n$
  $m = n$
  $n = t$
  return m

The time complexity is decided by the number of iterations of the while-loop, say k. Let m_i and n_i be the values of m and n at the end of the ith iteration, respectively.
We observe that
(1) $n_k = 0$, $n_i \geq 1$ for $i = 0, \ldots, k-1$, and $n_0 > \cdots > n_k$;
(2) $m_i = n_{i-1} > n_i$, for $i = 1, \ldots, k$;
(3) $n_i = m_{i-1} \bmod n_{i-1} < \frac{m_{i-1}}{2} = \frac{n_{i-2}}{2}$ for $i = 2, \ldots, k$. (Note: If $a \geq b$, then $a \bmod b < \frac{a}{2}$.)

- Multiplying two integers x and y:
  Assume that x (multiplicand) and y (multiplier) have the same number of figures, $n = 2^k$, denoted $(x)_n$ and $(y)_n$.
  Then $(x)_n = 10^{\frac{n}{2}} \cdot (a)_{\frac{n}{2}} + (b)_{\frac{n}{2}}$ and $(y)_n = 10^{\frac{n}{2}} \cdot (c)_{\frac{n}{2}} + (d)_{\frac{n}{2}}$.
  So $(x)_n \cdot (y)_n =$
  $10^n((a)_{\frac{n}{2}} \cdot (c)_{\frac{n}{2}}) + 10^{\frac{n}{2}}((a)_{\frac{n}{2}} \cdot (d)_{\frac{n}{2}} + (b)_{\frac{n}{2}} \cdot (c)_{\frac{n}{2}}) + (b)_{\frac{n}{2}} \cdot (d)_{\frac{n}{2}}$.
  The problem of multiplying two integers of size n is now reduced to four subproblems of multiplying two integers of size $\frac{n}{2}$.
  $T(n) = 4T(\frac{n}{2}) + n = \Theta(n^2)$ by the master theorem.
  Now let $s = (a)_{\frac{n}{2}} \cdot (c)_{\frac{n}{2}}$, $t = (b)_{\frac{n}{2}} \cdot (d)_{\frac{n}{2}}$, and
  $r = ((a)_{\frac{n}{2}} + (b)_{\frac{n}{2}}) \cdot ((c)_{\frac{n}{2}} + (d)_{\frac{n}{2}})$. Then
  $(x)_n \cdot (y)_n = 10^n s + 10^{\frac{n}{2}}(r - s - t) + t$. Only three subproblems have to be solved in the recursion.
  $T(n) = 3T(\frac{n}{2}) + n = \Theta(n^{\log_2 3}) = \Theta(n^{1.585})$ by the master theorem.

Average-case analysis

- Definition:
  I_n: Any instance of size n;
  $t(I_n)$: Time (# of basic steps) spent on I_n by the algorithm;
  $p(I_n)$: Probability that I_n appears as input;
  $T(n)$: Average-case time complexity of any instance of size n, $T(n) = \sum_{\forall I_n} p(I_n)t(I_n)$.

- Insertion sort:
  Assumptions: (1) Distinct items;
  (2) Each permutation with equal probability to occur.
  $T(n) =$ average # of pairwise comparisons $= \sum_{j=2}^{n} c_j$,
  where c_j is the average # of comparisons to insert $A[j]$ (key) into the sorted$A[1 \ldots j-1]$.
  For key, there are j possible positions, with probability $\frac{1}{j}$ for each.
  So $c_j = \frac{1}{j} \cdot 1 + \frac{1}{j} \cdot 2 + \cdots + \frac{1}{j} \cdot (j-1) + \frac{1}{j} \cdot (j-1) = \frac{1}{2}(j+1) - \frac{1}{j}$.
  Therefore,
  $T(n) = \sum_{j=2}^{n} (\frac{1}{2}(j+1) - \frac{1}{j}) = \frac{1}{4}n^2 + \frac{3}{4}n - H_n = \Theta(n^2)$.

- Binary search
  Given $A[1 \ldots n]$ sorted and x (a query). Is x in A?
  Assumptions: (1) $x \in A$;
  (2) Distinct items in A;
  (3)$Pr(x = A[i]) = \frac{1}{n}$ for any $i = 1, 2, \ldots, n$;
  (4) $n = 2^k - 1$ for some k.
  Binary search can be illustrated by a decision tree of n nodes (full binary tree with k levels).
  Example: What is the decision tree for $n = 7$?
  Average-case time is then the average length of a path from the root to a tree node.

Level	# of nodes	amount to add to average length
1	1	$\frac{1}{n} \cdot 1$
2	2	$\frac{1}{n} \cdot 2 + \frac{1}{n} \cdot 2$
3	4	$\frac{1}{n} \cdot 3 + \frac{1}{n} \cdot 3 + \frac{1}{n} \cdot 3 + \frac{1}{n} \cdot 3$
$\dots$	$\dots$	$\dots$
k	2^{k-1}	$\frac{2^{k-1}}{n} \cdot k$

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{k} \frac{2^{i-1}}{n} \cdot i \\
&= \frac{1}{n} \sum_{i=1}^{k} i \cdot 2^{i-1} \\
&= \frac{1}{n}(k \cdot 2^k - 2^k + 1) \\
&= k + \frac{k}{n} - 1 \\
&= \Theta(\log n).
\end{aligned}
$$

- Binary search trees (BST)
  What is the average number of comparisons, $T(n)$, needed to insert n distinct random elements into an initially empty BST?
  First, $T(0) = T(1) = 0$.
  Assume that $A = (a_1, \ldots, a_n)$ is the list given and that $B = (b_1, \ldots, b_n)$ is A sorted increasingly. That A is a random sequence implies that a_1 is equally likely to be b_j for any $1 \leq j \leq n$. Consider the tree obtained after the insertion of all n numbers, which has $a_1(b_j)$ as the root, $b_1, \ldots, b_{j-1}$ in the left subtree, and $b_{j+1}, \ldots, b_n$ in the right subtree.

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

  How to solve this recurrence?

Amortized analysis

- Amortization:
  1. To put money aside at intervals, as a sinking fund, for gradual payment of a debt;
  2. To average the time (cost) required to perform a sequence of data structure operations over all operations performed.

- Motivation:
  - A sequence of n data structure related operations, rather than just a single operation, is performed.
  - An operation may change the data structure, thus affect the next operation.
  - What is the total time complexity of the entire sequence?

- Compare three types of analysis
  - Worst-case analysis: Sum of worst-case time of each operation (which may never be achieved, thus not tight and overly pessimistic).
  - Average-case analysis: Averaging over all possible inputs and involving probability.
  - **Amortized analysis**: Averaging over a worst-case sequence to determine an amortized cost for each operation, which can be the total cost of the sequence divided by the number of operations, i.e., $T(n)/n$, or any costs as long as they add up to the total cost $T(n)$. Amortized analysis often gives a tight worst-case time bound.

- Three techniques:
  - **Aggregate analysis**: Attempt to consider the entire sequence as a whole and show for all n, a sequence of n operations takes worst-case time $T(n)$ in total, therefore, the amortized cost per operation is $T(n)/n$.
  - **Accounting method (banker's view)**: Represent prepaid work as credit stored with specific objects within the data structure.
  - **Potential method (physicist's view)**: Represent prepaid work as potential energy that can be released for future operations. The potential is associated with the data structure as a whole rather than with specific objects within the data structure.

- Example: Stack manipulation
  - Three types operations: $push(S, x)$, $pop(S)$ and $multipop(S, k)$.
  - Assume a sequence contains n operations defined above and initially the stack is empty.
  - Question: What is the total time/cost of a sequence of n operations?
  - Worst-case analysis: $O(n^2)$
  - Aggregate analysis: $2n = O(n)$. (At most n items pushed to the stack and each may result in a later pop.)

- Stack manipulation (continued)
  - Accounting method:
    - Coin-operated computer: 1 credit $\Rightarrow$ 1 time unit $\Rightarrow$ 1 push/pop.
    - Sequence of operations: $O_1, \ldots, O_n$, where operation O_i is allocated with credits c_i.
    - Assumptions:
      (1) Unused credits are carried over to later operations;
      (2) Operations can borrow credits as long as any debt is paid off eventually.
    - If all sequences of length n can be performed with the allocated credits, the total time is no larger than $\sum_{i=1}^{n} c_i$.
    - Key: How to choose c_i.

- Stack manipulation (continued)
  - Potential method:
    - Define a potential function $\Phi : D \to R$.
    - Assume that an operation takes t_i time units to change the data structure from D_{i-1} to D_i.
    - Then the amortized time of the operation is

$$a_i = t_i + \Phi(D_i) - \Phi(D_{i-1}),$$

and the total time for the sequence is

$$\sum_{i=1}^{n} t_i = \sum_{i=1}^{n} (a_i - \Phi(D_i) + \Phi(D_{i-1}))$$

$$= \Phi(D_0) - \Phi(D_n) + \sum_{i=1}^{n} a_i \leq \sum_{i=1}^{n} a_i \quad \text{if} \;\; \Phi(D_0) \leq \Phi(D_n).$$

    - Key: How to choose Φ.

- Example: Binary counter
  - A k-bit binary counter is implemented by an array $A[0 \ldots k-1]$, where $A.length = k$.
  - A binary number x stored in the counter can be defined as $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Initially $x = 0$, thus $A[i] = 0$ for $i = 0, 1, \ldots, k-1$.
  - The operation of interest is to increment the value in the counter.

    INCREMENT(A)
    ```
    i = 0
    while i < A.length and A[i] == 1
        A[i] = 0
        i = i + 1
    if i < A.length
        A[i] = 1
    ```
  - Starting from $x = 0$, what is the total time of performing a sequence of n INCREMENT operations?

- Binary counter (continued)
  - Worst-case analysis: $O(kn)$
  - Aggregate analysis: Consider a 4-bit counter as an example. Let $n = 8$.

x	$A[3]$	$A[2]$	$A[1]$	$A[0]$	total cost
0	0	0	0	0	0
1	0	0	0	1	$0+1=1$
2	0	0	1	0	$1+2=3$
3	0	0	1	1	$3+1=4$
4	0	1	0	0	$4+3=7$
5	0	1	0	1	$7+1=8$
6	0	1	1	0	$8+2=10$
7	0	1	1	1	$10+1=11$
8	1	0	0	0	$11+4=15$

  - Observe that the total cost is also the number of bit-flips.

- Binary counter (continued)
  - Accounting method: 1 credit $\Rightarrow$ 1 time unit $\Rightarrow$ 1 bit flip. Note that to perform an INCREMENT, there is only one bit that is flipped from 0 to 1 but zero or more bit that are flipped from 1 to 0.

- Binary counter (continued)
  - Potential method: The underlining data structure D is the k-bit counter (array). Define the potential $\Phi(D) = \#$ of 1's in the counter. Since $\Phi(D_0) = 0$ and $\Phi(D_n) \geq 0$, then $\Phi(D_0) \leq \Phi(D_n)$.

Probabilistic analysis

- Overview: When applying the concept of randomization to algorithms, there are usually two views. One is to consider the input to be structured random or following some probabilistic distribution while the algorithms are deterministic without randomness. This results in the average-case analysis. The other is to make no assumption about the input but to allow algorithms to behave random with the use of random process (such as the random number generator). This results in the probabilistic analysis.
- Some simple examples of randomized algorithms: randomized linear search and randomized quick sort.

- Example: Contention resolution
  There are n processes $P_1, P_2, \ldots, P_n$, each competing for access to a single shared database. Imagine time as being divided into discrete rounds. The database can be accessed by at most one process in a single time round; if more than one processes attempt to access it simultaneously, all processes are locked out for the duration of that round. Assuming no communication exists among processes, what is a good protocol with which the processes can access the database on a somewhat regular and equitable basis?
  A randomized algorithm to smooth out contention: For some carefully selected $0 < p \leq 1$, each process will attempt to access the database in each round with probability p, regardless of the decisions of other processes.

- Probabilistic analysis: What is the probability for a process to succeed in accessing the database in a round?
  - For any process P_i and time round t, define $A[i,t]$ to be the event that P_i attempts to access the database in round t. Clearly, $Pr[A[i,t]] = p$ and $Pr[\overline{A[i,t]}] = 1 - p$.
  - Define $S[i,t]$ to be the event that process P_i succeeds in accessing the database in round t. So $S[i,t] = A[i,t] \wedge (\wedge_{j \neq i} \overline{A[j,t]})$ and $Pr[S[i,t]] = Pr[A[i,t]] \cdot \Pi_{j \neq i} Pr[\overline{A[j,t]}] = p(1-p)^{n-1}$. The maximum success probability $Pr[S[i,t]]$ is achieved by setting $p = 1/n$, yielding $Pr[S[i,t]] = (1/n)(1-(1/n))^{n-1}$.
  - A helpful result from basic calculus: (a) The function $(1-(1/n))^n$ converges monotonically from $1/4$ up to $1/e$ as n increases from 2; (b) The function $(1-(1/n))^{n-1}$ converges monotonically from $1/2$ down to $1/e$ as n increases from 2.
  - From the above result, we get $1/(en) \leq Pr[S[i,t]] \leq 1/(2n)$, and hence $Pr[S[i,t]]$ is asymptotically equal to $\Theta(1/n)$.

- Probabilistic analysis: What is the probability that a process has not yet succeeded after a certain number of rounds?
  - Define $F[i,t]$ to be the event that process P_i does not succeed in any of the rounds 1 through t. Then $Pr[F[i,t]] = Pr[\wedge_{r=1}^{t} \overline{S[i,r]}] = \Pi_{r=1}^{t} Pr[\overline{S[i,r]}] = [1-(1/n)(1-(1/n))^{n-1}]^t \leq (1-1/(en))^t$.
  - If we set $t = \lceil en \rceil$, we get $Pr[F[i,t]] \leq (1-1/(en))^{\lceil en \rceil} \leq (1-1/(en))^{en} \leq 1/e$.
  - If we set $t = \lceil en \rceil \cdot (c \ln n)$, then $Pr[F[i,t]] \leq (1-1/(en))^t = ((1-1/(en))^{\lceil en \rceil})^{c ln n} \leq e^{-c \ln n} = n^{-c}$.
  - Conclusion: Asymptotically, after $\Theta(n)$ rounds, the probability that P_i has not yet succeeded is bounded by a constant; and between then and round $\Theta(n \ln n)$, this probability drops to a small quantity, bounded by an inverse polynomial in n.

- Probabilistic analysis: What is the probability that all processes succeed at least once in a given number of rounds?
  - Define F_t be the event that some process has not yet succeeded in accessing the database after t rounds. Then $F_t = \vee_{i=1}^{n} F[i,t]$.
  - The Union Bound: Given events $E_1, E_2, \ldots, E_n$, $Pr[\vee_{i=1}^{n} E_i] \leq \sum_{i=1}^{n} Pr[E_i]$. Or in words, the probability of the union of events is uppper-bounded by the sums of their individual probabilities.
  - $Pr[F_t] \leq \sum_{i=1}^{n} Pr[F[i,t]]$. To make the sum small thus tight for the bound, $Pr[F[i,t]]$ for each i needed to be significantly smaller than $1/n$. Choosing $t = \Theta(n)$ will not be good enough. But if we choose $t = \lceil en \rceil \cdot (c \ln n)$, we will have $Pr[F[i,t]] \leq n^{-c}$ for each i, which is what we want. Specifically, let $t = 2\lceil en \rceil \cdot \ln n$, we have $Pr[F_t] \leq \sum_{i=1}^{n} Pr[F[i,t]] \leq n \cdot n^{-2} = 1/n$.
  - Conclusion: With probability at least $1 - (1/n)$, all processes succeed in accessing the database at least once within $t = 2\lceil en \rceil \cdot \ln n$ rounds.

Design of Algorithm: By Data Structures

Data structures for disjoint sets

- Purposes:
  (1) Use good data structures to achieve algorithm efficiency;
  (2) Use amortization (it's the aggregate analysis) to give tight estimation of algorithm complexity.

- Problem: Initially, we are given n singletons: $S_i = \{i\}$ for $i = 1, \ldots, n$. We wish to execute a sequence of m operations of the following two types:
  (1) *union*(S_i, S_j) returns $S_i \cup S_j$, where S_i and S_j are disjoint;
  (2) *find*(i) returns the name of the set containing i.
- How can we organize the data (sets in this case) such that any sequence of intermixed *union*'s and *find*'s can be performed efficiently?

- Data structure: Set $\Rightarrow$ tree (arbitrary); Set name $\Rightarrow$ root; Sets $\Rightarrow$ forest (use parent array).
  Initially, there are n singletons, corresponding to n single-node trees in a forest, which is represented by a parent array of size n with 0 in each entry. ($parent[i]$ gives the parent of i in the forest. If $parent[i] = 0$, i is a root.)

- Implementation of *union* (by rank): Initially, each singleton node has a rank of 0. To union two sets, or trees, make the tree whose root has a smaller rank a subtree of the root of the other tree. If both roots before the union have the same rank, increment the rank of the root of the combined tree by one. $union(i, j)$ takes $O(1)$ time.
- Implementation of *find* (with path compression): Two passes are needed between the node and its root. One is to find the root and the other is to do the path compression. $find(i)$ takes $O(d)$, where d is the depth of i in the tree, or the distance between i and the root.

Note: *union* can also be done by size but it does not result in a different time complexity from union-by-rank.

- Complexity: Given an intermixed sequence σ of q *union*'s and p *find*'s, i.e., $m = q + p$. What is the worst-case time complexity for executing σ on an initial collection of n singletons $\{1\}, \ldots, \{n\}$?
  The worst-case analysis gives an overly pessimistic bound of $O(q + pn)$, or $O(mn)$.
  With an amortized analysis, Tarjan (1975) proved a worst-case time of $O(m \cdot \alpha(n))$, where α is the inverse of the famous Ackermann's function.
  What we will show next is the slightly loose time of $O(m \cdot \log^* n)$, which is an earlier result given by Hopcroft and Ullman (1973). Both $\alpha(n)$ and $\log^* n$ are extremely slow-growing functions. We choose the $\log^* n$ version since its proof is a little bit easier.

- Observations about ranks:
  We observe that if there are only *union*'s but no *find*'s in σ, then the rank of a node is the height of the tree rooted at that node. However, if there are *find*'s in σ, then path compression may decrease the height of the root thus making its rank to be just an upper bound of the height. Additionally, the only time when the rank of a node may be changed (increased) is when it is the root of a tree. Once a node becomes a non-root, it will never be changed back to a root again, thus its rank will remain the same from that point.

- Properties of ranks:
  P1: As the *find* operation follows the path to the root, the ranks of nodes its encounters are strictly increasing.
  P2: Any node i with rank r has at least 2^r descendants (including itself).
  P3: There are at most $\frac{n}{2^r}$ nodes of rank r.

- Definitions: $F(0) = 1$ and $F(n) = 2^{F(n-1)}$ for $n \geq 1$.
- Let G be the inverse of F, i.e., $G(n) = \min\{k \geq 0 | F(k) \geq n\}$. Note that $G(F(n)) = n$ and $G(n) = \log^* n$.

n	0	1	2	3	4	5	$\cdots$	16	17
$F(n)$	1	2	4	16	65536	$2^{2^{2^{2^{2}}}}$ $\cdots$	$\cdots$	$\cdots$	$\cdots$
$G(n)$	0	0	1	2	2	3	$\cdots$	3	4

- We next distribute ranks into groups by putting rank r into group $G(r)$. Thus, group 0 contains ranks 0 and 1; group 1 contains just rank 2; group 2 contains ranks 3 and 4; group 3 contains ranks from 5 to 16; and finally group $G(n)$ contains the largest possible rank, n.

Group	Ranks in the group
0	$0, 1$
1	2
2	$3, 4$
3	$5, \ldots, 16$
$\ldots$	$\ldots$
$G(n)$	$\ldots, n$
g	$F(g-1)+1, \ldots, F(g)$

It is easy to see that in any sequence σ, the total cost of q *union*'s is $O(q)$.
What is the total cost of p *find*'s?
Observe that the total cost of *find*'s can be measured by the total number of hops via tree edges toward the root for all *find*'s.
Let it be written as the sum of three types of cost, i.e.,
$T = T_1 + T_2 + T_3$, where

- T_1 is the total hops of those *find*'s that make only one hop to reach the root. Thus $T_1 = O(p)$.
- T_2 is the total hops between nodes with ranks in different groups. Prove that $T_2 = O(p \cdot G(n))$.
- T_3 is the total hops between nodes with ranks in the same group. What is T_3?

$$
\begin{aligned}
T_3 &\leq S \\
&\leq \sum_{\forall g} \sum_{\forall u \in g} (F(g) - F(g-1)) \\
&\leq \sum_{\forall g} \sum_{\forall u \in g} F(g) \\
&\leq \sum_{\forall g} (F(g) \cdot (\#\text{ nodes in group } g)) \\
&\leq \sum_{\forall g} (F(g) \cdot \sum_{r=F(g-1)+1}^{F(g)} \frac{n}{2^r}) \qquad \text{(By P3)} \\
&\leq \sum_{\forall g} (F(g) \cdot \frac{n}{F(g)}) \\
&\leq nG(n)
\end{aligned}
$$

The total cost for executing σ with $m = q + p$ operations is the sum of

- $O(q)$ for q *union*'s and
- $O(p) + O(pG(n)) + O(nG(n)))$ for p *find*'s.

So the total time is $O(q + pG(n) + nG(n))$, which is just a little more than $O(m + n)$ since $G(n) = \log^* n$ is an extremely slow-growing function, e.g. $G(2^{65534}) = 5$.
Or the total time is $O(m \log^* n)$ if $m = \Theta(n)$, which is always the case in practice.

Binary heaps

- The (binary) heap is a data structure H of keys that supports $Insert(H, x)$, $ExtractMax(H)$, $FindMax(H)$, and $MakeHeap(H)$. It is represented as a left-complete binary tree with the heap property that the key of a parent is no smaller than the keys of its children (called max-heap). It can be implemented as an array. For example, $16, 14, 10, 8, 7, 9, 3, 2, 4, 1$.

- A heap with n keys has height $\Theta(\log n)$ and the key of the root node is the maximum.
- Why use an array to implement a heap? In array $H[1 \ldots n]$ that represents a heap, if the parent of $H[i]$ is $H[\lfloor i/2 \rfloor]$, the left child of $H[i]$ is $H[2i]$, and the right child of $H[i]$ is $H[2i+1]$.

- *Insert* and *ExtractMax* can be done in $O(\log n)$ time using the up-heap process and the down-heap process respectively.
- *FindMax* can be done in $O(1)$ time.
- *MakeHeap* can be interpreted in two ways. It may mean to make an empty heap, which can be done in $O(1)$ time.
  It may also mean to convert a binary left-complete tree (array) into a heap, which can be done using *Insert* n times or using down-heap $n/2$ times in $O(n \log n)$ time or $O(n)$ time respectively.
- We use *MakeHeap*() and *MakeHeap*(H) to distinguish the two alternatives.
- The heap-sort algorithm: $O(n \log n)$ worst-case time.

Fibonacci heaps

Here, we switch to the concept of min-heap to be consistent with the textbook. Also we assume that *MakeHeap* is to make an empty Fibonacci heap.

- Mergeable heaps: A data structure that supports not only *Insert*, *ExtractMin*, *FindMin*, and *MakeHeap*, but also supports the following new operations: $Union(H_1, H_2)$ to make a new heap of elements in H_1 and H_2, $DecreaseKey(H, x, k)$ to decrease the key of x to a smaller value k, and $Delete(H, x)$ to delete x from H. Assume for *DecreaseKey* and *Delete*, the position of x in H is known.
- Mergeable heaps can be implemented by binary heaps with time $O(\log n)$, $O(\log n)$, $O(1)$, $O(1)$ or $O(n)$, $O(n)$, $O(\log n)$, $O(\log n)$ corresponding to the operations listed above. (Note: for *DecreaseKey*, apply up-heap at x and for *Delete*, swap the key of x with last key in the array and apply down-heap.)

- A Fibonacci heap (Fredman and Tarjan 1986) is a collection of rooted trees (not necessarily binary) that are min-heap ordered. Pointers are used to connect the tree nodes in the following ways.
  - The roots are connected into a circular doubly linked list (in any order), with a pointer to the root with the minimum key
  - For any node, there is a pointer to its parent and a pointer to any one of its children.
  - All sibling nodes are connected into a circular doubly linked list
  - A few marked nodes (all unmarked initially, will only be marked in *DecreaseKey*, and may be unmarked again in *ExtractMin*)