CIS 1210: Data Structures and Algorithms

Course Lecture Notes

CIS 1210 Course Staff [*]

Draft of: April 4, 2023

University of Pennsylvania

[*] see acknowledgments on next page

Goal

There is no one book that covers everything that we want to cover in CIS 1210. The goal of these notes is for students to find all course lecture material in one place, and in one uniform format.

Acknowledgments

These lecture notes were compiled by CIS 1210 Course Staff, including Steven Bursztyn, Rajiv Gandhi, John Geyer, and Robin Tan for CIS 1210 at the University of Pennsylvania.

These lecture notes are a work in progress, and we appreciate the students and Head TAs who have helped make small edits.

Some chapters include attributions to other courses or textbooks from which we adapted our lecture notes, with some parts copied directly.

We also gratefully acknowledge the many others who made their course materials freely available online.

Disclaimer

These notes are designed to be a supplement to the lecture. They may or may not cover all the material discussed in the lecture (and vice versa).

Material varies from semester to semester, and this book may contain more information than what will be covered in lecture (i.e., material you may not be responsible for knowing). Please refer to the course website for assigned readings.

Errata

If you find any mistakes, please email the professor(s) and Head TAs so we can fix them.

Feedback

If you believe a certain topic is explained poorly, or could use additional examples or explanations, please reach out to the course staff. As mentioned, this is a work in progress and we would love to do what we can to make this resource as helpful as possible.

Table of Contents

Review of Terms, Proofs, and Probability 1

1.1 Review of Proofs and Proof Techniques

The *unique factorization theorem* states that every positive number can be uniquely represented as a product of primes. More formally, it can be stated as follows.

> Given any integer $n > 1$, there exist a positive integer k, distinct prime numbers $p_1, p_2, \ldots, p_k$, and positive integers $e_1, e_2, \ldots, e_k$ such that
>
> $$n = p_1^{e_1} p_2^{e_2} p_3^{e_3} \cdots p_k^{e_k}$$
>
> and any other expression of n as a product of primes is identical to this except, perhaps, for the order in which the factors are written.

Example. Prove that $\sqrt{2}$ is irrational using the unique factorization theorem.

Solution. Assume for the purpose of contradiction that $\sqrt{2}$ is rational. Then there are numbers a and b $(b \neq 0)$ such that

$$\sqrt{2} = \frac{a}{b}$$

Squaring both sides of the above equation gives

$$\begin{aligned} 2 &= \frac{a^2}{b^2} \\ a^2 &= 2b^2 \end{aligned}$$

Let $S(m)$ be the sum of the number of times each prime factor occurs in the unique factorization of m. Note that $S(a^2)$ and $S(b^2)$ is even. Why? Because the number of times that each prime factor appears in the prime factorization of a^2 and b^2 is exactly twice the number of times that it appears in the prime factorization of a and b. Then, $S(2b^2)$ must be odd. This is a contradiction as $S(a^2)$ is even and the prime factorization of a positive integer is unique.

Example. Prove or disprove that the sum of two irrational numbers is irrational.

Solution. The above statement is false. Consider the two irrational numbers, $\sqrt{2}$ and $-\sqrt{2}$. Their sum is $0 = 0/1$, a rational number.

Example. Show that there exist irrational numbers x and y such that x^y is rational.

Solution. We know that $\sqrt{2}$ is an irrational number. Consider $\sqrt{2}^{\sqrt{2}}$.

Case I: $\sqrt{2}^{\sqrt{2}}$ is rational.
In this case we are done by setting $x = y = \sqrt{2}$.

Case II: $\sqrt{2}^{\sqrt{2}}$ is irrational.

In this case, let $x = \sqrt{2}^{\sqrt{2}}$ and let $y = \sqrt{2}$. Then, $x^y = \left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = (\sqrt{2})^2 = 2$, which is an integer and hence rational.

Induction

Induction is a proof technique that relies on the following logic: If I have a statement $P(n)$ such that the statement is true for $n = 1$, i.e. $P(n) = true$, *and* I know that whenever $P(k)$ is true for any $k \geq 1$, then $P(k+1)$ is also true, then the statement must be true for all integers $n \geq 1$. The typical intuitive examples of induction are:

- Suppose I have a line of dominoes. If I know I can knock down the first domino, and I know that if the k-th domino falls, it will knock over the $k+1$-th domino, then all the dominoes will fall.
- Suppose I have a ladder. If I know that I can get on the ladder, and I know that for each step of the ladder it is possible to get to the next step, then I can climb the whole ladder.

There are plenty of other examples. In general, an inductive proof of a claim P consists of proving that a **base case (BC)** holds (usually $P(0)$ or $P(1)$), making an **induction hypothesis (IH)** that assumes that $P(k)$ is true for some value k that is greater than or equal to the base case, and lastly an **induction step (IS)** which uses the IH to prove that $P(k+1)$ holds. Please note that induction can only be used to prove properties about integers–you can't use induction to prove statements about real numbers.

Example. Prove via induction that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Solution. We proceed via induction on n.

Base Case: The base case occurs when $n = 1$. $\sum_{i=1}^{1} i = 1$ and $\frac{1(1+1)}{2} = 1$, and so the claim holds in the base case.

Induction Hypothesis: Assume that for some integer $k \geq 1$ we have that $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$.

Induction Step: We must show that $\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$. Indeed:

$$\begin{aligned}
\sum_{i=1}^{k+1} i &= (k+1) + \sum_{i=1}^{k} i \\
&= (k+1) + \frac{k(k+1)}{2} && \text{(by the induction hypothesis)} \\
&= \frac{2(k+1)}{2} + \frac{k(k+1)}{2} \\
&= \frac{2(k+1) + k(k+1)}{2} \\
&= \frac{(k+1)(k+2)}{2}
\end{aligned}$$

Note how we used the induction hypothesis in the induction step. If you find yourself doing a proof by induction without invoking the induction hypothesis, you are probably doing it incorrectly (or induction is not necessary).

Example. Prove using induction that, for any positive integer n, if $x_1, x_2, \ldots, x_n$ are n distinct real numbers, then no matter how the parenthesis are inserted into their product, the number of multiplications used to compute the product is $n - 1$.

Solution. Let $P(n)$ be the property that "If $x_1, x_2, \ldots, x_n$ are n distinct real numbers, then no matter how the parentheses are inserted into their product, the number of multiplications used to compute the product is $n - 1$".

Base Case: $P(1)$ is true, since x_1 is computed using 0 multiplications.

Induction Hypothesis: Assume that $P(j)$ is true for all j such that $1 \leq j \leq k$.

Induction Step: We want to prove $P(k+1)$. Consider the product of $k+1$ distinct factors, $x_1, x_2, \ldots, x_{k+1}$. When parentheses are inserted in order to compute the product of factors, some multiplication must be the final one. Consider the two terms, of this final multiplication. Each one is a product of at most k factors. Suppose the first and the second term in the final multiplication contain f_k and s_k factors. Clearly, $1 \leq f_k, s_k \leq k$. Thus, by induction hypothesis, the number of multiplications to obtain the first term of the final multiplication is $f_k - 1$ and the number of multiplications to obtain the second term of the final multiplication is $s_k - 1$. It follows that the number of multiplications to compute the product of $x_1, x_2, \ldots, x_k, x_{k+1}$ is

$$(f_k - 1) + (s_k - 1) + 1 = f_k + s_k - 1 = k + 1 - 1 = k$$

1.2 Graphs

A **graph** $G = (V, E)$ is a set of **vertices** (also called **nodes**) V together with a set of **edges** $E \subseteq V \times V$, where $V \times V$ denotes the cartesian product of V with itself. We denote an edge from u to v as (u, v). A graph can be undirected, in which case we consider the edges (u, v) and (v, u) to be the same edge, or it can be directed, in which case we distinguish (u, v) from (v, u). In CIS 121, we don't worry about so-called "self-loops", i.e. edges of the form (v, v). The **degree** of a vertex v, denoted $\deg(v)$ is the number of edges incident on it (for directed graphs, we distinguish between in-degree and out-degree).

A **path** from u to v is a sequence $u = v_0,\ v_1,\ \ldots,\ v_n = v$ such that $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$. The **length** of a path is the number of edges in the path. A **simple path** is a path containing distinct vertices, i.e. for all $i \neq j$, $v_i \neq v_j$.

A **cycle** is a sequence $u = v_0,\ v_1, \ldots, v_n = u$, essentially a path with the same start and end point. Graphs that contain cycles are called **cyclic** while graphs that don't contain cycles are called **acyclic**.

A **connected** graph is an undirected graph such that for any two vertices u and v, there is a path from u to v. A **strongly connected** graph extends this definition to directed graphs: for any two vertices u and v, there is a path from u to v *and* a path from v to u. A **(strongly) connected component** of a graph G is a subgraph of G that is (strongly) connected and maximal, i.e., it is (strongly) connected and not contained in any larger (strongly) connected component.

Trees

A **tree** is an undirected, connected acyclic graph. Some special defining properties of trees are given below:

- A tree with n nodes is connected and has exactly $n-1$ edges
- For any vertices u and v in a tree T, there is a unique path from u to v
- Tree's are minimially connected: that is, a tree T is connected, but the removal of any edge from T will disconnect it
- A tree is acyclic, but adding *any* edge to the tree will create a cycle.

Any of the above can be used as the definition of a tree. A vertex with degree 1 is called a **leaf**. A **forest** is an acyclic, undirected graph. A forest is just the union of one or more trees.

A **binary tree** is a tree such that every vertex has degree at most 3. Usually we consider the tree to be **rooted** at some root vertex r. The **height** of a binary tree is the number of *edges* in the longest path from the root node to a leaf. Also, instead of considering the neighbors of vertices in a binary tree, we normally call them the left/right children. The **ancestors, descendants,** and **parents** of nodes in a binary tree are defined exactly how you think they should be.

Example. Prove that every graph G with n vertices and m edges has at least $n-m$ connected components.

Solution. We prove the claim via induction on m.

Base Case: $m = 0$. When there are no edges in G, there are trivially n connected components—one for each vertex.

Induction Hypothesis: Assume that for a given graph G with n vertices and k edges (where k is a non-negative integer), there are at least $n-k$ connected components.

Induction Step: Given a graph G with n vertices and $k+1$ edges, we want to show that G has at least $n-(k+1)$ connected components.

We first construct a subgraph of G with n vertices and k edges, G', by removing an arbitrary but particular edge $\{u, v\}$. By IH, G' has at least $n-k$ connected components. We then add back edge $\{u, v\}$, and consider two cases with respect to our vertices u and v.

Case I: u and v belong to the same connected component in G'.
In this case, we find that adding back $\{u, v\}$ does not alter the connected components in the graph, and thus G has at least $n-k > n-(k+1)$ connected components.

Case II: u and v do not belong to the same connected component in G'.
First let, C_u and C_v be the connected components for u and v respectively. We find that adding back $\{u, v\}$ will result in C_u and C_v combining (while all other connected components remain unchanged), and thus reduce the number of connected components by exactly 1. Thus, G has at least $n-k-1 = n-(k+1)$ connected components.

In either case we find that the number of connected components in G is at least $n-(k+1)$, completing our induction step and our proof.

Example. Prove that every connected graph with n vertices has at least $n-1$ edges.

Solution. We will prove the contrapositive, i.e., a graph G with $m \leq n-2$ edges is disconnected. From the result of the previous problem, we know that the number of components of G is at least

$$n - m \geq n - (n-2) = 2$$

which means that G is disconnected. This proves the claim.

One could also have proved the above claim directly by observing that a connected graph has exactly one connected component. Hence, $1 \geq n - m$. Rearranging the terms gives us $m \geq n-1$.

Example. Prove that every tree with at least two vertices has at least two leaves and deleting a leaf from an n-vertex tree produces a tree with $n-1$ vertices.

Solution. A connected graph with at least two vertices has an edge. In an acyclic graph, an endpoint of a maximal non-trivial path (a path that is not contained in a longer path) has no neighbors other than its only neighbor on the path. Hence, the endpoints of such a path are leaves.

Let v be a leaf of a tree T and let $T' = T - v$. A vertex of degree 1 belongs to no path connecting two vertices other than v. Hence, for any two vertices $u, w \in V(T')$, every path from u to w in T is also in T'. Hence T' is connected. Since deleting a vertex cannot create a cycle, T' is also acyclic. Thus, T' is a tree with $n-1$ vertices.

Example. Prove that if G is a tree on n vertices then G is connected and has $n-1$ edges.

Solution. We can prove this by induction on n. The property is clearly true for $n = 1$ as G has 0 edges. Assume that any tree with k vertices, for some $k \geq 1$, has $k-1$ edges. We want to prove that a tree G with $k+1$ vertices has k edges. Let v be a leaf in G, which we know exists as G is a tree with at least two vertices. Thus $G' = G - \{v\}$ is connected. By induction hypothesis, G' has $k-1$ edges. Since $deg(v) = 1$, G has k edges.

Eulerian and Hamiltonian Graphs

An *Eulerian circuit* is a closed walk in which each edge appears exactly once. A graph is *Eulerian* if it contains an Eulerian circuit. A *Hamiltonian circuit* is a closed walk in which each vertex appears exactly once. A graph is *Hamiltonian* if it contains a Hamiltonian circuit.

To determine whether a graph is Hamiltonian or not is significantly harder than determining whether a graph is Eulerian or not. In this class we study the characterization of Eulerian graphs.

Example. If $\delta(G) \geq 2$ then G contains a cycle.

Solution. Let P be a longest path (actually, any *maximal* path suffices) in G and let u be an endpoint of P. Since P cannot be extended, every neighbor of u is a vertex in P. Since $deg(u) \geq 2$, u has a neighbor $v \in P$ via an edge that is not in P. The edge $\{u, v\}$ completes the cycle with the portion of P from v to u.

Example. Prove that a connected graph G is Eulerian iff every vertex in G has even degree.

Solution. *Necessity:* To prove that "if G is Eulerian then every vertex in G has even degree". Let C denote the Eulerian circuit in G. Each passage of C through a vertex uses two incident edges and the first edge is paired with the last at the first vertex. Hence every vertex has even degree.

Sufficiency: To prove that "if every vertex in G has even degree then G is Eulerian". We will prove this using induction on the number of edges, m.

Base Case: $m = 0$. In this case G has only one vertex and that itself forms a Eulerian circuit.

Induction Hypothesis: Assume that the property holds for any graph G with n vertices and j edges, for all j such that $n - 1 \leq j \leq k$.

Induction Step: We want to prove that the property holds when G has n vertices and $k + 1$ edges. Since G has at least one edge and because G is connected and every vertex of G has even degree, $\delta(G) \geq 2$. From the result of the previous problem, G contains a cycle, say C. Let G' be the graph obtained from G by removing the edges in $E(C)$. Since C has either 0 or 2 edges at every vertex of G, each vertex in G' also has even degree. However, G' may not be connected. By induction hypothesis, each connected component of G' has an Eulerian circuit. We can now construct an Eulerian circuit of G as follows. Traverse C, but when a component of G' is entered for the first time, we detour along the Eulerian circuit of that component. The circuit ends at the vertex where we began the detour. When we complete the traversal of C, we have completed an Eulerian circuit of G.

Alternative Proof for the Sufficiency Condition: Let G be a graph with all degrees even and let

$$W = v_0 e_0 \dots e_{l-1} v_l$$

be the longest walk in G using no edge more than once. Since W cannot be extended all edges incident on v_l are part of W. Since all vertices in G have even degree it must be that $v_l = v_0$. Thus W is a closed walk. If W is Eulerian then we are done. Otherwise, there must be an edge in $E[G] \setminus E[W]$ that is incident on some vertex in W. Let this edge be $e = \{u, v_i\}$. Then the walk

$$u e v_i e_i \dots e_{l-1} v_l e_0 v_0 e_1 \dots e_{i-1} v_i$$

is longer than W, a contradiction.

1.3 Probability

Example. Suppose we flip two fair coins. What is the probability that both tosses give heads given that one of the flips results in heads? What is the probability that both tosses give heads given that the first coin results in heads?

Solution. We consider the following events to answer the question.

A: event that both flips give heads.
B: event that one of the flips gives heads.
C: event that the first coin flip gives heads.

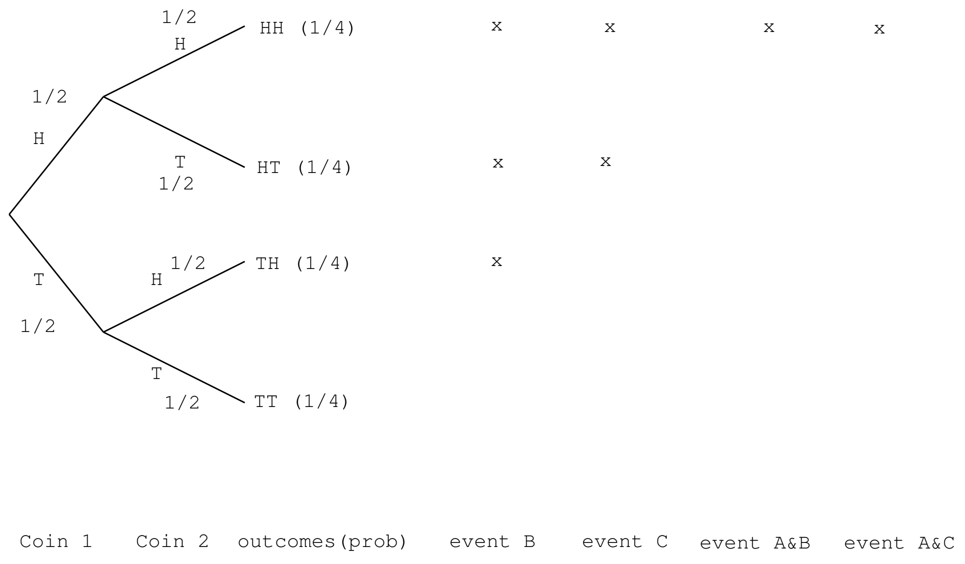
Let's first calculate $\Pr[A|B]$.

$$\Pr[A|B] = \frac{\Pr[A \cap B]}{\Pr[B]} = \frac{\Pr[A]}{\Pr[B]} = \frac{1/4}{3/4} = \frac{1}{3}.$$

Similarly we can calculate $\Pr[A|C]$ as follows.

$$\Pr[A|C] = \frac{\Pr[A \cap C]}{\Pr[C]} = \frac{\Pr[A]}{\Pr[C]} = \frac{1/4}{1/2} = \frac{1}{2}.$$

The above analysis also follows from the tree diagram in the figure below.



The Total Probability Theorem. Consider events E and F. Consider a sample point $\omega \in E$. Observe that ω belongs to either F or $\overline{F}$. Thus, the set E is a disjoint union of two sets: $E \cap F$ and $E \cap \overline{F}$. Hence we get

$$\begin{aligned} \Pr[E] &= \Pr[E \cap F] + \Pr[E \cap \overline{F}] \\ &= \Pr[F] \times \Pr[E|F] + \Pr[\overline{F}] \times \Pr[E|\overline{F}] \end{aligned}$$

In general, if $A_1, A_2, \ldots, A_n$ form a partition of the sample space and if $\forall i, \Pr[A_i] > 0$, then for any event B in the same probability space, we have

$$\Pr[B] = \sum_{i=1}^{n} \Pr[A_i \cap B] = \sum_{i=1}^{n} \Pr[A_i] \times \Pr[B|A_i]$$

Example. A medical test for a certain condition has arrived in the market. According to the case studies, when the test is performed on an affected person, the test comes up positive 95% of the times and yields a "false negative" 5% of the times. When the test is performed on a person not suffering from the medical condition the test comes up negative in 99% of the cases and yields a "false positive" in 1% of the cases. If 0.5% of the population actually have the condition, what is the probability that the person has the condition given that the test is positive?

Solution. We will consider the following events to answer the question.

C: event that the person tested has the medical condition.
$\overline{C}$: event that the person tested does not have the condition.
P: event that the person tested positive.

We are interested in $\Pr[C|P]$. From the definition of conditional probability and the total probability theorem we get

$$\begin{aligned}
\Pr[C|P] &= \frac{\Pr[C \cap P]}{\Pr[P]} \\
&= \frac{\Pr[C]\Pr[P|C]}{\Pr[P \cap C] + \Pr[P \cap \overline{C}]} \\
&= \frac{\Pr[C]\Pr[P|C]}{\Pr[C]\Pr[P|C] + \Pr[\overline{C}]\Pr[P|\overline{C}]} \\
&= \frac{0.005 \times 0.95}{0.005 \times 0.95 + 0.995 \times 0.01} \\
&= 0.323
\end{aligned}$$

This result means that 32.3% of the people who are tested positive actually suffer from the condition!

1.4 Linearity of Expectation

One of the most important properties of expectation that simplifies its computation is the *linearity of expectation.* By this property, the expectation of the sum of random variables equals the sum of their expectations. This is given formally in the following theorem.

Theorem. For any finite collection of random variables $X_1, X_2, \ldots, X_n$,

$$\mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i]$$

Example. Suppose that n people leave their hats at the hat check. If the hats are randomly returned what is the expected number of people that get their own hat back?

Solution. Let X be the random variable that denotes the number of people who get their own hat back. Let $X_i, 1 \leq i \leq n$, be the random variable that is 1 if the ith person gets his/her own hat back and 0 otherwise. Clearly,

$$X = X_1 + X_2 + X_3 + \ldots + X_n$$

By linearity of expectation we get

$$\mathbf{E}[X] = \sum_{i=1}^{n} \mathbf{E}[X_i] = \sum_{i=1}^{n} \frac{(n-1)!}{n!} = n \times \frac{1}{n} = 1$$

Example. The following pseudo-code computes the minimum of n distinct numbers that are stored in an array A. What is the expected number of times that the variable *min* is assigned a value if the array A is a random permutation of the n elements?

```
FindMin($A, n$)
    $min \leftarrow A[1]$
    for $i \leftarrow 2$ to $n$ do
        if ($A[i] < min$) then
            $min = A[i]$
    return $min$
```

Solution. Let X be the random variable denoting the number of times that *min* is assigned a value. We want to calculate $\mathbf{E}[X]$. Let X_i be the random variable that is 1 if *min* is assigned $A[i]$ and 0 otherwise. Clearly,

$$X = X_1 + X_2 + X_3 + \cdots + X_n$$

Using the linearity of expectation we get

$$\begin{aligned}\mathbf{E}[X] &= \sum_{i=1}^{n} \mathbf{E}[X_i] \\ &= \sum_{i=1}^{n} \Pr[X_i = 1]\end{aligned}$$

Note that $\Pr[X_i = 1]$ is the probability that $A[i]$ contains the smallest element among the elements $A[1], A[2], \ldots, A[i]$. Since the smallest of these elements is equally likely to be in any of the first i locations, we have $\Pr[X_i = 1] = \frac{1}{i}$.

1.5 Probability Distributions

Tossing a coin is an experiment with exactly two outcomes: heads ("success") with a probability of, say p, and tails ("failure") with a probability of $1 - p$. Such an experiment is called a *Bernoulli trial.* Let Y be a random variable that is 1 if the experiment succeeds and is 0 otherwise. Y is called a *Bernoulli* or an *indicator* random variable. For such a variable we have

$$\mathbf{E}[Y] = p \cdot 1 + (1 - p) \cdot 0 = p = \Pr[Y = 1]$$

Thus for a fair coin if we consider heads as "success" then the expected value of the corresponding indicator random variable is 1/2.

A sequence of Bernoulli trials means that the trials are independent and each has a probability p of success. We will study two important distributions that arise from Bernoulli trials: the *geometric distribution* and the *binomial distribution.*

The Geometric Distribution

Consider the following question. Suppose we have a biased coin with heads probability p that we flip repeatedly until it lands on heads. What is the distribution of the number of flips? This is an example of a *geometric distribution.* It arises in situations where we perform a sequence of independent trials until the first success where each trial succeeds with a probability p.

Note that the sample space Ω consists of all sequences that end in H and have exactly one H. That is

$$\Omega = \{H, TH, TTH, TTTH, TTTTH, \ldots\}$$

For any $\omega \in \Omega$ of length i, $\Pr[\omega] = (1-p)^{i-1}p$.

Definition. A *geometric random variable* X with parameter p is given by the following distribution for $i = 1, 2, \ldots$:

$$\Pr[X = i] = (1-p)^{i-1}p$$

We can verify that the geometric random variable admits a valid probability distribution as follows:

$$\sum_{i=1}^{\infty}(1-p)^{i-1}p = p\sum_{i=1}^{\infty}(1-p)^{i-1} = \frac{p}{1-p}\sum_{i=1}^{\infty}(1-p)^{i} = \frac{p}{1-p}\cdot\frac{1-p}{1-(1-p)} = 1$$

Note that to obtain the second-last term we have used the fact that $\sum_{i=1}^{\infty} c^i = \frac{c}{1-c}$, $|c| < 1$.

Let's now calculate the expectation of a geometric random variable, X. We can do this in several ways. One way is to use the definition of expectation.

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{i=0}^{\infty} i \Pr[X = i] \\
&= \sum_{i=0}^{\infty} i(1-p)^{i-1}p \\
&= \frac{p}{1-p}\sum_{i=0}^{\infty} i(1-p)^{i} \\
&= \left(\frac{p}{1-p}\right)\left(\frac{1-p}{(1-(1-p))^2}\right) \qquad \left(\because \sum_{i=0}^{\infty} kx^k = \frac{x}{(1-x)^2}, \text{ for } |x| < 1.\right) \\
&= \left(\frac{p}{1-p}\right)\left(\frac{1-p}{p^2}\right) \\
&= \frac{1}{p}
\end{aligned}$$

Another way to compute the expectation is to note that X is a random variable that takes on non-negative values. From a theorem proved in last class we know that if X takes on only non-negative values then

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i]$$

Using this result we can calculate the expectation of the geometric random variable X. For the geometric random variable X with parameter p,

$$\Pr[X \geq i] = \sum_{j=i}^{\infty}(1-p)^{j-1}p = (1-p)^{i-1}p\sum_{j=0}^{\infty}(1-p)^{j} = (1-p)^{i-1}p \times \frac{1}{1-(1-p)} = (1-p)^{i-1}$$

Therefore

$$\mathbf{E}[X] = \sum_{i=1}^{\infty}\Pr[X \geq i] = \sum_{i=1}^{\infty}(1-p)^{i-1} = \frac{1}{1-p}\sum_{i=1}^{\infty}(1-p)^{i} = \frac{1}{1-p}\cdot\frac{1-p}{1-(1-p)} = \frac{1}{p}$$

Binomial Distributions

Consider an experiment in which we perform a sequence of n coin flips in which the probability of obtaining heads is p. How many flips result in heads?

If X denotes the number of heads that appear then

$$\Pr[X = j] = \binom{n}{j}p^j(1-p)^{n-j}$$

Definition. A *binomial* random variable X with parameters n and p is defined by the following probability distribution on $j = 0, 1, 2, \ldots, n$:

$$\Pr[X = j] = \binom{n}{j}p^j(1-p)^{n-j}$$

We can verify that the above is a valid probability distribution using the binomial theorem as follows

$$\sum_{j=0}^{n}\binom{n}{j}p^j(1-p)^{n-j} = (p+(1-p))^n = 1$$

What is the expectation of a binomial random variable X? We can calculate $\mathbf{E}[X]$ is two ways. We first calculate it directly from the definition.

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{j=0}^{n} j\binom{n}{j}p^j(1-p)^{n-j} \\
&= \sum_{j=0}^{n} j\frac{n!}{j!(n-j)!}p^j(1-p)^{n-j} \\
&= \sum_{j=1}^{n} j\frac{n!}{j!(n-j)!}p^j(1-p)^{n-j} \\
&= \sum_{j=1}^{n} \frac{n!}{(j-1)!(n-j)!}p^j(1-p)^{n-j} \\
&= np\sum_{j=1}^{n} \frac{(n-1)!}{(j-1)!((n-1)-(j-1))!}p^{j-1}(1-p)^{(n-1)-(j-1)} \\
&= np\sum_{k=0}^{n-1} \frac{(n-1)!}{k!((n-1)-k)!}p^{k}(1-p)^{(n-1)-k} \\
&= np\sum_{k=0}^{n-1} \binom{n-1}{k}p^{k}(1-p)^{(n-1)-k} \\
&= np
\end{aligned}$$

The last equation follows from the binomial expansion of $(p+(1-p))^{n-1}$.

We can obtain the result in a much simpler way by using the linearity of expectation. Let $X_i, 1 \le i \le n$ be the indicator random variable that is 1 if the ith flip results in heads and is 0 otherwise. We have $X = \sum_{i=1}^{n} X_i$. By the lineartity of expectation we have

$$\mathbf{E}[X] = \sum_{i=1}^{n}\mathbf{E}[X_i] = \sum_{i=1}^{n} p = np$$

What is the variance of the binomial random variable X? Since $X = \sum_{i=1}^{n} X_i$, and $X_1, X_2, \ldots, X_n$ are independent we have

$$\begin{aligned}
\mathrm{Var}[X] &= \sum_{i=1}^{n}\mathrm{Var}[X_i] \\
&= \sum_{i=1}^{n}\mathbf{E}[X_i^2] - \mathbf{E}[X_i]^2 \\
&= \sum_{i=1}^{n}(p-p^2) \\
&= np(1-p)
\end{aligned}$$

Examples

Example. Consider a sequence of n tosses of a fair coin. Define a "run" to be a sequence of consecutive tosses that all produce the same result (i.e. heads or tails). For example, if $n = 7$, then the sequence of outcomes

$$H\ H\ H\ T\ T\ H\ T$$

has four runs, namely $(H\ H\ H)$, $(T\ T)$, (H) , and (T). Given a sequence of n tosses of a fair coin, what is the expected number of runs you will see?

Solution. Let X be the random variable denoting the number of runs we see. We wish to compute $\mathbf{E}[X]$. Define the indicator random variables

$$X_i = \begin{cases} 1 & \text{there is a run beginning at index } i \\ 0 & \text{otherwise} \end{cases}$$

Note that $X_1 = 1$ since the first outcome always begins a new run. Moreover, we see that

$$X = X_1 + ... + X_n = \sum_{i=1}^{n} X_i$$

and so by the linearity of expectation,

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i]$$

Recall that for an indicator random variable Y, $\mathbf{E}[Y] = 1 \cdot P(Y = 1) + 0 \cdot P(Y = 0) = P(Y = 1)$, hence for each i, we have

$$\mathbf{E}[X_i] = P(X_i = 1)$$

Now a run begins at index i whenever the $i-1$-th outcome and the i-th outcome are different. For $i > 1$, this occurs with probability $\frac{1}{2}$. Hence

$$\mathbf{E}[X_i] = \begin{cases} 1 & i = 1 \\ \frac{1}{2} & i > 1 \end{cases}$$

Thus

$$\mathbf{E}[X] = \sum_{i=1}^{n} \mathbf{E}[X_i] = 1 + \sum_{i=2}^{n} \mathbf{E}[X_i] = 1 + \sum_{i=2}^{n} \frac{1}{2} = 1 + \frac{n-1}{2} = \frac{n+1}{2}$$

Example. Find the expected length of an arbitrary run.

Solution. Let X be the length of an arbitrary run. The run can either be a run of heads or a run of tails. We thus **condition** on this case: let Y be the random variable that is 1 if the run is a run of heads and 0 if it is a run of tails. Then $P(Y = 1) = P(Y = 0) = \frac{1}{2}$. Computing the expectation of X by conditioning on Y gives:

$$\mathbf{E}[X] = \mathbf{E}[X|Y = 1]P(Y = 1) + \mathbf{E}[X|Y = 0]P(Y = 0)$$

Given that $Y = 1$, i.e. that the run is a run of heads, X is a geometric random variable with parameter $p = \frac{1}{2}$, since finding the length of the run starting with a head is equivalent to finding the number of coin tosses we make until seeing the first tail. Hence $\mathbf{E}[X|Y = 1] = 2$, since the expected value of a geometric random variable with parameter p is $\frac{1}{p}$. Symmetrically, given that $Y = 0$, X is also a geometric random variable with parameter $p = \frac{1}{2}$. Hence $\mathbf{E}[X|Y = 0] = 2$ as well. This gives:

$$\mathbf{E}[X] = 2 \cdot \frac{1}{2} + 2 \cdot \frac{1}{2} = 2$$

Note that because of the symmetry, we could have assumed WLOG that the run started with heads and proceeded from there. However, in the case of a biased coin, we don't have symmetry and thus conditioning is the way to go.

Example. We are trying to collect n different coupons that can be obtained by buying cereal boxes. The objective is to collect at least one coupon of each of the n types. Assume that each cereal box contains exactly one coupon and any of the n coupons is equally likely to occur. How many cereal boxes do we expect to buy to collect at least one coupon of each type?

Solution. Let the random variable X denote the number of cereal boxes bought until we have at least one coupon of each type. We want to compute $\mathbf{E}[X]$. Let X_i be the random variable denoting the number of boxes bought to get the ith new coupon. Clearly,

$$X = X_1 + X_2 + X_3 + \ldots + X_n$$

Using the linearity of expectation we have

$$\mathbf{E}[X] = \mathbf{E}[X_1] + \mathbf{E}[X_2] + \mathbf{E}[X_3] + \ldots + \mathbf{E}[X_n]$$

What is the distribution of random variable X_i? Observe that the probability of obtaining the ith new coupon is given by

$$p_i = \frac{n-(i-1)}{n} = \frac{n-i+1}{n}$$

Thus the random variable $X_i, 1 \leq i \leq n$ is a geometric random variable with parameter p_i.

$$\mathbf{E}[X_i] = \frac{1}{p_i} = \frac{n}{n-i+1}$$

Applying linearity of expectation, we get

$$\mathbf{E}[X] = \frac{n}{n} + \frac{n}{n-1} + \frac{n}{n-1} + \cdots + \frac{n}{2} + \frac{n}{1} = n\sum_{i=1}^{n} \frac{1}{i}$$

The summation $\sum_{i=1}^{n} \frac{1}{i}$ is known as the *harmonic number* $H(n)$ and $H(n) = \ln n + c$, for some constant $c < 1$.

Hence the expected number of boxes needed to collect n coupons is about $nH(n) < n(\ln n + 1)$.

Gale-Shapley Stable Matching | 2

2.1 Background and Intuition

Many problems in the real world involve pairing up two groups of people or things in such a way that everyone is "happy." For example, one may consider matching applicants to open job positions, performers to gigs, patients to hospital beds, and so on. What does it mean for everyone to be happy? Consider the following scenario: you and your friend are using career services to apply to internships at companies A and B. You really want to work for A, but your friend wants to work for B. On the other hand, both company A and B think you would be a much better employee than your friend.

Unfortunately, due to some logistical mix-up at the career services center, you are matched with company B and your friend is matched with company A. Clearly, you are unhappy, as is company A. We call this situation an *instability*. It is unstable in the sense that there's nothing stopping company A from firing your friend and offering you the job. You would of course accept (since company A was your top choice), but now your friend has no job, and company B has no employee. Moreover, it is not immediately clear that just forcing company B to hire your friend solves the problem: what if multiple companies/applicants are involved? The situation could spiral out of control.

One of the basic problems with the above process is that it is not self-enforcing: if people are allowed to act in their self-interest, then it risks breaking down.

As another example, suppose that career services instead matched you with A and your friend with B. In this case, you and your friend are happy, as is company A. Unfortunately, company B is unhappy. However, in this case there is no risk of spiraling out of control: company B may try to reach out to you to make you an offer, but you are happy where you are. In this case, the matching is *stable*.

Gale and Shapley studied scenarios like these and asked the following question: Given a set of preferences among employers and applicants, is it possible to assign applicants to employers so that for every employer E and every applicant A who is not scheduled to work for E, at least one of the following is true:

(1) E prefers every one of its accepted applicants to A
(2) A prefers her current situation over working for employer E

If this holds, the outcome is *stable*. That is, individual self-interest will prevent any applicant/employer deal from being made behind the scenes.

2.2 Formulating the Problem

In first analyzing this question, we will make some simplifications. The employer/applicant scenario has some significant asymmetries. For example:

- Employers typically look for multiple applicants, while applicants seek only a single employer
- Each applicant may not apply to every company
- The number of employers or company may differ

These notes were adapted from Kleinberg and Tardos' *Algorithm Design*

Here is an online tool, developed by Robin Tan, former head TA, for you to explore and visualize the Gale-Shapley algorithm: gale-shapley.com

We will make the following assumptions: each of n applicants applies to each of n companies, and each company wants a *single* applicant. Gale and Shapley when they worked on this problem phrased it in terms of a so-called stable marriage: each of n heterosexual men and n heterosexual women will be paired up for marriage.

Formally:

Consider a set $M = \{m_1, ..., m_n\}$ of n men and a set $W = \{w_1, ..., w_n\}$ of n women. Let $M \times W$ denote the set of all possible ordered pairs of the form (m, w) where $m \in M$ and $w \in W$.

Definition. A matching S is a subset of ordered pairs in $M \times W$ such that each member of M and each member of W appears in at most one pair in S.

A matching S' is called a perfect matching if it is a matching and has the property that each member of M and each member of W appears in *exactly* one pair of S'.

Intuitively, a perfect matching corresponds to a way of pairing off the men and the women in such a way that everyone ends up married to somebody of the opposite gender, and nobody is married to more than one person. There is neither singlehood nor polygamy.

Now we can add the notion of *preferences* to this setting.

Definition. Each man $m \in M$ ranks all the women. There are no ties allowed. We say that m prefers w to w' if m ranks w higher than w'. We will refer to the ordered ranking of m as his preference list.

Each woman, analogously, ranks all the men.

Note: There are no ties allowed in any man or woman's preference lists. As we go through the remainder of the analysis, you should think about which statements require this constraint and which do not.

Now that we've formulated the problem, how do we express the problem we faced in the introduction in this new notation? Using the motivating example as guidance, we should be worried about the following situation: there are two pairs, (m, w) and (m', w') in S such that m prefers w' to w and w' prefers m to m'. In this case, there is nothing to stop m and w' from abandoning their current partners and heading off together: this set of marriages is not self-enforcing.

Definition. Using the notation above, the pair (m, w') is called an instability with respect to the matching S. That is, (m, w') does not belong to S, but each of m and w' prefers the other to their partner in S.

Definition. A matching S is stable if:

1. it is perfect
2. there are no instabilities with respect to S

Our goal, then, is to find a stable matching. Two questions immediately spring to mind:

1. Does there exist a stable matching for every set of preference lists?
2. Given a set of preference lists, can we efficiently construct a stable matching if there is one?

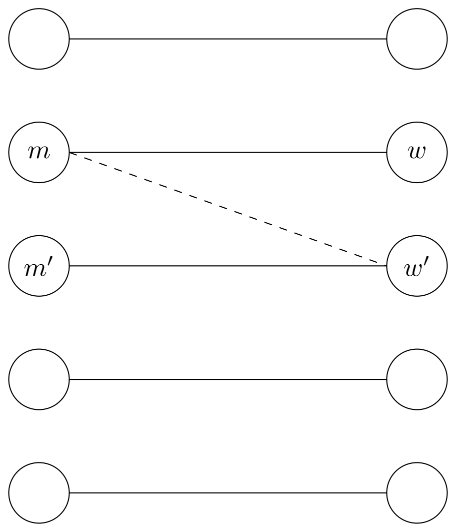


Figure 2.1: Perfect matching S with instability (m, w')

2.3 Examples

To illustrate the definitions from the previous section, consider the following two very simple instances of the stable matching problem:

First, suppose we have a set of two men, $M = \{m, m'\}$ and a set of two women $W = \{w, w'\}$. The preference lists are as follows:

m prefers w to w'
m' prefers w to w'
w prefers m to m'
w' prefers m to m'

Intuitively, this case represents complete agreement: the men agree on the order of the women and the women agree on the order of the men. There is a unique stable matching in this case, consisting of the pairs (m, w) and (m', w'). The other perfect matching, consisting of the pairs (m', w) and (m, w') is NOT stable: the pair (m, w) would form an instability with respect to this matching.

Here's a more interesting example: suppose the preferences are:

m prefers w to w'
m' prefers w' to w
w prefers m' to m
w' prefers m to m'

In this case, the two men's preferences mesh perfectly with each other (they rank different women first), and the two women's preferences likewise mesh perfectly with each other. However, the men's preferences clash completely with the women's preferences.

In this second example, there are two different stable matchings. The matching consisting of the pair (m, w) and (m', w') is stable because both men are as happy as possible and so neither would leave their matched partner. But the matching consisting of the pairs (m', w) and (m, w') is also stable, for the complementary reason that both women are as happy as possible. This is an important point to remember: *it is possible for an instance to have more than one stable matching*

2.4 Designing an Algorithm

We still have two questions to answer, namely is there always a stable matching and how do we find one if/when it exists? The following provides the answer:

Proposition 1. *Given a set of n men and a set of n women, there exists a stable matching for every set of preference lists among the men and women.*

We will prove this claim by giving an algorithm that takes the preference lists and constructs a stable matching. This will answer both of the above questions simultaneously.

Before constructing the algorithm, let's summarize the basic ideas that are relevant to the problem:

- Initially, everyone is unmarried. Suppose an unmarried man m chooses the woman w who ranks highest on his preference list and *proposes* to her. Can we declare immediately that (m, w) will be one of the pairs in our final stable matching? Not necessarily: at some point in the future, a man m' whom w prefers more than m may propose to her. On the other hand, it may be dangerous for w to reject m right away: she may never receive a proposal from someone she ranks as highly as m. So a natural idea would be to have the pair (m, w) enter into an intermediate state—*engagement*.
- Suppose we are now at a state in which some men and women are *free*—not engaged—and some are engaged. The next step could look like this: An arbitrary man m chooses the highest-ranked woman w to whom he has not yet proposed, and he proposes to her. If w is also free, then m and w become engaged. Otherwise, w is already engaged to some other man m'. In this case, she determines which of m or m' ranks higher on her preference list: this man becomes engaged to w and the other man becomes free.
- Finally, the algorithm will terminate when no one is free. At this moment, all engagements are declared final, and the resulting perfect matching is returned.

Here's a more concise description of the *Gale-Shapley Algorithm*:

Gale-Shapley Algorithm for Stable Matching

Input: A set of n men, M, a set of n women, W, and the associated preference lists

Output: A stable matching

```
Initially all $m \in M$ and $w \in W$ are free

While there is a free man $m$ who hasn't proposed to every woman
    Choose such a man $m$
    Let $w$ be the highest ranked woman in $m$'s preference list
        to whom he has not yet proposed

    If $w$ is free then
        $(m, w)$ become engaged

    Else $w$ is currently engaged to $m'$
        If $w$ prefers $m'$ to $m$ then
            $m$ remains free
        Else $w$ prefers $m$ to $m'$
```

```
            $(m, w)$ become engaged
            $m'$ becomes free

Return the set $S$ of engaged pairs
```

2.5 Runtime of the GS Algorithm

Before proving the correctness of the algorithm, we first prove its runtime. That is, we will answer the question: how many iterations are needed for the algorithm to terminate?

Proposition 2. *The GS algorithm terminates after at most n^2 iterations of the `while` loop*

Proof. A useful strategy for upper-bounding the running time of an algorithm is to find a measure of *progress*. That is, we want to find a precise way of saying that each step taken by the algorithm brings it closer to termination.

In the GS algorithm, each iteration consists of some man proposing (for the only time) to a woman he has never proposed to before. So if we let $\mathcal{P}(t)$ denote the set of pairs (m, w) such that m has proposed to w by the end of iteration t, we see that for all t, the size of $\mathcal{P}(t + 1)$ is strictly greater than the size of $\mathcal{P}(t)$. But there are only n^2 possible pairs of men and women in total, so that value of $\mathcal{P}(\cdot)$ can increase at most n^2 times over the course of the algorithm. Thus there can be at most n^2 iterations. $\square$

An important note about the above algorithm: there are many quantities that would not have worked well as a *progress measure* for the GS algorithm, since they need not strictly increase in each iteration. For example, the number of free individuals could remain constant from one iteration to the next, as could the number of engaged pairs. Thus these quantities could not be used directly in giving an upper bound on the maximum possible number of iterations, in the style of the above proof.

2.6 Correctness of the GS Algorithm

While we showed above that the GS algorithm terminates, it is not obvious at all that this algorithm returns a stable matching, or even a perfect matching. We will prove through a series of intermediate facts that it does indeed return a stable matching.

Consider the view of a woman w during the execution of the GS algorithm. For a while, no one has proposed to her, and she is free. Then a man m may propose to her, and she becomes engaged. As time goes on, she may receive additional proposals, accepting those that increase the rank of her partner. Hence:

Lemma 1. *w remains engaged from the point at which she receives her first proposal. The sequence of partners to which she is engaged gets better and better (in terms of her preference list)*

The view of a man m during the execution of the algorithm is very different. He is free until he proposes to the highest ranked woman on his list. At this point, he may or may not become engaged. As time goes on, he may alternate between being free and being engaged, however it is easy to see that:

Lemma 2. *The sequence of women to whom m proposes gets worse and worse (in terms of his preference list)*

Now, let us establish that the set S returned at the termination of the algorithm is in fact a perfect matching. Why is this not immediately obvious? Essentially, we have to show that no man can "fall off" the end of his preference list. That is, the only way for the `while` loop to exit is for there to be no free man. In this case, the set of engaged couple would indeed be a perfect matching. The following lemma and its proof establish this claim.

Lemma 3. *If m is free at some point in the execution of the algorithm, then there is a woman to whom he has not yet proposed*

Proof. Seeking contradiction, suppose there comes a point when m is free but has already proposed to every woman. By **Lemma 1**, each of the n women is engaged at this point in time. Since the set of engaged pairs forms a matching, there must also be n engaged men at this point in time. But there are only n men in total, and m is not engaged, so this is a contradiction. □

Lemma 4. *The set S returned at termination is a perfect matching*

Proof. The set of engaged pairs always forms a matching. Let us suppose for the sake of contradiction that the algorithm terminates with a free man m. At termination, it must be the case that m has already proposed to every woman, for otherwise the `while` loop would not have exited. But this contradicts **Lemma 3**, which says that there cannot be a free man who has proposed to every woman. □

Now we have the tools to prove the main property of the algorithm, namely that it results in a stable matching.

Proposition 3. *Consider an execution of the GS algorithm that returns a set of pairs S. The set S is a stable matching.*

Proof. We know by **Lemma 4** that S is a perfect matching. Thus, it suffices to show S has no instabilities. Seeking contradiction, suppose we had the following instability: there are two pairs (m, w) and (m', w') in S such that

- m prefers w' to w and
- w' prefers m to m'

In the execution of the algorithm that produced S, the last proposal of m was, by definition, to w. Now we ask: Did m propose to w' at some earlier point in this execution? If he didn't, then w must occur higher than w' on m's preference list, contradicting out assumption that m prefers w' to w. If he did, then he was rejected by w' in favor of some other man m'' whom w' prefers to m. m' is the final partner of w', so either $m'' = m'$ or by **Lemma 1**, w' prefers her final partner m' to m''. Either way, this contradicts our assumption that w' prefers m to m'.

It follows that S is a stable matching. □

2.7 Extensions

Now that we've shown the GS algorithm constructs a stable matching, we can now consider some further questions about the behavior of the GS algorithm and its relation to the properties of different stable matchings.

To begin, recall that we saw an example earlier in which there could be multiple stable matchings. The preference lists in this example were as follows:

m prefers w to w'
m' prefers w' to w
w prefers m' to m
w' prefers m to m'

Now, in any execution of the GS algorithm, m will become engaged to w, m' will become engaged to w' (perhaps in the other order), and things will stop there. Thus the *other* stable matching, consisting of the pairs (m', w) and (m, w') is not attainable from an execution of the GS algorithm in which the men propose. On the other hand, it would be reached if we ran a version of the algorithm in which the women propose. You can imagine that in larger examples (i.e. with $n > 2$), it is possible to have an even larger collection of possible stable matching, many of them not achievable by the GS algorithm.

This example shows a certain "unfairness" in the GS algorithm favoring men. If the men's preferences mesh perfectly (they all list different women as their first choice), then in all runs of the GS algorithm, all men end up matched with their first choice, independent of the preferences of the women. If the women's preferences clash completely with the men's preferences (as was the case in this example), then the resulting stable matching is as bad as possible for the women. So this simple set of preferences lists compactly summarizes a world in which *someone* is destined to end up unhappy: woman are unhappy if men propose, and men are unhappy if woman propose.

Let's now see if we can generalize this notion of "unfairness".

Note that the above example illustrates that the GS algorithm is actually underspecified: as long as there is a free man, we are allowed to choose *any* free man to make the next proposal. Different choices specify different executions of the algorithm. This is why, to be careful, we stated in **Proposition 3** to "Consider an execution of the GS algorithm that returns a set of pairs S" rather than "Consider the set S returned by the GS algorithm."

This raises a natural question: Do all executions of the GS algorithm yield the same matching? It turns out, the answer is "Yes."

All Executions Yield the Same Matching

It turns out that the easiest and most informative way to prove this claim is the uniquely characterize the matching that is obtained from the GS algorithm and then show that all executions result in the matching with this characterization.

First, let us introduce some relevant definitions:

Definition. A woman w is a <u>valid partner</u> of a man m if there is a stable matching that contains the pair (m, w).

A woman w is the best valid partner of a man m, denoted $best(m)$, if w is a valid partner of m and no woman whom m ranks higher than w is a valid partner of his.

Define the set

$$S^* = \{(m, best(m)) \mid m \in M\}$$

That is, the set where each man is paired with his best valid partner. We have the following result:

Proposition 4. *Every execution of the GS algorithm results in the set S^*.*

Before proving this, let's appreciate this statement. First of all, it is not obvious at all that S^* is even a matching, much less a stable matching. After all, why couldn't two men have the same best valid partner? Second, the result shows that the GS algorithm gives the best possible outcome for every man simultaneously: there is no stable matching in which any of the men could have hoped to do better. Lastly, it answers the question above by showing that the order of proposals in the GS algorithm has absolutely no effect on the final outcome.

Despite all this, the proof is not so difficult:

Proof. Seeking contradiction, suppose that some execution ε of the GS algorithm results in a matching S in which some man is paired with a woman who is not his best valid partner. Since men propose in decreasing order of preference, this means that some man is rejected by a valid partner during the execution ε of the algorithm. So consider the first moment during the execution ε in which some man, say m, is rejected by a valid partner w. Again, since men propose in decreasing order of preference, and since this is the first time such a rejection has occurred, it must be that w is m's best valid partner, i.e. $w = best(m)$.

The rejection of m by w may have happened either because m proposed and was turned down in favor of w's existing partner, or because w broke her engagement to m in favor of a better proposal. But either way, at this moment, w forms or continues an engagement with a man m' whom she prefers to m.

Since w is a valid partner of m, there exists some stable matching S' containing the pair (m, w). Now we ask: Who is m' paired with in the matching S'? Suppose it is a woman $w' \neq w$.

Since the rejection of m by w was the first rejection of a man by a valid partner in the execution ε, it must be that m' had not been rejected by any valid partner at the point in ε when he became engaged to w. Since he proposed in decreasing order of preference, and since w' is clearly a valid partner of m', it must be that m' prefers w to w'. But we have already seen that w prefers m' to m, for in execution ε she rejected m in favor of m'. Since $(m', w) \notin S'$, it follows that (m', w) is an instability in S'.

This contradicts our claim that S' is stable, and hence contradicts our initial assumption. $\square$

So for the men, the GS algorithm is ideal. Unfortunately the same cannot be said for the women.

Definition. m is a valid partner of a woman w if there is a stable matching that contains the pair (m, w).

m is the worst valid partner of w, denoted $worst(w)$, if m is a valid partner of w and no man whom w ranks lower than m is a valid partner of hers.

Proposition 5. *In the stable matching S^*, each woman is paired with her worst valid partner. That is,*

$$S^* = \{(m, best(m) \mid m \in M\} = \{(worst(w), w) \mid w \in W\}$$

Proof. Seeking contradiction, suppose there were a pair (m, w) in S^* such that m is not the worst valid partner of w. Then there is a stable matching S' in which w is paired with a man m' whom she likes less than m. In S', m is paired with a woman $w' \neq w$. Since w is the best valid partner of m, and w' is a valid partner of m, we see that m prefers w to w'.

But from this it follows that (m, w) is an instability in S', contradicting the claim that S' is stable, and hence contradicting our initial assumption. □

Thus, we find that the simple example above, in which the men's preferences clashed with the women's, hinted at a very general phenomenon: for any input, the side that does the proposing in the GS algorithm ends up with the best possible stable matching (from their perspective), while the side that does not do the proposing correspondingly ends up with the worst possible stable matching.

Note: The above analysis was performed under the assumption that there were no ties in any of the preference lists, and that the number of men and women were equal. What parts of the analysis would change if these assumptions didn't hold?

3 Greatest Common Divisor

3.1 Definitions

We first begin by introducing some intuitive definitions:

Definition. Let $a, b \in \mathbb{Z}$. We call an integer d a common divisor of a and b provided $d \mid a$ and $d \mid b$.

For example, the common divisors of 24 and 30 are $-6, -3, -2, -1, 1, 2, 3$, and 6.

Definition. Let $a, b \in \mathbb{Z}$. We call an integer d the greatest common divisor of a and b provided

(1) d is a common divisor of a and b AND
(2) if d' is *any* common divisor of a and b, then $d' \le d$

We generally denote the greatest common divisor of a and b as $\gcd(a, b)$.

Going back to the example above, we have $\gcd(24, 30) = 6$. Note also that $\gcd(-24, -30) = 6$. If two numbers have a gcd, then it is unique. This justifies the use of the phrase "*the* greatest common divisor."

3.2 Calculating the GCD

A naive way to compute the GCD of two numbers is to simply list all of their common factors and choose the largest. This brute force solution gives rise to the following algorithm:

1. Let a and b be two positive integers
2. For every positive integer k from 1 to $\min(a, b)$, see whether $k \mid a$ and $k \mid b$. If so, save that number k in a list l
3. Return the largest number in the list l as $\gcd(a, b)$.

While this algorithm certainly works, its runtime is abysmal. Even for moderately large numbers (for example, $a = 34902$ and $b = 34299883$), the algorithm needs to perform many, many divisions.

A faster and more clever algorithm for finding the GCD of two numbers a and b was first described in Euclid's *Elements* over 2000 years ago. Not only is it extremely fast, but it is also easy to implement as a computer program.

The central idea behind the algorithm is based on the following proposition:

Proposition 1. *Let a and b be positive integers and let $c = a \mod b$. Then*

$$\gcd(a, b) = \gcd(b, c)$$

In other words, for positive integers a and b, we have

$$\gcd(a, b) = \gcd(b, a \mod b)$$

These notes were adapted from Scheinerman's *Mathematics: A Discrete Introduction*

Proof. As above, let a and b be positive integers and let $c = a \mod b$. This means that $a = qb + c$ where $0 \leq c < b$. Let $d = \gcd(a, b)$ and $e = \gcd(b, c)$. We must show that $d = e$. To do this, we will prove that $d \leq e$ and $e \leq d$.

First we will show that $d \leq e$. Since $d = \gcd(a, b)$, we know that $d \mid a$ and $d \mid b$. Hence, since $c = a - qb$ we must have $d \mid c$. Thus d is a common divisor of b and c. Since e is the *greatest* common divisor of b and c, we have $d \leq e$.

Now, we will show that $e \leq d$. Since $e = \gcd(b, c)$ we know $e \mid b$ and $e \mid c$. Since $a = qb + c$, we must have that $e \mid a$ as well. Since $e \mid a$ and $e \mid b$, e is a common divisor of a and b. However, since d is the *greatest* common divisor of a and b, this tells us that $e \leq d$.

We have shown that $d \leq e$ and $e \leq d$, and hence $d = e$. That is, $\gcd(a, b) = \gcd(b, c)$. □

To illustrate how the above proposition enables us to calculate GCDs efficiently, we compute $\gcd(689, 234)$. The brute force algorithm described earlier would have us try all possible common divisors from 1 to 234 and select the largest, requiring hundreds of divisions!

Instead, we can use the above proposition to make life easier. Let $a = 689$ and $b = 234$. Performing one division, we see that $c = 689 \mod 234 = 221$. The proposition tells us that to find $\gcd(689, 234)$, it is sufficient to find $\gcd(234, 221)$. Let's record this step:

$$689 \mod 234 = 221 \quad \Rightarrow \quad \gcd(689, 234) = \gcd(234, 221)$$

Now all we have to do is calculate $\gcd(234, 221)$. Using the same idea now with $a = 234$ and $b = 221$, we have $c = 234 \mod 221 = 13$. Note that so far we have only performed two divisions. Let's record this step:

$$234 \mod 221 = 13 \quad \Rightarrow \quad \gcd(234, 221) = \gcd(221, 13)$$

Now the problem is reduced to $\gcd(221, 13)$. Note that the numbers are significantly smaller than the original 689 and 234. To complete the computation, we perform a third and final division to find that $221 \mod 13 = 0$, i.e. $13 \mid 221$. This tells us that the greatest common divisor of 13 and 221 is 13. Recording this final step gives:

$$221 \mod 13 = 0 \quad \Rightarrow \quad \gcd(221, 13) = 13$$

We are finished! After only three divisions, we have found that

$$\gcd(689, 234) = \gcd(234, 221) = \gcd(221, 13) = 13$$

The steps we just performed are precisely the Euclidean algorithm. Here is a formal, general description:

Euclid's Algorithm for Greatest Common Divisor

Input: Positive integers a and b

Output: $\gcd(a, b)$

```
(1) Let $c = a \mod b$
(2) If $c = 0$, return $b$
(3) Otherwise ($c \neq 0$), calculate and return $\gcd(b, c)$
```

Let's see how the algorithm works for $a_0 = 63$ and $b_0 = 75$.

- The first step calculates $c_0 = a_0 \mod b_0 = 63 \mod 75 = 63$
- Next, since $c_0 \neq 0$, we now compute $\gcd(b_0, c_0) = \gcd(75, 63)$.
- Now we restart the process with $a_1 = 75$ and $b_1 = 63$. We find $c_1 = 75 \mod 63 = 12$. Since $12 \neq 0$, we now compute $\gcd(b_1, c_1) = \gcd(63, 12)$
- We restart again with $a_2 = 63$, $b_2 = 12$. We find $c_2 = 63 \mod 12 = 3$. Since $3 \neq 0$, we now compute $\gcd(b_2, c_2) = \gcd(12, 3)$
- With $a_3 = 12$ and $b_3 = 3$, we find $c_3 = 12 \mod 3 = 0$. Hence we return $b_3 = 3$ and we are finished.

Here is an overview of the calculation in chart form:

a	b	c
63	75	63
75	63	12
63	12	3
12	3	0

Note how the answer is found using only four divisions.

Another way to visualize the computation is by using a list. The first two entries are a and b. Now we extend the list by computing the mod of the last two entries of the list. When we reach 0 we stop. The next-to-last entry of the list is the GCD of a and b. In this example, this process would look like:

```
63 75
63 75 63
63 75 63 12
63 75 63 12 3
63 75 63 12 3 0
```

and so we would return 3, just as before.

3.3 Correctness of Euclid's Algorithm

To prove the correctness of Euclid's algorithm, we will use the "smallest-counterexample" method. This is a proof technique closely related to induction and sometimes useful for proving the correctness of recursive algorithms. It works by assuming the algorithm fails on some input, then considering the *smallest* such input, then proving that, in fact, there must be some smaller input for which the algorithm failed, giving a contradiction.

Proposition 2. *Euclid's algorithm as described above correctly computes* $\gcd(a, b)$ *for any positive integers* a *and* b.

Proof. Suppose for the sake of contradiction that Euclid's algorithm failed to correctly compute the GCD. Let a and b be two positive integers for which the algorithm failed, and let them be such that $a + b$ is as small as possible.

It may be the case that $a < b$. If this is so, then the first pass of Euclid's algorithm will simply interchange the values of a and b (as we saw above when computing $\gcd(63, 75)$), since if $a < b$, then $a \mod b = a$ and so $\gcd(b, a \mod b) = \gcd(b, a)$.

Hence, WLOG we may assume $a \geq b$.

The first step of the algorithm is to compute $c = a \mod b$. In the case that $c = 0$, $a \mod b = 0$ which implies $b \mid a$. Since $b > 0$ by assumption, b is clearly the largest divisor of b. Since $b \mid a$, b is the GCD of a and b. Since the algorithm returns b in this case, the algorithm returns the correct result. Thus it must be the case that $c \neq 0$.

Recall, we may write $a = qb + c$ where $0 < c < b$ (the inequality is strict since we now know $c \neq 0$). We also have assumed WLOG that $b \leq a$. Hence,

$$\begin{aligned} & c < b \\ + \quad & b \leq a \\ \Rightarrow \quad & b + c < a + b \end{aligned}$$

Thus, b and c are positive integers with $b + c < a + b$.

Now, if the algorithm correctly computed $\gcd(b, c)$, then Proposition 1 would imply that the algorithm would've also correctly computed $\gcd(a, b)$, which we know isn't the case by our assumption. Hence the algorithm must have incorrectly computed $\gcd(b, c)$. But since $b + c < a + b$, this means the algorithm failed on a pair of integers with sum smaller than $a + b$, contradicting our assumption.

Hence, Euclid's algorithm always returns the greatest common divisor of the positive integers it is given. $\square$

3.4 Runtime of Euclid's Algorithm

How fast is Euclid's algorithm? That is, how many divisions do we have to perform? It turns out that after two rounds of Euclid's algorithm, the integers with which we are working have decreased by at least 50%. The following proposition is the main tool needed to prove this assertion:

Proposition 3. *Let $a, b \in \mathbb{Z}$ with $a \geq b > 0$. Let $c = a \mod b$. Then $c < \frac{a}{2}$.*

Proof. We consider two cases: (1) $a < 2b$ and (2) $a \geq 2b$.

Case (1) $a < 2b$.
We know that $2b > a > 0$ and $a \geq b > 0$, so we have $a > 0$ and $a - b \geq 0$ but $a - 2b < 0$. Hence the quotient when a is divided by b is 1. So the remainder in a divided by b is $c = a - b$. Now rewriting $a < 2b$ as $b > \frac{a}{2}$, we have

$$c = a - b < a - \frac{a}{2} = \frac{a}{2}$$

which is exactly what we wanted.

Case (2) $a \geq 2b$, which can be rewritten $b \leq \frac{a}{2}$
The remainder, upon division of a by b is strictly less than b (this is always true when a and b are positive integers). Hence $c < b$ and so we have

$$c < b \leq \frac{a}{2}$$

In both cases, we found $c < \frac{a}{2}$ $\square$

Now, recall we may assume that we start Euclid's algorithm with $a \geq b$; if not, then the first pass will reverse a and b, and from then on, the numbers come in decreasing order. That is, if the numbers produced by Euclid's algorithm are listed as

$$(a,\ b,\ c,\ d,\ e,\ f, \ldots,\ 0)$$

then assuming $a \geq b$, we have

$$a \geq b \geq c \geq d \geq e \geq f \geq \cdots \geq 0$$

By Proposition 3, the numbers c and d are less than half as large as a and b, respectively. Similarly, the numbers e and f are less than half as large as c and d, respectively, and thus less than one-fourth of a and b respectively. Hence:

Every two steps of Euclid's algorithm decreases the integers with which we are working to less than half their current values

If we begin with (a, b), then after two steps, the numbers are less than $\left(\frac{1}{2}a, \frac{1}{2}b\right)$, and four steps later, less than $\left(\frac{1}{4}a, \frac{1}{4}b\right)$ and six steps later, less than $\left(\frac{1}{8}a, \frac{1}{8}b\right)$, and so on. In general, after $2t$ passes of Euclid's algorithm, the numbers are less than $(2^{-t}a, 2^{-t}b)$.

Euclid's algorithm stops when the second number reaches 0. Since the numbers in Euclid's algorithm are integers, this is the same as when the second number is less than 1. That is, the second number reaches 0 as soon as

$$2^{-t}b \leq 1$$

Taking logs of both sides tells us that $t \geq \log_2 b$. Thus, after at most $2\log_2 b$ passes, the algorithm has completed its work. The algorithm is thus $O(\log b)$.

For a concrete example, how many divisions would Euclid's algorithm perform if a and b were huge numbers (say 1000 digits each)? If $b \approx 10^{1000}$, then the number of steps performed is bounded by

$$2\lg\left(10^{1000}\right) = 2000\lg 10 \approx 2000 \times 3.4 = 6800$$

Compare this with the naive brute-force algorithm.

Insertion Sort | 4

4.1 Insertion Sort

The problem of sorting is defined as:

- **Input**: n integers in array $A[1..n]$
- **Output**: A sorted in increasing order

Here we present INSERTION-SORT, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

We present our pseudocode for INSERTION-SORT, which takes as a parameter an array $A[1..n]$ containing a sequence of length n that is to be sorted. Note that in the code, $n = A.length$. The algorithm sorts the input numbers **in place**. That is, it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time. The input array A contains the sorted output sequence when the INSERTION-SORT procedure is finished.

```
INSERTION-SORT(A):
  for $j \leftarrow 2$ to $A.length$
      $key = A[j]$
      // Insert $A[j]$ into the sorted sequence $A[1..j-1]$
      $i = j - 1$
      while $i > 0$ and $A[i] > key$
          $A[i+1] = A[i]$
          $i = i - 1$
      $A[i+1] = key$
```

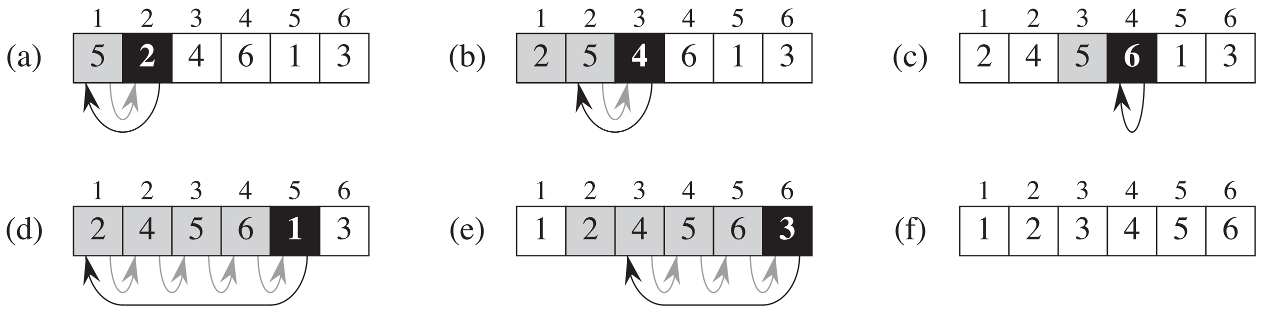


Figure 4.1: From CLRS, the operation of INSERTION-SORT on the array $A = [5, 2, 4, 6, 1, 3]$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)-(e)** The iterations of the **for** loop of lines 1-8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

These notes were adapted from CLRS Chapter 2

Now, we must ask ourselves two questions: does this algorithm work, and does it have good performance?

4.2 Correctness of Insertion Sort

Once you figure out what INSERTION-SORT is doing, you may think that it's "obviously" correct. However, if you didn't know what it was doing and just got the above code, maybe this wouldn't be so obvious. Additionally, for algorithms that we'll study in the future, it won't always be obvious that it works, and so we'll have to prove it. To warm us up for those proofs, let's carefully go through a proof of correctness of INSERTION-SORT.

We'll prove the correctness of INSERTION-SORT formally using a **loop invariant**:

> At the start of each iteration of the **for** loop of lines 1-8, the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order.

To use loop invariants, we must show three things:

1. **Initialization**: It is true prior to the first iteration of the loop.
2. **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration
3. **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Note that this is basically **mathematical induction**: the initialization is the base case, and the maintenance is the inductive step).

In the case of insertion sort, we have:

Initialization: Before the first iteration (which is when $j = 2$), the subarray $A[1..j-1]$ is just the first element of the array, $A[1]$. This subarray is sorted, and consists of the elements that were originally in $A[1..1]$.

Maintenance: Suppose $A[1..j-1]$ is sorted. Informally, the body of the for loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$ and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4-7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1..j]$ then consists of the elements originally in $A[1..j]$, but in sorted order. Incrementing j for the next iteration of the for loop then preserves the loop invariant.

Termination: The condition causing the for loop to terminate is that $j > n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. By the initialization and maintenance steps, we have shown that the subarray $A[1..n+1-1] = A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order. Observing that the subarray $A[1..n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

4.3 Running Time of Insertion Sort

The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed. We assume that a constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the i^{th} line takes time c_i, where c_i is a constant.

```
INSERTION-SORT(A)                                   cost     times
1  for $j = 2$ to $A.length$                        $c_1$    $n$
2      $key = A[j]$                                 $c_2$    $n-1$
3      // Insert $A[j]$ into the sorted
           sequence $A[1..j-1]$.                    0        $n-1$
4      $i = j - 1$                                  $c_4$    $n-1$
5      while $i > 0$ and $A[i] > key$               $c_5$    $\sum_{j=2}^{n} t_j$
6          $A[i+1] = A[i]$                          $c_6$    $\sum_{j=2}^{n} (t_j - 1)$
7          $i = i - 1$                              $c_7$    $\sum_{j=2}^{n} (t_j - 1)$
8      $A[i+1] = key$                               $c_8$    $n-1$
```

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.

To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of the *cost* and *times* columns. Let t_j represent the number of times the **while** loop test is executed on the j^{th} iteration of the **for** loop.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + +c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

So if I asked you how long an algorithm takes to run, you'd probably say, "it depends on the input you give it." In this class we mostly consider the "worst-case running time", which is the longest running time for any input of size n.

For insertion sort, what are the worst and best case inputs? The best case input is a sorted array, because you never have to move elements, and the worst case input is a reverse sorted array (i.e., decreasing order).

In the best case, $t_j = 1$ for all j, and thus we can rewrite $T(n)$ as:

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

We can express this running time as $an + b$ for constants a and b that depend on the statement costs c_i; it is thus a **linear function** of n.

In the worst case, we must compare each element $A[j]$ with each element in the entire sorted subarray $A[1..j-1]$, so $t_j = j$ for all j. Substituting in $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$, we can rewrite $T(n)$ as:

$$T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i; it is thus a **quadratic function** of n.

Running Time and Growth Functions 5

5.1 Measuring Running Time of Algorithms

One way to measure the running time of an algorithm is to implement it and then study its running time by executing it on various inputs. This approach has the following limitations.

- It is necessary to implement and execute an algorithm to study its running time experimentally.
- Experiments can be done only on a limited set of inputs and may not be indicative of the running time on other inputs that were not included in the experiment.
- It is difficult to compare the efficiency of two algorithms unless they are implemented and executed in the same software and hardware environments.

Analyzing algorithms analytically does not require that the algorithms be implemented, takes into account all possible inputs, and allows us to compare the efficiency of two algorithms independent of the hardware and software environment.

5.2 RAM Model of Computation

The model of computation that we use to analyze algorithms is called the *Random Access Machine*(RAM). In this model of computation we assume that high-level operations that are independent of the programming language used take 1 time step. Such high-level operations include the following: performing arithmetic operations, comparing two numbers, assigning a value to a variable, calling a function, returning from a function, and indexing into an array. Note that actually, each of the above high-level operations take a few primitive low-level instructions whose execution time depends on the hardware and software environment, but this is bounded by a constant.

Note that in this model we assume that each memory access takes exactly one time step and we have unlimited memory. The model does not take into account whether an item is in the cache or on the disk.

While some of the above assumptions do not hold on a real computer, the assumptions are made to simplify the mathematical analysis of algorithms. Despite being simple, RAM captures the essential behavior of computers and is an excellent model for understanding how an algorithm will perform on a real computer.

5.3 Average Case and Worst Case

As we saw in class (Insertion Sort), an algorithm works faster on some inputs than others. How should we express the running time of an algorithm? Let $T(n)$ denote the running time (number of high-level steps) of algorithm on an input of size n. Note that there are 2^n input instances with n bits. Below are three possibilities of inputs that we will consider.

- **A typical input:** The problem with considering a typical input is that different applications will have very different typical inputs.
- **Average Case:** Average case analysis is challenging. It requires us to define a probability distribution on the set of inputs, which is a difficult task.

- **Worst Case:** In this measure we consider the input on which the algorithm performs the slowest. When we say that $T(n)$ is the worst case running time of an algorithm we mean that the algorithm takes no more than $T(n)$ steps for *any* input of size n. In other words, it is an upper bound on the running time of the algorithm for any input. This measure is a nice clean mathematical definition and is easiest to analyze.

5.4 Order of Growth

To simplify our analysis we will ignore the lower-order terms in the function $T(n)$ and the multiplicative constants in front. That is, it is the *rate of growth* or the *order of growth* that will be of interest to us. We also ignore the function for small values of n and focus on the *asymptotic* behavior as n becomes very large. Some reasons are as follows.

- The multiplicative constants depends on how fast the machine is and the precise definition of an operation.
- Counting every operation that the algorithm takes is tedious and more work than it is worth.
- It is much more significant whether the time complexity is $T(n) = n^2$ or n^3 than whether it is $T(n) = 2n^2$ or $T(n) = 3n^2$.

Definitions of Asymptotic Notations – $O, \Omega, \Theta, o, \omega$

Note: the limit definition only applies if the limit $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)}$ exists.

If $f(n)$ is in $O(g(n))$ (pronounced "big-oh of g of n"), $g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Definition. O-notation

$f(n) \in O(g(n))$ if there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

$f(n) \in O(g(n))$ if $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$ or a constant.

If $f(n)$ is in $\Omega(g(n))$ (pronounced "omega of g of n"), $g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Definition. Ω-notation

$f(n) \in \Omega(g(n))$ if there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$.

$f(n) \in \Omega(g(n))$ if $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ or a constant.

If $f(n)$ is $\Theta(g(n))$, $g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Definition. Θ-notation

$f(n) \in \Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$.

Thus, we can say that $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

$f(n) \in \Theta(g(n))$ if $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} =$ a nonzero constant.

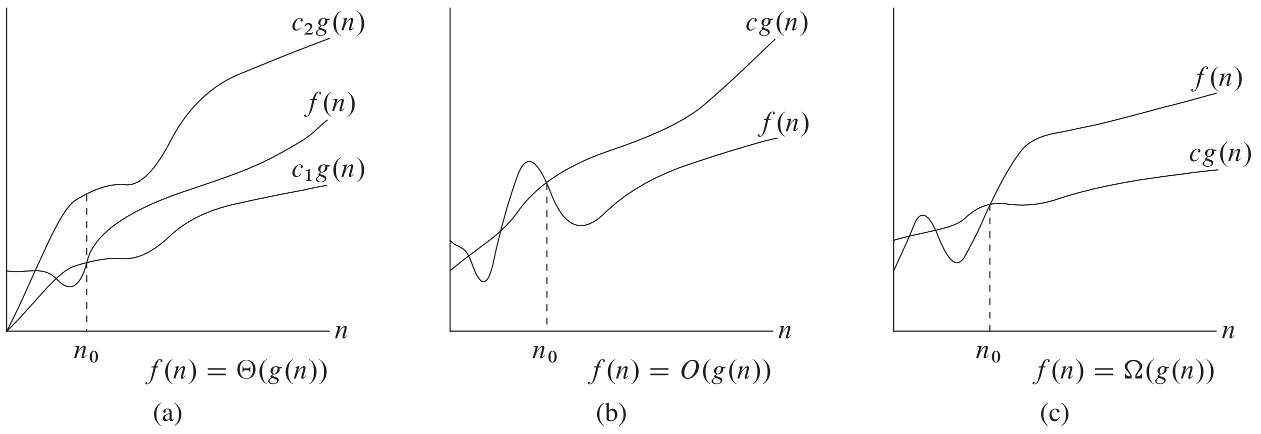


Figure 5.1: From CLRS, graphic examples of the O, Ω and Θ notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

If $f(n)$ is in $o(g(n))$ (pronounced "little-oh of g of n"), $g(n)$ is an ***upper bound that is not asymptotically tight*** for $f(n)$.

Definition. o-notation

$f(n) \in o(g(n))$ if for any positive constant $c > 0$, there exists a positive constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.

$f(n) \in o(g(n))$ if $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

If $f(n)$ is in $\omega(g(n))$ (pronounced "little-omega of g of n"), $g(n)$ is a ***lower bound that is not asymptotically tight*** for $f(n)$.

Definition. ω-notation

$f(n) \in \omega(g(n))$ if for any positive constant $c > 0$, there exists a positive constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$.

$f(n) \in \omega(g(n))$ if $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Note: the limit definition only applies if the limit $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)}$ exists.

Example. Prove that $7n - 2 = \Theta(n)$.

Solution. Note that $7n - 2 \leq 14n$. Thus $7n - 2 = O(n)$ follows by setting $c = 14$ and $n_0 = 1$.

To show that $7n-2 = \Omega(n)$, we need to show that there exists positive constants c and n_0 such that $7n-2 \geq cn$, for all $n \geq n_0$. This is the same as showing that for all $n \geq n_0$, $(7 - c)n \geq 2$. Setting $c = 1$ and $n_0 = 1$ proves the lower bound. Hence $7n - 2 = \Theta(n)$.

Example. Prove that $10n^3 + 55n \log n + 23 = O(n^3)$.

Solution. The left hand side is at most $88n^3$. Thus setting $c = 88$ and $n_0 = 1$ proves the claim.

Example. Prove that $3^{60} = O(1)$.

Solution. This follows because $3^{60} \leq 3^{60} \cdot 1$, for all $n \geq 0$.

Example. Prove that $5/n = O(1/n)$.

Solution. This follows because $5/n \leq 5 \cdot (1/n)$, for all $n \geq 1$.

Example. Prove that $n^2/8 - 50n = \Theta(n^2)$.

First Solution: $n^2/8 - 50n \leq n^2$. To prove the upperbound, we need to prove that

$$0 \leq \frac{n^2}{8} - 50n \leq c_2 n^2, \forall n \geq n_0$$

Setting $c_2 = 1$ and $n_0 = 800$ satisfies the above inequalities.

To show that $n^2/8 - 50n = \Omega(n^2)$, we need to show that there exists positive constants c_1 and n_0 such that $n^2/8 - 50n \geq c_1 n^2$, for all $n \geq n_0$. This is the same as showing that for all $n \geq n_0$, $(1/8 - c_1)n \geq 50$, or equivalently $(1 - 8c_1)n \geq 400$. Setting $c_1 = 1/16$ and $n_0 = 800$ proves the lower bound.

Hence setting $c_1 = 1/16, c_2 = 1$, and $n_0 = 800$ in the definition of $\Theta(\cdot)$ proves that $n^2/8 - 50n = \Theta(n^2)$.

Second Solution: Note that $\lim_{n\to\infty}((n^2/8 - 50n)/n^2) = \lim_{n\to\infty}(n - 400)/8n = \lim_{n\to\infty}(1 - 400/n)/8 = 1/8$. Thus the claim follows.

Example. Prove that $\lg n = O(n)$.

Solution. We will show using induction on n that $\lg n \leq n$, for all $n \geq 1$.
Base Case: $n = 1$: the claim holds for this case as the left hand side is 0 and the right hand side is 1.

Induction Hypothesis: Assume that $\lg k \leq k$, for some $k \geq 1$.

Induction Step: We want to show that $\lg(k + 1) \leq (k + 1)$. We have

$$\begin{aligned} \text{LHS} &= \lg(k+1) \\ &\leq \lg(2k) \\ &= \lg 2 + \lg k \\ &\leq 1 + k \qquad \text{(using induction hypothesis)} \end{aligned}$$

Example. Prove that $3n^{100} = O(2^n)$.

Solution. We will show using induction on n that $3n^{100} \leq 3(100^{100})(2^n)$, for all $n \geq 200$. That is $c = 3(100^{100})$ and $n_0 = 200$.

Base Case: $n = 200$: the claim holds for this case as the left hand side is $3(100^{100})(2^{100})$ and the right hand side is $3(100^{100})(2^{200})$.

Induction Hypothesis: Assume that $3k^{100} \leq 3(100^{100})(2^k)$, for some $k \geq 200$.

Induction Step: We want to show that $3(k+1)^{100} \leq 3(100^{100})(2^{k+1})$.

$$\begin{aligned}
\text{LHS} &= 3(k+1)^{100} \\
&\leq 3\left(k^{100} + 100k^{99} + \binom{100}{2}k^{98} + \cdots + 1\right) \qquad \text{(using Binomial Theorem)} \\
&\leq 3\left(k^{100} + 100k^{99} + 100^2k^{98} + \cdots + 100^{100}k^0)\right) \\
&= 3k^{100}\left(1 + (100/k) + (100/k)^2 + (100/k)^3 + \cdots + (100/k)^{100}\right) \\
&\leq 3(100^{100})(2^k)\sum_{i=0}^{\infty}(1/2)^i \qquad \text{(using induction hypothesis and the fact that } k \geq 200) \\
&\leq 3(100^{100})(2^{k+1})
\end{aligned}$$

Another way to prove the claim is as follows. Consider $\lim_{n\to\infty} 3n^{100}/2^n$. This can be written as

$$\lim_{n\to\infty} 2^{\lg 3n^{100}}/2^n = \lim_{n\to\infty} 2^{\lg 3n^{100} - n} = 0.$$

Example. Prove that $7n - 2$ is not $\Omega(n^{10})$.

Solution. Note that $\lim_{n\to\infty} n^{10}/(7n-2) = \infty$ and hence $7n - 2 = o(n^{10})$. Thus it cannot be $\Omega(n^{10})$.

We can also prove the claim as follows using the definition of Ω. Assume for the sake of contradiction that $7n - 2 = \Omega(n^{10})$. This means that for some constants c and n_0, $7n - 2 \geq cn^{10}$, for all $n \geq n_0$. This implies that

$$\begin{aligned}
7n &\geq cn^{10} \\
7 &\geq cn^9
\end{aligned}$$

This is a contraction as the left hand side is a constant and the right had side keeps growing with n. More specifically, the right hand side becomes greater than the left hand side as soon as $n > \sqrt[9]{7}c^{-1/9}$.

Example. Prove that $n^{1+0.0001}$ is not $O(n)$.

Solution. Assume for the sake of contradiction that $n^{1+0.0001} = O(n)$. This means that for some constants c and n_0, $n^{1+0.0001} \leq cn$, for all $n \geq n_0$. This is equivalent to $n^{0.0001} \leq c$, for all $n \geq n_0$ This is impossible as the left hand side grows unboundedly with n. More specifically, when $n > c^{10^4}$, the left hand side becomes greater than c.

5.5 Properties of Asymptotic Growth Functions

We now state without proof some of the properties of asymptotic growth functions.

1. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$.
2. If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ then $f(n) = \Omega(h(n))$.
3. If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.
4. Suppose f and g are two functions such that for some other function h, we have $f(n) = O(h(n))$ and $g(n) = O(h(n))$. Then $f(n) + g(n) = O(h(n))$.

We now state asymptotic bounds of some common functions.

1. Recall that a polynomial is a function that can be written in the form
$$a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d \text{ for some integer constant } d > 0 \text{ and } a_d \text{ is non-zero.}$$
Let f be a polynomial of degree d in which the coefficient $a_d > 0$. Then $f = O(n^d)$.
2. For every $b > 1$ and any real numbers $x, y > 0$, we have $(\log_b n)^x = O(n^y)$. Note that $(\log_b n)^x$ is also written as $\log_b^x n$.
3. For every $r > 1$ and every $d > 0$, we have $n^d = O(r^n)$.

Example. As a result of the above bounds we can conclude that asymptotically, $10^5 \log^{38} n$ grows slower than $n/10^{10}$.

Example. Prove that $n^{10} = o(2^{\lg^3 n})$ and $2^{\lg^3 n} = o(2^n)$.

Solution. Note that
$$2^{\lg^3 n} = (2^{\lg n})^{\lg^2 n} = n^{\lg^2 n}.$$
Since we know that $10 = o(\lg^2 n)$, we conclude that
$$n^{10} = o(2^{\lg^3 n})$$
Since we also know that any poly-logarithmic function in n such as $\lg^3 n$ is asymptotically upper-bounded by a polynomial function in n, we conclude that
$$\lg^3 n = O(n)$$
Actually, $\lg^3 n = o(n)$ and hence
$$2^{\lg^3 n} = o(2^n).$$

6 Analyzing Runtime of Code Snippets

Example. Consider the following code fragment.

```
for (i = 0; i < n; i++)
    for (j = 0; j < i; j=j+10)
       print (''run time analysis'')
```

Give a tight bound on the running time of this code fragment.

Solution. For each value of i, the inner loop executes $i/10$ times. Thus the running time of the body of the outer loop is at most $c(i/10)$, for some positive constant c. Hence the total running time of the code fragment is given by

$$\sum_{i=0}^{n-1} c\left\lceil \frac{i}{10} \right\rceil \leq \sum_{i=0}^{n-1} c\left(\frac{i}{10}+1\right) = \frac{c(n-1)n}{20} + cn \leq 2cn^2 = O(n^2)$$

We will now show that $\sum_{i=0}^{n-1} c\lceil \frac{i}{10} \rceil = \Omega(n^2)$. Note that

$$\sum_{i=0}^{n-1} c\lceil \frac{i}{10} \rceil \geq \sum_{i=0}^{n-1} \frac{ci}{10} = c(n-1)n/20$$

We want to find positive constants c' and n_0, such that for all $n \geq n_0$,

$$\frac{c(n-1)n}{20} \geq c'n^2$$

This is equivalent to showing that $n(c - 20c') \geq c$. This is true when $c' = c/40$ and $n \geq 2$. Thus, the running time of the code fragment is $\Omega(n^2)$.

Example. Consider the following code fragment.

```
i = n
while (i >= 10) do
  i = i/3
  for j = 1 to n do
    print (''Inner loop'')
```

What is an upper-bound on the running time of this code fragment? Is there a matching lower-bound?

Solution. The running time of the body of the inner loop is $O(1)$. Thus the running time of the inner loop is at most $c_1 n$, for some positive constant c_1. The body of the outer loop takes at most $c_2 n$ time, for some positive constant c_2 (note that the statement $i = i/3$ takes $O(1)$ time). Suppose the algorithm goes through t iterations of the while loop. At the end of the last iteration of the while loop, the value of i is $n/3^t$. We know that the code fragment surely finishes when $n/3^t \leq 1$, solving which gives us $t \geq \log_3 n$. This means that the number of iterations of the while loop is at most $O(\log n)$. Thus the total running time is $O(n \log n)$.

We will now show that the running time is $\Omega(n \log n)$. We will lower-bound the number of iterations of the outer loop. Note that when the value of i is more than 10 (say, 3^3), the outer loop has not terminated. Solving $n/3^t \geq 3^3$, gives us that $\log_3 n - 3$ is a lower bound on the number of iterations of the outer loop. For each iteration of the outer loop, the inner loop runs n times. Thus the total running time is at least $cn(\log_3 n - 3)$, for some positive constant c. Note that $cn(\log_3 n - 3) \geq c'n \log n$, when $c' = c/2$ and $n \geq 3^{15}$. Thus the running time is $\Omega(n \log n)$.

Example. Consider the following code fragment.

```
for i = 0 to n do
  for j = n down to 0 do
    for k = 1 to j-i do
        print (k)
```

What is an upper-bound on the running time of this algorithm? What is the lower bound?

Solution. Note that for a fixed value of i and j, the innermost loop goes through $\max\{0, j-i\} \leq n$ times. Thus the running time of the above code fragment is $O(n^3)$.

To find the lower bound on the running time, consider the values of i, such that $0 \leq i \leq n/4$ and values of j, such that $3n/4 \leq j \leq n$. Note that for each of the $n^2/16$ different combinations of i and j, the innermost loop executes at least $n/2$ times. Thus the running time is at least

$$(n^2/16)(n/2) = \Omega(n^3)$$

Example. Consider the following code fragment.

```
for i = 1 to n do
  for j = 1 to i*i do
    for k = 1 to j do
        print (k)
```

Give a tight-bound on the running time of this algorithm? We will assume that n is a power of 2.

Solution. Note that the value of j in the second for-loop is upper bounded by n^2 and the value of k in the innermost loop is also bounded by n^2. Thus the outermost for-loop iterates for n times, the second for-loop iterates for at most n^2 times, and the innermost loop iterates for at most n^2 times. Thus the running time of the code fragment is $O(n^5)$.

We will now argue that the running time of the code fragment is $\Omega(n^5)$. Consider the following code fragment.

```
for i = n/2 to n do
  for j = (n/4)*(n/4) to (n/2)*(n/2)  do
    for k = 1 to (n/4)*(n/4) do
        print (k)
```

Note that the values of i, j, k in the above code fragment form a subset of the corresponding values in the code fragment in question. Thus the running time of the new code fragment is a lower bound on the running time of the code fragment in question. Thus the running time of the code fragment in question is at least $n/2 \cdot 3n^2/16 \cdot n^2/16 = \Omega(n^5)$.

Thus the running time of the code fragment in question is $\Theta(n^5)$.

Example. Consider the following code fragment. We will assume that n is a power of 2.

```
for (i = 1; i <= n; i = 2*i) do
  for j = 1 to i do
        print (j)
```

Give a tight-bound on the running time of this algorithm?

Solution. Observe that for $0 \le k \le \lg n$, in the k^{th} iteration of the outer loop, the value of $i = 2^k$. Thus the running time $T(n)$ of the code fragment can be written as follows.

$$\begin{aligned}
T(n) &= \sum_{k=0}^{\lg n} 2^k \\
&= 2^{\lg n + 1} - 1 \\
&= 2n - 1 \\
&= \Theta(n) \quad (c_1 = 1, c_2 = 2, n_0 = 1)
\end{aligned}$$

What is wrong with the following argument that the running time of the above code fragment is $\Omega(n \log n)$?

```
for (i = n/128; i<= n; i = 2*i) do
  for j = 1 to n/128 do
        print (j)
```

The outer loop runs $\Omega(\lg n)$ times and the inner loop runs $\Omega(n)$ times and hence the running time is $\Omega(n \log n)$.[*]

Discussion: Consider a problem X with an algorithm A.

- Algorithm A runs in time $O(n^2)$. This means that the worst case asymptotic running time of algorithm A is upper-bounded by n^2. Is this bound tight? That is, is it possible that the run-time analysis of algorithm A is loose and that one can give a tighter upper-bound on the running time?
- Algorithm A runs in time $\Theta(n^2)$. This means that the bound is tight, that is, a better (tighter) bound on the worst case asymptotic running time for algorithm A is not possible.
- Problem X takes time $O(n^2)$. This means that there is an algorithm that solves problem X on *all* inputs in time $O(n^2)$.
- Problem X takes $\Theta(n^{1.5})$. This means that there is an algorithm to solve problem X that takes time $O(n^{1.5})$ and no algorithm can do better.

Logarithm Facts: Below are some facts on logarithms that you may find useful.

i. $\log_a b = \frac{1}{\log_b a}$
ii $\log_a b = \frac{\log_c b}{\log_c a}$
iii $a^{\log_a b} = b$
iv $b^{\log_a x} = x^{\log_a b}$

[*] Answer: the issue with the argument is that the outer loop only runs a constant number of times (8 to be specific, since $2^7 = 128$), and therefore the outer loop runs in $\Omega(1)$ time instead of $\Omega(\lg n)$, which is why it's $\Theta(n)$, as the proof above shows.

7 Divide & Conquer and Recurrence Relations

7.1 Computing Powers of Two

Consider the problem of computing 2^n for any non-negative integer n. Below are four similar looking algorithms to solve this problem.

```
powerof2(n)
  if n = 0
    return 1
  else
    return 2 * powerof2(n-1)

powerof2(n)
  if n = 0
    return 1
  else
    return powerof2(n-1)+ powerof2(n-1)

powerof2(n)
  if n = 0
    return 1
  else
    tmp = powerof2(n-1)
    return tmp + tmp

powerof2(n)
  if n = 0
    return 1
  else
    tmp = powerof2(floor(n/2))
    if (n is even) then
      return tmp * tmp
    else
      return 2 * tmp * tmp
```

The recurrence for the first and the third method is $T(n) = T(n-1) + O(1)$. The recurrence for the second method is $T(n) = 2T(n-1) + O(1)$, and the recurrence for the last method is $T(n) = T(n/2) + c$ (assuming that n is a power of 2). In all cases the base case is $T(0) = 1$.

We will solve these recurrences. The recurrence for the first and the third method can be solved as follows.

$$\begin{aligned}
T(n) &= T(n-1) + c \\
&= T(n-2) + 2c \\
&= T(n-3) + 3c \\
&\dots \\
&\dots \\
&= T(n-k) + kc
\end{aligned}$$

The recursion bottoms out when $n - k = 0$, i.e., $k = n$. Thus, we get

$$\begin{aligned}
T(n) &= T(0) + kc \\
&= 1 + nc \\
&= \Theta(n)
\end{aligned}$$

The recurrence for the second method can be solved as follows.

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

The recursion bottoms out when $n - k = 0$, i.e., $k = n$. Thus, we get

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

The recurrence for the fourth method can be solved as follows.

$$\begin{aligned}
T(n) &= T(n/2) + c \\
&= T(n/2^2) + 2c \\
&= T(n/2^3) + 3c \\
&\dots \\
&\dots \\
&= T(n/2^k) + kc
\end{aligned}$$

The recursion bottoms out when $n/2^k < 1$, i.e., when $k > \lg n$. Thus, we get

$$\begin{aligned} T(n) &= T(0) + c(\lg n + 1) \\ &= 1 + \Theta(\lg n) \\ &= \Theta(\lg n) \end{aligned}$$

7.2 Linear Search and Binary Search

The input is an array A of elements in any arbitrary order and a key k and the objective is to output true, if k is in A, false, otherwise. Below is a recursive function to solve this problem.

```
LinearSearch (A[lo .. hi], k)
  if lo > hi then
    return False
  else
    return (A[hi] == k) or LinearSearch(A[lo..hi-1], k)
```

The recurrence relation to express the running time of `LinearSearch` is given by $T(n) = T(n-1) + c$, with the base case being $T(0) = 1$. We have already solved this recurrence and it yields a running time of $T(n) = \Theta(n)$.

If the input array A is already sorted, we can do significantly better using *binary search* as follows.

```
BinarySearch (A[lo .. hi], k)
  if lo > hi then
     return False
  else
     mid = floor(lo+hi/2)
     if A[mid] = k then
        return True
     else if A[mid] < k then
        return BinarySearch(A[mid+1 .. hi], k)
     else
        return BinarySearch(A[lo .. mid-1], k)
```

The running time of this method is given the recurrence $T(n) = T(n/2) + c$, with the base case being $T(0) = 1$. As we have seen before, this recurrence yields a running time of $T(n) = \Theta(\log n)$.

7.3 MergeSort

Below is a recursive version of insertion sort that we studied a couple of lectures ago.

```
InsertionSort(A[lo..hi])
  if lo = hi then
    return A
  else
    A' = InsertionSort(A[lo..hi-1])
    Insert(A', A[hi])  // insert element A[hi] into the sorted array A'
```

Note that the `Insert` function takes $\Theta(n)$ time for an input array of size n. Thus the running time of Insertion sort is given by the following recurrence.

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

It is easy to see that this recurrence yields a running time of $T(n) = \Theta(n^2)$.

To motivate the idea behind the next sorting algorithm (Merge Sort), let's rewrite `InsertionSort` function as follows.

```
InsertionSort(A[lo..hi])
  if lo = hi then
    return A
  else
    // Merge combines two sorted arrays into one sorted array
    Merge(InsertionSort(A[lo..hi-1]), InsertionSort(A[hi..hi]))
```

The function `Merge` is as follows.

```
Merge(A[1..p], B[1..q])
  if p = 0 then
    return B
  if q = 0 then
    return A
  if A[1] <= B[1] then
    return prepend(A[1], Merge(A[2..p], B[1..q]))
  else
    return prepend(B[1], Merge(A[1..p], B[2..q]))
```

Note that the running time of `Merge` is $O(p+q)$. The second recursive call to `InsertionSort` takes $O(1)$ time and hence the running time of `InsertionSort` still is $\Theta(n^2)$.

Observe that in `InsertionSort` the input array A is partitioned into two arrays, one of size $|A| - 1$ and another of size 1. In Merge Sort, we partition the input array of size n in two equal halves (assuming n is a power of 2). Below is the function.

```
MergeSort(A[1..n])
if n = 1 then
  return A
else
  return Merge(MergeSort(A[1..n/2]), MergeSort(A[n/2+1..n]))
```

The running time of `MergeSort` is given by the following recurrence.

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n/2) + cn, & n \geq 2 \end{cases}$$

We can also solve recurrences by guessing the overall form of the solution and then figure out the constants as we proceed with the proof. Below are some examples.

Example. Consider the following recurrence for the `MergeSort` algorithm.

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

Prove that $T(n) = O(n \lg n)$.

Solution. We will first prove the claim by expanding the recurrence as follows.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2^2 T(n/2^2) + 2n \\ &= 2^3 T(n/2^3) + 3n \\ &\dots \\ &\dots \\ &= 2^k T(n/2^k) + kn \end{aligned}$$

The recursion bottoms out when $n/2^k = 1$, i.e., $k = \lg n$. Thus, we get

$$\begin{aligned} T(n) &= 2^{\lg n} T(1) + n \lg n \\ &= \Theta(n \log n) \end{aligned}$$

We will now prove that $T(n) = O(n \lg n)$ by using strong induction on n. We will show that for some constant c, whose value we will determine later, $T(n) \leq cn \lg n$, for all $n \geq 2$.

Induction Hypothesis: Assume that the claim is true when $n = j$, for all j such that $2 \leq j \leq k$. In other words, $T(j) \leq cj \lg j$.

Base Case: $n = 2$. The left hand side is given by $T(2) = 2T(1) + 2 = 4$ and the right hand side is $2c$. Thus the claim is true for the base case when $c \geq 2$.

Induction Step: We want to show that for $k \geq 2$, $T(k+1) \leq c(k+1)\lg(k+1)$. We have

$$\begin{aligned} T(k+1) &= 2T\left(\frac{k+1}{2}\right) + (k+1) \\ &\leq 2c\left(\left(\frac{k+1}{2}\right)\lg\left(\frac{k+1}{2}\right)\right) + (k+1) \\ &= c(k+1)(\lg(k+1) - \lg 2) + (k+1) \\ &= c(k+1)\lg(k+1) - (c-1)(k+1) \\ &\leq c(k+1)\lg(k+1) \qquad \text{(since } c \geq 2\text{)} \end{aligned}$$

Example. Consider the following recurrence that you may want to try to solve on your own before reading the solution.

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n/2) + n^2, & n \geq 2 \end{cases}$$

Prove that $T(n) = \Theta(n^2)$.

Solution. Clearly, $T(n) = \Omega(n^2)$ (because of the n^2 term in the recurrence). To prove that $T(n) = O(n^2)$, we will show using strong induction that for some constant c, whose value we will determine later, $T(n) \leq cn^2$, for all $n \geq 1$.

Induction Hypothesis: Assume that the claim is true when $n = j$, for all j such that $1 \leq j \leq k$. In other words, $T(j) \leq cj^2$.

Base Case: $n = 1$. The claim is clearly true as the left hand side and the right hand side, both equal 1.

Induction Step: We want to show that $T(k+1) \leq c(k+1)^2$. We have

$$\begin{aligned} T(k+1) &= 2T\left(\frac{k+1}{2}\right) + (k+1)^2 \\ &\leq 2c\left(\frac{k+1}{2}\right)^2 + (k+1)^2 \\ &= \left(\frac{c}{2} + 1\right)(k+1)^2 \end{aligned}$$

We want the right hand side to be at most cn^2. This means that we want $c/2 + 1 \leq c$, which holds when $c \geq 2$. Thus we have shown that $T(n) \leq 2n^2$, for all $n \geq 1$, and hence $T(n) = O(n^2)$.

7.4 More Recurrence Practice

Example. The running time of Karatsuba's algorithm for integer multiplication is given by the following recurrence. Solve the recurrence. Assume that n is a power of 2.

$$T(n) = \begin{cases} 3T(n/2) + n, & n \geq 2 \\ 1, & \text{n=1} \end{cases}$$

Solution.

$$\begin{aligned}T(n) &= 3T(n/2) + n\\ &= 3^2T(n/2^2) + 3n/2 + n\\ &= 3^3T(n/2^3) + (3/2)^2n + 3n/2 + n\\ &\dots\\ &\dots\\ &= 3^kT(n/2^k) + n\sum_{i=0}^{k-1}(3/2)^i\end{aligned}$$

The recursion bottoms out when $n/2^k = 1$, i.e., $k = \lg n$. Thus, we get

$$\begin{aligned}T(n) &= 3^{\log_2 n}T(1) + n\sum_{i=0}^{\lg n-1}(3/2)^i\\ &= 3^{\log_2 n} + n\left(\frac{(3/2)^{\lg n}-1}{3/2-1}\right)\\ &= n^{\log_2 3} + 2n(n^{\lg(3/2)} - 1) \qquad \text{(log fact derived from change of base rule)}\\ &= n^{\log_2 3} + 2n(n^{\lg 3-\lg 2)} - 1)\\ &= n^{\log_2 3} + 2n(n^{\lg 3-1}) - 2n\\ &= n^{\log_2 3} + 2n^{\lg 3} - 2n\\ &= 3n^{\log_2 3} - 2n\\ &= \Theta(n^{\lg 3}) \qquad \text{(can be argued by setting } c = 3 \text{ and } n_0 = 1)\end{aligned}$$

Example. Find the running time expressed by the following recurrence. Assume that n is a power of 3.

$$T(n) = \begin{cases} 5T(n/3) + n^2, & n \geq 2\\ 1, & \text{n=1}\end{cases}$$

Solution.

$$\begin{aligned}T(n) &= 5T(n/3) + n^2\\ &= 5^2T(n/3^2) + 5(n/3)^2 + n^2\\ &= 5^3T(n/3^3) + 5^2(n/3^2)^2 + 5n^2/3^2 + n^2\\ &\dots\\ &\dots\\ &= 5^kT(n/3^k) + n^2\left(1 + \frac{5}{9} + \left(\frac{5}{9}\right)^2 + \ldots + \left(\frac{5}{9}\right)^{k-1}\right)\end{aligned}$$

The recursion bottoms out when $n/3^k = 1$, i.e., $k = \log_3 n$. Thus, we get

$$\begin{aligned}
T(n) &= 5^{\log_3 n} T(1) + n^2 \sum_{i=0}^{k-1} (5/9)^i \\
&= 5^{\log_3 n} + n^2 \left(\frac{(5/9)^{\log_3 n} - 1}{(5/9) - 1} \right) \\
&= 5^{\log_5 n / \log_5 3} + \frac{9n^2}{4} \left(1 - (n^{\log_3 5}/n^2)\right) \\
&= n^{\log_3 5} + \frac{9n^2}{4} - \frac{9n^{\log_3 5}}{4} \\
&= \frac{9n^2}{4} - \frac{5n^{\log_3 5}}{4} \\
&= \Theta(n^2)
\end{aligned}$$

7.5 Simplified Master Theorem

Obviously, it would be nice if there existed some kind of shortcut for computing recurrences. The Simplified Master Theorem provides this shortcut for recurrences of the following form:

Theorem 1. *Let $a \geq 1$, $b > 1, k \geq 0$ be constants and let $T(n)$ be a recurrence of the form*

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k)$$

defined for $n \geq 0$. The base case $T(1)$ can be any constant value. Then

Case 1: *if $a > b^k$, then $T(n) = \Theta\left(n^{\log_b a}\right)$*
Case 2: *if $a = b^k$, then $T(n) = \Theta\left(n^k \log_b n\right)$*
Case 3: *if $a < b^k$, then $T(n) = \Theta\left(n^k\right)$*

The statement of the full Master Theorem and its proof will be presented in CIS 320.

Example. Solve the following recurrences using the Simplified Master Theorem. Assume that n is a power of 2 or 3 and $T(1) = c$ for some constant c.

a. $T(n) = 4T(n/2) + n$
b. $T(n) = T(n/3) + n$
c. $T(n) = 9T(n/3) + n^{2.5}$
d. $T(n) = 8T(n/2) + n^3$

Solution.

a. $a = 4, b = 2$, and $k = 1$. Thus case 1 of the Simplified Master Theorem applies and hence $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$.
b. $a = 1, b = 3$, and $k = 1$. Thus case 3 of the Simplified Master Theorem applies and hence $T(n) = \Theta(n)$.
c. $a = 9, b = 3$, and $k = 2.5$. Thus case 3 of the Simplified Master Theorem applies and hence $T(n) = \Theta(n^{2.5})$.
d. $a = 8, b = 2$, and $k = 3$. Thus case 2 of the Simplified Master Theorem applies and hence $T(n) = \Theta(n^3 \log_2 n)$.

Quicksort 8

8.1 Deterministic Quicksort

In quicksort, we first decide on the pivot. This could be the element at any location in the input array. The function `Partition` is then invoked. `Partition` accomplishes the following: it places the pivot in the location that it should be in the output and places all elements that are at most the pivot to the left of the pivot and all elements greater than the pivot to its right. Then we recurse on both parts. The pseudocode for Quicksort is as follows.

```
QSort(A[lo..hi])
  if hi <= lo then
    return
  else
    pIndex = floor((lo+hi)/2)  (this could have been any location)
    loc = Partition(A, lo, hi, pIndex)
    QSort(A[lo..loc-1])
    QSort(A[loc+1..hi])
```

One possible implementation of the function `Partition` is as follows.

```
Partition(A, lo, hi, pIndex)
  pivot = A[pIndex]
  swap(A, pIndex, hi)
  left = lo
  right = hi-1
  while left <= right do
     if (A[left] <= pivot) then
        left = left + 1
     else
        swap(A, left, right)
        right = right - 1
  swap(A, left, hi)
  return left
```

The worst case running time of the algorithm is given by

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

Hence the worst case running time of `QSort` is $\Theta(n^2)$.

The quicksort algorithm performs poorly when the pivot is chosen to be an element that is near the min or max of the array. For example, consider an instance of quicksort where the input is an array in descending order of elements and where we always select the first element as the pivot.

8.2 Randomized Quicksort

In the randomized version of quicksort, we pick a pivot uniformly at random from all possibilities. We will now show that the expected number of comparisons made in randomized quicksort is equal to $2n \ln n + O(n) = \Theta(n \log n)$.

Theorem 1. *For any input array of size n, the expected number of comparisons made by randomized quicksort is*

$$2n \ln n + O(n) = \Theta(n \log n)$$

Proof. Let $y_1, y_2, ..., y_n$ be the elements in the input array A *in sorted order*. Let X be the random variable denoting the total number of pair-wise comparisons made between elements of A. Let $X_{i,j}$ be the random variable denoting the total number of times elements y_i and y_j are compared during the algorithm.

We make the following observations:

- Comparisons between elements in the input array are done only in the function `Partition`
- Two elements are compared if and only if one of them is a pivot.

Let $X^k_{i,j}$ be an indicator random variable that is 1 if and only if elements y_i and y_j are compared in the k-th call to `Partition`. Then we have:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j} \qquad \text{and} \qquad X_{i,j} = \sum_k X^k_{i,j}$$

We will now calculate $\mathbf{E}[X_{i,j}]$. By the linearity of expectation, we have

$$\mathbf{E}[X_{ij}] = \sum_k \mathbf{E}[X^k_{ij}] = \sum_k \Pr[X^k_{ij} = 1]$$

Let t be the iteration of the first call to `Partition` during which one of the elements from $y_i, y_{i+1}, ..., y_j$ is used as the pivot (think about why such an iteration must exist). From our observations above, note that for all times before t, y_i and y_j are never compared, so $X^k_{i,j} = 0$ for all $k < t$. If one of y_i or y_j is chosen as the t-th pivot, then $X^t_{i,j} = 1$, otherwise $X^t_{i,j} = 0$ and y_i and y_j will be separated into different sublists and hence will never be compared again. Hence $X^k_{i,j} = 0$ for all $k > t$.

Now, since there are $j - i + 1$ elements in the list $y_i, y_{i+1}, ..., y_j$, and the pivot is chosen randomly, the probability that one of y_i or y_j is chosen as the pivot is $\frac{2}{j-i+1}$. Hence:

$$\mathbf{E}[X_{i,j}] = \Pr[X^t_{i,j} = 1] = \frac{2}{j - i + 1}$$

Now we use this result and apply the linearity of expectation to get:

$$
\begin{aligned}
\mathbf{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{E}[X_{i,j}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{k=2}^{n} \frac{2}{k}(n-k+1) \qquad \text{(see the note below)} \\
&= (n+1)\sum_{k=2}^{n} \frac{2}{k} - 2(n-1) \\
&= 2(n+1)\sum_{k=1}^{n} \frac{1}{k} - 2(n-1) - 2(n+1) \\
&= 2(n+1)\left(\ln n + c\right) - 4n \qquad \text{where } 0 \le c < 1 \\
&= 2n \ln n + O(n)
\end{aligned}
$$

Here we have used the fact that the harmonic function, $H(n) = \sum_{k=1}^{n} \frac{1}{k}$ is at most $\ln n + c$ for some constant $0 \le c < 1$.

Also, the third equality follows by expanding the sum as

$$
\begin{aligned}
\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} &= \frac{2}{2} + \frac{2}{3} + ... + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} \\
&+ \frac{2}{2} + \frac{2}{3} + ... + \frac{2}{n-2} + \frac{2}{n-1} \\
&+ \frac{2}{2} + \frac{2}{3} + ... + \frac{2}{n-2} \\
&\vdots \\
&+ \frac{2}{2}
\end{aligned}
$$

and grouping the columns together.

□

Counting Inversions 9

9.1 Introduction and Problem Description

We will continue with our study of divide and conquer algorithms by considering how to count the number of inversions in a list of elements. This is a problem that comes up naturally in many domains, such as collaborative filtering and meta-search tools on the internet. A core issue in applications like these is the problem of comparing two rankings: you rank a set of n movies, and then a collaborative filtering system consults its database to look for other people who had "similar" rankings in order to make suggestions. But how does one measure similarity?

Suppose you and another person rank a set of n movies, labelling them from 1 to n accordingly. A natural way to compare your ranking with the other persons is to count the number of pairs that are "out of order." Formally, we will consider the following problem: We are given a sequence of n numbers, $a_1, ..., a_n$, which we assume are distinct. We want to define a measure that tells us how far this list is from being in ascending order. The value of the measure should be 0 if $a_1 < a_2 < ... < a_n$ and should increase as the numbers become more scrambled, taking on the largest possible value if $a_n < a_{n-1} < ... < a_1$.

A natural way to quantify this notion is by counting the number of *inversions*.

Definition. Given a sequence of numbers $a_1, a_2, ..., a_n$, we say indices $i < j$ form an inversion if $a_i > a_j$. That is, if a_i and a_j are out of order.

As an example, consider the sequence

$$2, 4, 1, 3, 5$$

There are three inversions in this sequence: $(2, 1), (4, 1)$, and $(4, 3)$. A simple way to count the number of inversions when n is small is to draw the sequence of input numbers in the order they're provided, and below that in ascending order. We then draw a line between each number in the top list and its copy in the lower list. Each crossing pair of line segments corresponds to an inversion.

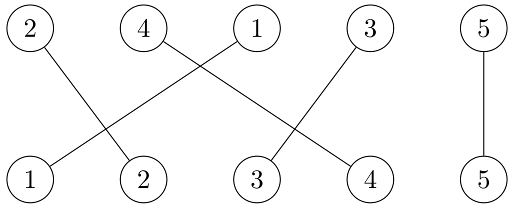


Figure 9.1: There are three inversions: $(2, 1), (4, 1)$, and $(4, 3)$.

Note how the number of inversions is a measure that smoothly interpolates between complete agreement (when the sequence is in ascending order, then there are no inversions) and complete disagreement (if the sequence is in descending order, then every pair forms an inversion and so there are $\binom{n}{2}$ of them).

These notes were adapted from Kleinberg and Tardos' *Algorithm Design*

9.2 Designing an Algorithm

A naive way to count the number of inversions would be to simply look at every pair of numbers $(a_i,\ a_j)$ and determine whether they constitute an inversion. This would take $O(n^2)$ time.

As you can imagine, there is a faster way that runs in $O(n \log n)$ time. Note that since there can be a quadratic number of inversions, such an algorithm must be able to compute the total number without every looking at each inversion individually. The basic idea is to use a divide and conquer strategy.

Similar to the other divide and conquer algorithms we've seen, the first step is to divide the list into two pieces: set $m = \left\lceil \frac{n}{2} \right\rceil$ and consider the two lists $a_1, ..., a_m$ and $a_{m+1}, ..., a_n$. Then we conquer each half by recursively counting the number of inversions in each half separately.

Now, how do we combine the results? We have the number of inversions in each half separately, so we must find a way to count the number of inversions of the form (a_i, a_j) where a_i and a_j are in different halves. We know that the recurrence $T(n) = 2T(n/2) + O(n)$, $T(1) = 1$ has solution $T(n) = O(n \log n)$, and so this implies that we must be able to do this part in $O(n)$ time if we expect to find an $O(n \log n)$ solution.

Note that the first-half/second-half inversions have a nice form: they are precisely the pairs (a_i, a_j) where a_i is in the first half, a_j is in the second half, and $a_i > a_j$.

To help with counting the number of inversions between the two halves, we will make the algorithm recursively sort the numbers in the two halves as well. Having the recursive step do a bit more work (sorting as well as counting inversions) will make the "combining" portion of the algorithm easier.

Thus the crucial routine in this process is Merge-and-Count. Suppose we have recursively sorted the first and second halves of the list and counted the inversions in each. We know have two sorted lists A and B, containing the first and second halves respectively. We want to produce a single sorted list C from their union, while also counting the number of pairs (a, b) with $a \in A$ and $b \in B$ and $a > b$.

This is closely related to a problem we have previously encountered: namely the "combining" step in MergeSort. There, we had two sorted lists A and B and we wanted to merge them into a single sorted list in $O(n)$ time. The difference here is that we want to do something extra: not only should we produce a single sorted list from A and B, but we should also count the number of inverted pairs (a, b) where $a \in A$, $b \in B$ and $a > b$ as we do so.

We can do this by walking through the sorted lists A and B, removing elements from the front and appending them to the sorted list C. In a given step, we have a *current* pointer into each list, showing our current position. Suppose that these pointers are currently at elements a_i and b_j. In one step, we compare the elements a_i and b_j, remove the smaller one from its list, and append it to the end of list C.

This takes care of the merging. To count the number of inversions, note that because A and B are sorted, it is actually very easy to keep track of the number of inversions we encounter. Every time element a_i is appended to C, no new inversions are encountered since a_i is smaller than everything left in list B and it comes before all of them. On the other hand, if b_j is appended to list C, then it is smaller than *all* of the remaining items in A and it comes after all of them, so we increase our count of the number of inversions by the number of elements remaining in A. This is the crucial idea: in constant time, we have accounted for a potentially large number of inversions.

Merge-and-Count

Input: Two sorted lists A and B

Output: A sorted list containing all elements in A and B, as well as the number of inversions assuming all elements in A precede those in B.

```
Merge-and-Count(A,B)
    L = []
    currA = 1
    currB = 1
    count = 0
    While currA <= A.length and currB <= B.length:
        a = A[currA]
        b = B[currB]
        if a < b then
            L.append(a)
            currA = currA + 1
        else
            L.append(b)
            remaining = A.length - currA + 1
            count = count + remaining
            currB = currB + 1
    Once one list is empty, append the remainder of the other
    list to $L$

    Return count and L
```

We use this Merge-and-Count routine in a recursive procedure that simultaneously sorts and counts the number of inversions in a list L.

Sort-and-Count

Input: A list L of n distinct numbers

Output: A sorted version of L as well as the number of inversions in L.

```
Sort-and-Count(L)
    if n = 1 then
        return 0 and L

    m = $\lceil n/2 \rceil$
    A = L[1,...,m]
    B = L[m+1,...,n]
    $(r_A, A)$ = Sort-and-Count(A)
    $(r_B, B)$ = Sort-and-Count(B)
    $(r\ ,\ L)$ = Merge-and-count(A,B)

    return $r_A + r_B + r$ and the sorted list $L$
```

9.3 Runtime

The running time of Merge-and-Count can be bounded by the analogue of the argument we used for the original Merge algorithm: each iteration of the `while` loop takes constant time, and in each iteration, we add some element to the output that will never be seen again. Thus the number of iterations can be at most the sum of the initial lengths of A and B, and so the total running time is $O(n)$.

Since Merge-and-Count runs in $O(n)$ time, the recurrence for the runtime of Sort-and-Count is

$$T(n) = \begin{cases} 2T(n/2) + O(n) & n > 1 \\ 1 & \text{otherwise} \end{cases}$$

This is exactly the same recurrence as we saw in the analysis of MergeSort, so we can immediately say the runtime of Sort-and-Count is

$$T(n) = O(n \log n)$$

Selection Problem 10

10.1 Introduction to Problem

Definition. The ith order statistic of a set of n elements is the ith smallest element.

Definition. A median, informally, is the "halfway point" of the set, occurring at $i = \lfloor (n+1)/2 \rfloor^*$

* When n is even, two medians exist. However, for simplicity, we will choose the *lower median.*

Here we address the problem of selecting the ith order statistic from a set of n distinct numbers. We assume for convenience that the set contains distinct numbers, although virtually everything that we do extends to the situation in which a set contains repeated values. The **selection problem** is defined as:

- **Input**: n integers in array $A[1..n]$, and an integer $i \in \{1, 2, ..., n\}$
- **Output**: the ith smallest number in A

We can solve the selection problem in $O(n \lg n)$ time, since we can sort the numbers using merge sort and then simply index the ith element in the output array. Here, we present a faster algorithm which achieves $O(n)$ running time in the worst case.

If we could find element e such that $rank(e) = n/2$ (the *median*) in $O(n)$ time we could make Quicksort run in $\Theta(n \log n)$ time worst case. We could just exchange e with the last element in A in the beginning of Partition and thus ensure that A is always partitioned in the middle.

10.2 Selection in Worst-Case Linear Time

We now examine a selection algorithm whose running time is $O(n)$ in the worst case. The algorithm Select finds the desired element by recursively partitioning the input array. Here, however, we guarantee a good split upon partitioning the array. Select uses the deterministic partitioning algorithm Partition from Quicksort, but modified to take the element to partition around as an input parameter.

The Select algorithm determines the ith smallest of an input array of $n > 1$ distinct elements by executing the following steps. (If $n = 1$, then Select merely returns its only input value as the ith smallest.)

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining $n \mod 5$ elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups by first Insertion-Sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
3. Use Select recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 2. (If there are an even number of medians, then by our convention, x is the *lower median.*)
4. Partition the input array around the median-of-medians x using the modified version of Partition. Let k be one more than the number of elements on the low side of the partition, so that x is the kth smallest element and there are $n - k$ elements on the high side of the partition.
5. If $i = k$, then return x. Otherwise, use Select recursively to find the ith smallest element on the low side if $i < k$, or the $(i - k)$th smallest element on the high side if $i > k$.

These notes were adapted from CLRS Chapter 9.3

To analyze the running time of SELECT, we first determine a lower bound on the number of elements that are greater than the partitioning element x. The figure below helps us to visualize this bookkeeping. At least half of the medians found in step 2 are greater than or equal to the median-of-medians x (remember, we assumed distinct elements). Thus, at least half of the $\lceil n/5 \rceil$ groups contribute at least 3 elements that are greater than x, except for the one group that has fewer than 5 elements if 5 does not divide n exactly, and the one group containing x itself. Discounting these two groups, it follows that the number of elements greater than x is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

Similarly, at least $3n/10 - 6$ elements are less than x. Thus, in the worst case, step 5 calls SELECT recursively on at most $7n/10 + 6$ elements.

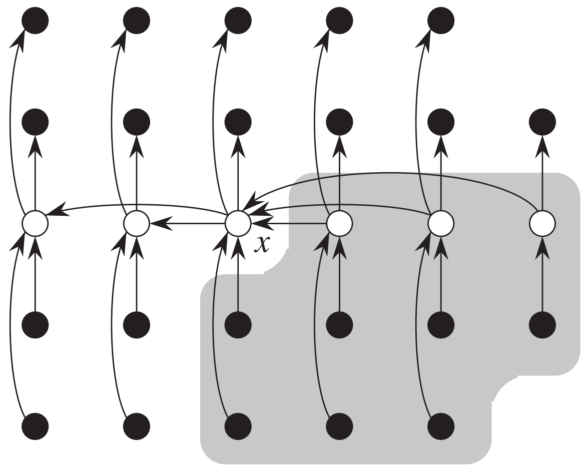


Figure 10.1: From CLRS, this figure helps understand the analysis of SELECT. The n elements are represented by small circles, and each group of 5 elements occupies a column. The medians of the groups are whitened, and the median-of-medians x is labeled. (When finding the median of an even number of elements, we use the lower median.) Arrows go from larger elements to smaller, from which we can see that 3 out of every full group of 5 elements to the right of x are greater than x, and 3 out of every group of 5 elements to the left of x are less than x. The elements known to be greater than x appear on a shaded background.

We can now develop a recurrence for the worst-case running time $T(n)$ of the algorithm SELECT. Steps 1, 2, and 4 take $O(n)$ time. (Step 2 consists of $O(n)$ calls of INSERTION-SORT on sets of size $5 = O(1)$.) Step 3 takes time $T(\lceil n/5 \rceil)$, and step 5 takes time at most $T(7n/10 + 6)$, assuming that T is monotonically increasing. We make the assumption, which seems unmotivated at first, that any input of fewer than 140 elements requires $O(1)$ time; the origin of the magic constant 140 will be clear shortly. We can therefore obtain the recurrence

$$T(n) \leq \begin{cases} O(1), & \text{if } x < 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n), & \text{otherwise.} \end{cases}$$

We show that the running time is linear by substitution. More specifically, we will show that $T(n) \leq cn$ for some suitably large constant c and all $n > 0$. We begin by assuming that $T(n) \leq cn$ for some suitably large constant c and all $n < 140$; this assumption holds if c is large enough. We also pick a constant a such that the function described by the $O(n)$ term above (which describes the non-recursive component of the running time of the algorithm) is bounded above by an for all $n > 0$. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

$$\begin{aligned} T(n) &\leq c\lceil n/5\rceil + c(7n/10+6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

which is at most cn if

$$\begin{aligned} 0 &\geq -cn/10 + 7c + an \\ c &\geq 10a(n/(n-70)) && \text{(when } n > 70\text{)} \end{aligned}$$

Therefore, since we assume $n \geq 140^*$, we have $n/(n-70) \leq 2$, and so choosing $c \geq 20a$ satisfies the equation, showing that the worst-case running time of SELECT is therefore linear (i.e., runs in $O(n)$ time).

* Note that there is nothing special about the constant 140; we could replace it by any integer strictly greater than 70 and then choose c accordingly.

Closest Pair 11

11.1 Closest Pair

Today, we consider another application of divide-and-conquer, which comes from the field of computational geometry. We are given a set P of n points in the plane, and we wish to find the closest pair of points $p, q \in P$ (see (a) in the figure below). This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall that, given two points $p = (p_x, p_y)$ and $q = (q_x, q_y)$, their (Euclidean) distance is

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Clearly, this problem can be solved by brute force in $O(n^2)$ time, by computing the distance between each pair, and returning the smallest. Today, we will present an $O(n \log n)$ time algorithm, which is based a clever use of divide-and-conquer.

Before getting into the solution, it is worth pointing out a simple strategy that fails to work. If two points are very close together, then clearly both their x-coordinates and their y-coordinates are close together. So, how about if we sort the points based on their x-coordinates and, for each point of the set, we'll consider just nearby points in the list. It would seem that (subject to figuring out exactly what "nearby" means) such a strategy might be made to work. The problem is that it could fail miserably. In particular, consider the point set of (b) in the figure below. The points p and q are the closest points, but we can place an arbitrarily large number of points between them in terms of their x-coordinates. We need to separate these points sufficiently far in terms of their y-coordinates that p and q remain the closest pair. As a result, the positions of p and q can be arbitrarily far apart in the sorted order. Of course, we can do the same with respect to the y-coordinate. Clearly, we cannot focus on one coordinate alone*.

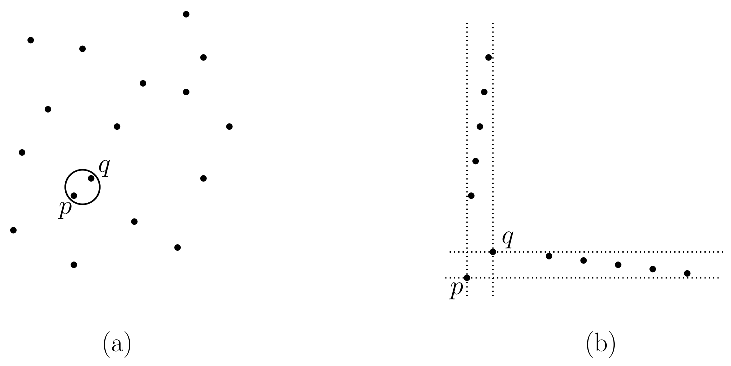


Figure 11.1: (a) The closest pair problem and (b) why sorting on x- or y-alone doesn't work.

These notes were adapted from the University of Maryland's CMSC 451 course

* While the above example shows that sorting along any one coordinate axis may fail, there is a variant of this strategy that can be used for computing nearest neighbors approximately. This approach is based on the observation that if two points are close together, their projections onto a *randomly oriented vector* will be close, and if they are far apart, their projections onto a randomly oriented vector will be far apart in expectation. This observation underlies a popular nearest neighbor algorithm called *locality sensitive hashing*

11.2 Divide and Conquer Algorithm

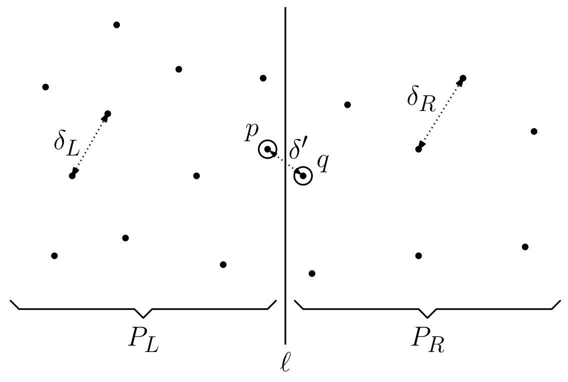
Let us investigate how to design an $O(n \log n)$ time divide and conquer approach to the problem. The input consists of a set of points P, represented, say, as an array of n elements, where each element stores the (x, y) coordinates of the point. (For simplicity, let's assume there are no duplicate x-coordinates.) The output will consist of a single number, being the closest distance. It is easy to modify the algorithm to also produce the pair of points that achieves this distance.

For reasons that will become clear later, in order to implement the algorithm efficiently, it will be helpful to begin by *presorting* the points, both with respect to their x- and y-coordinates. Let P_x be an array of points sorted by x, and let P_y be an array of points sorted by y. We can compute these sorted arrays in $O(n \log n)$ time. Note that this initial sorting is done only once. In particular, the recursive calls do not repeat the sorting process. Like any divide-and-conquer algorithm, after the initial base case, our approach involves three basic elements: divide, conquer, and combine.

Base Case: If $|P| < 3$, then just solve the problem by brute force in $O(1)$ time.

Divide: Otherwise, partition the points into two subarrays P_L and P_R based on their x-coordinates. In particular, imagine a vertical line ℓ that splits the points roughly in half (see figure below). Let P_L be the points to the left of ℓ and P_R be the points to the right of ℓ.

In the same way that we represented P using two sorted arrays, we do the same for P_L and P_R. Since we have presorted P_x by x-coordinates, we can determine the median element for ℓ in constant time. After this, we can partition each of arrays P_x and P_y in $O(n)$ time each.



Conquer: Compute the closest pair *within* each of the subsets P_L and P_R each, by invoking the algorithm recursively. Let δ_L and δ_R be the closest pair distances in each case (see figure above). Let $\delta = \min(\delta_L, \delta_R)$.

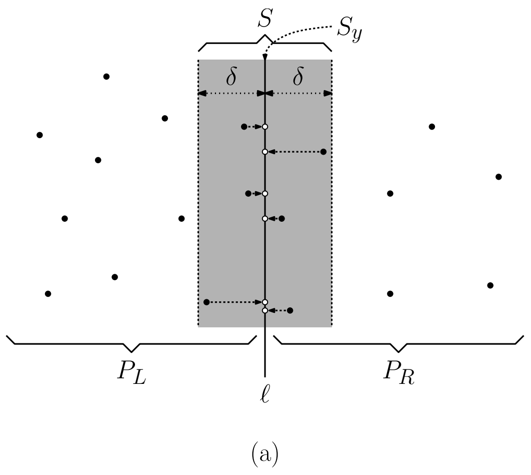
Combine: Note that δ is not necessarily the final answer, because there may be two points that are very close to one another but are on opposite sides of ℓ. To complete the algorithm, we want to determine the closest pair of points *between* the sets, that is, the closest points $p \in P_L$ and $q \in P_R$ (see figure above). Since we already have an upper bound on the closest pair, it suffices to solve the following restricted problem: if the closest pair (p, q) are within distance δ, then we will return such a pair, otherwise, we may return any pair. (This restriction is very important to the algorithm's efficiency.) In the next section, we'll show how to solve this restricted problem in $O(n)$ time. Given the closest such pair (p, q), let $\delta' = \|pq\|$. We return $\min(\delta, \delta')$ as the final result.

Assuming we can solve the "combine" step in $O(n)$ time, it follows that the algorithm's running time is given by $T(n) = 2T(n/2) + n$, and (as in Mergesort) the overall running time is $O(n \log n)$, as desired.

11.3 Closest Pair Between the Sets

To finish up the algorithm, we need to compute the closest pair p and q, where $p \in P_L$ and $q \in P_R$. As mentioned above, because we already know of the existence of two points within distance δ of each other, this algorithm is allowed to fail, if there is no such pair that is closer than δ. The input to our algorithm consists of the point set P, the x-coordinate of the vertical splitting line ℓ, and the value of $\delta = min(\delta_L, \delta_R)$. Recall that our goal is to do this in $O(n)$ time.

This is where the real creativity of the algorithm enters. Observe that if such a pair of points exists, we may assume that both points lie within distance δ of ℓ, for otherwise the resulting distance would exceed δ. Let S denote this subset of P that lies within a vertical strip of width $2 \cdot \delta$ centered about ℓ (see (a) in figure below)*.



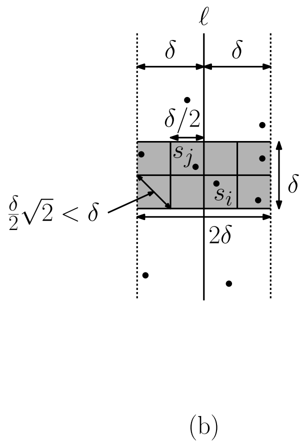


Figure 11.2: Closest pair in the strip.

How do we find the closest pair within S? Sorting comes to our rescue. Let $S_y = \langle s_1, ..., s_m \rangle$ denote the points of S sorted by their y-coordinates (see (a) in figure above). At the start of the notes, we asserted that considering the points that are close according to their x- or y-coordinate alone is not sufficient. It is rather surprising, therefore, that this does work for the set S_y.

The key observation is that if S_y contains two points that are within distance δ of each other, these two points must be within a constant number of positions of each other in the sorted array S_y. The following lemma formalizes this observation.

Lemma 1. *Given any two points $s_i, s_j \in S_y$, if $\|s_i s_j\| \leq \delta$, then $|j - i| \leq 7$.*

Proof. Suppose that $\|s_i s_j\| \leq \delta$. Since they are in S they are each within distance δ of ℓ. Clearly, the y-coordinates of these two points can differ by at most δ. So they must both reside in a rectangle of width 2δ and height δ centered about ℓ (see (b) in figure above). Split this rectangle into eight identical squares each of side length $\delta/2$. A square of side length x has a diagonal of length $x\sqrt{2}$, and no two points within such

* You might be tempted to think that we have pruned away many of the points of P, and this is the source of efficiency, but this is not generally true. It might very well be that *every* point of P lies within the strip, and so we cannot afford to apply a brute-force solution to our problem.

a square can be farther away than this. Therefore, the distance between any two points lying within one of these eight squares is at most

$$\frac{\delta\sqrt{2}}{2} = \frac{\delta}{\sqrt{2}} < \delta$$

Since each square lies entirely on one side of ℓ, no square can contain two or more points of P, since otherwise, these two points would contradict the fact that δ is the closest pair seen so far. Thus, there can be at most eight points of S in this rectangle, one for each square. Therefore, $|j - i| \leq 7$. □

Avoiding Repeated Sorting One issue that we have not yet addressed is how to compute S_y. Recall that we cannot afford to sort these points explicitly, because we may have n points in S, and this part of the algorithm needs to run in $O(n)$ time. This is where presorting comes in. Recall that the points of P_y are already sorted by y-coordinates. To compute S_y, we enumerate the points of P_y, and each time we find a point that lies within the strip, we copy it to the next position of array S_y. This runs in $O(n)$ time, and preserves the y-ordering of the points.

By the way, it is natural to wonder whether the value "8" in the statement of the lemma is optimal. Getting the best possible value is likely to be a tricky geometric exercise. Our textbook proves a weaker bound of "16". Of course, from the perspective of asymptotic complexity, the exact constant does not matter.

```
closestPair(P = (Px, Py)) {
    n = |P|
    if (n <= 3) solve by brute force                // base case
    else {
        Find the vertical line L through P's median // divide
        Split P into PL and PR (split Px and Py as well)
        dL = closestPair(PL)                        // conquer
        dR = closestPair(PR)
        d = min(dL, dR)
        for (i = 1 to n) {                          // create Sy
            if (Py[i] is within distance d of L)
                append Py[i] to Sy
        }
        d' = stripClosest(Sy)                       // closest in strip
        return min(d, d')                           // overall closest
    }
}

stripClosest(Sy) {                              // closest in strip
    m = |Sy|
    d' = infinity
    for (i = 1 to m) {
        for (j = i+1 to min(m, i+7)) {          // search neighbors
            if (dist(Sy[i], Sy[j]) <= d')
                d' = dist(Sy[i], Sy[j])         // new closest found
        }
    }
    return d'
}
```

Integer Multiplication | 12

12.1 Introduction and Problem Statement

The problem we consider is extremely basic: how do we multiply two integers? In elementary school, you were taught a concrete (and fairly efficient) algorithm to multiply two n-digit numbers x and y. You first compute a "partial product" by multiplying each digit of y separately by x, and then you add up all the partial products, shifting where necessary.

$$\begin{array}{r} 12 \\ \underline{\times 13} \\ 36 \\ \underline{+12} \\ 156 \end{array} \qquad\qquad \begin{array}{r} 1100 \\ \underline{\times 1101} \\ 1100 \\ 0000 \\ 1100 \\ \underline{+\ 1100} \\ 10011100 \end{array}$$

Figure 12.1: The elementary school algorithm for multiplying two integers in decimal and binary representation.

Note that while in elementary school, you learned how to multiply decimal numbers, in computer science, we care about binary numbers. Either way, the same algorithm works.

How long does this algorithm take to multiply two n bit numbers? Counting a single operation on a pair of bits as one primitive step in the computation, it takes $O(n)$ time to compute each partial product, and $O(n)$ time to combine it in with the running sum of partial products so far. Since there are n partial products, this gives a total running time of $O(n^2)$.

In fact, there exists a faster algorithm which uses the divide and conquer strategy to achieve a much better runtime.

12.2 Designing the Algorithm

The improved algorithm is based on a more clever way to break up the product into partial sums. Let's assume we are in base 2 (it doesn't really matter) and we are given two binary numbers x and y which we want to multiply. We start by writing x as

$$x = x_1 \cdot 2^{n/2} + x_0$$

In other words, x_1 corresponds to the $n/2$ higher-order bits and x_0 corresponds to the $n/2$ lower-order bits. Similarly, write $y = y_1 \cdot 2^{n/2} + y_0$. Then we have

$$\begin{aligned} xy &= \left(x_1 \cdot 2^{n/2} + x_0\right)\left(y_1 \cdot 2^{n/2} + y_0\right) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

These notes were adapted from Kleinberg and Tardos' *Algorithm Design*

Hence we have reduced the problem of solving a single n-bit instance (multiplying the two n-bit numbers x and y) into the problem of solving four $n/2$-bit instances (computing the products x_1y_1, x_1y_0, x_0y_1, and x_0y_0). So we have a first candidate for a divide-and-conquer algorithm: recursively compute the results for these four $n/2$-bit instances, and then combine them using the above equation. The combining requires a constant number of additions and shifts of $O(n)$-bit numbers, so it takes $O(n)$ time. Thus the running time is given by the recurrence

$$T(n) = 4T(n/2) + cn$$

(where we have ignored the base case for simplicity). Here c is just a constant.

Unfortunately, you can check that solving this recurrence gives us $T(n) = \Theta(n^2)$, which is no better than the elementary school algorithm!

In order to speed up our algorithm, we can try to reduce the number of recursive calls made from four to three. In that case, we'd have $T(n) = \Theta(n^{\log_2 3}) = O(n^{1.59})$ which is much better than $\Theta(n^2)$.

Now in order to compute the value of the expression

$$x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$$

using only three recursive calls instead of four, we need to use a little trick. Remember, as part of our computation, we need to compute $(x_1y_0 + x_0y_1)$. In particular, we don't need to explicitly compute x_1y_0 and x_0y_1 separately: we just need their sum!

Hence consider the result of the single multiplication $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$. This has the four products above added together, at the cost of a single recursive multiplication. If we now also determine x_1y_1 and x_0y_0 by recursion, then we get the outermost terms explicitly, and we get the middle term by subtracting x_1y_1 and x_0y_0 away from $(x_1 + x_0)(y_1 + y_0)$.

This gives the full algorithm:

Fast Integer Multiplcation

Input: Two n-bit numbers x and y, where n is assumed to be a power of 2.

Output: The product xy

```
Recursive-Multiply($x, y$)
    Let $x = x_1 \cdot 2^{n/2} + x_0$
    Let $y = y_1 \cdot 2^{n/2} + y_0$

    a = $x_1 + x_0$
    b = $y_1 + y_0$
    c = Recursive-Multiply(a, b)
    p = Recursive-Multiply($x_1, y_1$)
    q = Recursive-Multiply($x_0, y_0$)

    Return p*$2^n$ + (c - p - q)*$2^{n/2}$ + q
```

12.3 Runtime

As stated above, the running time is $T(n) = \Theta\left(n^{\lg 3}\right) = O(n^{1.59})$. This comes from the following recurrence:

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

where c is a constant and where we ignore the base case.

We can then solve the recurrence (ignoring the constant c) as follows:

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

Assuming a base case of $T(1) \leq 1$, the recurrence bottoms out when $n/2^k = 1 \quad \Rightarrow \quad k = \log_2 n$. Plugging this in gives:

$$\begin{aligned}
3^k T(n/2^k) + n \cdot \sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i &= 3^k + n \cdot \left(\frac{\left(\frac{3}{2}\right)^k - 1}{\frac{3}{2} - 1}\right) \\
&= 3^{\log_2 n} + n \cdot \left(\frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{3}{2} - 1}\right) \\
&= n^{\log_2 3} + 2n \cdot \left(n^{\log_2 3 - 1} - 1\right) \\
&= n^{\log_2 3} + 2n^{\log_2 3} - 2n \\
&= 3n^{\log_2 3} - 2n \\
&= \Theta\left(n^{\log_2 3}\right)
\end{aligned}$$

Above we used the fact that $n^{\log_b a} = a^{\log_b n}$.

Stacks and Queues 13

13.1 The Stack ADT

An *abstract data type* (ADT) is an abstraction of a data structure. It specifies the type of data stored and different operations that can be performed on the data. It's like a Java interface—it specifies the name and definition of the methods, but hides their implementations.

In the stack ADT, the data can be arbitrary objects and the main operations are `push` and `pop` which allow insertions and deletions in a last-in, first-out manner. One way to implement a stack is to use an array. Note that an array has a fixed size, so in a fixed capacity implementation of the stack, the number of items in the stack can be no more than the size of the array. This implies that to use the stack one must estimate the maximum size of the stack ahead of time. To make the stack have unlimited capacity, we will adjust dynamically the size of the array so that it is both sufficiently large to store all of the items and not so large so as to waste an excessive amount of space. We will consider two strategies—the *incremental strategy* in which the array capacity is increased by a constant amount when the stack size equals the array capacity and the *doubling strategy*, in which the size of the array is doubled when the stack size equals the array capacity. Since arrays cannot be dynamically resized, we have to create a new array of increased size and copy the elements from the old array to the new array.

We will first analyze the incremental strategy.

```
push(obj)
    // s: stack size
    // a: array capacity
    // c: initial array size and also the increment in array size
    A[s] = obj
    s = s + 1
    if s == a then
        a = a + c
        copy contents of old array to the new array
```

An example sequence of operations is below:

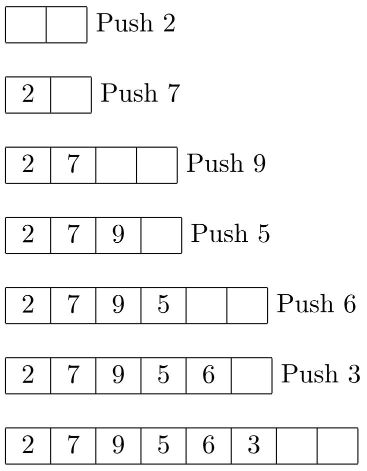


Figure 13.1: A sequence of `Push` operations using the incremental strategy. Here $c = 2$.

Let c be the initial size of the array and the amount by which the array size is increased during each expansion. Consider a sequence of n push operations. Note that after every c push operations, the array expansion happens. The n push operations cost n. The cost of the first expansion is $c + 2c$ (since c elements are being copied and we need to allocate a new array of size $2c$), the cost of the second expansion is $2c + 3c$ (since $2c$ elements are being copied and we need to allocate a new array of size $3c$), and so on. We can separate these expressions into two parts: the cost of allocations and the cost of copying. The cost of allocations is $2c + 3c + ... + n + (n + c)$ and the cost of copying is $c + 2c + ... + n$.

Thus the total cost of n consecutive push operations is given by

$$\begin{aligned}T(n) &= \text{cost of pushes} + \text{cost of copying} + \text{cost of allocations}\\ &= n + (c + 2c + ... + n) + (2c + 3c + ... + n + c)\\ &= n + c(1 + 2 + ... + n/c) + c(2 + 3 + ... + n/c + 1)\\ &= n + c \cdot \left(\frac{n/c(n/c+1)}{2}\right) + c\left(\frac{(n/c+1)(n/c+2)}{2} - 1\right)\\ &= \Theta(n^2)\end{aligned}$$

since c is a constant.

We will now analyze the doubling method. The pseudocode is:

```
push(obj)
    // s: stack size
    // a: array capacity
    A[s] = obj
    s = s + 1
    if s == a then
        a = 2 * a
        copy contents of old array to the new array
```

We will be using *amortized analysis*, which means that we will be finding the time-averaged cost for a sequence of operations. In other words, the amortized runtime of an operation is the time required to perform a sequence of operations averaged over all the operations performed. Let $T(n)$ be the total time needed to perform a series of n push operations. Then the amortized time of a single push operation is $T(n)/n$. Note that this is different from the notion of "average case analysis"—we're not making any assumptions about inputs begin chosen at random, nor are we assuming any probability distribution over the input. We are just averaging over time. Also note that the total real cost of a sequence of operations will be bounded by the total of the amortized costs of all the operations.

Let s denote the number of objects in the stack at any given time and let a be the array capacity at any given time. When $s < a$, `push(obj)` is a constant time operation. However, when the stack is full, i.e. when $s = a$, we double the size of the array. Thus the cost of `push(obj)` in this case is $O(s)$, as we have to copy s items from the old array into the new array (the cost of allocating and freeing the array is $O(s)$). The worst case cost of a push operation is $O(n)$, and hence the cost of n push operations is $O(n \log n)$ (since there are $O(\log n)$ expansions). Is this tight?

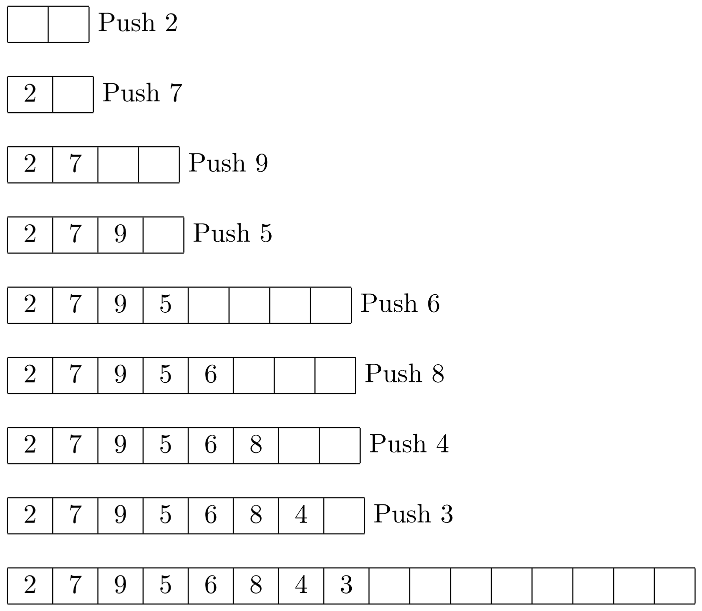


Figure 13.2: A sequence of `Push` operations using the doubling strategy.

The above analysis did NOT use amortization. Let's see if we can get a better bound using amortized analysis. If we start from an empty stack, what is the cost of a sequence of n push operations? As before, the cost of the n push operations ignoring expansions is n. The cost of allocating new arrays is at most $1 + 2 + 4 + ... + n + 2n < 4n$ and the cost of copying elements is at most $1 + 2 + 4 + ... + n/2 + n < 2n$. Thus the total cost is at most $7n = O(n)$. The amortized cost of an operation is $7 = O(1)$, even though the worst case time complexity of a single push operation is $O(n)$.

Another way to analyze the doubling scheme is as follows: Each element will be allocated 7 dollars. Once the element is pushed, it uses 1 dollar. When the array needs to be doubled, only elements that have never been moved before pay for all the elements that are being moved to the new array. Since the number of such elements is exactly half the number of all elements in the stack, each never-been-moved element pays a cost of 6 for the move–2 for allocating space for itself and copying itself over, 2 for allocating space of one of the elements in the first half of the stack and copying it over, and 2 for allocating two more empty slots since the array is being doubled. Thus the total cost incurred by each element is 7, and hence $T(n) \leq 7n = O(n)$.

Similarly, in `pop()`, after removing the object from the stack, if the stack size is significantly less than the array capacity, then we resize the array (we don't want to waste space). More specifically, when the stack size is equal to one-fourth of the array size, then we reduce the size of the array to half its current capacity. After resizing, the array is still half full and can accomodate a substantial number of push and pop operations before having to resize again. The pseudocode for the pop operations is as follows:

```
pop()
    item = A[s]
    s = s - 1
    if s < a/4 then
        a = a/2
```

Why do we only resize when the array is less than one fourth full, rather than one half full? Consider what would happen if a malicious user pushed elements onto the stack until it resized up. Then it popped a single element—this would trigger another resizing down. Then it pushed a single element—this would trigger another resizing up. And so on. In this worst case, every push/pop operation would require copying elements, leading to very bad running times. Resizing the array when it is less than one-fourth full prevents this "thrashing" problem.

13.2 Queues

A queue is a collection of objects that is based on a first-in, first-out policy. The main operations in a Queue ADT are `enqueue(obj)` and `dequeue`. The `enqueue` operation inserts an element at the end of the queue. The `dequeue` operation removes and returns the element at the front of the queue. Queues can also be implemented using expandable arrays. However, unlike a stack, in a queue we need to keep track of both the head and the tail of the queue. The head of the queue points to the first element in the queue and the tail points to the last element in the queue. Note that when we dequeue an element, the element is removed from the head of the queue and when an element is enqueued, that element is inserted at the tail of the queue. When the tail points to the end of the array, it may not mean that the array is full. This is because some elements may have been popped off and the head of the queue may not be pointing to the beginning of the array. That is, there may be room at the beginning of the array. To address this we use a wrap-around implementation. This way, we expand the array only when every slot in the array contains an element, i.e., when the queue size equals the array capacity. When copying the queue elements into the new array, we can "unwind" the queue so that the head points to the beginning of the array.

Note that this is only one possible implementation of a queue. There are countless others, including singly- or doubly-linked lists, circular linked lists, etc.

14 Binary Heaps and Heapsort

A heap is a type of data structure. One of the interesting things about heaps is that they allow you to find the largest element in the heap in $O(1)$ time. (Recall that in certain other data structures, like arrays, this operation takes $O(n)$ time.) Furthermore, extracting the largest element from the heap (i.e., finding and removing it) takes $O(\log n)$ time. These properties make heaps very useful for implementing a *priority queue*, which we'll get to later. They also give rise to an $O(n \log n)$ sorting algorithm, Heapsort, which works by repeatedly extracting the largest element until we have emptied the heap.

14.1 Definitions and Implementation

Definition. The binary heap data structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap. In a max-heap, the max-heap property is that for every node i other than the root, $A[\text{Parent}(i)] \geq A[i]$, that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.

A min-heap is organized in the opposite way; the min-heap property is that for every node i other than the root, $A[\text{Parent}(i)] \leq A[i]$. The smallest element in a min-heap is at the root.

Definition. Viewing a heap as a tree, we define the height of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is $\lfloor \lg n \rfloor$.

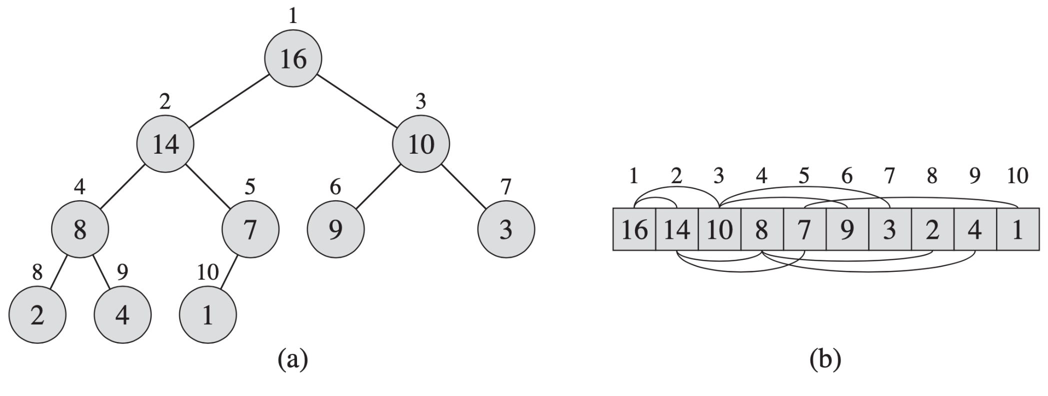


Figure 14.1: A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

These notes were adapted from CLRS Chapter 6 and CS 161 at Stanford University

As mentioned above, you can implement a heap as an array. This array is essentially populated by "reading off" the numbers in the tree, from left to right and from top to bottom (i.e., level order).

The root is stored at index 1, and if a node is at index i, then:

- Parent(i): **return** $\lfloor i/2 \rfloor$
- Left(i): **return** $2i$
- Right(i): **return** $2i + 1$

Furthermore, for the heap array A, we store two properties: $A.length$, which is the number of elements in the array, and $A.heapsize$, which is the number of array elements that are actually part of the heap. Even though A is potentially filled with numbers, only the elements in $A[1..A.heapsize]$ are actually part of the heap.

Note: The leaves of the heap are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, ..., n$, so there are $\sim n/2 = O(n)$ elements which are at the leaves.

We shall see that the basic operations on heaps run in time at most proportional to the height of the tree and thus take $O(\lg n)$ time. The remainder of this chapter presents some basic procedures and shows how they are used in a sorting algorithm and a priority-queue data structure.

- The Max-Heapify procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The Build-Max-Heap procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The Heapsort procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The Max-Heap-Insert, Heap-Extract-Max, Heap-Increase-Key, Heap-Maximum procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.

Note: we will be discussing max-heaps in the following sections, with the understanding that the algorithms for min-heaps are analogous.

14.2 Maintaining the Heap Property

In order to maintain the max-heap property, we call the procedure Max-Heapify. Its inputs are an array A and an index i into the array. When it is called, Max-Heapify assumes that the binary trees rooted at Left(i) and Right(i) are max-heaps, but that $A[i]$ might be smaller than its children, thus violating the max-heap property. Max-Heapify lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

```
Max-Heapify($A, i$):
  $l$ = Left($i$)
  $r$ = Right($i$)
  if $l \leq A.heapsize$ and $A[l] > A[i]$
      $largest = l$
  else $largest = i$
  if $r \leq A.heapsize$ and $A[r] > A[largest]$
      $largest = r$
  if $largest \neq i$
      exchange $A[i]$ with $A[largest]$
      Max-Heapify($A, largest$)
```

The figure below illustrates how Max-Heapify works. At each step, the largest of the elements $A[i]$, $A[\textsc{Left}(i)]$, and $A[\textsc{Right}(i)]$ is determined, and its index is stored in *largest*. If $A[i]$ is largest, then the subtree rooted at node i is already a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element, and $A[i]$ is swapped with $A[largest]$, which causes node i and its children to satisfy the max-heap property. The node indexed by *largest*, however, now has the original value $A[i]$, and thus the subtree rooted at largest might violate the max-heap property. Consequently, we call Max-Heapify recursively on that subtree.

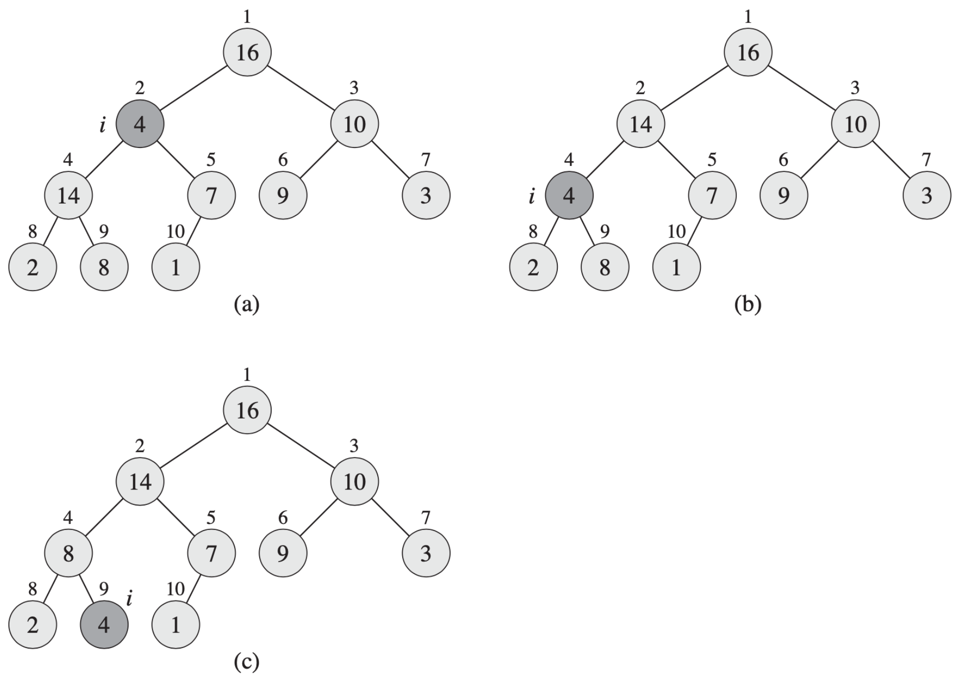


Figure 14.2: The action of Max-Heapify$(A, 2)$, where $A.heapsize = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call Max-Heapify$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call Max-Heapify$(A, 9)$ yields no further change to the data structure.

The running time of Max-Heapify on a subtree of size n rooted at a given node i is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\textsc{Left}(i)]$, and $A[\textsc{Right}(i)]$, plus the time to run Max-Heapify on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children's subtrees each have size at most $2n/3$—the worst case occurs when the bottom level of the tree is exactly half full—and therefore we can describe the running time of Max-Heapify by the recurrence

$$T(n) \leq T(2n/3) + \Theta(1)$$

which by the Simplified Master Theorem is $T(n) = O(\lg n)$. Alternatively, we can characterize the running time of Max-Heapify on a node of height h as $O(h)$.

14.3 Building a Heap

The Build-Max-Heap procedure runs Max-Heapify on all the nodes in the heap, starting at the nodes right above the leaves and moving towards the root, to convert an array $A[1..n]$, where $n = A.length$, into a max-heap. We start at the bottom because in order to run Max-Heapify on a node, we need the subtrees of that node to already be heaps.

Recall that the elements in the subarray $A[(\lfloor n/2 \rfloor + 1)..n]$ are all leaves of the tree, and so each is a 1-element heap to begin with. The procedure Build-Max-Heap goes through the remaining nodes of the tree and runs Max-Heapify on each one.

```
Build-Max-Heap($A$):
  $A.heapsize = A.length$
  for $i = \lfloor A.length/2 \rfloor$ downto 1
      Max-Heapify($A, i$)
```

See the figure on the next page for a worked-through example of Build-Max-Heap.

Correctness

To show why Build-Max-Heap works correctly, we use the following loop invariant: at the start of each iteration of the **for** loop of lines 2–3, each node $i+1, i+2, ..., n$ is the root of a max-heap.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, ..., n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call Max-Heapify(A, i) to make node i a max-heap root. Moreover, the Max-Heapify call preserves the property that nodes $i+1, i+2, ..., n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, $i = 0$. By the loop invariant, each node $1, 2, ..., n$ is the root of a max-heap. In particular, node 1 is a max-heap.

Runtime

Simple Upper Bound: We can compute a simple upper bound on the running time of Build-Max-Heap as follows. Each call to Max-Heapify costs $O(\lg n)$ time, and Build-Max-Heap makes $O(n)$ such calls. Thus, the running time is $O(n \lg n)$. This upper bound, though correct, is not asymptotically tight.

Tighter Upper Bound: Recall that an n-element heap has height $\lfloor \lg n \rfloor$. In addition, an n element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of any height h (this is Exercise 6.3-3 in CLRS, and we won't solve it here, but you can prove it by induction). The time required by Max-Heapify when called on a node of height h is $O(h)$, and so we can express the total cost of Build-Max-Heap as being bounded from above by

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)^* = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

Hence, we can build a max-heap from an unordered array in $O(n)$ time.

* evaluate this summation by substituting $x = 1/2$ into CLRS formula $(A.8)$

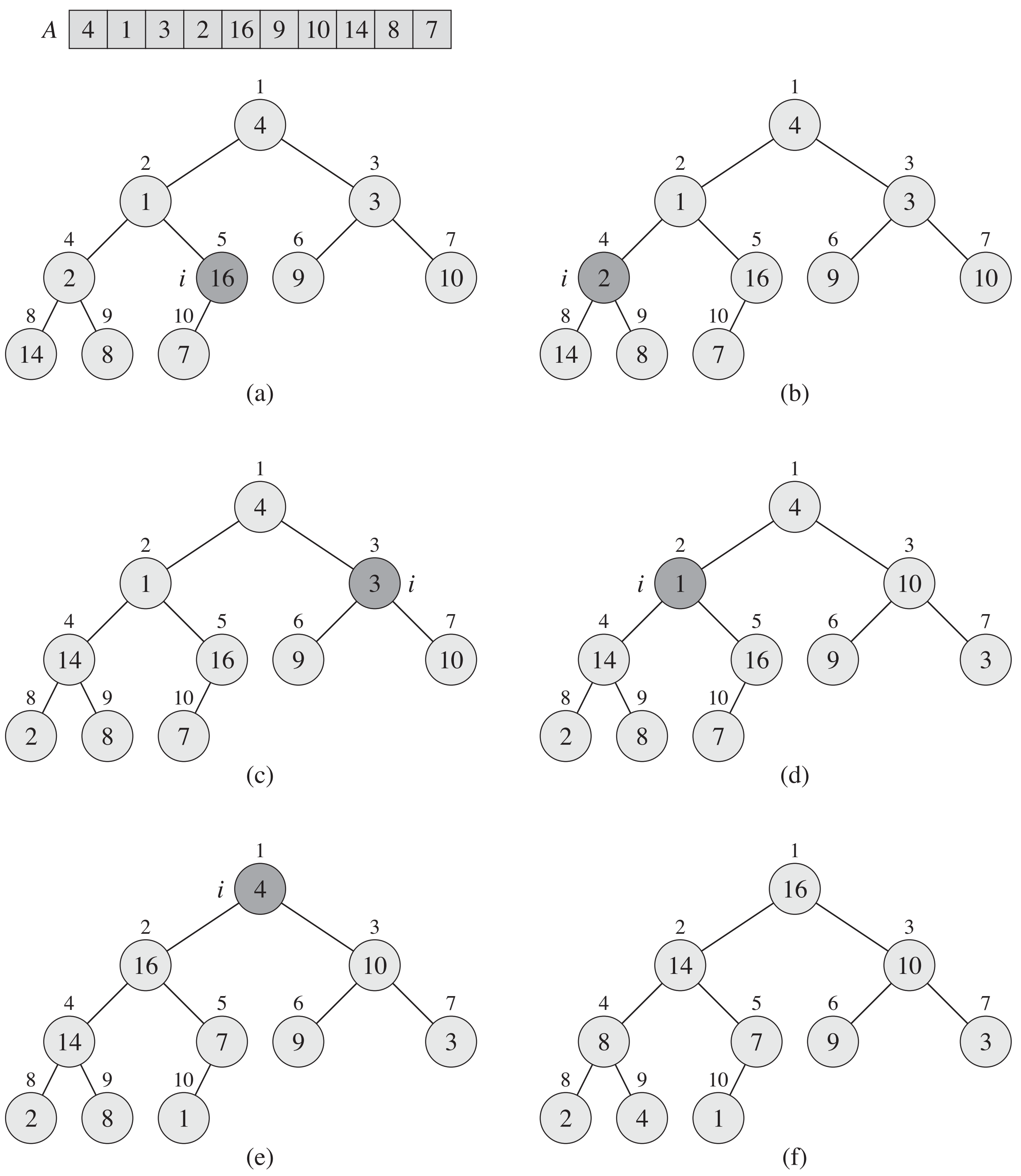


Figure 14.3: The operation of Build-Max-Heap, showing the data structure before the call to MAX-HEAPIFY in line 3 of Build-Max-Heap. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call Max-Heapify(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the for loop in Build-Max-Heap. Observe that whenever Max-Heapify is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after Build-Max-Heap finishes.

14.4 Heapsort

Heapsort is a way of sorting arrays. The Heapsort algorithm starts by using Build-Max-Heap to build a max-heap on the input array $A[1..n]$, where $n = A.length$. Since the maximum element of the array is stored at the root $A[1]$, we can put it into its correct final position by exchanging it with $A[n]$. If we now discard node n from the heap–and we can do so by simply decrementing $A.heapsize$–we observe that the children of the root remain max-heaps, but the new root element might violate the max-heap property. All we need to do to restore the max-heap property, however, is call Max-Heapify$(A, 1)$, which leaves a max-heap in $A[1..n-1]$. The Heapsort algorithm then repeats this process for the max-heap of size $n-1$ down to a heap of size 2.

```
Heapsort($A$):
  Build-Max-Heap($A$)
  **for** $i = A.length$ **downto** 2
      exchange $A[1]$ with $A[i]$
      $A.heapsize = A.heapsize - 1$
      Max-Heapify($A, 1$)
```

See the figure on the next page for a worked-through example of Heapsort.

14.5 Priority Queues

Earlier we said that heaps could be used to implement priority queues. A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A max-priority queue supports the following operations:

- Insert(S, x) inserts the element x into the set S.
- Maximum(S) returns the element of S with the largest key.
- Extract-Max(S) removes and returns the element with the largest key.
- Increase-Key(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

In addition, there are min-priority queues, which support the analogous operations Minimum, Extract-Min, and Decrease-Key.

A max-priority queue can be used to schedule jobs on a shared computer, where each job has a priority level. Every time a job is finished, we pick the highest-priority job to run next, and we do this using Extract-Max. Furthermore, we can add new jobs to the queue by calling Insert.

In addition, min-priority queues are useful in Dijkstra's algorithm for finding shortest paths in graphs, which we will see later in class.

The procedure Heap-Maximum implements the Maximum operation in $\Theta(1)$ time.

```
Heap-Maximum($A$):
  **return** $A[1]$
```

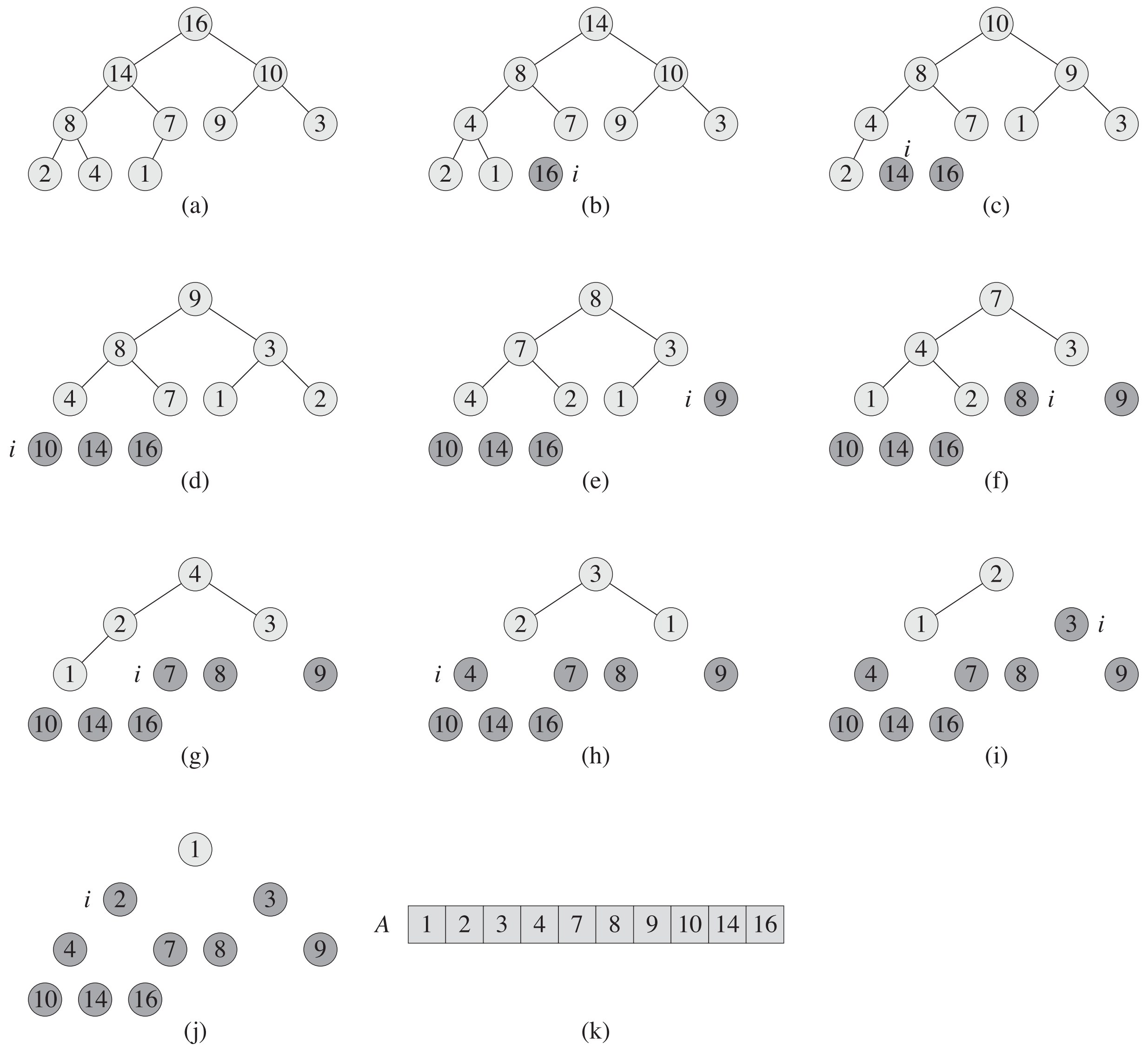


Figure 14.4: The operation of Heapsort. **(a)** The max-heap data structure just after Build-Max-Heap has built it in line 1. **(b)–(j)** The max-heap just after each call of Max-Heapify in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

The procedure Heap-Extract-Max implements the Extract-Max operation. It is similar to Heapsort, where we swapped the first element with the last element of the array, decreased the size of the heap by 1 and then called Max-Heapify.

```
Heap-Extract-Max($A$):
    if $A.heapsize < 1$
        error "heap underflow"
    $max = A[1]$
    $A[1] = A[A.heapsize]$
    $A.heapsize = A.heapsize - 1$
    Max-Heapify($A, 1$)
    return $max$
```

The running time of Heap-Extract-Max is $O(\lg n)$, since it performs only a constant amount of work on top of the $O(\lg n)$ time for Max-Heapify.

The procedure Heap-Increase-Key implements the Increase-Key operation. We increase the key of the node, then "push" the node up through the heap until it is greater than all its children. We do this by repeatedly swapping the node with its parent.

```
Heap-Increase-Key($A, i, key$):
    if $key < A[i]$
        error "new key is smaller than current key"
    $A[i] = key$
    while $i > 1$ and $A[\text{Parent}(i)] < A[i]$
        exchange $A[i]$ with $A[\text{Parent}(i)]$
        $i = \text{Parent}(i)$
```

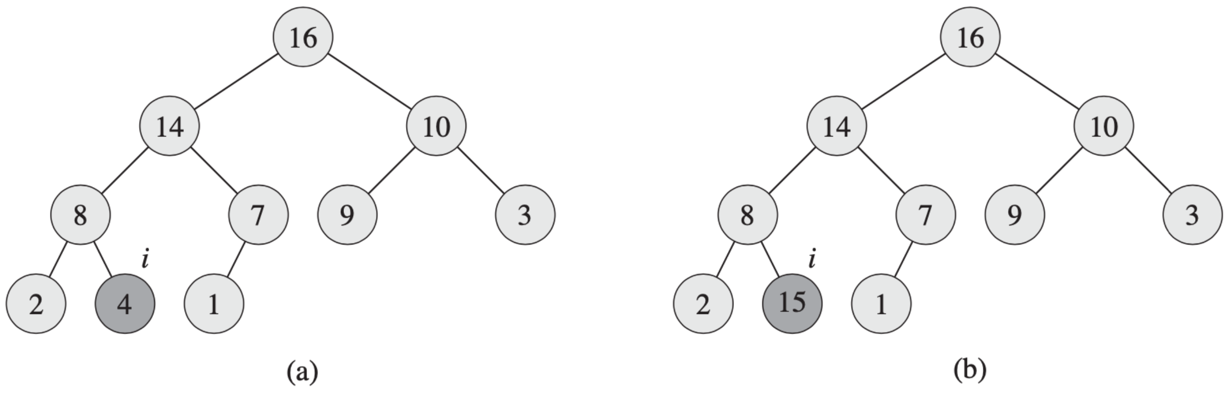
The running time of Heap-Increase-Key on an n-element heap is $O(\lg n)$, since the path traced from the node updated in line 3 to the root has length $O(\lg n)$. See the figure on the next page for a worked-through example of Heap-Increase-Key.

The procedure Max-Heap-Insert implements the Insert operation. It takes as an input the key of the new element to be inserted into max-heap A. The procedure first expands the max-heap by adding to the tree a new leaf whose key is $-\infty$. Then it calls Heap-Increase-Key to set the key of this new node to its correct value and maintain the max-heap property.

```
Max-Heap-Insert($A, key$):
    $A.heapsize = A.heapsize + 1$
    $A[A.heapsize] = -\infty$
    Heap-Increase-Key($A, A.heapsize, key$)
```

The running time of Max-Heap-Insert on an n-element heap is $O(\lg n)$.

In summary, a heap can support any priority-queue operation on a set of size n in $O(\lg n)$ time.



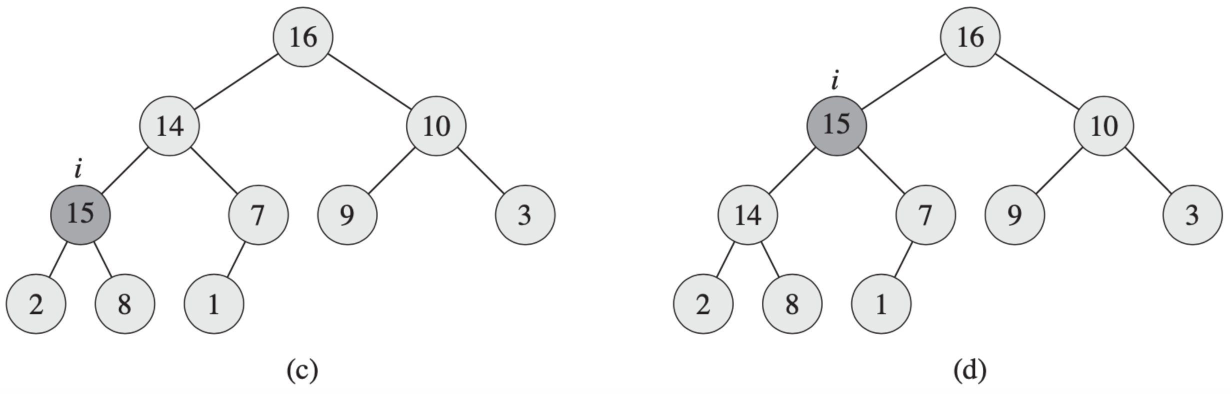


Figure 14.5: The operation of Heap-Increase-Key. **(a)** The max-heap of the previous figure with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{Parent}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

Huffman Coding 15

15.1 From Text to Bits

Computers ultimately operate on sequences of bits. As a result, one needs encoding schemes that take text written in richer alphabets (such as the alphabets underpinning human languages) and converts this text into strings of bits. Moreover, in many cases, it is useful to do so in a way that compresses the data. For example, when large files need to be shipped across communication networks, or stored on hard disks, it is important to represent them as compactly as possible without losing information. As a result, compression algorithms are a big focus in research. Before we jump into data compression algorithms, though, let's return to the problem of turning characters into bit sequences.

The simplest way to do this would be to use a fixed number of bits for each symbol in the alphabet, and then just concatenate the bit strings for each symbol to form the text. Since you can form 2^b different sequences using b bits, you would need $\lceil \lg n \rceil$-bit symbols to encode an alphabet of n characters. For example, to encode the 26 letters of the English alphabet, plus "space," you'd need 5 bit symbols. In fact, encoding schemes like ASCII work precisely this way, except they use a larger number of bits per symbol (eight in the case of ASCII) in order to handle larger alphabets which include capital letters, parenthesis, and a bunch of other special symbols.

This method certainly gets the job done, but we should ask ourselves: is there anything more we could ask for from an encoding scheme, maybe with respect to data compression? We can't just use fewer bits per encoding: if we only used four bits in the example above, we could only have $2^4 = 16$ symbols. However, what if we used fewer bits for *some* characters? As it currently stands (using the above example), we use five bits for every character, so obviously the average number of bits per character is five. In some cases, though, we may be able to use fewer than five bits per character *on average*. Think about it: letters like e, t and a get used much more frequently than q and x (by more than an order of magnitude, in fact). So it's a big waste to translate them all into the same number of bits. Instead, we could use a small number of bits for the frequent letters, and a larger number of bits for the less frequent ones, and hope to end up using fewer than five bits per letter when we average over a long string of text.

15.2 Variable-Length Encoding Schemes

In fact, this idea was used by Samuel Morse when developing *Morse Code*, which translates letters to dots and dashes (you can think of these as 0's and 1's). Morse consulted local printing presses to get frequency estimates for the letters in English and constructed his namesake code accordingly ("e" is a single dot—0, "t" a single dash—1, "a" is dot-dash—01, etc.).

Morse code, however, uses such short strings for letters that the encoding of words becomes ambiguous. For example, the string 0101 could correspond to any one of $eta, aa, etet$, or aet. Morse dealt with this ambiguity by putting pauses between letters. While this certainly eliminates ambiguity, it also means that now we require three symbols (dot, dash, pause) instead of only two (dot, dash). Since our goal is to translate into bits, we need to use a different method.

These notes were adapted from Kleinberg and Tardos' *Algorithm Design*

Prefix Codes

The ambiguity in Morse code arises because there are pairs of letters where the bit string that encodes one is a *prefix* of the bit string that encodes the other. To eliminate this problem, and hence to obtain an encoding scheme that has a well-defined interpretation for every sequence of bits, we need the encoding to be *prefix-free*:

Definition. A prefix code for a set S of letters is a function γ that maps each letters $x \in S$ to some sequence of zeros and ones, in such a way that for any distinct $x, y \in S$, the sequence $\gamma(x)$ is not a prefix of the sequence $\gamma(y)$.

Why does this ensure every encoding has a well-defined interpretation? Suppose we have a text consisting of letters $x_1x_2...x_n$. If we encode each letter as a bit sequence using an encoding function γ, then concatenate all the bit sequences together to get $\gamma(x_1)\gamma(x_2)...\gamma(x_n)$, then reconstructing the original text can be done by following these steps:

- Scan the bit sequence from left to right
- As soon as you've seen enough bits to match the encoding of some letter, output this as the first letter of the text. This must be the correct first letter, since no shorter or longer prefix of the bit sequence could encode any other letter
- Now delete the corresponding set of bits from the front of the message and iterate.

As an example, suppose $S = \{a, b, c, d, e\}$ and we have an encoding γ specified by

$$\begin{aligned}\gamma(a) &= 11\\ \gamma(b) &= 01\\ \gamma(c) &= 001\\ \gamma(d) &= 10\\ \gamma(e) &= 000\end{aligned}$$

Given the string 0010000011101, we see that c must be the first character, so now we are left with 0000011101. e is clearly the second character, leaving 0011101. c is next, leaving 1101. Hence a followed by b end the string. The final translation is *cecab*. The fact that γ is prefix free makes the translation straightforward.

Optimal Prefix Codes

We ultimately want to give more frequent characters shorter encodings. First we must introduce some notation.

Assume S is our alphabet. For each character $x \in S$, we assume there is a frequency f_x representing the fraction of letters in the text equal to x. That is, we assume a probability distribution over the letters in the texts we will be encoding. In a text with n letters, we expect nf_x of them to be equal to x. Also, we must have

$$\sum_{x \in S} f_x = 1$$

since the frequencies must sum to 1.

If we use a prefix code γ to encode a text of length n, what is the total length of the encoding? Writing $|\gamma(x)|$ to denote the number of bits in the encoding $\gamma(x)$, the expected encoding length of the full text is given by

$$\text{encoding length} = \sum_{x \in S} n f_x \cdot |\gamma(x)| = n \sum_{x \in S} f_x \cdot |\gamma(x)|$$

That is, we simply take the sum over all letters $x \in S$ of the number of times x occurs in the text (nf_x) times the length of the bit string $\gamma(x)$ used to encode x. The average number of bits per letter is simply the encoding length divided by the number of letters:

$$\text{Average bits per letter} = \text{ABL}(\gamma) = \sum_{x \in S} f_x \cdot |\gamma(x)|$$

Using the previous example where $S = \{a, b, c, d, e\}$, suppose the frequencies were given by

$$f_a = 0.32 \quad f_b = 0.25 \quad f_c = 0.20 \quad f_d = 0.18 \quad f_e = 0.05$$

Then using the same encoding from above, we have

$$\text{ABL}(\gamma) = 0.32 \cdot 2 + 0.25 \cdot 2 + 0.20 \cdot 3 + 0.18 \cdot 2 + 0.05 \cdot 3 = 2.25$$

Note that a fixed length encoding with five characters would require $\lceil \lg 5 \rceil = 3$ bits per character. Thus using the code γ reduces the bits per letter from 3 to 2.25, a saving of 25%. In fact, there are even better encodings for this example.

Definition. An optimal prefix code is as efficient as possible. That is, given an alphabet S and a set of frequencies for the letters in S, it minimizes the average number of bits per letter $\text{ABL}(\gamma) = \sum_{x \in S} f_x \cdot |\gamma(x)|$

This leads us to the underlying question: Given an alphabet S and a set of frequencies for the letters in S, how do we produce an optimal prefix code?

15.3 Huffman Encoding

Binary Trees and Prefix Codes

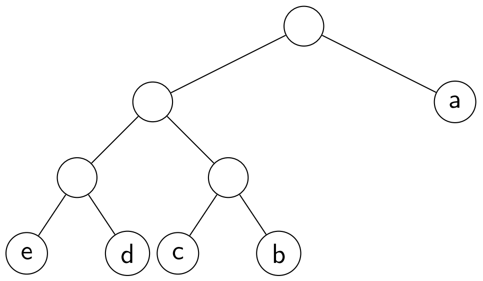
A key insight into this problem is developing a tree-based representation of prefix codes. Suppose we take a rooted tree T in which each node that is not a leaf has exactly two children; i.e. T is a *binary tree*. Further, suppose that the number of leaves is equal to the size of the alphabet S, and we label each leaf with a distinct letter in S.

Such a labeled binary tree T naturally describes a prefix code. For each letter $x \in S$, we follow the path from the root to the leaf labeled x. Each time the path goes from a node to its left child, we write down a 0, and each time the path goes from a node to its right child, we write down a 1. We take the resulting string of bits as the encoding for x.

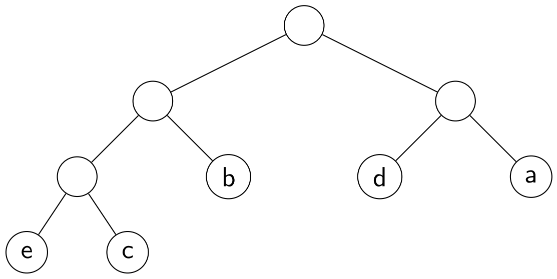
Lemma 1. *The encoding of S constructed from T as described above is a prefix code.*

Proof. In order for the encoding of x to be a prefix of the encoding for some other character y, the path from the root to x would have to be a prefix of the path from the root to y. But this is the same as saying that x would be on the path from the root to y, which isn't possible since x is a leaf. □

In fact, this relationship between binary trees and prefix codes goes the other way too: given a prefix code γ, we can build a binary tree recursively as follows. We start with a root. All letters $x \in S$ whose encodings begin with a 0 will be leaves in the left subtree of the root, and all the letters $y \in S$ whose encodings begin with a 1 will be leaves in the right subtree of the root. Then, just build the two subtrees recursively using this rule.



Character	Encoding
a	1
b	011
c	010
d	001
e	000



Character	Encoding
a	11
b	01
c	001
d	10
e	000

Figure 15.1: Examples of the correspondence between prefix codes and binary trees.

Thus by this equivalence, the search for an optimal prefix code can be viewed as the search for a binary tree T, together with a labeling of leaves of T, that minimizes the average number of bits per letter. Moreover, this average quantity has a natural interpretation in terms of the structure of T: the length of the encoding of a letter $x \in S$ is simply the length of the path from the root to the leaf labeled with x, a quantity which has a special name:

Definition. The depth of a leaf in a binary tree is the length of the path from the root to that leaf. The depth of a leaf v in tree T is denoted $\text{depth}_T(v)$.

Thus we seek the labeled tree that minimizes the weighted average of the depths of all leaves, where the average is weighted by the frequencies of the letters that label the leaves. We use ABL(T) to denote this quantity. If γ is the prefix code corresponding to T, then

$$\text{ABL}(T) = \sum_{x \in S} f_x \cdot \text{depth}_T(x) = \sum_{x \in S} f_x \cdot |\gamma(x)| = \text{ABL}(\gamma)$$

One thing to note immediately about the optimal binary tree T is that it is *full.* That is, each node that is not a leaf has exactly two children. To see why, suppose the optimal tree weren't full. Then we could take any node with only one child, remove it from the tree, and replace it by its lone child. This would create a new tree T' such that $\text{ABL}(T') < \text{ABL}(T)$, contradicting the optimality of T. We record this fact below:

Lemma 2. *The binary tree corresponding to an optimal prefix code is full.*

A First Attempt: The Top-Down Approach

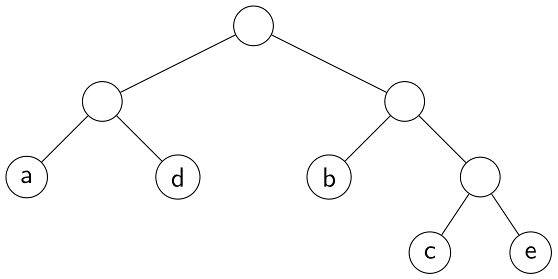
Intuitively, our goal is to produce a labeled binary tree in which the leaves are as close to the root as possible. This will give us a small average leaf depth.

A natural way to do this would be to try building a tree from the top down by "packing" the leaves as tightly as possible, perhaps using a divide-and-conquer method. So suppose we try to split the alphabet S into two sets, S_1 and S_2 such that the total frequency of the letters in each set is as close to 1/2 as possible. We then recursively construct prefix codes for S_1 and S_2 independently, then make these the two subtrees of the root.

In fact, this type of encoding scheme is called the *Shannon-Fano* code, named after two major figures in information theory, Claude Shannon and Robert Fano. Shannon-Fano codes work OK in practice, but unfortunately it doesn't lead to an optimal prefix code. For example, consider our five letter alphabet $S = \{a, b, c, d, e\}$ with frequencies

$$f_a = 0.32 \quad f_b = 0.25 \quad f_c = 0.20 \quad f_d = 0.18 \quad f_e = 0.05$$

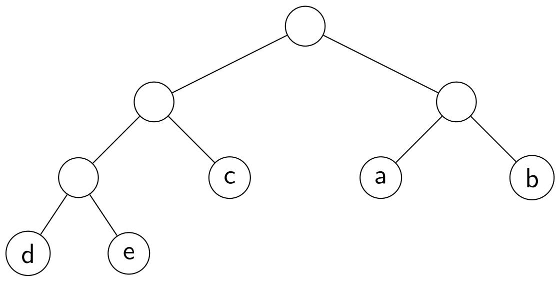
There is a unique way to split the alphabet into two sets of equal frequency: $\{a, d\}$ and $\{b, c, e\}$. Recursing on $\{a, d\}$, we can use a single bit to encode each. Recursing on $\{b, c, e\}$, we need to go one more level deeper in the recursion, and again, there is a unique way to divide the set into two subsets of equal frequency. The resulting code corresponds to:



Character	Encoding
a	00
b	10
c	110
d	01
e	111

Figure 15.2: A non-optimal prefix code for the given alphabet S.

However, it turns out this is NOT optimal. In fact, the optimal code is given by:



Character	Encoding
a	10
b	11
c	01
d	000
e	001

Figure 15.3: An optimal prefix code for the given alphabet S.

You can verify this is better than the Shannon-Fano tree by computing the value of ABL(T) for each of the above trees. A quick way to see that the tree in Figure 3 is better than the Shannon-Fano is to notice that the only characters whose depth differ between the trees are c and d. Since d has a lower frequency than c, it should have greater depth.

David Huffman, a graduate student who had taken a class by Fano, decided to take up the problem for himself. This would eventually lead to Huffman's algorithm and Huffman coding, which always produce optimal prefix-free codes.

Huffman's Algorithm

In searching for an efficient algorithm or solving a tough problem, it often helps to assume—as a thought experiment—that you know something about the optimal solution, and then to see how you would make use of this partial knowledge in finding the complete solution. Applying this technique to the current problem, we ask: What if we knew the binary tree T^* that corresponded to the optimal prefix code, but not the labeling of the leaves? To complete the solution, we would need to figure out which letter should label which leaf of T^*, and then we'd have our code.

It turns out, this is rather simple to do.

Proposition 1. *Suppose that u and v are leaves of T^*, such that depth(u) $<$ depth(v). Further, suppose that in a labeling of T^* corresponding to an optimal prefix code, leaf u is labeled with character $y \in S$ and leaf v is labeled with $z \in S$. Then $f_y \geq f_z$.*

Proof. We will use a common technique called an *exchange argument* to prove this. Seeking contradiction, suppose that $f_y < f_z$. Consider the code obtained by exchanging the labels at the nodes u and v. In the expression for the average number of bits per letter, $\text{ABL}(T^*) = \sum_{x \in S} f_x \cdot \text{depth}(x)$, the effect of the exchange is as follows: the multiplier on f_y increases from depth(u) to depth(v), and the multiplier on f_z decreases by the same amount. All other terms remain the same.

Thus the change to the overall sum is $(\text{depth}(v) - \text{depth}(u))(f_y - f_z)$. If $f_y < f_z$, this change is a negative number, contradicting the supposed optimality of the prefix code that we had before the exchange. □

For example, as state above, it is easy to see that the tree in Figure 2 is not optimal, since we can exchange d and c to achieve the more optimal code in Figure 3.

This proposition tells us how to label the tree T^* should someone give it to us: We go through the leaves in order of increasing depth, assigning letters in order of decreasing frequency. Note that, among labels assigned to leaves at the same depth, *it doesn't matter which label we assign to which leaf.* Since the depths are all the same, the corresponding multipliers in the expression $\text{ABL}(T^*) = \sum_{x \in S} f_x \cdot |\gamma(x)|$ are all the same too.

Now we have reduced the problem to finding the optimal tree T^*. How do we do this? It helps to think about the labelling process from the bottom-up, i.e. by considering the leaves at the greatest depths (which thus receive the labels with the lowest frequencies). Specifically, let v be a leaf in T^* whose depth is a large as possible. Leaf v has a parent u, and since T^* must be full, u has another child, w, which is a *sibling* of v. Note that w is also a leaf in T^*, otherwise, v would not be the deepest possible leaf.

Now, since v and w are siblings at the greatest depth, our level-by-level labelling process will reach them last. The leaves at this deepest level will receive the lowest frequency letters, and since we argued above that the order in which we assign these letters to the leaves within this level doesn't matter, there is an optimal labelling in which v and w get the two lowest-frequency letters over all:

Lemma 3. *There is an optimal prefix code, corresponding to tree T^*, in which the two lowest-frequency letters are assigned to leaves that are siblings in T^*.*

Designing the Algorithm

Suppose that y and z are the two lowest frequency letters in S (ties can be broken arbitrarily). Lemma 3 tells us that it is safe to assume that y and z are siblings in an optimal prefix tree. In fact, this directly suggests an algorithm: we replace y and z with a meta-letter w, whose frequency is $f_y + f_z$. That is, we link y and z together in the prefix tree, making them siblings with a common parent w. This step makes the alphabet one letter smaller, so we can recursively find a prefix code for the smaller alphabet. Once we reach an alphabet with one letter, we are done: this one letter is the root of our tree, and we can "unravel" or "open up" the process to obtain a prefix code for the original alphabet S.

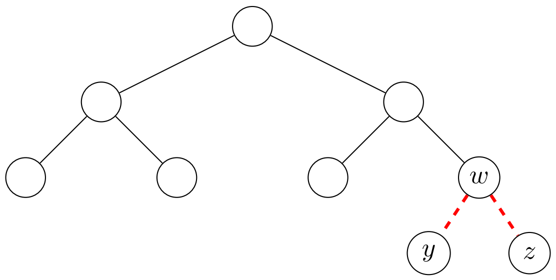


Figure 15.4: We take the two lowest frequency letters y and z, combine them to form a meta-letter w, and recurse.

Huffman's Algorithm

Input: A set of characters S, together with the frequencies of each character.

Output: An optimal, prefix-free code for S.

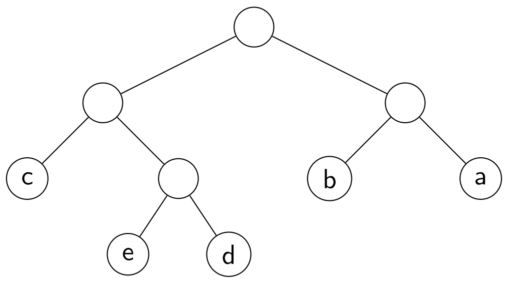
```
Huffman(S)
    If |S| = 2 then
        Encode one letter using 0 and the other letter using 1
    Else
        Let y,z be the two lowest-frequency letters

        Form a new alphabet S' by deleting y and z and
            replacing them with a new letter w of frequency $f_w$

        Recursively construct a prefix code for S' with tree T'

        Define a prefix code for S as follows:
            Start with T'
            Take the leaf labeled w and add y and z as its
                two children
```

Before jumping into the analysis of the algorithm, let's trace out how it would work on our running example:

$S = \{a, b, c, d, e\}$ with frequencies

$$f_a = 0.32 \quad f_b = 0.25 \quad f_c = 0.20 \quad f_d = 0.18 \quad f_e = 0.05$$

We'd first merge d and e into a single letter, denoted (ed), of frequency $0.18 + 0.05 = 0.23$. We now recurse on the alphabet $S' = \{a, b, c, (ed)\}$. The two lowest frequency letters are now c and (ed), so we merge these into the single letter $(c(ed))$ of frequency $0.20 + 0.23 = 0.43$. Next we merge a and b to get the alphabet $\{(ba), (c(ed))\}$. Lastly, we perform one final merge to get the single letter $((c(ed))(ba))$, and we are done. Here, the parenthesis structure is meant to represent the structure of the corresponding prefix tree, which looks like:



Note the contrast with the Shannon-Fano codes. Instead of a divide-and-conquer, top-down approach, Huffman's algorithm uses a bottom-up, "greedy" approach. Here, "greedy" just means that at each step, we make local, short-sighted decision (merge the two lowest frequency nodes without worrying about how they will fit into the overall structure) that leads to a globally optimal outcome (an optimal prefix code for the whole alphabet).

Correctness of Huffman's Algorithm

Theorem 1. *The Huffman code for a given alphabet S achieves the minimum average number of bits per letter of any prefix code.*

Proof. Since the algorithm has a recursive structure, it is natural to try to prove optimality by induction on the size of the alphabet (CLRS provides a non-inductive proof). Clearly it is optimal for one and two-letter alphabets (it uses only one bit per letter). So suppose inductively that it is optimal for all alphabets of size $k - 1$, where $k - 1 \geq 2$ and consider an input consisting of an alphabet of size k.

Now, let y and z be the lowest frequency letters in S (ties broken arbitrarily). The algorithm proceeds by merging y and z into a single meta-letter w with $f_w = f_y + f_z$, and forming a new alphabet $S' = (S \cup \{w\}) - \{y, z\}$. The algorithm then recurses on S'. By the induction hypothesis, this recursive call produces an optimal prefix code for S', represented by a labeled binary tree T'. It then extends this into a tree T for S by attaching the leaves labeled y and z as the children of the node in T' labelled w.

We have the following relationship:

Lemma 4. *With T, T', and w defined as above, we have:*

$$ABL(T') = ABL(T) - f_w$$

Proof. The depth of each letter in S other than y and z is the same in both T and T'. Also, the depths of y and z in T are each one greater than the depth of w in T'. Using this, plus the fact that $f_w = f_y + f_z$, we have

$$\begin{aligned}
\text{ABL}(T) &= \sum_{x\in S} f_x \cdot \text{depth}_T(x) \\
&= f_y \cdot \text{depth}_T(y) + f_z \cdot \text{depth}_T(z) + \sum_{x\neq y,z} f_x \cdot \text{depth}_T(x) \\
&= (f_y + f_z)(1 + \text{depth}_{T'}(w)) + \sum_{x\neq y,z} f_x \cdot \text{depth}_{T'}(x) \\
&= f_w + f_w \cdot \text{depth}_{T'}(w) + \sum_{x\neq y,z} f_x \cdot \text{depth}_{T'}(x) \\
&= f_w + \sum_{x\in S'} f_x \cdot \text{depth}_{T'}(x) \\
&= f_w + \text{ABL}(T')
\end{aligned}$$

□

Now we can prove optimality as follows: Seeking contradiction, suppose the tree T produced by Huffman's algorithm is not optimal. Then there exists some labeled binary tree Z such that $\text{ABL}(Z) < \text{ABL}(T)$. By **Lemma 3**, we can WLOG assume Z has y and z as siblings.

If we delete the leaves labeled y and z from Z, and label their former parent with w, we get a tree Z' that defines a prefix code for S'. In the same way that T is obtained from T', the tree Z is obtained from Z' by adding leaves from y and z below w. Thus **Lemma 4** applies to Z and Z' as well, so $\text{ABL}(Z') = \text{ABL}(Z) - f_w$.

But we have assumed that $\text{ABL}(Z) < \text{ABL}(T)$. Subtracting f_w from both sides of this inequality tells us that $\text{ABL}(Z') < \text{ABL}(T')$, contradicting the optimality of T' as a prefix code for S' (which was given by the induction hypothesis). □

Running Time

Let n be the number of characters in the alphabet S. The recursive calls of the algorithm define a sequence of $n-1$ iterations over smaller and smaller alphabets until reaching the base case. Identifying the lowest frequency letters can be done in a single scan of the alphabet in $O(n)$ time, and so summing over the $n-1$ iterations, the runtime is $T(n) \approx n + (n-1) + ... + 2 + 1 = \Theta(n^2)$.

However, note that the algorithm requires finding the minimum of a set of elements. Thus we should expect to use a data structure that makes it easy to find the minimum of a set of elements quickly. A priority queue works well here. In this case, the keys are the frequencies. In each iteration, we just extract the minimum twice (to get the two lowest frequency letters), and then insert a new letter whose key is the sum of these two minimum frequencies, building the tree as we go. Our priority queue then contains a representation of the alphabet needed for the next iteration.

Using a binary-heap implementation of a priority queue, we know that each insertion and extract-min operation take $O(\log n)$ time. Hence summing over all n iterations gives a total running time of $O(n \log n)$.

This leads to a cleaner formulation of Huffman's algorithm:

Huffman's Algorithm

Input: A set of characters S, together with the frequencies of each character.

Output: An optimal, prefix-free code for S.

```
Huffman(S)
    Let Q be a priority queue containing all elements in S
        with frequencies as keys

    for i = 1 to n do
        allocate a new node z
        x = ExtractMin(Q)
        y = ExtractMin(Q)
        z.left = x
        z.right = y
        $f_z = f_x + f_y$
        Insert(Q, z)

    return ExtractMin(Q) // the root of the tree
```

15.4 Extensions

While Huffman works well in many cases, it by no means is the solution for all data compression problems. For example, consider transmitting black-and-white images, where each image is a 1000-by-1000 array of black or white pixels. Further, suppose a typical image is almost entirely white: only about 1000 of the million pixels are black. If we wanted to compress this image using prefix codes, it wouldn't work so well: we'd have a text of length one million over the two letter alphabet {black, white}. As a result, the text is already encoded using one-bit per letter, so no significant compression is really possible.

Intuitively, though, such images should be highly compressible. One possible compression scheme is to represent each image by a set of (x, y) coordinates denoting which pixels are black.

Another drawback of prefix codes is that they cannot *adapt* to changes in text. For example, suppose we are trying to encode the output of a program that produces a long sequence of letters from the set $\{a, b, c, d\}$. Further suppose that for the first half of this sequence, the letters a and b occur equally frequently, while c and d do not occur at all. Huffman's algorithm would view the entire text as one where each letter is equally frequent (all letters occur $1/4$ of the time), and thus assign two bits per letter.

But what's really happening here is that the frequency remains stable for half the text, and then it changes radically. So one could get away with just one bit per letter plus a bit of extra overhead. Consider, for example, the following scheme:

- Begin with an encoding in which a is represented by 0 and b is represented by 1
- Halfway into the sequence, insert an instruction that says "Change the encoding to 0 represents c and 1 represents d" (this could be some kind of reserved special character).
- Use the new encoding for the rest of the sequence.

The point is that investing a small amount of space to describe a new encoding can pay off many times over if it reduces the average number of bits per letter over a long run of text that follows. Such approaches, which change the encoding in midstream, are called *adaptive* compression schemes, and for many kinds of data they lead to significant improvements over the static Huffman method.

Graph Traversals: BFS and DFS 16

16.1 Graphs and Graph Representations

A **graph** is a set of **vertices** and **edges** connecting those vertices. Formally, we define a graph G as $G = (V, E)$ where $E \subset V \times V$. For ease of analysis, the variables n and m typically stand for the number of vertices and edges, respectively. Graphs can come in two flavors, **directed** or **undirected**. If a graph is undirected, it must satisfy the property that $(i, j) \in E$ iff $(j, i) \in E$ (i.e., all edges are bidirectional). In undirected graphs, $m \leq \frac{n(n-1)}{2}$. In directed graphs, $m \leq n(n-1)$. Thus, $m = O(n^2)$ and $\log m = O(\log n)$. A **connected** graph is a graph in which for any two nodes u and v there exists a path from $u \rightsquigarrow v$. For an undirected connected graph $m \geq n - 1$. A **sparse** graph is a graph with few edges (for example, $\Theta(n)$ edges) while a **dense** graph is a graph with many edges (for example, $m = \Theta(n^2)$).

Graph Representations

A common issue is the topic of how to represent a graph's edges in memory. There are two standard methods for this task.

1. An **adjacency matrix** uses an arbitrary ordering of the vertices from 1 to $|V|$. The matrix consists of an $n \times n$ binary matrix such that the (i, j)-th element is 1 if (i, j) is an edge in the graph, 0 otherwise.
2. An **adjacency list** consists of an array A of $|V|$ lists, such that $A[u]$ contains a linked list of vertices v such that $(u, v) \in E$ (the neighbors of u). In the case of a directed graph, it's also helpful to distinguish between outgoing and ingoing edges by storing two different lists at $A[u]$: a list of v such that $(u, v) \in E$ (the out-neighbors of u) as well as a list of v such that $(v, u) \in E$ (the in-neighbors of u).

What are the tradeoffs between these two methods? To help our analysis, let $deg(v)$ denote the degree of v, or the number of vertices connected to v. In a directed graph, we can distinguish between out-degree and in-degree, which respectively count the number of outgoing and incoming edges.

- The adjacency matrix can check if (i, j) is an edge in G in constant time, whereas the adjacency list representation must iterate through up to $deg(i)$ list entries
- The adjacency matrix takes $\Theta(n^2)$ space, whereas the adjacency list takes $\Theta(m + n)$ space.
- The adjacency matrix takes $\Theta(n)$ operations to enumerate the neighbors of a vertex v since it must iterate across an entire row of the matrix. The adjacency list takes $deg(v)$ time.

What's a good rule of thumb for picking the implementation? One useful property is the sparsity of the graph's edges. If the graph is **sparse**, and the number of edges is considerably less than the max ($m \ll n^2$), then the adjacency list is a good idea. If the graph is **dense** and the number of edges is nearly n^2, then the matrix representation makes sense because it speeds up lookups without too much space overhead. Of course, some applications will have lots of space to spare, making the matrix feasible no matter the structure of the graphs. Other applications may prefer adjacency lists even for dense graphs. Choosing the appropriate structure is a balancing act of requirements and priorities.

Note: in this course, please assume the graphs are given to you as adjacency lists unless otherwise specified.

These notes were adapted from Kleinberg and Tardos' *Algorithm Design* and from CS 161 at Stanford University.

16.2 Connectivity

Suppose we are given a graph $G = (V, E)$ represented as an adjacency list and two particular nodes $s, t \in V$. A natural question to ask is: Is there a path from s to t in G? This is the problem of *s-t connectivity.* As a side note, s stands for "source" and t stands for "target." One can also think of the connectivity problem as a traversal problem: from s, which nodes are reachable? A traversal of the graph beginning at s would answer such a question.

We will introduce two natural algorithms for solving this problem: breadth-first search (BFS) and depth-first search (DFS). Later in the course, you will see several other uses for these algorithms beyond graph connectivity.

Before introducing specific algorithms, we can consider a more general, high-level description of a process to find the nodes reachable from a source s:

```
$R$ will consist of nodes to which $s$ has a path
$R$ = $\{s\}$
while there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
    Add $v$ to $R$
```

In fact in an undirected graph, this "algorithm" finds the vertices in the connected component containing s. The reason "algorithm" is in quotes is because the above process is under-specified: in the `while` loop, how do we decide which edge to consider next? The BFS and DFS algorithms below give two different ways to do this.

16.3 Breadth-First Search (BFS)

Perhaps the simplest algorithm for graph traversal is *breadth-first search* (BFS) in which we explore outward from s in all possible directions, visiting nodes one "layer" at a time. Thus, we start at s and visit all nodes that are joined by an edge to s—this is the first layer of the search. We then visit all additional nodes that are joined by an edge to any node in the first layer—this is the second layer. We continue this way until no new nodes are encountered.

We can define "layers" more formally:

- $L_0 = \{s\}$
- L_{k+1} is the set of all nodes that do not belong in $\bigcup_{i=0}^{k} L_i$ and that have an edge to some node in L_k.

One can also think about BFS not in terms of layers, but in terms of a tree T rooted at s. More specifically, for each node $v \neq s$, consider the moment when v is first discovered by the BFS algorithm. This happens when some node u in layer L_k is begin examined, and we find that it has an edge to the previously unseen node v. At this moment, we add the edge (u, v) to the tree T—u becomes the parent of v. We call the tree T produced this way a *breadth-first search tree.*

The algorithm is described below. We will use a queue to determine which nodes to visit next, and we will use an array `discovered` to keep track of which nodes have been visited. For simplicity, we assume the vertices are integers numbered from 1 to $|V|$. The first implementation keeps track of the BFS tree via the `parent` array, while the second implementation keeps track of both the BFS tree *and* the levels L_i. Depending on the scenario, one may be more useful than the other.

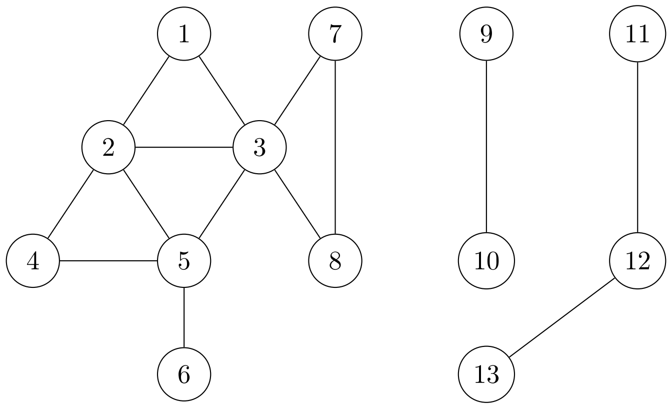
Breadth-First Search

Input: A graph $G = (V, E)$ implemented as an adjacency list and a source vertex s.

Output: A BFS traversal of G

```
BFS(G,s)
    for each v ∈ V do
        discovered[v] = FALSE
        parent[v] = NIL

    Let Q be an empty Queue
    Q.enqueue(s)
    discovered[s] = TRUE

    while Q is not empty do
        v = Q.dequeue()
        for each u ∈ Adj[v] do
            if discovered[u] = FALSE then
                discovered[u] = TRUE
                Q.enqueue(u)
                parent[u] = v
--------------------------------------------------------------------------


BFS(G,s)
    for each v ∈ V do
        discovered[v] = FALSE

    discovered[s] = TRUE
    L[0] = {s}
    i = 0
    while L[i] is not empty
        Let L[i+1] be a new list
        for each v ∈ L[i] do
            for each u ∈ Adj[v] do
                if discovered[u] = FALSE then
                    discovered[u] = TRUE
                    parent[u] = v
                    L[i+1].append(u)
        i = i + 1
```

Note: the order in which we visit each node in a particular level doesn't matter. There could be multiple BFS trees.



To see how the algorithm works, suppose we start with s as node 1 in the above graph. With $L_0 = \{1\}$, the first layer of the search would be $L_1 = \{2, 3\}$, the second layer would be $L_2 = \{4, 5, 7, 8\}$, and the third layer would be just $L_3 = \{6\}$. At this point, there are no further nodes that could be added. Note in particular that nodes 9 through 13 are never reached by the search.

For the BFS started at $s = 1$ on the tree in the figure above, the BFS tree is shown below. The solid edges are the edges in T, the dashed edges are edges in G that aren't in T. The first few steps of the execution that produces this tree can be described in words:

- ▶ Starting from 1, $L_1 = \{2, 3\}$.
- ▶ Layer L_2 is then grown by considering the nodes in L_1 in any order. If we examine node 2 first, then we discover nodes 4 and 5, so 2 becomes their parent. When we examine node 3, we discover 7 and 8 (5 has already been discovered), so 3 becomes their parent.
- ▶ We then consider the nodes in L_2 (WLOG we consider them in ascending order). The only new node discovered is node 6, which is discovered through node 5 and so becomes the child of node 5.
- ▶ No new nodes are discovered when node 6 is examined. The BFS traversal thus terminates.

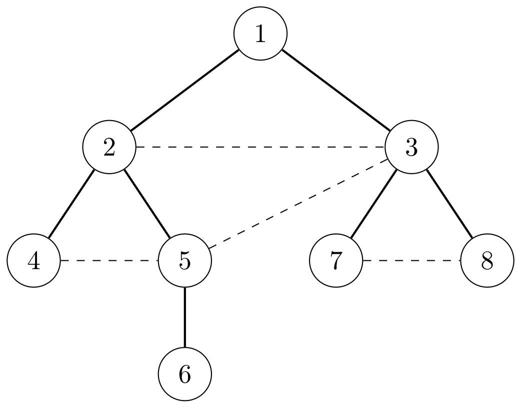


Figure 16.1: The BFS tree with $s = 1$.

BFS Properties

We now prove some properties related to BFS and BFS trees.

If we define the *distance* between two nodes as the minimum number of edges on a path joining them, we have the following property:

Proposition 1. *For each $k \geq 1$, the layer L_k produced by BFS consists of all nodes at distance exactly k from s. Moreover, there is path from s to t in G if and only if t appears in some layer (i.e. iff t is visited by the BFS traversal).*

Proposition 2. *Let G be a graph in which vertices x and y share an edge. Let T be a BFS tree of G. In T, let vertices x and y belong to layers L_i and L_j , respectively. Then i and j differ by at most 1.*

Proof. Seeking contradiction, suppose WLOG that $i < j - 1$, or equivalently $i + 1 < j$. Consider the point in the BFS algorithm when the edges incident to x were begin examined. Since x belongs to layer L_i, the only nodes discovered from x belong to layers L_{i+1} and earlier. Hence if y is a neighbor of x, then it should have been discovered by this point at the latest and hence should belong to L_{i+1} or earlier. Since $i + 1 < j$, we have a contradiction. □

Runtime of BFS

Proposition 3. *Running BFS on a graph $G = (V, E)$ given as an adjacency list takes $O(m + n)$ time, where $n = |V|$ and $m = |E|$. That is, BFS is a linear time algorithm.*

Proof. It is easy to see that the algorithm is $O(n^2)$: it visits each node once, and examines all its neighbors, performing $O(1)$ work for each. Since there are n nodes, each having $O(n)$ neighbors, the runtime is $O(n^2)$.

To get a tighter bound, note that for each node, we only examine its neighbors, so for each node $v \in V$ we do $O(\deg(v))$ work. Since

$$\sum_{v \in V} \deg(v) = 2m$$

by the handshake lemma, the runtime is $O(n + 2m) = O(n + m)$. □

16.4 Depth-First Search (DFS)

While BFS explores nodes level-by-level, depth-first search searches "deeper" in the graph whenever possible. In particular, DFS explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search "back tracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then DFS selects one of them as a new source and it repeats the search from that source. The algorithm repeats until it has discovered every vertex.

Note that unlike BFS—which creates a tree—DFS creates a *forest*, called the depth-first search forest. Moreover, the DFS algorithm will keep track of several things:

- At every point in the algorithm, a vertex will have one of three *colors*: white (undiscovered), grey (discovered and currently examining), or black (finished).
- Each vertex v will be *timestamped* with its discovery time $v.d$ and finish time $v.f$. These are related to the color of v: $v.d$ is the time when v is first colored grey, and $v.f$ is the time when v is colored black.

- At the end of the algorithm, each edge will be classified either as a tree edge, back edge, forward edge, or cross edge.

We will discuss each of these in depth. First, let us present the algorithm. DFS can be implemented either recursively or using a stack. We give a recursive version here (the implementation using a stack is very similar to the BFS implementation using a queue, except with "queue" replaced with stack, "enqueue" replaced with "push," and "dequeue" replaced with pop throughout). As before, the DFS forest is represented by a `parent` array. Start/finish times and colors are also maintained using arrays:

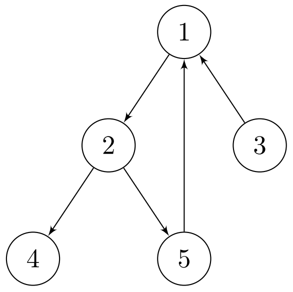
Depth-First Search

Input: A graph $G = (V, E)$ implemented as an adjacency list.

Output: A DFS traversal of G

```
DFS(G)
    for each v ∈ V do
        color[v] = WHITE

    time = 0

    for each v ∈ V do
        if color[v] = WHITE then
            DFS-VISIT(G, v)

--------------------------------------------------

DFS-VISIT(G,u)
    time = time + 1
    d[u] = time
    color[u] = GRAY
    for each v ∈ Adj[u] do
        if color[v] = WHITE then
            parent[v] = u
            DFS-VISIT(G,v)      // go deeper into graph
    color[u] = BLACK            // all u's neighbors explored
    time = time + 1
    f[u] = time
```

DFS works for both directed and undirected graphs. Consider the graph below:



A DFS traversal of this graph is given in Figure 2. In that traversal, we visit nodes in numerical order (this is arbitrary: DFS can visit the nodes in *any* order). Pay careful attention to the start/finish times of each node and how these relate to each node's color. Also, be sure to note which edges are included in the DFS forest (the red edges) and which aren't. Note that there are two trees in the resulting forest: one consists of the vertices $\{1, 2, 4, 5\}$ and the other consists of the lone vertex $\{3\}$.

Runtime of DFS

The procedure `DFS-VISIT` is called once per vertex. At each vertex v, we iterate through v's neighbors (potentially calling `DFS-VISIT` on the neighbor), and thus this iteration takes $\sum_{v \in V} \deg(v) = O(m)$ time, not including the calls to `DFS-VISIT`. The runtime of DFS is therefore $O(n + m)$.

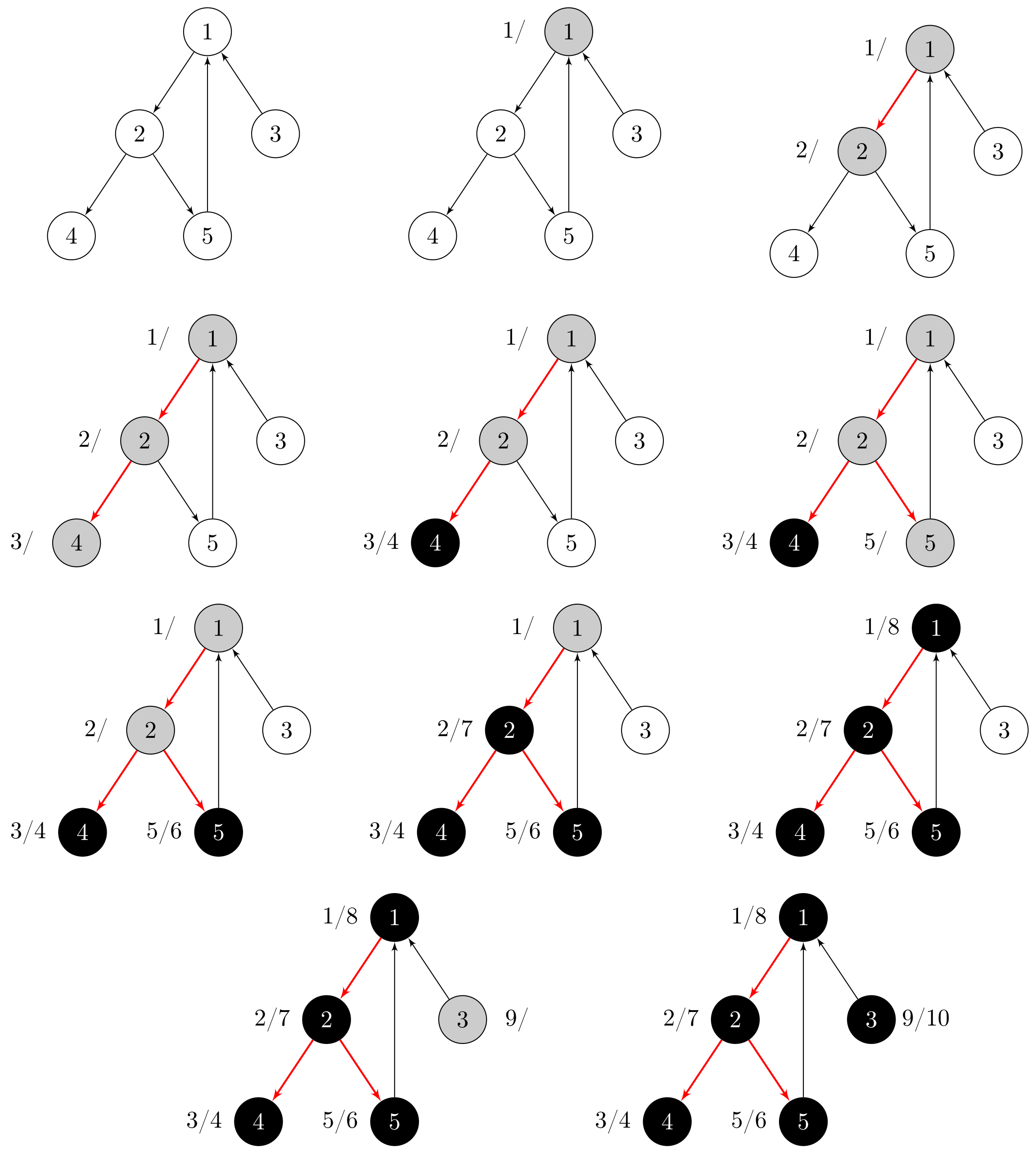


Figure 16.2: A DFS traversal. The start/finish times are given for each node, and the red edges are those that are included in the DFS forest.

DFS Properties and Extensions

Here we explore some of the reasons for keeping track of node colors and start/finish times. We will see that they reveal valuable information about the structure of a graph.

First note that each DFS traversal naturally gives rise to a DFS forest: u is a parent of v if and only if `DFS-VISIT(G,v)` was called during the search of u's adjacency list. Additionally, v is a descendant of u if and only if v is discovered during the time in which u is gray. We record that in the following lemma, as it will be important later on:

Lemma 1. *In the DFS forest, v is a descendant of u if and only if v is discovered during the time in which u is gray.*

Another important property of DFS is that discovery and finish times have *parenthesis structure.* If we represent the discovery of vertex u with a left parenthesis "(u" and represent its finishing by a right parenthesis "u)" then the history of discoveries and finishings makes a well-formed expression in the sense that the parentheses are properly nested. For example, in the DFS traversal from figure 2, the parenthesis structure looks like:

$$\begin{array}{cccccccccc} (1 & (2 & (4 & 4) & (5 & 5) & 2) & 1) & (3 & 3) \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array}$$

where the bottom line represents time.

The following theorem characterizes this parenthesis structure and gives its relationship to the structure of the DFS forest:

Theorem 1. ***(Parenthesis Theorem)*** *In any DFS of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:*

- *The intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the DFS forest*
- *The interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a DFS tree*
- *The interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a DFS tree*

Proof. First, consider the case in which $u.d < v.d$ (i.e. u was discovered first). There are two subcases, according to whether $v.d < u.f$ or not. If $v.d < u.f$, then v was discovered while u was still gray, which implies that v is a descendant of u. Moreover, since v was discovered more recently than u, all of its outgoing edges are explored, and v is finished, before the search returns to and finishes u. In this case, therefore, the interval $[v.d, v.f]$ is entirely contained within the interval $[u.d, u.f]$. In the other subcase, $u.f < v.d$, and since $u.d < u.f$ always, we have

$$u.d < u.f < v.d < v.f$$

Thus the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint. Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.

The case in which $v.d < u.d$ is symmetric. □

Corollary 1. ***(Nesting of Descendants' Intervals)*** *Vertex v is a proper descendant of vertex u in the DFS forest for a (directed or undirected) graph G if and only if $u.d < v.d < v.f < u.f$.*

The next theorem is extremely important, and vies another characterization of when one vertex is a descendant of another in the DFS forest.

Theorem 2. ***(White Path Theorem (WPT))*** *In a DFS forest of a graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $u.d$ when the search discovers u, there is a path from u to v in G consisting entirely of white vertices.*

Proof.
(⇒): If $v = u$, then the path from u to v contains just vertex u, which is still white when we set the value of $u.d$. Now suppose that v is a proper descendant of u in the DFS forest. By the previous corollary, $u.d < v.d$ and so v is white at time $u.d$. Since v was an arbitrary descendant of u, this implies all vertices on the unique simple path from u to v in the DFS forest are white at time $u.d$.

(⇐): Seeking contradiction, suppose there is a path of white vertices from u to v at time $u.d$, but v does not become a descendant of u in the DFS tree containing u. WLOG assume that every vertex other than v along the path becomes a descendant of u (otherwise, let v be the closest vertex to u along the path that doesn't become a descendant of u). Let w be the predecessor of v in the path, so that w is a descendant of u (it may be the case that $w = u$). Again by the previous corollary, $w.f < u.f$. Because v must be discovered after u is discovered, but before w is finished, we have

$$u.d < v.d < w.f \leq u.f$$

The parenthesis theorem implies that the interval $[v.d, v.f]$ is entirely contained within the interval $[u.d, u.f]$. By the previous corollary, v must after all be a descendant of u, a contradiction. □

Classifying Edges

Another interesting property of DFS is that it can be used to classify the edges of the input graph $G = (V, E)$. The type of each edge can provide important information about a graph. For example, we will prove later that a directed graph is acyclic if and only if a DFS search on the graph yields no back edges.

We define four types of edges below:

Definition. Let $G = (V, E)$ be any graph and let G_{DFS} be the DFS forest produced by a DFS on G.

1. Tree Edges are edges in the DFS forest G_{DFS}. (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. Back Edges are those edges (u, v) connected a vertex u to an ancestor v in a DFS tree in G_{DFS}.
3. Forward Edges are those non-tree edges (u, v) connected a vertex u to a descendant v in a DFS tree in G_{DFS}.
4. Cross Edges are all other edges. They can go between vertices in the same DFS tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different DFS tree in G_{DFS}

For example, in the DFS traversal from Figure 2, edge $(1, 2)$ is a tree edge, edge $(5, 1)$ is a back-edge, edge $(3, 1)$ is a cross edge, and there are no forward edges (if we added the edge $(1, 4)$ to the graph and ran DFS in ascending order of vertices as before, then the edge $(1, 4)$ would be a forward edge in the DFS forest).

We can classify edge (u, v) based on the color of v when the edge is first explored:

- If v is white, (u, v) is a tree edge
- If v is gray, (u, v) is a back edge
- If v is black, (u, v) is either a forward or a cross edge

An undirected graph may entail some ambiguity about how to classify edges, since (u, v) and (v, u) are really the same edge. In such a case, we classify the edge as the *first* type in the classification list that applies. Equivalently, we classify the edge according to whichever of (u, v) or (v, u) the DFS encounters first.

Theorem 3. *In a DFS of an* ***undirected*** *graph G, every edge of G is either a tree edge or a back edge.*

Proof. Let (u, v) be an arbitrary edge of G, and suppose WLOG that $u.d < v.d$. Then the search must discover and finish v before it finishes u (while u is gray), since v is on u's adjacency list. If the first time that the search explores edge (u, v), it is in the direction from u to v, then v is undiscovered (white) until that time, for otherwise the search would have explored this edge already in the direction from v to u. Thus (u, v) becomes a tree edge. If the search explores (u, v) first in the direction from v to u, then (u, v) is a back edge, since u is still gray at the time the edge is first explored. □

Application of BFS: Bipartiteness 17

17.1 Definitions and Properties

Definition. A graph $G = (V, E)$ is bipartite if the set V can be partitioned into two sets X and Y such that every edge in E has one end in X and the other in Y.

Another way to think of bipartiteness is in terms of colors: a graph is bipartite if and only if it is two colorable. That is, it is possible to color its nodes using two colors (say red and blue) such that each edge has exactly one red end and one blue end.

We'd like to characterize bipartite graphs. In order to do so, let's consider some examples of bipartite graphs. Clearly, a triangle graph is not bipartite. More generally, any cycle of odd length cannot be bipartite. If $C = (v_1\ v_2\ ...\ v_{2k}\ v_{2k+1})$ is an odd cycle which we try to two-color, note that the even-indexed nodes and odd-indexed nodes must have different colors. Since there is an edge between v_1 and v_{2k+1}, both of which are the same color, it is impossible to two-color an odd cycle. Moreover, if a graph G simply contains an odd cycle, it cannot be two-colorable. Hence:

Proposition 1. *If a graph G is bipartite, then it has no odd cycles*

It turns out that the converse is true as well. We now discuss an algorithm that uses BFS to determine if a graph is bipartite. In analyzing the algorithm, we will prove the converse to the above proposition.

17.2 Algorithm

We want to determine if a graph G is bipartite. We may assume the graph G is connected, since otherwise we can run the algorithm on each connected component separately. The algorithm is very straightforward: essentially we try to two color the graph: if we succeed, the graph is two colorable, and if we fail, we argue that there cannot be any two-coloring. The algorithm proceeds by picking an arbitrary vertex $s \in V$ and coloring it red. There is no loss in doing this, since s must receive some color. Now, in order for G to be bipartite, all the neighbors of s must be colored blue, so we do this. It then follows that all the neighbors of *these* nodes must be red, their neighbors must be blue, and so on until the whole graph is colored.

Note that this description is essentially identical to BFS: we move outward from s, coloring nodes in "layers." In fact, an equivalent description of the algorithm could be: perform BFS. Color all nodes in $L_{2k},\ k \geq 0$ red and color all nodes in $L_{2k+1},\ k \geq 0$ blue.

It is obvious that if this algorithm results in a valid two coloring, the graph is bipartite. What we need to show, however, is that if the process fails (for example, if at some point we try to color a red node blue), that there is in fact *no* valid two-coloring of the graph.

17.3 Analysis

These notes were adapted from Kleinberg and Tardos' *Algorithm Design*

Proposition 2. *Let G be a connected graph, and let $L_0, L_1, \ldots$ be the layers produced by BFS starting at a node s. Then exactly one of the following two things must hold:*

1. *There is no edge of G joining two nodes of the same layer. In this case, G is a bipartite graph in which the nodes in even-numbered layers can be colored red and the nodes in odd-numbered layers can be colored blue.*
2. *There is an edge of G joining two nodes of the same layer. In this case, G contains an odd cycle, and so it cannot be bipartite.*

Proof. Let us first consider case 1. We know from our analysis of BFS that every edge in G connects nodes that are at most one layer apart. Case 1 assumes that every edge joins two nodes that are *exactly* one layer apart. Since the coloring procedure gives nodes in adjacent layers different colors, every edge connects nodes with opposite colors. Thus this coloring establishes G is bipartite.

Now consider case 2, and let $e = (u, v)$ be an edge joining two nodes u and v on the same layer, say j. Let w be the lowest common ancestor of u and v in the BFS tree, i.e. w is the ancestor of u and v whose layer number—say i—is as large as possible. Clearly, $i < j$. Consider the cycle C defined by following the $w - u$ path in the BFS tree, then the edge $e = (u, v)$, then the $v - w$ path in the BFS tree. The length of this cycle is $(j - i) + 1 + (j - i) = 2(j - i) + 1$, which is an odd number. Hence G contains an odd cycle and cannot be bipartite. □

Note this shows the converse to proposition 1: if G is not bipartite, then G must contain an odd cycle. We record the full statement below:

Theorem 1. *A graph G is bipartite if and only if it contains no odd cycles.*

Note: The above algorithm for determining bipartiteness runs in $O(m+n)$ time, since it is just a modified version of BFS. Essentially, this means that we can determine if a graph is two-colorable in the same amount of time that it takes just to read the graph as input. On the other hand, determining if a graph is k-colorable for $k > 2$ is a much harder problem. In fact, there is no known polynomial time solution to this problem. If you can come up with one, you'll get a \$1 million prize!

DAGs and Topological Sorting 18

18.1 DAGs

If an undirected graph is acyclic, then it is a forest: a collection of trees. What can be said of an acyclic, directed graph?

Definition. A DAG is a **D**irected **A**cyclic **G**raph

DAGs show up in many practical applications. One such example is a dependency network. In this case, DAGs are used to represent *precedence relations* or *dependencies*. Suppose we are given a list of tasks, which we represent as a graph. If task a cannot begin until task b finishes, we can draw an edge from b to a representing the fact that b must come before a in any scheduling of the tasks. Doing this for all the precedence relations gives a graph. In order for the graph to have meaning, it cannot contain any cycles, lest there be deadlock. Hence such a graph is a DAG.

18.2 Topological Sorting

Given a dependency graph, a natural thing to want to do would be so order the tasks in a way that all dependencies are respected, i.e. if task a depends on task b, then b comes before a in the order. In terms of the dependency graphs, this corresponds to an ordering of the vertices in such a way that all edges point from left to right.

Definition. A topological ordering of a graph G is an ordering of its nodes $v_1, v_2, ..., v_n$ such that for every edge (v_i, v_j), $i < j$. In other words, the edges point "forward" in the ordering.

We have the following result:

Proposition 1. *G is a DAG if and only if it has a topological ordering.*

Proof. $(\Leftarrow)$: Seeking contradiction, suppose G has a topological ordering $v_1, v_2, ..., v_n$ and also has a cycle C. Let v_i be the lowest-indexed node on the cycle and let v_j be the node on C just before v_i. Obviously (v_j, v_i) is an edge in G. By by the choice of i, we have $j > i$, which contradicts the assumption that $v_1, v_2, ..., v_n$ is a topological ordering.

In order to prove the other direction, we will provide an algorithm below that, given a DAG, produces a topological sorting. □

Before we describe the algorithm, note how useful the above theorem is in visualizing a DAG. Often, it is much easier to understand the structure of a DAG when looking at it from the perspective of its topological ordering. See Figure 1 for an example.

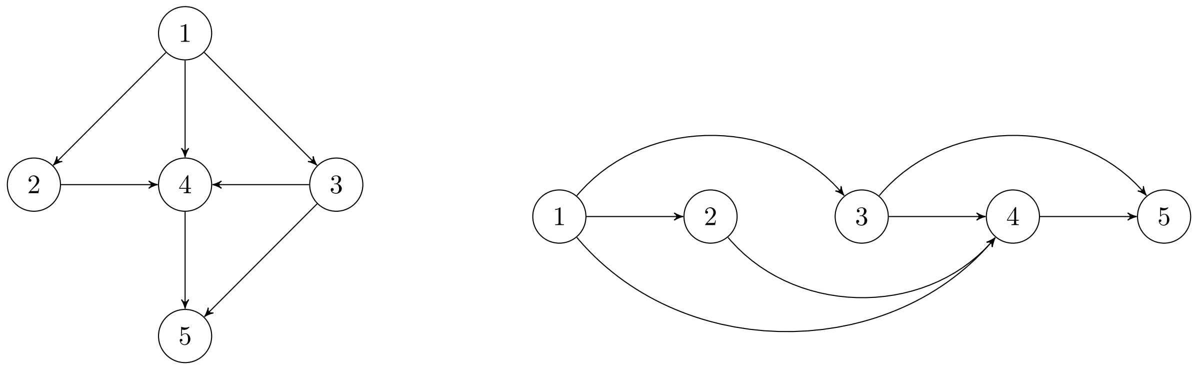


Figure 18.1: Two representations of the same graph. Note how the topologically ordered representation may be easier to reason about in some cases.

18.3 Kahn's Algorithm

The first step in finding a topological ordering is to ask: Which node comes first? Such a node must have no incoming edges. Is it guaranteed that such a node always exists? The answer is yes:

Proposition 2. *In every DAG, there is a node with no incoming edges.*

Proof. We will show that a graph in which every node has an incoming edge must contain a cycle. Let G be such a graph. Pick any node v in G. Now we just follow the edges backward: since v has an incoming edge, say (v_1, v), walk backward to v_1. Since v_1 has an incoming edge, say (v_2, v_1), walk backward to v_2. Continue this process. After $n+1$ steps, we must have visited some node twice. If we let C denote the sequence of nodes encountered between successive visits to this node, then clearly C contains a cycle. □

In fact, this proposition is all we need to prove the converse to Theorem 1, namely that every DAG has a topological ordering. We can prove it by induction on the number of nodes as follows:

Proof. (Continued from above)

$(\Rightarrow)$:

Base Case: A DAG with one node has a topological ordering consisting of just that node. The claim is also clear for DAGs with two nodes u and v: such a DAG contains at most one edge (u, v), so a topological ordering is just u followed by v.

Induction Hypothesis: Let $k \geq 2$ and assume all DAGs with k nodes have a topological ordering.

Induction Step: Let G be a DAG on $k+1$ nodes. Let v be a node with no incoming edges (which exists by the previous proposition). Place v first in the topological ordering. Consider the graph G' formed by removing v and all its incident edges from G. G' is a DAG (deleting a vertex cannot create a cycle) with k nodes, so by the IH, there exists a topological ordering of the vertices in G'. Appending v to the front of this ordering gives a topological ordering for G. □

This proof gives rise to a recursive procedure for computing a topological ordering of a graph G:

- Find a node v with no incoming edges.
- Put v next in the topological ordering.

- Remove v and all edges incident on v from the graph.
- Repeat.

The running time of this algorithm can be computed as follows: identifying a node with in-degree 0 can be done in $O(n)$ time by scanning through the adjacency list. Since this algorithm runs for n iterations, the total runtime is $O(n^2)$.

It turns out, using the right data structure, we can get this down to $O(m + n)$. Instead of scanning through the entire adjacency list to find a node with in-degree 0 at each iteration, we can maintain a list of such nodes, adding to it as necessary. That way, finding a node with no incoming edges is $O(1)$, not $O(n)$. To do this, we maintain the indegree of each node. Each time we remove a node from the graph, we decrement the indegree of all of its outneighbors. If the indegree of any of these nodes becomes 0, then we add it to our list. This leads to Kahn's algorithm:

Kahn's Algorithm for Topological Ordering

Input: A DAG $G = (V, E)$, given as an adjacency list

Output: A topological ordering of the nodes in G.

```
TopoSort(G):
    Let L be a new queue        // nodes of in-degree 0
    Let OUT be a new list       // topological ordering

    // compute in-degree of each node
    for each v ∈ V
        for each u ∈ Adj[v]
            in[u] = in[u] + 1

    // initially populate L
    for each v ∈ V
        if in[v] = 0 then
            L.enqueue(v)

    while L is not empty do
        v = L.dequeue()
        OUT.append(v)
        for each u ∈ Adj[v]
            in[u] = in[u] - 1
            if in[u] = 0 then
                L.enqueue(u)

    return OUT
```

The runtime of Kahn's algorithm can be computed as follows: scanning through the adjacency list to compute the in-degree of every node is $O(m + n)$, as is initially populating the list `L`. The `while` loop examines each vertex exactly once, then scans through it's neighbors, performing $O(1)$ work for each. The runtime is thus $O\left(n + \sum_{v \in V} \deg(v)\right) = O(m + n)$. Hence the entire algorithm is $O(m + n)$.

When the graph G is sparse, $O(m + n)$ is a significant improvement over the $O(n^2)$ algorithm from above.

Note: Topologically sorting a DAG only takes $O(m+n)$ time. This means that it if you are given a problem where the input graph is a DAG, most of the time, it is useful to topologically sort it as a first step. Since most graph algorithms take $\Omega(m+n)$ time anyway, this step is essentially "free." If nothing else, topologically sorting the graph will probably make the problem easier to reason about.

18.4 Tarjan's Algorithm

We will present another algorithm that topologically sorts a DAG, however, we will not analyze its correctness. You should try to do this on your own as an exercise to see how well you understand the material. Moreover, if you understand why this algorithm works, it will help a lot in understanding Kosaraju's algorithm later on in the course.

The algorithm is called Tarjan's algorithm. At first glance, it seems quite magical that it actually outputs a topological ordering. Nevertheless, we are still able to prove that it does in fact work.

Tarjan's Algorithm for Topological Ordering

Input: A DAG $G=(V,E)$, given as an adjacency list

Output: A topological ordering of the nodes in G.

```
TopoSort(G)
    Perform a DFS on G

    Return the nodes in decreasing order of finish time
```

Proof. In order to prove that Tarjan's Algorithm for topological ordering gives a valid topological ordering for a DAG, we simply need to show that given a DAG, $G=(V,E)$, that $\forall(u,v)\in E$, in any DFS traversal of G, $u.f > v.f$.

Case 1: $u.d < v.d$. At time $u.d$, there exists a white path from u to v, namely the edge (u,v), and thus v is a descendant of u by the White Path Theorem. By Parenthesis Theorem, we know that this means that $u.d < v.d < v.f < u.f$, and thus we have shown that $u.f > v.f$ in case 1.

Case 2: $u.d > v.d$. Since G is a DAG, there is no path from v to u, and thus, $v.f < u.d$. For any vertex x, $x.d < x.f$, meaning that $v.d < v.f < u.d < u.f$, showing that $u.f > v.f$ in case 2 as well.

Strongly Connected Components 19

19.1 Introduction and Definitions

We now consider a classic application of depth-first search: decomposing a directed graph into its strongly connected components. This section shows how to do so using two depth-first searches. Many algorithms that work with directed graphs begin with such a decomposition. After decomposing the graph into strongly connected components, such algorithms run separately on each one and then combine the solutions according to the structure of connections among components.

Definition. A strongly connected component of a directed graph $G = (V, E)$ is a maximal* set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is, vertices u and v are reachable from each other.

*By "maximal" we mean that no proper superset of the vertices forms a strongly connected component. However, note that a single graph may have multiple strongly connected components of different sizes.

Note: connected components are defined for undirected graphs, and strongly connected components are defined for directed graphs.

If we wanted to find the connected components in an undirected graph, we could simply run depth first search (or breadth first search). Each depth first search tree consists of the vertices that are reachable from a given start node, and those nodes are all reachable from each other. However, finding the strongly connected components in a directed graph is not as easy, because u can be reachable from v without v being reachable from u.

Definition. Suppose we decompose a graph into its strongly connected components, and build a new graph, where each strongly connected component is a "giant vertex". We call this graph G^{SCC} or the component graph. Then this new graph forms a directed acyclic graph, as in the figure below.

Formally, define $G^{SCC} = (V^{SCC}, E^{SCC})$ with respect to graph G as follows:

- V^{SCC}: Contains a vertex v_i for each SCC C_i in G.
- E^{SCC}: Contains edge $(v_i, v_j) \in E^{SCC}$ if G contains directed edge (x, y) where $x \in C_i$ and $y \in C_j$.

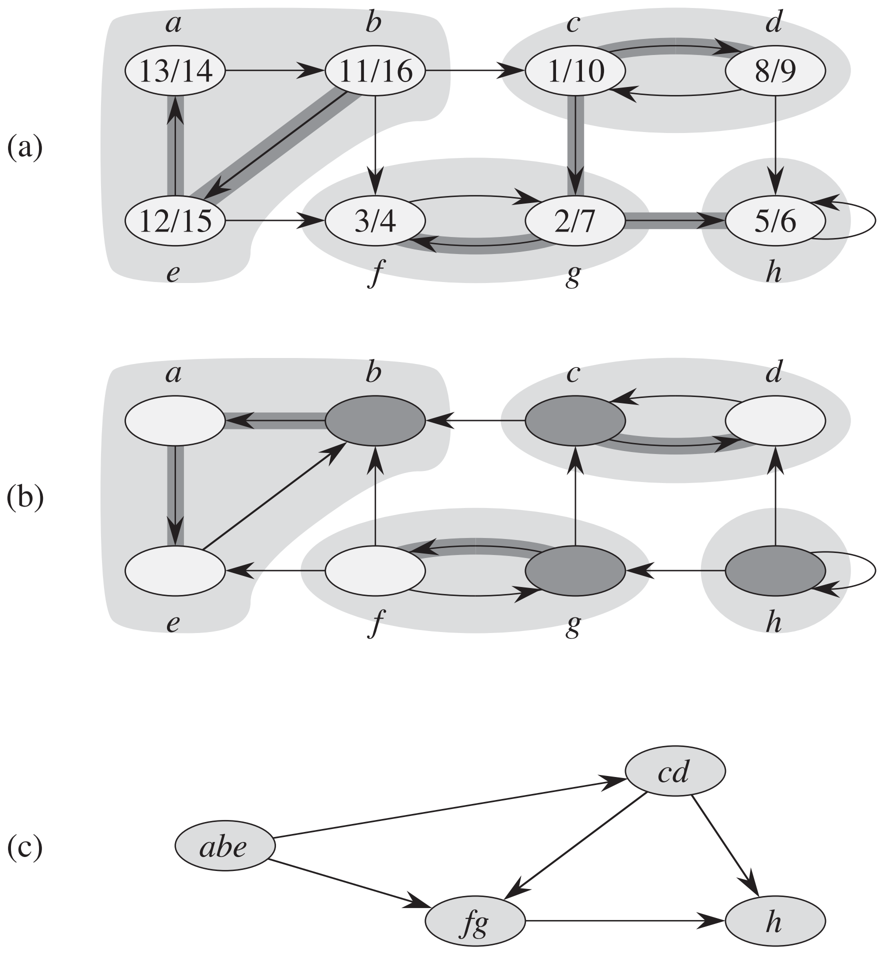


Figure 19.1: **(a)** A directed graph G. Each shaded region is a strongly connected component of G. Each vertex is labeled with its discovery and finishing times in a depth-first search, and tree edges are shaded. **(b)** The graph G^T, the transpose of G, with the depth-first forest computed in line 3 of Kosaraju shown and tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices b, c, g, and h, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of G^T. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.

19.2 Kosaraju's Algorithm

Note: we will first dive into the algorithm, and then will explain the idea behind it and why it works. This algorithm is difficult to understand, so please ask if you have questions!

Our algorithm for finding strongly connected components of a graph $G = (V, E)$ uses the transpose of G: $G^T = (V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$. That is, E^T consists the edges of G with their directions reversed. Given an adjacency-list representation of G, the time to create G^T is $O(V + E)$.

It is interesting to observe that G and G^T have exactly the same strongly connected components: u and v are reachable from each other in G if and only if they are reachable from each other in G^T.

The following linear-time (i.e., $\Theta(V + E)$) algorithm[*] computes the strongly connected components of a directed graph $G = (V, E)$ using two depth-first searches, one on G and one on G^T.

```
KOSARAJU(G):
  call DFS($G$) to compute finishing times $u.f$ for each vertex $u$
  compute $G^T$
  call DFS($G^T$), but in the main loop of DFS, consider the vertices
      in order of decreasing $u.f$ (as computed in line 1)
  output the vertices of each tree in the depth-first forest formed in line 3
      as a separate strongly connected component
```

Proof of Correctness

The idea behind this algorithm comes from a key property of the component graph: G^{SCC} is a DAG.

Lemma 1. *Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$ and $u', v' \in C'$, and suppose that G contains a path from $u \rightsquigarrow u'$. Then G cannot also contain a path $v' \rightsquigarrow v$.*

Proof. If G contains a path $v' \rightsquigarrow v$, then it contains paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$. Thus, u and v' are reachable from each other, thereby contradicting the assumption that C and C' are distinct strongly connected components. □

We will see that by considering vertices in the second depth-first search in decreasing order of the finishing times that were computed in the first depth-first search, we are, in essence, visiting the vertices of the component graph (each of which corresponds to a strongly connected component of G) in topologically sorted order.

Because the KOSARAJU procedure performs two depth-first searches, there is the potential for ambiguity when we discuss $u.d$ or $u.f$. In this section, these values always refer to the discovery and finishing times as computed by the first call of DFS, in line 1.

We extend the notation for discovery and finishing times to sets of vertices. If C is a component, we let $d(C) = \min_{u \in U}(u.d)$ be the earliest discovery time out of any vertex in C and $f(C) = \max_{u \in U}(u.f)$ be the latest finishing time out of any vertex in C.

[*] In CLRS and some other places, you will see this algorithm called the STRONGLY-CONNECTED-COMPONENTS algorithm. It was invented by Kosaraju, who in fact received his PhD at Penn!

The following lemma and its corollary give a key property relating strongly connected components and finishing times in the first depth-first search.

Lemma 2. *Let C and C' be distinct strongly connected components in a directed graph $G = (V, E)$. Suppose there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then, $f(C) > f(C')$.*

This lemma implies that if vertices are visited in reverse order of finishing time, then the components will be visited in topologically sorted order. That is, if there is an edge from component 1 to component 2, then component 1 will be visited before component 2.

Proof. Consider two cases:

1. $d(C) < d(C')$ (i.e., component C was discovered first). Let x be the first vertex discovered in C. Then at time $x.d$, all vertices in C and C' are white, so there is a path from x to each vertex in C consisting only of white vertices. There is also a path from x to each vertex in C' consisting of only white vertices (because you can follow the edge (u, v)). By the white path theorem, all vertices in C and C' become descendants of x in the depth first search tree.

   Now x has the latest finishing time out of any of its descendants, due to the parenthesis theorem. However, you can see this if you look at the structure of the recursive calls – we call DFS-Visit on x, and then recurse on the nodes that will become its descendants in the depth-first-search tree. So the inner recursion has to finish before the outer recursion, which means x has the latest finishing time out of all the nodes in C and C'. Therefore, $x.f = f(C) > f(C')$.

2. $d(C) > d(C')$ (i.e., component C' was discovered first). Let y be the first vertex discovered in C'. At time $y.d$, all vertices in C are white, and there is a path from y to each vertex in C' consisting only of white vertices. By the white path theorem, all vertices in C' are descendants of y in the depth-first-search tree, so $y.f = f(C')$.

   However, there is no path from C' to C (because we already have an edge from C to C', and if there was a path in the other direction, then C and C' would just be one component). Now all of the vertices in C are white at time $y.d$ (because $d(C) > d(C')$). And since no vertex in C is reachable from y, all the vertices in C must still be white at time $y.f$. Therefore, the finishing time of any vertex in C is greater than $y.f$, and $f(C) > f(C')$.

□

The following corollary tells us that each edge in G^T that goes between different strongly connected components goes from a component with an earlier finishing time (in the first depth-first search) to a component with a later finishing time.

Corollary 1. *Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then, $f(C) < f(C')$.*

Proof. Since $(u, v) \in E^T$, we have $(v, u) \in E$. Because the strongly connected components of G and G^T are the same, the previous lemma implies that $f(C) < f(C')$. □

This corollary provides the key to understanding why the strongly connected components algorithm works. Let us examine what happens when we perform the second depth-first search, which is on G^T.

We start with the strongly connected component C whose finishing time $f(C)$ is maximum. The search starts from some vertex $x \in C$, and it visits all vertices in C. By the corollary above, G^T contains no edges from C to any other strongly connected component, and so the search from x will not visit vertices in any other component. Thus, the tree rooted at x contains exactly the vertices of C. Having completed visiting all vertices in C, the search in line 3 selects as a root a vertex from some other strongly connected component C' whose finishing time $f(C')$ is maximum over all components other than C. Again, the search will visit all vertices in C', but by the corollary above, the only edges in G^T from C' to any other component must be to C, which we have already visited. In general, when the depth-first search of G^T in line 3 visits any strongly connected component, any edges out of that component must be to components that the search already visited. Each depth-first tree, therefore, will be exactly one strongly connected component.

The following theorem formalizes this argument.

Lemma 3. Kosaraju*'s algorithm correctly computes the strongly connected components of the directed graph G provided as its input.*

Proof. We argue by induction on the number of depth-first trees found in the depth-first search of G^T in line 3 that the vertices of each tree form a strongly connected component.

Induction Hypothesis: the first k trees produced in line 3 are strongly connected components.

Base Case: $k = 0$ is trivially true

Inductive Step: assume that each of the first k depth-first trees produced in line 3 is a strongly connected component, and we consider the $(k+1)$st tree produced. Let the root of this tree be vertex u, and let u be in strongly connected component C. Because of how we choose roots in the depth-first search in line 3, $u.f = f(C) > f(C')$ for any strongly connected component C' other than C that has yet to be visited.

By the inductive hypothesis, at the time that the search visits u, all other vertices of C are white. By the white-path theorem, therefore, all other vertices of C are descendants of u in its depth-first tree. Moreover, by the inductive hypothesis and by the corollary above, any edges in G^T that leave C must be to strongly connected components that have already been visited.

Thus, no vertex in any strongly connected component other than C will be a descendant of u during the depth-first search of G^T. Thus, the vertices of the depth-first tree in G^T that is rooted at u form exactly one strongly connected component, which completes the inductive step and the proof. □

Here is another way to look at how the second depth-first search operates. Consider the component graph $(G^T)^{SCC}$ of G^T. If we map each strongly connected component visited in the second depth-first search to a vertex of $(G^T)^{SCC}$, the second depth-first search visits vertices of $(G^T)^{SCC}$ in the reverse of a topologically sorted order. If we reverse the edges of $(G^T)^{SCC}$, we get the graph $((G^T)^{SCC})^T$. Because $((G^T)^{SCC})^T = G^{SCC}$, the second depth-first search visits the vertices of G^{SCC} in topologically sorted order.

Shortest Path 20

20.1 The Shortest Path Problem

As we've seen, graphs are often used to model networks in which one travels from one point to another. As a result, a basic algorithmic problem is to determine the shortest path between nodes in a graph.

The concrete setup of the shortest paths problem is as follows. Assume we're given a directed* graph $G = (V, E)$ with arbitrary non-negative weights on edges. The shortest path in G from source node s to destination node t is the directed path that minimizes its sum of edge weights. Let $d(s, t)$ denote this sum, also called the distance between s and t.

While BFS was able to solve this problem for unweighted graphs, we will need a more robust algorithm to solve the single-source shortest path problem on graphs with non-negative edge weights.

Here, we will focus on the single-source shortest-path (SSSP) problem: given a graph $G = (V, E)$ we want to find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$. You will learn other variants in CIS 320.

20.2 Dijkstra's Algorithm

In 1959, Edsger Dijkstra proposed a very simple greedy algorithm to solve the single-source shortest-paths problem for the case in which all edge weights are non-negative. In this section, therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

The key idea that Dijkstra will maintain as an invariant is that $\forall t \in V$, the algorithm computes an estimate $dist[t]$ of the distance of t from the source such that:

- At any point in time, $dist[t] \geq d(s, t)$, and
- when t is finished, $dist[t] = d(s, t)$

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and *relaxes* all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their *dist* values. Finding the shortest path from $s \rightsquigarrow t$ is made easy by using the *parent* pointers computed by the algorithm, and traversing them in reverse from $parent[t]$ to s.

Dijkstra's Algorithm

Input: A graph $G = (V, E)$ implemented as an adjacency list and a source vertex $s \in V$.

Output: $\forall t \in V$, return $dist[t]$ (shortest distance to t) and $parent[t]$ (node on a shortest $s \rightsquigarrow t$ path that occurs right before t)

These notes were adapted from Kleinberg and Tardos' *Algorithm Design* and CLRS Ch. 24.3

* We should mention that although the problem is specified for a directed graph, we can handle the case of an undirected graph easily.

```
Dijkstra(G, s)
    for each v ∈ V do
        dist[v] = ∞
        parent[v] = NIL

    dist[s] = 0

    S = ∅
    Q = min-priority queue on all vertices, keyed by dist value

    while Q is not empty do
        u = Extract-Min(Q)
        S = S ∪ {u}
        for each v ∈ Adj[u] do
            // edge relaxation step
            if v ∈ Q and dist[v] > dist[u] + w(u, v) then
                dist[v] = dist[u] + w(u, v)        // decrease-key operation
                parent[v] = u
```

To get a better sense of what the algorithm is doing, consider the snapshot of its execution depicted in the figure below. At the point the picture is drawn, two iterations have been performed: the first added node u, and the second added node v. In the following iteration, the node x will be added since $dist[x] > dist[u] + w(u, x)$. Note that attempting to add y or z to the set S at this point would lead to an incorrect value for their shortest-path distances; ultimately, they will be added because of their edges from x.

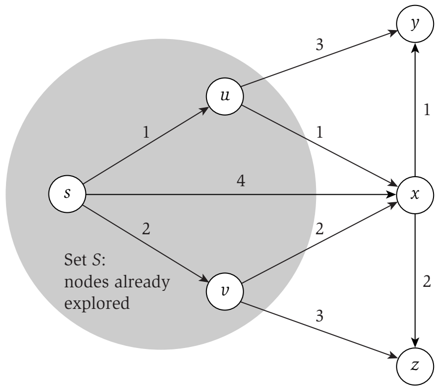


Figure 20.1: A snapshot of the execution of Dijkstra's Algorithm. The next node that will be added to the set S is x, due to the path through u.

Analyzing the Algorithm

We see in this example that Dijkstra's Algorithm is doing the right thing and avoiding recurring pitfalls: growing the set S by the wrong node can lead to an overestimate of the shortest-path distance to that node. The question becomes: Is it always true that when Dijkstra's Algorithm adds a node v, we get the true shortest-path distance to v?

We now answer this by proving the correctness of the algorithm, showing that the paths P_u really are shortest paths. Dijkstra's Algorithm is greedy in the sense that we always form the shortest new $s \rightsquigarrow v$ path we can make from a path in S followed by a single edge. We prove its correctness using a variant of our first style of analysis: we show that it "stays ahead" of all other solutions by establishing, inductively, that each time it selects a path to a node v, that path is shorter than every other possible path to v.

Proposition 1. *Consider the set S at any point in the algorithm's execution. For each $u \in S$, the path P_u is a shortest $s \rightsquigarrow u$ path.*

Note that this fact immediately establishes the correctness of Dijkstra's Algorithm, since we can apply it when the algorithm terminates, at which point S includes all nodes.

Proof. We prove this by induction on the size of S.

Base Case: $|S| = 1$ holds since we have $S = \{s\}$ and $dist[s] = 0$.

Induction Hypothesis: Suppose the claim holds when $|S| = k$ for some value of $k \geq 1$.

Induction Step: We now grow S to size $k + 1$ by adding the node v. Let (u, v) be the final edge on our $s \rightsquigarrow v$ path P_v.

By induction hypothesis, P_u is the shortest $s \rightsquigarrow u$ path for each $u \in S$. Now consider any other $s \rightsquigarrow v$ path P; we wish to show that it is at least as long as P_v. In order to reach v, this path P must leave the set S somewhere; let y be the first node on P that is not in S, and let $x \in S$ be the node just before y.

The situation is now as depicted in the figure below, and the crux of the proof is very simple: P cannot be shorter than P_v because it is already at least as long as P_v by the time it has left the set S. Indeed, in iteration $k + 1$, Dijkstra's Algorithm must have considered adding node y to the set S via the edge (x, y) and rejected this option in favor of adding v. This means that there is no path from s to y through x that is shorter than P_v. But the subpath of P up to y is such a path, and so this subpath is at least as long as P_v. Since edge lengths are non-negative, the full path P is at least as long as P_v as well.

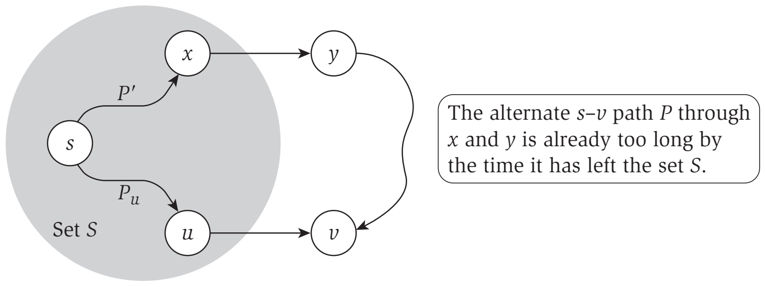

x
y
P′
s
P_u
Set S
u
v
The alternate s–v path P through x and y is already too long by the time it has left the set S.

Figure 20.2: The shortest path P_v and an alternate $s \rightsquigarrow v$ path P through the node y.

□

Here are two observations about Dijkstra's Algorithm and its analysis. First, the algorithm does not always find shortest paths if some of the edges can have negative lengths. (Do you see where the proof breaks?) Many shortest-path applications involve negative edge lengths, and a more complex algorithm—due to Bellman and Ford—is required for this case. You will explore this topic in-depth in CIS 320.

The second observation is that Dijkstra's Algorithm is, in a sense, even simpler than we've described here. Dijkstra's Algorithm is really a "continuous" version of the standard BFS algorithm for traversing a graph, and it can be motivated by the following physical intuition. Suppose the edges of G formed a system of pipes filled with water, joined together at the nodes; each edge (u, v) has length $w(u, v)$ and a fixed cross-sectional area. Now suppose an extra droplet of water falls at node s and starts a wave from s. As the wave expands out of node s at a constant speed, the expanding sphere of wavefront reaches nodes in increasing order of their distance from s. It is easy to believe (and also true) that the path taken by the wavefront to get to any node v is a shortest path. Indeed, it is easy to see that this is exactly the path to v found by Dijkstra's Algorithm, and that the nodes are discovered by the expanding water in the same order that they are discovered by Dijkstra's Algorithm.

Implementation and Running Time

Now that we have proved the correctness of Dijkstra, we'll look at how long it will take to run.

It maintains the min-priority queue Q by calling three priority-queue operations: Build-Heap, Extract-Min, and Decrease-Key. The algorithm calls Build-Heap once an all n nodes, and calls Extract-Min once per vertex. Because each vertex $u \in V$ is added to set S exactly once, each edge in the adjacency list $Adj[u]$ is examined in the for loop exactly once during the course of the algorithm. Since the total number of edges in all the adjacency lists is $|E|$, this for loop iterates a total of $|E|$ times, and thus the algorithm calls Decrease-Key at most $|E|$ times overall.

The running time of Dijkstra's algorithm depends on how we implement the min-priority queue. In this course, we focus on implementing the min-priority queue with a binary min-heap. Each Extract-Min operation then takes time $O(\lg V)$. As before, there are $|V|$ such operations. The time to build the binary min-heap is $O(V)$. Each Decrease-Key operation takes time $O(\lg V)$, and there are still at most $|E|$ such operations. The total running time is therefore $O((V + E) \lg V)$, which is $O(E \lg V)$ if all vertices are reachable from the source.

The fastest implementation of Dijkstra uses Fibonacci heaps, and is $O(V \lg V + E)$. This is beyond the scope of this class.

20.3 Shortest Path in DAGs

As seen in the previous section, Dijkstra's algorithm is slowed down through the use of the priority queue.

By relaxing the edges of a weighted DAG $G = (V, E)$ according to a topological sort of its vertices, we can compute shortest paths from a single source in $\Theta(V + E)$ time. Shortest paths are always well defined in a DAG, since even if there are negative-weight edges, no negative-weight cycles can exist. The algorithm starts by topologically sorting the DAG to impose a linear ordering on the vertices. If the DAG contains a path from vertex u to vertex v, then u precedes v in the topological sort. We make just one pass over the vertices in the topologically sorted order. As we process each vertex, we relax each edge that leaves the vertex.

Shortest Path in DAG Algorithm

Input: A DAG $G = (V, E)$ implemented as an adjacency list and a source vertex $s \in V$.

Output: $\forall t \in V$, return $dist[t]$ (shortest distance to t) and $parent[t]$ (node on a shortest $s \rightsquigarrow t$ path that occurs right before t)

```
DAG-Shortest-Path(G, s)
  topologically sort the vertices of G
```

```
for each v ∈ V do
    dist[v] = ∞
    parent[v] = NIL

dist[s] = 0

for each vertex u, taken in topologically sorted order do
    for each vertex v ∈ Adj[u] do
        if dist[v] > dist[u] + w(u, v) then
            dist[v] = dist[u] + w(u, v)
            parent[v] = u
```

The running time of this algorithm is easy to analyze. As covered previously, topological sort takes $\Theta(V+E)$ time. Initializing the *dist* and *parent* arrays takes $\Theta(V)$ time. The for loop that iterates through the vertices in topological order makes one iteration per vertex. Altogether, that for loop relaxes each edge exactly once. (We have used an aggregate analysis here.) Because each iteration of the inner for loop takes $\Theta(1)$ time, the total running time is $\Theta(V+E)$, which is linear in the size of an adjacency-list representation of the graph.

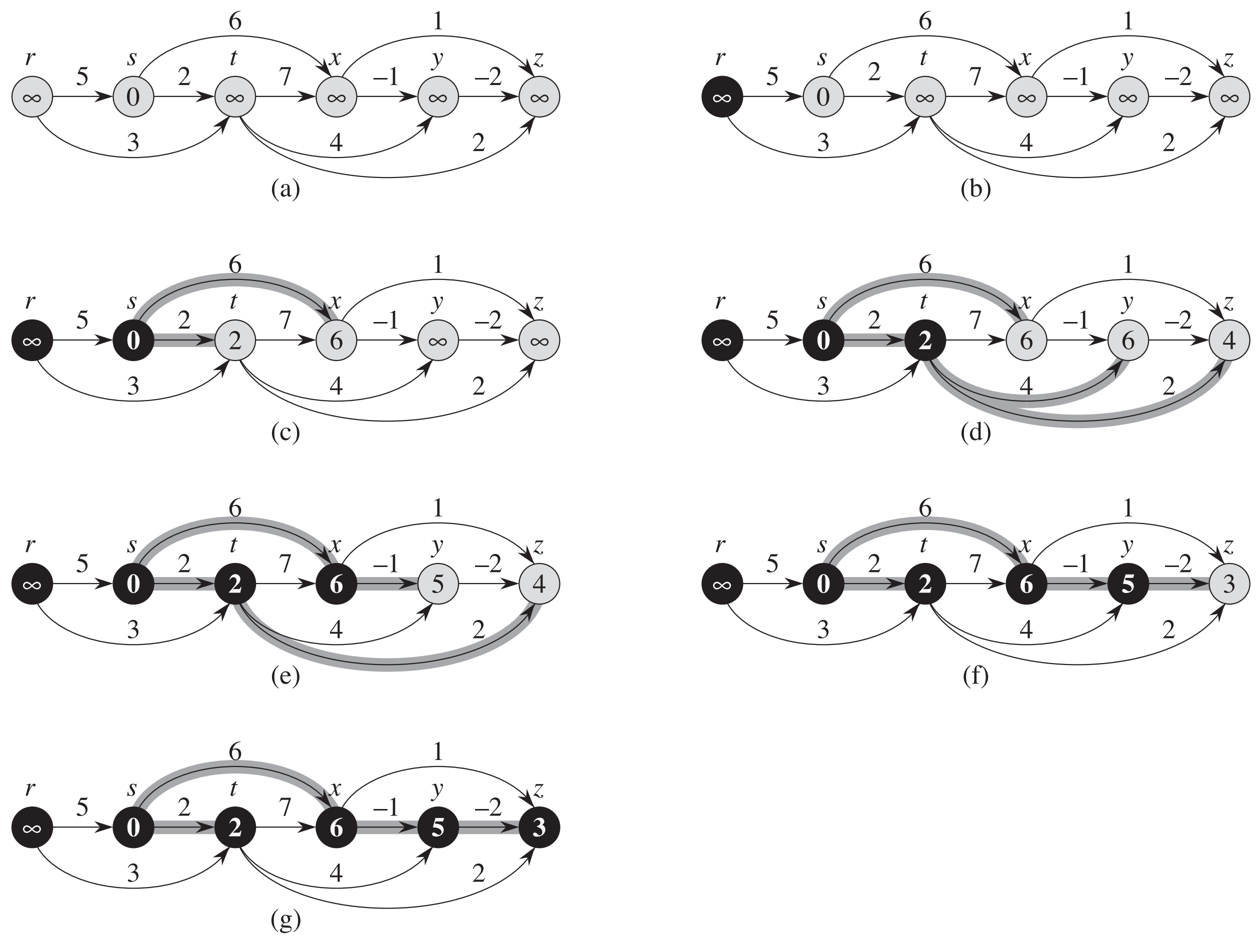


Figure 20.3: The execution of the algorithm for shortest paths in a DAG. The vertices are topologically sorted from left to right. The source vertex is s. The *dist* values appear within the vertices, and shaded edges indicate the *parent* values. **(a)** The situation before the first iteration of the second for loop. **(b)-(g)** The situation after each iteration of the second for loop. The newly blackened vertex in each iteration was used as u in that iteration. The values shown in part **(g)** are the final values.

The following proof shows that the algorithm correctly computes the shortest paths.

Proof. Let v be the first vertex in the topological ordering for which $dist[v] \neq \delta(s, v)$, where $\delta(s, v)$ is the shortest path distance from $s \rightsquigarrow v$ in G. Let $parent[v] = u$. Let the actual shortest path from $s \rightsquigarrow v$ in G be given by $s \rightsquigarrow w \rightarrow v$.

Case 1: $dist[v] = \infty$: Note that w comes before v in the topological ordering, and $dist[w] = \delta(s, w)$. Thus, by the time the outer for loop finishes processing w, we must have updated $dist[v]$ to a value smaller than ∞. This is a contradiction.

Case 2: $dist[v] < \infty$: Recall that $parent[v] = u$. Note that both u and w come before v in the topological ordering. By the time the outer for loop finishes processing both u and w, the value of $dist[v]$ is no more than $\min(dist[u] + w(u, v), dist[w] + w(w, v))$. Since $dist[u] = \delta(s, u)$ and $dist[w] = \delta(s, w)$, it must be that $dist[v]$ is no more than $\delta(s, w) + w(w, v)$. This is a contradiction. □

Minimum Spanning Trees 21

21.1 Introduction and Background

Consider a very natural problem: we are given a set of locations $V = \{v_1, v_2, ..., v_n\}$. We want to build a road system that connects these locations. Building a road between locations (v_i, v_j) costs some amount of money, $c(v_i, v_j) > 0$. Hence we want our road system to be as cheap as possible.

As the notation suggests, we can model this as a graph problem: we are given a set of vertices $V = \{v_1, v_2, ..., v_n\}$, a set of edges E, and a mapping $w : E \to \mathbb{R}^+$ from edges to positive real numbers (we will discuss applications with negative real numbers later). Here w is called the *weight function*. We assume the graph $G = (V, E)$ is connected, otherwise one can apply the results of this section to each connected component separately.

Stated this way, our goal is to find a subset of edges $T \subseteq E$ such that the graph (V, T) is connected and the total cost, defined as $w(T) = \sum_{e \in T} w(e)$, is as small as possible. We call such a graph T a minimum weight spanning subgraph:

Definition. A spanning subgraph of G is a subgraph of G that contains all the vertices in G.

A minimum weight spanning subgraph or minimum spanning subgraph is a spanning subgraph whose total cost is the minimum over all other spanning subgraphs.

Note We will often abuse notation and interchange T and the graph (V, T). For example, we may say "consider the graph T" even though T is technically a set of edges. In most cases, there is no ambiguity.

When the edge weights are all positive, we have the following result:

Proposition 1. *Let T be a minimum cost set of edges as described above. Then (V, T) is a tree.*

Proof. We know T is connected by how it is defined. Seeking contradiction, supposed T contained a cycle C. Choose any edge e on this cycle, and remove it from T. This forms a new graph with edge set $T' = T - \{e\}$. Clearly this resulting graph is still connected, however it's weight is $w(T') = w(T) - w(e) < w(T)$ since all the edge weights were positive. This contradicts the assumption that T is the minimum cost set of edges that connect the graph. □

Hence the graph T is in fact a *minimum spanning tree* (MST), and our goal will be to design an algorithm that efficiently computes the MST of a given graph.

Clearly the above proposition doesn't hold if there are negative weight edges: the critical part of the proof above was that e had positive weight. On the other hand, if we allow some edges to have weight 0, then even though there may be multiple minimum spanning subgraphs, there will always exist at least one MST. Why? Note that by the above proof, there cannot be any cycles in which every edge has positive weight. That is, every cycle has at least one edge with weight 0. Hence if you remove all such edges, you get rid of all the cycles, and the resulting graph has the same weight as the original. Moreover, it is a tree.

These notes were adapted from Kleinberg and Tardos' *Algorithm Design*

21.2 MST Algorithms

It turns out that many of the algorithms used to find MSTs are very simple greedy procedures. We will consider two of the more popular algorithms here.

Prim's Algorithm

Prim's algorithm is very similar to Dijkstra's shortest-paths algorithm. Let $G = (V, E)$ be a connected graph. We choose any starting node $s \in V$ and greedily grow a tree outward from s by simply adding the node that can be attached as cheaply as possible to the partial tree we already have.

More rigorously, we maintain a set T, where initially $T = \{s\}$. At each step, we consider all the vertices in $V - T$ which have an edge to some vertex in T. From these, we choose the one that has the minimum weight edge.

It is similar to Dijkstra's algorithm in that we will be maintaining a priority queue to determine which vertex is the one that has the minimum weight edge to some vertex in T. In fact, the pseudocode is identical to Dijkstra's, the only difference being how we update the keys in the priority queue. The tree T in the pseudocode is represented by parent pointers, while the set $V - T$ is represented by the vertices in the priority queue.

Prim's Algorithm for MSTs

Input: A connected graph $G = (V, E)$ given as an adjacency list, a weight function $w : E \to \mathbb{R}^+$, and a starting node s.

Output: An MST of G.

```
Prim(G, s)
    Let PQ be a priority queue containing every vertex in G
    for each v ∈ V do
        v.key = ∞
        parent[v] = NIL

    s.key = 0

    while PQ is not empty do
        v = PQ.ExtractMin()
        for each u ∈ Adj[v] do
            if u ∈ PQ and w(u,v) < u.key then
                u.key = w(u,v)
                parent[u] = v

    Return the parent array
```

Another way to phrase Prim's algorithm is that it starts with $T = \{s\}$ and at each step, chooses the lightest weight edge crossing the cut $(T, V - T)$ (here we use T as a set of vertices) and adds it to the growing tree.

$V - T$ represents all the vertices still in the priority queue. This may be easier to visualize, as in the figure below:

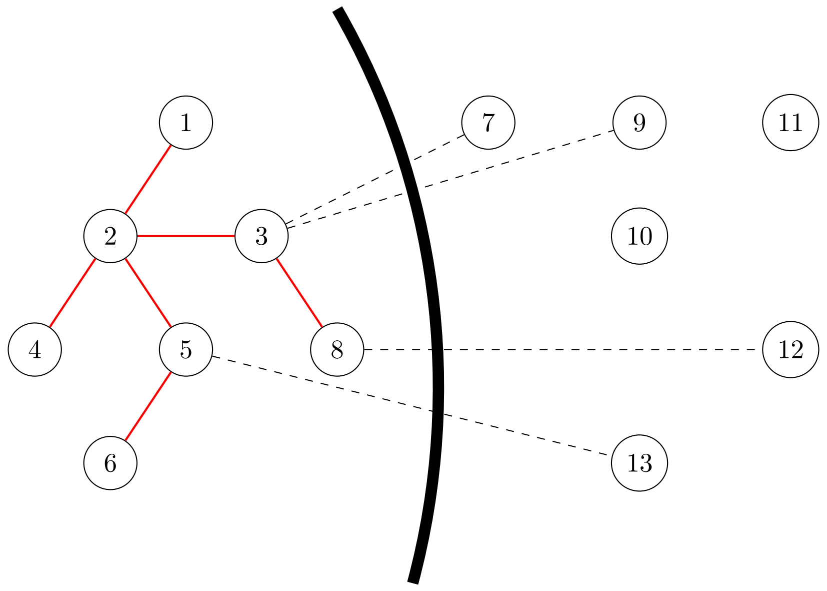


Figure 21.1: Prim's Algorithm in the middle of execution. The red lines denote the tree T that is being built. The next step in the algorithm will be to choose the lightest edge crossing the $(T, V-T)$ cut and add it to T. Here, that means choosing the lightest weight dotted line. In this picture, vertices 10 and 11 have keys equal to ∞ in the priority queue.

We will postpone a proof of correctness for Prim's algorithm until later. The runtime is easily seen to be $O(m \lg n)$ by analogy to Dijkstra's algorithm.

Kruskal's Algorithm

Kruskal's algorithm takes a different approach: it begins with a graph T that has no edges. Then it iterates through the edges in increasing order of weight. For each edge e, if adding e to T doesn't create a cycle, then $T = T \cup \{e\}$, otherwise you discard e and move on to the next edge. At the end of the algorithm, T will be an MST.

In order to determine if adding an edge (u, v) creates a cycle, we will need to use the Union Find (UF) data structure, which has the following methods:

- `MakeSet(x)`: creates a set with the single element x
- `Find(x)`: returns the ID of the set to which x belongs
- `Union(x,y)`: combines the set containing x and the set containing y

More details are provided in the Union Find notes, which also discusses the runtimes of the above functions.

Using UF, we can implement Kruskal's algorithm as follows (note how we don't need to specify a starting vertex like we do in Prim's algorithm):

Kruskal's Algorithm for MSTs

Input: A connected graph $G = (V, E)$ given as an adjacency list, a weight function $w : E \to \mathbb{R}^+$.

Output: An MST of G.

```
Kruskal(G)
    $T = \emptyset$
    Sort the edges in $E$ in increasing order of weight.
    for each $v \in V$ do
        MakeSet($v$)

    for each $e = (u, v) \in E$ do
        if Find($u$) $\neq$ Find($v$)
            $T = T \cup \{e\}$
            Union($u, v$)

    return $T$
```

Sorting the edges takes $O(m \log m) = O(m \log n)$ time, and as the Union Find notes prove, this $O(m \log n)$ term in fact dominates the runtime of all the UF operations combined. Hence Kruskal's also runs in $O(m \log n)$ time, just like Prim's (and Dijkstra's).

Reverse-Delete

Prim's algorithm and Kruskal's algorithm both take different approaches to finding an MST. Prim's algorithm starts with a source node s, then progressively grows an MST outwards from s. On the other hand, Kruskal's algorithm starts with a bunch of different trees (initially just single nodes), and combines these trees by adding edges until there is just one single tree.

In fact, there is another algorithm for finding MSTs called "Reverse Delete" which is kind of like a backwards-Kruskal. It starts with the full graph $G = (V, E)$, then iterates through the edges in *decreasing* order of weight, deleting edges as long as they don't disconnect the graph. The pseudocode is below:

Reverse-Delete Algorithm for MSTs

Input: A connected graph $G = (V, E)$ given as an adjacency list, a weight function $w : E \to \mathbb{R}^+$.

Output: An MST of G.

```
Reverse-Delete(G)
    $T = E$
    Sort the edges in $E$ in decreasing order of weight.

    for each $e = (u, v) \in E$ do
        if $T - \{e\}$ is connected
            $T = T - \{e\}$

    return $T$
```

Checking if $T - \{e\}$ is connected can be done in $O(n+m)$ time using BFS, hence the algorithm can be implemented in $O(m \lg n + m(n+m)) = O(m^2)$, where $n = O(m)$ since we assume the input graph is connected.

It may be surprising that there are so many efficient greedy algorithms for solving the MST problem. The proof of correctness for both Prim's and Kruskal's will shed some light on why this is the case.

21.3 Correctness of Prim's, Kruskal's, and Reverse-Delete

Since Prim's and Kruskal's algorithms work by repeatedly inserting edges from a partial solution, it will be useful to characterize when an edge is "safe" to include in the MST. We will also provide a characterization of edges that are guaranteed not to be in any MST. In this analysis we will assume that all the edge weights of the input graph $G = (V, E)$ are distinct–you should think about how to argue that this assumption is WLOG. We will revisit this topic at the end of the notes.

We have the following two important results:

Proposition 2. ***(Cut Property)*** *Let $G = (V, E)$ be a connected, undirected graph. Let S be any subset of nodes that is neither empty nor equal to all of V. Let $e = (u, v)$ be the minimum cost edge with one end in S and the other in $V - S$. Then every MST of G contains e.*

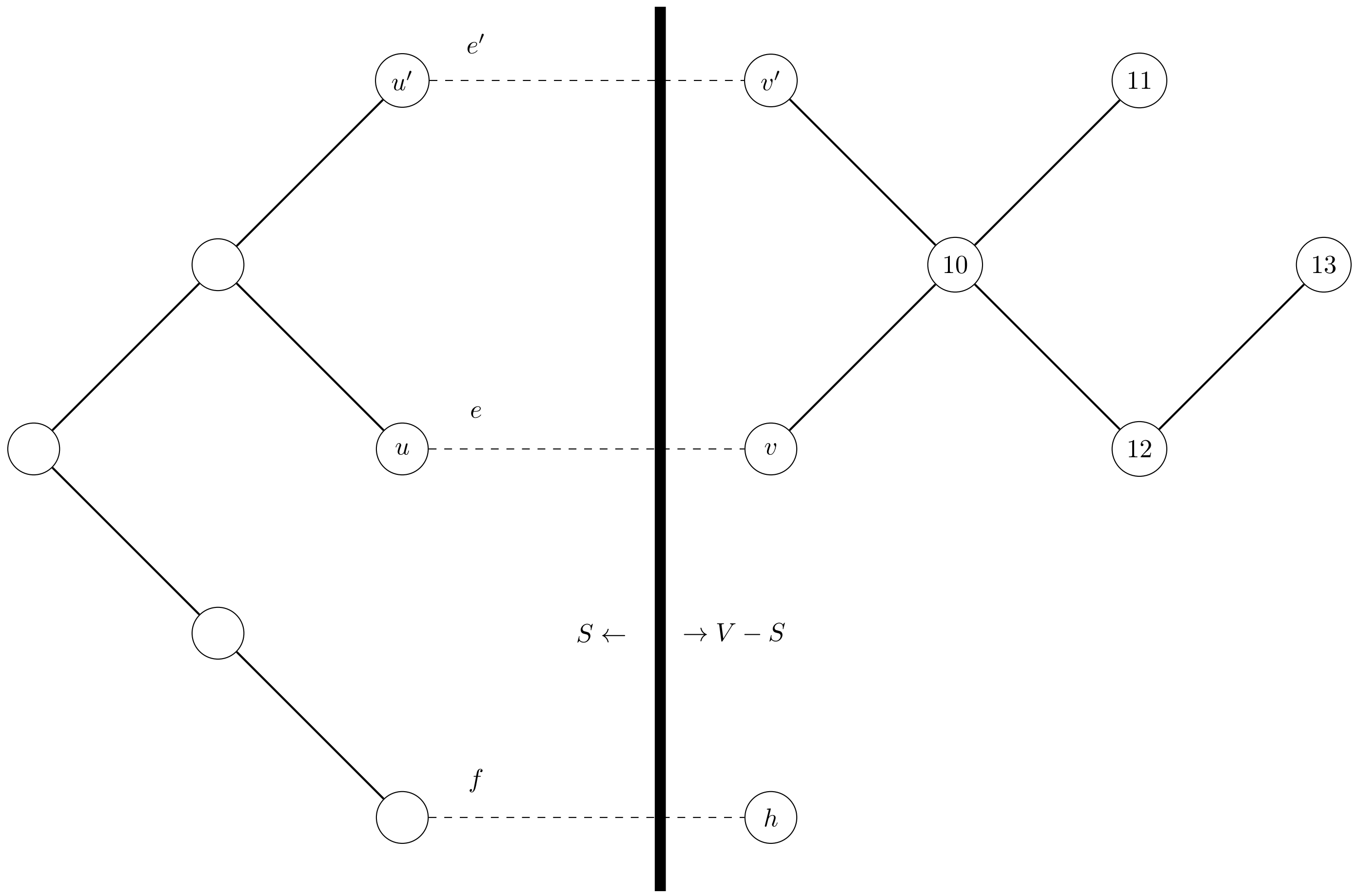
Proof. Let T be a spanning tree that does not contain e. We need to show that T does not have the minimum possible cost. To do this, we will use an exchange argument: we need to find an edge e' in T that is more expensive than e such that $T' = T - \{e'\} \cup \{e\}$ is a tree with lower total weight than T.

Since T is a spanning tree, there must be a path P from u to v in T. Since one end of e is in S and the other is in $V - S$, the path P must cross the $(S, V - S)$ cut at some point. Let $e' = (u', v')$ be the edge on P that crosses this cut.

Now consider $T' = T - \{e'\} \cup \{e\}$. T' is a spanning tree: since T was a spanning tree, any path in T that went through e' can be "re-routed" in T' through the edge e. Also, T' is acyclic because the only cycle in $T' \cup \{e'\}$ is the one composed of e and the path P, and this cycle is not present in T' due to the deletion of e'.

Now, since e was the lightest edge crossing the $(S, V - S)$ cut, we must have $w(e) < w(e')$. Hence the weight of T' must be strictly less than the weight of T, since T' and T are the same except for the edges e and e'. □

Note the choice of the edge e' is important–you need to make sure that T' is in fact a spanning tree. If you just choose *any* edge in T that crosses the $(S, V - S)$ cut, such as the edge f in the figure below, then it is not guaranteed that $T' = T - \{f\} \cup \{e\}$ is a spanning tree.



While the Cut Property is all that is needed to prove the correctness of Prim's and Kruskal's algorithm, we include the following important result since it is useful for analyzing properties of MSTs (and can also be used to prove the correctness of the Reverse-Delete algorithm):

Proposition 3. ***(Cycle Property)*** *Let $G = (V, E)$ be a connected, undirected graph. Let C be any cycle in G and let $e = (u, v)$ be the heaviest edge in C. Then e does not belong to any MST of G.*

Proof. Let T be a spanning tree that contains e. We will show T doesn't have the minimum possible weight. First, delete e from T: this partitions the nodes into two components, S (which contains u) and $V - S$ (which contains v). Consider the cycle C. $C - \{e\}$ is just a path from u to v and hence must cross the cut $(S, V - S)$ at some point. Let e' be the edge in $C - \{e\}$ such that e' has one endpoint in S and the other in $V - S$. Then $T' = T - \{e\} \cup \{e'\}$ is a spanning tree of G (the argument is similar to the one in the proof of the cut property) and since e was the heaviest edge on C, we know $w(e) > w(e')$ and thus the weight of T' is strictly less than the weight of T. Hence T doesn't have the lowest possible weight, completing the proof. $\square$

With the cut property, the correctness of Prim's and Kruskal's are straightforward.

Prim's Algorithm: Correctness

In each iteration of Prim's, there is a set $S \subseteq V$ on which a partial spanning tree has been constructed, and a node v and edge e are added to minimize $\min_{e=(u,v)\ \mid\ u \in S} w(e)$. Hence by definition, e is the lightest edge crossing the $(S, V-S)$ cut so by the cut property, e is in every MST. Since Prim's outputs a spanning tree such that every edge in the spanning tree is contained in an MST, Prim's outputs an MST.

Kruskal's Algorithm: Correctness

Consider any edge $e = (u, v)$ added by Kruskal's algorithm, and let S be the set of all nodes to which u has a path at the moment just before e is added. Clearly $u \in S$ but $v \notin S$ since adding e doesn't create a cycle. Moreover, no edge from S to $V - S$ has been encountered yet, since any such edge could have been added without creating a cycle, and hence would have already been added by Kruskal's algorithm. Thus e is the lightest edge crossing the $(S, V-S)$ cut, and so it belongs to every MST. Since by construction, Kruskal's algorithm outputs a spanning tree, it follows this output is actually an MST.

Reverse-Delete Algorithm: Correctness

The correctness of the reverse delete algorithm follows immediately from the cycle property. Consider any edge $e = (u, v)$ removed by Reverse-Delete. At the time e is removed, it lies on some cycle C; and since it is the first edge on C encountered by the algorithm in decreasing order of edge costs, it must be the heaviest edge on C. Thus by the cycle property e doesn't belong to any MST. Moreover, by construction, the output of Reverse-Delete is a spanning tree, since it never removes an edge that disconnects the graph and it will never end while there is a cycle in G.

21.4 Eliminating the Assumption that All Edge Weights are Distinct

Suppose we are given an instance of the MST problem in which certain edges have the same weight. To apply the results from this section, we need to specify some way to "break ties" among edges with the same weights (i.e. some way to impose an ordering over the edges with the same weights). Suppose we perturb each edge weight by small, different numbers so that they all become distinct yet maintain their relative ordering. For example, define

$$\delta = \min_{e,e' \in E} |w(e) - w(e')|$$

to be the smallest difference in weights between two edges. We can then construct the weight function w' such that for each set of edges $\{e_0, e_1, ..., e_k\}$ with the same weight $w(e_0) = w(e_1) = ... = w(e_k)$, the weight function w' is defined by $w'(e_i) = w(e_i) + i \cdot \frac{\delta}{m}$. For edges e that aren't part of a tie, we can set $w'(e) = w(e)$. This ensures that

(1) The edges in G maintain their relative order (i.e. if $w(e) < w(e')$ then $w'(e) < w'(e')$
(2) There are no ties

What we have essentially done is arbitrarily break ties between the edges with the same weights in a way that allows us to apply the results from the above sections.

Now, any minimum spanning tree T for a graph with the new edge weight function w' is also a MST for the graph with the edge weight function w. Moreover, we can restate the cut and cycle properties to account for tied edge weights:

Proposition 4. ***(Cut Property)*** *Let $G = (V, E)$ be a connected, undirected graph. Let S be any subset of nodes that is neither empty nor equal to all of V. Let $e = (u, v)$ be a minimum cost edge with one end in S and the other in $V - S$. Then some MST of G contains e.*

Proposition 5. ***(Cycle Property)*** *Let $G = (V, E)$ be a connected, undirected graph. Let C be any cycle in G and let $e = (u, v)$ be a heaviest edge in C. Then e does not belong to some MST of G.*

Note the subtle difference: here we say that if e is *some* (not "the") minimum weight edge crossing the cut, then *there exists* an MST containing e. Similarly, if e is *a* heaviest edge on a cycle, then *there exists* an MST that doesn't contain e.

Union Find 22

22.1 Introduction

Some applications involve grouping n distinct elements into a collection of disjoint sets. These applications often need to perform two operations in particular: finding the unique set that contains a given element and unioning two sets. This chapter explores methods for maintaining a data structure that supports these operations.

Definition. A disjoint-set data structure maintains a collection $\mathcal{S} = \{S_1, S_2, ..., S_k\}$ of disjoint dynamic sets. We identify each set by a representative, which is some member of the set (for our purposes in this course, it does not matter which).

In this course, we will refer to this data structure as Union Find (coming from the name of the operations it supports).

We wish to support the following operations:

Make-Set(x) creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.

Union(x, y) unions the two sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. If these two sets are not disjoint, then they must be the same set (because non-disjoint sets are not allowed), and thus no change is made. Otherwise, a new set S is created with the elements in $S_x \cup S_y$, with any valid representative element (implementation specific detail we will cover soon). The original sets S_x and S_y are destroyed. In practice, we often absorb the elements of one of the sets into the other set.

Find(x) returns a pointer to the representative element of the set containing x. If x and y belong to the same set (i.e., they are connected), Find(x) == Find(y).

A common use case is to check if two nodes are in the same connected component. This is easily accomplished:

```
Connected-Components($G$):
  for each vertex $v \in G.V$
      Make-Set($v$)
  for each edge $(u, v) \in G.E$
      if Find($u$) $\neq$ Find($v$)
          Union($u, v$)
```

```
Same-Component($u, v$):
  return Find($u$) == Find($v$)
```

22.2 Union by Rank

As shown in the figure below, one way to store a set is as a directed tree. Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree. This root element is a convenient *representative*, or *name*, for the set. It is distinguished from the other elements by the fact that its parent pointer is a self-loop.

These notes were adapted from CLRS Chapter 21 and S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani's *Algorithm* Textbook

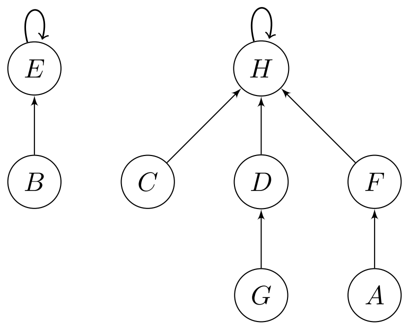


Figure 22.1: A directed-tree representation of two sets $\{B, E\}$ and $\{A, C, D, F, G, H\}$.

In addition to a parent pointer π, each node also has a *rank* that, for the time being, should be interpreted as the height of the subtree hanging from that node.

```
MAKE-SET($x$):
    $\pi(x) = x$
    $rank(x) = 0$
```

```
FIND($x$):
    while $x \neq \pi(x)$
        $x = \pi(x)$
    return $x$
```

As can be expected, MAKE-SET is a constant-time operation. On the other hand, FIND follows parent pointers to the root of the tree and therefore takes time proportional to the height of the tree. The tree actually gets built via the third operation, UNION, and so we must make sure that this procedure keeps trees shallow.

Merging two sets is easy: make the root of one point to the root of the other. But we have a choice here. If the representatives (roots) of the sets are r_x and r_y, do we make r_x point to r_y or the other way around?

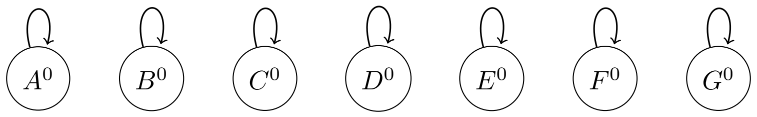
Definition. Since tree height is the main impediment to computational efficiency, a good strategy is to *make the root of the shorter tree point to the root of the taller tree.* This way, the overall height increases only if the two trees being merged are equally tall. Instead of explicitly computing heights of trees, we will use the rank numbers of their root nodes–which is why this scheme is called union by rank.

By design, the rank of a node is exactly the height of the subtree rooted at that node. This means, for instance, that as you move up a path toward a root node, the rank values along the way are strictly increasing. However, as you will see later, the rank becomes just an upper bound on the height of the node, since it is not maintained when used along with path compression.

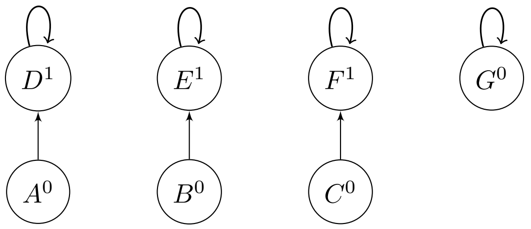
```
UNION($x, y$):
    $r_x$ = FIND($x$)
    $r_y$ = FIND($y$)
    if $r_x = r_y$
        return // already in the same set
    if $rank(r_x) > rank(r_y)$
        $\pi(r_y) = r_x$
    else
        $\pi(r_x) = r_y$
        if $rank(r_x) = rank(r_y)$
            $rank(r_y) = rank(r_y) + 1$
```

Here is an example of a sequence of disjoint-set operations (Make-Set and Union) to show how path compression works. The superscript in the node denotes the rank of the node.

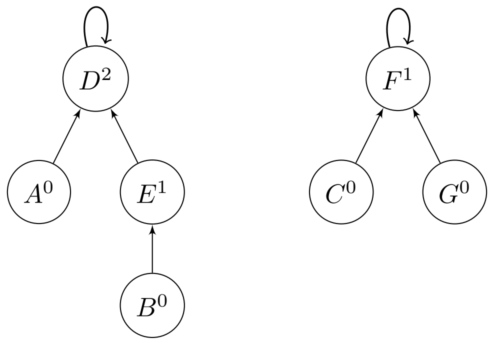
After Make-Set(A), Make-Set(B), ..., Make-Set(G):



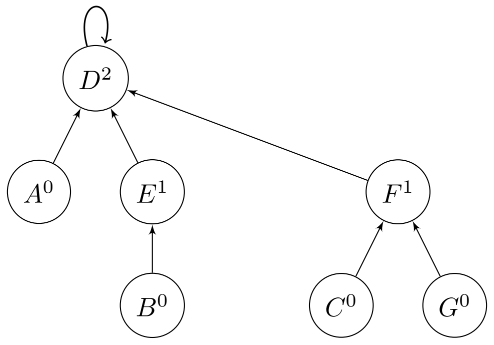
After Union(A, D), Union(B, E), and Union(C, F):



After Union(C, G) and Union(E, A):



After Union(B, G):



We note that the following three properties emerge:

Property 1. For any non-root x, $rank(x) < rank(\pi(x))$.

A root node with rank k is created by the merger of two trees with roots of rank $k-1$.

Property 2. Any root node of rank k has at least 2^k nodes in its tree.

This extends to internal (nonroot) nodes as well: a node of rank k has at least 2^k nodes in the subtree rooted at that node. After all, any internal node was once a root, and neither its rank nor its set of descendants has changed since then. Moreover, different rank-k nodes cannot have common descendants, since by Property 1 any element has at most one ancestor of rank k.

Property 3. If there are n elements overall, there can be at most $n/2^k$ nodes of rank k.

This last observation implies, crucially, that the maximum rank is $\log n$. Therefore, all the trees have height $\leq \log n$, and this is an upper bound on the running time of FIND and UNION.

22.3 Path Compression

With the data structure as presented so far, Kruskal's algorithm becomes $O(|E| \log |V|)$ for sorting the edges (remember, $\log |E| \approx \log |V|$) plus another $O(|E| \log |V|)$ for the FIND and UNION operations that dominate the rest of the algorithm. So there seems to be little incentive to make our data structure any more efficient.

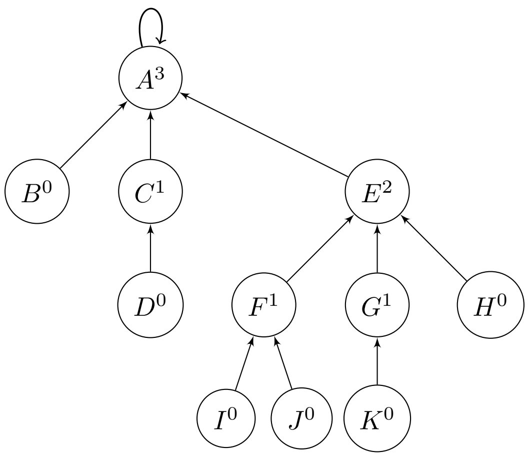
But what if the edges are given to us sorted? Or if the weights are small (say, $O(|E|)$) so that sorting can be done in linear time? Then the data structure part becomes the bottleneck, and it is useful to think about improving its performance beyond $\log n$ per operation. As it turns out, the improved data structure is useful in many other applications.

But how can we perform UNION's and FIND's faster than $\log n$? The answer is, by being a little more careful to maintain our data structure in good shape. As any housekeeper knows, a little extra effort put into routine maintenance can pay off handsomely in the long run, by forestalling major calamities. We have in mind a particular maintenance operation for our union-find data structure, intended to keep the trees short.

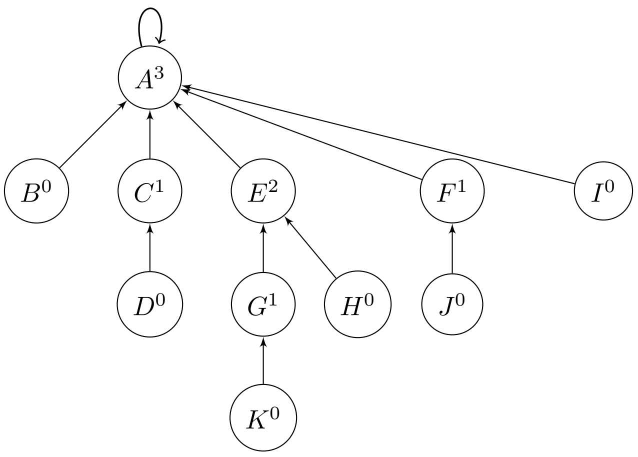
Definition. During each FIND, when a series of parent pointers is followed up to the root of a tree, we will change all these pointers so that they point directly to the root (see figure below). This path compression heuristic only slightly increases the time needed for a find and is easy to code.

```
FIND($x$):
  if $x \neq \pi(x)$
      $\pi(x) = $ FIND($\pi(x)$)
  return $\pi(x)$
```

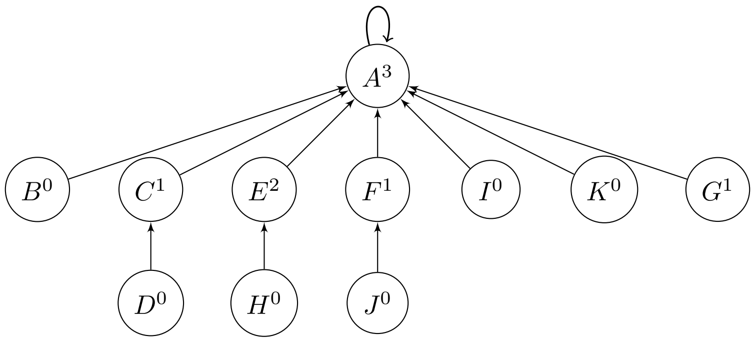
The following example demonstrates the effect of path compression.



After Find(I):



After Find(K):



The benefit of this simple alteration is long-term rather than instantaneous and thus necessitates a particular kind of analysis: we need to look at sequences of FIND and UNION operations, starting from an empty data structure, and determine the average time per operation. This amortized cost turns out to be just barely more than $O(1)$, down from the earlier $O(\log n)$.

Think of the data structure as having a "top level" consisting of the root nodes, and below it, the insides of the trees. There is a division of labor: FIND operations (with or without path compression) only touch the insides of trees, whereas UNION's only look at the top level. Thus path compression has no effect on UNION operations and leaves the top level unchanged.

We now know that the ranks of root nodes are unaltered, but what about *nonroot* nodes? The key point here is that once a node ceases to be a root, it never resurfaces, and its rank is forever fixed. Therefore the ranks of all nodes are unchanged by path compression, even though these numbers can no longer be interpreted as tree heights. In particular, properties 1-3 (from before) still hold.

If there are n elements, their rank values can range from 0 to $\log n$ by Property 3. Let's divide the nonzero part of this range into certain carefully chosen intervals, for reasons that will soon become clear:

$$\{1\}, \{2\}, \{3, 4\}, \{5, 6, ..., 16\}, \{17, 18, ..., 2^{16} = 65536\}, \{65537, 65538, ..., 2^{65536}\}$$

Each group is of the form $\{k+1, k+2, ..., 2^k\}$, where k is a power of 2. The number of groups is $\log^* n$, which is defined to be the number of successive log operations that need to be applied to n to bring it down to 1 (or below 1). For instance, $\log^* 1000 = 4$ since $\log\log\log\log 1000 \leq 1$. In practice there will just be the first five of the intervals shown; more are needed only if $n \geq 2^{65536}$, in other words never.

In a sequence of FIND operations, some may take longer than others. We'll bound the overall running time using some creative accounting. Specifically, we will give each node a certain amount of pocket money, such that the total money doled out is at most $n \log^* n$ dollars. We will then show that each find takes $O(\log^* n)$ steps, plus some additional amount of time that can be "paid for" using the pocket money of the nodes involved-one dollar per unit of time. Thus the overall time for m FIND's is $O(m \log^* n)$ plus at most $O(n \log^* n)$.

In more detail, a node receives its allowance as soon as it ceases to be a root, at which point its rank is fixed. If this rank lies in the interval $\{k+1, ..., 2^k\}$, the node receives 2^k dollars. By Property 3, the number of nodes with rank $> k$ is bounded by

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + ... \leq \frac{n}{2^k}$$

Therefore the total money given to nodes in this particular interval is at most n dollars, and since there are $\log^* n$ intervals, the total money disbursed to all nodes is $\leq n \log^* n$.

Now, the time taken by a specific FIND is simply the number of pointers followed. Consider the ascending rank values along this chain of nodes up to the root. Nodes x on the chain fall into two categories: either the rank of $\pi(x)$ is in a higher interval than the rank of x, or else it lies in the same interval. There are at most $\log^* n$ nodes of the first type, so the work done on them takes $O(\log^* n)$ time. The remaining nodes—whose parents' ranks are in the same interval as theirs—have to pay a dollar out of their pocket money for their processing time.

This only works if the initial allowance of each node x is enough to cover all of its payments in the sequence of FIND operations. Here's the crucial observation: each time x pays a dollar, its parent changes to one of higher rank. Therefore, if x's rank lies in the interval $\{k+1, ..., 2^k\}$, it has to pay at most 2^k dollars before its parent's rank is in a higher interval; whereupon it never has to pay again.[*]

[*] If you are interested in a more in-depth analysis of the $\log^* n$ proof, you can consult *CLRS 21.4: Analysis of union by rank with path compression.*

Hashing 23

A hash table is a commonly used data structure to store an unordered set of items, allowing constant time inserts, lookups and deletes (in expectation). Every item consists of a unique identifier called a *key* and a piece of information. For example, the key might be a Social Security Number, a driver's license number, or an employee ID number. The way in which a hash table stores a item depends only on its key, so we will only focus on the key here, but keep in mind that each key is usually associated with additional information that is also stored in the hash table.

A hash table supports the following operations:

- INSERT(k): Insert k into the hash table
- SEARCH(k): Check if k is in the hash table
- DELETE(k): Delete the key k from the hash table

23.1 Direct-Address Tables

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a key drawn from the universe $U = \{0, 1, ..., m-1\}$, where m is not too large. We will assume that no two elements have the same key.

To represent the dynamic set, we use an array, or *direct-address table*, denoted by $T[0..m-1]$, in which each position, or slot, corresponds to a key in the universe U. The figure below illustrates the approach; slot k points to an element in the set with key k. If the set contains no element with key k, then $T[k] = \text{NIL}$.

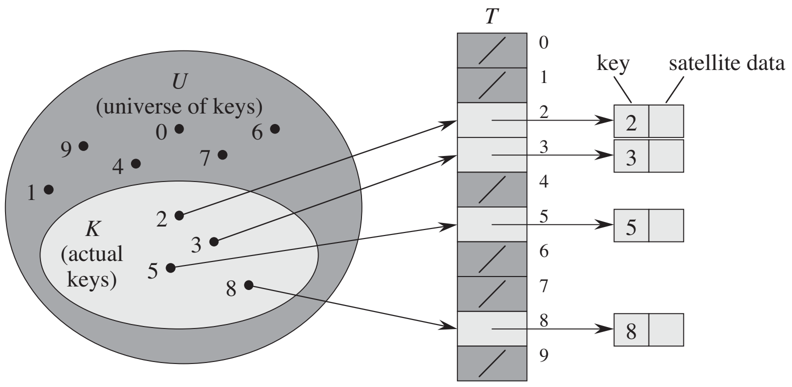


Figure 23.1: How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, ..., 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

The dictionary operations are trivial to implement, and each operation takes $O(1)$ time:

```
SEARCH(T, k):
    return T[k]
```

```
INSERT(T, x):
    T[x.key] = x
```

```
DELETE(T, k):
    T[k] = NIL
```

These notes were adapted from CLRS Chapter 11, though some of our proofs are approached differently.

This method would give us guaranteed $O(1)$ look-ups and inserts, but would take space $\Theta(|U|)$ which can be impracticably large. For example, assuming that the longest name at Penn is 20 characters long (including spaces), there would be 27^{20} strings in our universe. This is clearly inefficient because there is no way we are going to be storing that many names.

Thus, Direct Addressing could be a good option when U is small. But as U becomes very large, storing an array of size $|U|$ can be impractical due to memory limitations. Furthermore, often the set of actual keys K stored is much less than $|U|$, as such the extra space allocated to T would be wasted.

23.2 Hash Tables

Let $|U|$ be a very big universe, and let K be the set of keys stored in T. Let's also assume that $|K| \ll |U|$. We want a table T of size that is proportional to the number of keys stored in T, and we also want the runtime of the three operations to be $O(1)$. We will achieve $O(1)$ time per operation, but in the average case.

With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot $h(k)$; that is, we use a hash function h to compute the slot from the key k. Here, h maps the universe U of keys into the slots of a hash table $T[0..m-1]$ of size m. Formally, the hash function h is:

$$h : U \rightarrow 0, ..., m-1$$

where usually $m \ll |U|$. We say that an element with key k hashes to slot $h(k)$; we also say that $h(k)$ is the hash value of key k. The figure below illustrates the basic idea. The hash function reduces the range of array indices and hence the size of the array. Instead of a size of $|U|$, the array can have size m.

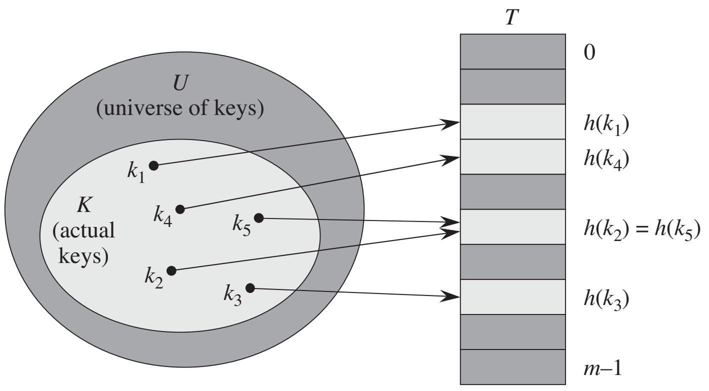


Figure 23.2: Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

By initializing a table of size $m \ll |U|$ and defining the hash function h, we've managed to save a lot of space. However, this improvement (like many things in life), comes at a cost: insertions and look-ups are now on average $O(1)$—worst case for some particular insert or lookup may be $O(n)$—instead of guaranteed $O(1)$. Furthermore, by the pigeonhole principle, two keys in U may now hash to the same slot, a situation known as a collision. Thus, while a well-designed, "random"-looking hash function can minimize the number of collisions, we still need a method for resolving the collisions that do occur. In this course we will cover separate chaining and open addressing, which are two common ways of resolving collisions

Collision Resolution by Chaining

In collision resolution by chaining, we place all the elements that hash to the same slot into the same linked list, as shown in the figure below. Slot j contains a pointer to the head of the list of all stored elements that hash to j; if there are no such elements, slot j contains NIL.

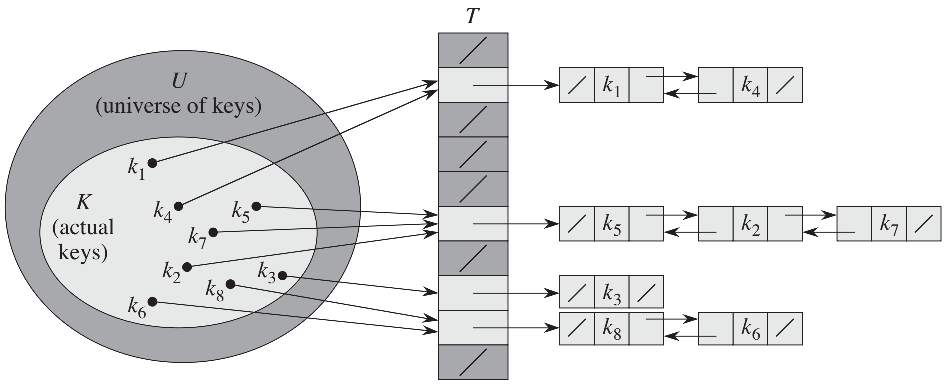


Figure 23.3: Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$. The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

The dictionary operations on a hash table T are easy to implement when collisions are resolved by chaining:

```
INSERT($T, x$):
    insert $x$ at the head of
    list $T[h(x.key)]$
```

```
SEARCH($T, k$):
    search for an element with
    key $k$ in list $T[h(k)]$
```

```
DELETE($T, x$):
    delete $x$ from the list
    $T[h(x.key)]$
```

Analysis of Hashing with Chaining

Given a hash table T with m slots that stores n elements, we define the load factor α for T as n/m, that is, the average number of elements stored in a chain. Our analysis will be in terms of α, which can be less than, equal to, or greater than 1.

The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n. The worst-case time for searching is thus $\Theta(n)$ plus the time to compute the hash function—no better than if we used one linked list for all the elements. Clearly, we do not use hash tables for their worst-case performance.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average. It intuitively follows that we seek a hash function that distributes the universe U of keys as evenly as possible among the m slots of our table. We can formalize this concept and give it a name:

Definition. The Simple Uniform Hashing Assumption (SUHA) states that any key k is equally likely to be mapped to any of the m slots in our hash table T, independently of where any other key is hashed to. In other words, the probability of hashing some key k not already present in the hash table into any arbitrary slot in T is $\frac{1}{m}$.

We assume that $O(1)$ time suffices to compute the hash value $h(k)$, so that the time required to search for an element with key k depends linearly on the length of the list $T[h(k)]$. Setting aside the $O(1)$ time required to compute the hash function and to access slot $h(k)$, let us consider the expected number of elements examined by the search algorithm, that is, the number of elements in the list $T[h(k)]$ that the algorithm checks to see whether any have a key equal to k. We consider two cases. In the first, the search is unsuccessful: no element in the table has key k. In the second, the search successfully finds an element with key k.

Theorem 1. *In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time* $\Theta(1+\alpha)$*, under the assumption of simple uniform hashing.*

Proof. The average time it takes to search for key k is the expected size of the linked list $T[h(k)]$.

$\mathbf{E}\Big[|T[h(k)]|\Big] = \sum_{i=1}^{n} \Pr\big[X_i \text{ is mapped to } h(k)\big] = \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m} = \alpha$

Thus, the expected number of elements examined in an unsuccessful search is α, and the total time required (including the time for computing $h(k)$) is $\Theta(1+\alpha)$. □

The situation for a successful search is slightly different, since each list is not equally likely to be searched. Instead, the probability that a list is searched is proportional to the number of elements it contains. Nonetheless, the expected search time still turns out to be $\Theta(1+\alpha)$.

Theorem 2. *In a hash table in which collisions are resolved by chaining, a successful search takes average-case time* $\Theta(1+\alpha)$*, under the assumption of simple uniform hashing.*

Proof. We assume that the element being searched for is equally likely to be any of the n elements stored in the table. The number of elements examined during a successful search for an element x is one more than the number of elements that appear before x in x's list. Because new elements are placed at the front of the list, elements before x in the list were all inserted after x was inserted. To find the expected number of elements examined, we take the average, over the n elements x in the table, of 1 plus the expected number of elements added to x's list after x was added to the list.

Let x_i denote the ith element inserted into the table, for $i = 1, 2, ..., n$, and let $k_i = x_i.key$. Let Z be the random variable denoting the search time in a successful search, let Z_i be a random variable denoting the search time in a successful search for x_i, let $Z_i^j = 1$ if x_j $(j \neq i)$ is mapped to same location as x_i, and let $Z_i^j = 0$ otherwise.

$$Z = \frac{Z_1 + Z_2 + ... + Z_n}{n}$$

By Linearity of Expectation, we get:

$$\begin{aligned}
\mathbf{E}[Z] &= \frac{1}{n}\sum_{i=1}^{n}(\mathbf{E}[Z_i]) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n}\mathbf{E}[Z_i^j]\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \frac{n-i}{m}\right)
\end{aligned}$$

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

Thus, the total time required for a successful search (including the time for computing the hash function) is $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$. □

What does this analysis mean? If the number of hash-table slots is at least proportional to the number of elements in the table, we have $n = O(m)$ and, consequently, $\alpha = n/m = O(m)/m = O(1)$. Thus, searching takes constant time on average. Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked, we can support all dictionary operations in $O(1)$ time on average.

23.3 Hash Functions

What makes a good hash function?

A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to. Unfortunately, we typically have no way to check this condition, since we rarely know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently.

Occasionally we do know the distribution. For example, if we know that the keys are random real numbers k independently and uniformly distributed in the range $0 \leq k < 1$ then the hash function $h(k) = \lfloor km \rfloor$ satisfies the condition of simple uniform hashing.

In practice, we can often employ heuristic techniques to create a hash function that performs well. Qualitative information about the distribution of keys may be useful in this design process. A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data.

Interpreting Keys as Natural Numbers

Most hash functions assume that the universe of keys is the set $\mathbb{N} = \{0, 1, 2, ...\}$ of natural numbers. Thus, if the keys are not natural numbers, we find a way to interpret them as natural numbers. For example, we can interpret a character string as an integer by summing the ASCII values and multiplying them by a constant c raised to a variable power. The string "pat" can therefore be represented as an integer: $\text{ASCII}(p) \cdot c^2+$ $\text{ASCII}(a) \cdot c^1+$ $\text{ASCII}(t) \cdot c^0$. We need to multiply that additional constant c raised to a variable power because if we didn't, then all permutations of the string would map to the same integer.

In the context of a given application, we can usually devise some such method for interpreting each key as a (possibly large) natural number. In Java, this is the job of the HashCode method. In what follows, we assume for simplicity that the keys are natural numbers.

The Division Method

In the division method for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is $h(k) = k \mod m$.

When using the division method, we usually avoid certain values of m. For example, m should not be a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k. Unless we know that all low-order p-bit patterns are equally likely, we are better off designing the hash function to depend on all the bits of the key. A prime not too close to an exact power of 2 is often a good choice for m. This is what has been found empirically.

The Multiplication Method

The multiplication method for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, we multiply this value by m and take the floor of the result. In short, the hash function is $h(k) = \lfloor m(kA \mod 1) \rfloor$ where "kA mod 1" means the fractional part of kA, that is, $kA - \lfloor kA \rfloor$. An advantage of the multiplication method is that the value of m is not critical. We typically choose it to be a power of 2 ($m = 2^p$ for some integer p), since we can then easily implement the function on most computers as follows.

Although this method works with any value of the constant A, it works better with some values than with others as has been shown empirically. $A \approx (\sqrt{5} - 1)/2 = .6180339887...$ is likely to work reasonably well.

23.4 Open Addressing

In open addressing, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table. No lists and no elements are stored outside the table, unlike in chaining. Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made; one consequence is that the load factor α can never exceed 1.

Of course, we could store the linked lists for chaining inside the hash table, in the otherwise unused hash-table slots, but the advantage of open addressing is that it avoids pointers altogether. Instead of following pointers, we *compute* the sequence of slots to be examined. The extra memory freed by not storing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

To perform insertion using open addressing, we successively examine, or probe, the hash table until we find an empty slot in which to put the key. Instead of being fixed in the order $\{0, 1, ..., m - 1\}$ (which requires $\Theta(n)$ search time), the sequence of positions probed *depends upon the key being inserted.* To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input. Thus, the hash function becomes:

$$h : U \times \{0, 1, ..., m - 1\} \to \{0, 1, ..., m - 1\}$$

With open addressing, we require that for every key k, the probe sequence

$$\langle h(k,0), h(k,1), ..., h(k,m-1)\rangle$$

be a permutation of $\langle 0, 1, ..., m-1\rangle$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up.

In the following pseudocode, we assume that the elements in the hash table T are keys with no satellite information; the key k is identical to the element containing key k. Each slot contains either a key or NIL (if the slot is empty).

```
INSERT($T, k$):
    for $i \leftarrow 0$ to $m-1$ do
        $j \leftarrow h(k,i)$
        if $T[j] ==$ NIL then
            $T[j] \leftarrow k$
            return $j$
    return "Error: hash table overflow!"
```

```
SEARCH($T, k$):
    for $i \leftarrow 0$ to $m-1$ do
        $j \leftarrow h(k,i)$
        if $T[j] == k$ then
            return $j$
        else if $T[j] ==$ NIL then
            return NIL
    return NIL
```

Deletion from an open-address hash table is difficult. When we delete a key from slot i, we cannot simply mark that slot as empty by storing NIL in it. If we did, we might be unable to retrieve any key k during whose insertion we had probed slot i and found it occupied. We can solve this problem by marking the slot, storing in it the special value DELETED instead of NIL. We would then modify INSERT to treat such a slot as if it were empty so that we can insert a new key there. We do not need to modify SEARCH, since it will pass over DELETED values while searching. When we use the special value DELETED, however, search times no longer depend on the load factor α, and for this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.

Definition. In our analysis, we assume uniform hashing: the probe sequence of each key is equally likely to be any of the $m!$ permutations of $\langle 0, 1, ..., m-1\rangle$. Uniform hashing generalizes the notion of simple uniform hashing defined earlier to a hash function that produces not just a single number, but a whole probe sequence. True uniform hashing is difficult to implement, however, and in practice suitable approximations (such as double hashing, defined below) are used.

We will examine three commonly used techniques to compute the probe sequences required for open addressing: linear probing, quadratic probing, and double hashing. These techniques all strive to guarantee that $\langle h(k,0), h(k,1), ..., h(k,m-1)\rangle$ is a permutation of $\langle 0, 1, ..., m-1\rangle$ for each key k. None of these techniques fulfills the assumption of uniform hashing, however, since none of them is capable of generating more than m^2 different probe sequences (instead of the $m!$ that uniform hashing requires). Double hashing has the greatest number of probe sequences and, as one might expect, seems to give the best results.

Linear Probing

Given an ordinary hash function $h' : U \to \{0, 1, ..., m-1\}$, which we refer to as an auxiliary hash function, the method of linear probing uses the hash function

$$h(k,i) = (h'(k) + i) \mod m$$

for $i = 0, 1, ..., m - 1$. Given key k, we first probe $T[h'(k)]$, i.e., the slot given by the auxiliary hash function. We next probe slot $T[h'(k) + 1]$, and so on up to slot $T[m - 1]$. Then we wrap around to slots $T[0], T[1], ...$ until we finally probe slot $T[h'(k) - 1]$. Because the initial probe determines the entire probe sequence, there are only m distinct probe sequences.

Linear probing is easy to implement, but does not work well in practice because it suffers from a problem known as primary clustering. Long runs of occupied slots build up, increasing the average search time. Clusters arise because an empty slot preceded by i full slots gets filled next with probability $(i + 1)/m$. Long runs of occupied slots tend to get longer, and the average search time increases.

Quadratic Probing

Quadratic probing uses a hash function of the form

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

where h' is an auxiliary hash function, c_1 and c_2 are positive auxiliary constants, and $i = 0, 1, ..., m - 1$. The initial position probed is $T[h'(k)]$; later positions probed are offset by amounts that depend in a quadratic manner on the probe number i. This method works much better than linear probing, but to make full use of the hash table, the values of c_1, c_2, and m are constrained. Also, if two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$. This property leads to a milder form of clustering, called secondary clustering. As in linear probing, the initial probe determines the entire sequence, and so only m distinct probe sequences are used.

Double Hashing

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. It uses a hash function of the form

$$h(k, i) = (h_1(k) + ih_2(k)) \mod m$$

where both h_1 and h_2 are auxiliary hash functions. The initial probe goes to position $T[h_1(k)]$; successive probe positions are offset from previous positions by the amount $h_2(k)$, modulo m. Thus, unlike the case of linear or quadratic probing, the probe sequence here depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary. Here's an example:

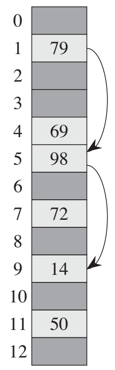


Figure 23.4: Double hashing. Here $m = 13$ with $h_1(k) = k \mod 13$ and $h_2(k) = 1 + (k \mod 11)$. Since $14 \equiv 1(\mod 13)$ and $14 \equiv 3(\mod 11)$, we insert the key 14 into empty slot 9, after examining slots 1 and 5 and finding them to be occupied.

Analysis of Open-Address Hashing

As in our analysis of chaining, we express our analysis of open addressing in terms of the load factor $\alpha = n/m$ of the hash table. Of course, with open addressing, at most one element occupies each slot, and thus $n \leq m$, which implies $\alpha \leq 1$.

Definition. We assume that we are using the uniform hashing assumption. In this idealized scheme, the probe sequence $\langle h(k,0), h(k,1), ..., h(k,m-1)\rangle$ used to insert or search for each key k is equally likely to be any of the $m!$ permutations of $\langle 0,1,...,m-1\rangle$. Of course, a given key has a unique fixed probe sequence associated with it; what we mean here is that, considering the probability distribution on the space of keys and the operation of the hash function on the keys, each possible probe sequence is equally likely.

We now analyze the expected number of probes for hashing with open addressing under the uniform hashing assumption, beginning with an analysis of the number of probes made in an unsuccessful search.

Theorem 3. *Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.*

Proof. Let X be the random variable that denotes the number of probes made in an unsuccessful search. We want to find $\mathbf{E}[X]$. From CIS 160, we know X is a non-negative integer random variable. Therefore,

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i]$$

. Let A_j be the event that the j^{th} probe is unsuccessful (i.e., the slot is full). Hence,

$$\begin{aligned}
\Pr[X \geq i] &= \Pr[A_1 \cap A_2 \cap ... \cap A_{i-1}] \\
&= \Pr[A_1]\Pr[A_2|A_1]\Pr[A_3|A_1 \cap A_2]\cdots\Pr[A_{i-1}|A_1 \cap A_2 \cap ... \cap A_{i-2}] \\
&= \frac{n}{m}\cdot\frac{n-1}{m-1}\cdot\frac{n-2}{m-2}\cdots\frac{n-i+2}{m-i+2} \\
&\leq \left(\frac{n}{m}\right)^{i-1} && \left(\text{Note that } \tfrac{n-k}{m-k} \leq \tfrac{n}{m}\right) \\
&= \alpha^{i-1}
\end{aligned}$$

Now, we plug that value back in, and find that:

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{i=1}^{\infty} \Pr[X \geq i] \\
&= \sum_{i=1}^{\infty} \alpha^{i-1} \\
&= \sum_{i=0}^{\infty} \alpha^{i} \\
&= \frac{1}{1-\alpha}
\end{aligned}$$

Alternate explanation: α is the fraction of the table that is full. $1-\alpha$ is the fraction of the table that is empty. Hence, the probability of finding an empty slot for a given probe is equal to $1-\alpha$. That is the success probability of a geometric distribution! X is a geometric random variable, and so we know that $\mathbf{E}[X] = 1/(1-\alpha)$. $\square$

Theorem 4. *Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most $\frac{1}{\alpha}\lg\frac{1}{1-\alpha}$, assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.*

Proof. Just as we did in our analysis for chaining, we will assume that $x_1, x_2, ..., x_n$ are elements inserted into T in that order. In expectation, the number of probes needed to insert any of the first $m/2$ elements is at most 2. So, in expectation, the total number of probes needed to insert the first $m/2$ elements is at most $2 \cdot \frac{m}{2} = m$.

Consider the insertion of the next $m/4$ elements $x_{m/2+1}, x_{m/2+2}, ..., x_{3m/4}$.

The number of probes needed (in expectation) to insert any of the next $m/4$ elements is at most 4. Hence, the total number of probes needed to insert $x_{m/2+1}, x_{m/2+2}, ..., x_{3m/4}$ is at most $4 \cdot \frac{m}{4} = m$.

Similarly, the total number of probes needed to insert the next $m/2^3 = m/8$ elements is at most $8 \cdot \frac{m}{8} = m$.

Thus, the total number of probes needed to insert the first $m/2$ elements, then the next $m/4$ elements, then the next $m/8$ elements, ..., then the next $m/2^i$ elements is at most $m \cdot i$.

Note that after inserting $m/2 + m/4 + m/8 + m/2^i$ elements in the table, the fraction of the table that is empty is equal to $1/2^i = 2^{-i}$.

Observe that $m \cdot i = -m \cdot \lg(2^{-i})$. After we insert $x_1, x_2, ..., x_n$, the fraction of the table that is empty is equal to $1-\alpha$. Therefore, the total number of probes needed to insert $x_1, x_2, ..., x_n$ is at most $-m \cdot \lg(2^{-i})$.

Hence, the average time to search for an element which we know exists in the table is at most $\frac{-m\lg(1-\alpha)}{n}$. This is equal to $-\frac{1}{\alpha}\lg(1-\alpha) = \frac{1}{\alpha}\lg\left(\frac{1}{1-\alpha}\right)$. [*] $\square$

[*] You can see a different proof for this same theorem in CLRS Ch. 11

Tries 24

24.1 Introduction

Here, we present string searching algorithms that preprocess the text. This approach is suitable for applications in which many queries are performed on a fixed text, so that the initial cost of preprocessing the text is compensated by a speedup in each subsequent query (for example, a website that offers pattern matching in Penn's *Almanac* or a search engine that offers Web pages containing the term *Penn*).

Definition. A trie (pronounced "try") is a tree-based data structure for storing strings in order to support fast pattern matching. The main application for tries is in information retrieval. Indeed, the name "trie" comes from the word "retrieval."

In an information retrieval application, we are given a collection S of strings, all defined using the same alphabet. The primary query operations that tries support are pattern and prefix matching. The latter operation involves being given a string X, and looking for all the strings in S that begin with X.

24.2 Standard Tries

Let S be a set of $|S|$ strings from alphabet Σ such that no string in S is a prefix of another string.

Definition. A standard trie for S is an ordered tree T with the following properties:

- Each node of T, except the root, is labeled with a character of Σ.
- The children of an internal node of T have distinct labels.
- T has $|S|$ leaves, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to a leaf v of T yields the string of S associated with v.

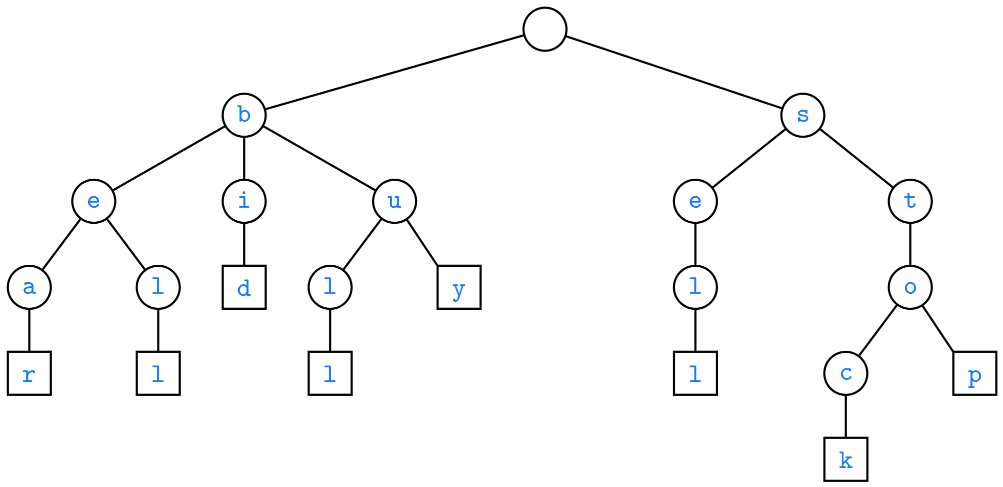


Figure 24.1: Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}

These notes were adapted from Goodrich and Tamassia's *Data Structures and Algorithms in Java, 4th edition* Chapter 13.3

Thus, a trie T represents the strings of S with paths from the root to the leaves of T. Note the importance of assuming that no string in S is a prefix of another string. This ensures that each string of S is uniquely associated with a leaf of T. (This is similar to the restriction for prefix codes with Huffman coding.)

We can always satisfy this assumption that no string in S is a prefix of another string by adding a special character (such as \$) that is not in the original alphabet Σ at the end of each string.

Note: this is an implementation detail that makes analysis simpler. When implementing tries, we *do* want to support words that are prefixes of other words, such as “pen” and “penguin”. There are two ways of accomplishing this: the special character way (e.g., adding a \$ or some other character not part of Σ at the end of each word), or by putting a special “end of word” marker (i.e., a field/variable) at the last character node of each word.

An internal node in a standard trie T can have anywhere between $1 \to |\Sigma|$ children. There is an edge going from the root r to one of its children for each character that is first in some string in the collection S. In addition, a path from the root of T to an internal node v at depth k corresponds to a k-character prefix. $X[0..k-1]$ of a string X of S. In fact, for each character c that can follow the prefix $X[0..k-1]$ in a string of the set S, there is a child of v labeled with character c. In this way, a trie concisely stores the common prefixes that exist among a set of strings.

Although it is possible that an internal node has up to Σ children, in practice the average degree of such nodes is likely to be much smaller. For example, the trie shown in the figure above has several internal nodes with only one child. On larger data sets, the average degree of nodes is likely to get smaller at greater depths of the tree, because there may be fewer strings sharing the common prefix, and thus fewer continuations of that pattern. Furthermore, in many languages, there will be character combinations that are unlikely to naturally occur.

The following proposition provides some important structural properties of a standard trie:

Proposition 1. *A standard trie storing a collection S of $|S|$ strings of total length n from an alphabet Σ has the following properties:*

- *The height of T is equal to the length of the longest string in S.*
- *T has $|S|$ leaves*
- *The number of nodes of T is at most $n+1$*

The worst case for the number of nodes of a trie occurs when no two strings share a common nonempty prefix; that is, except for the root, all internal nodes have one child.

A trie T for a set S of strings can be used to implement a set or map whose keys are the strings of S. Namely, we perform a search in T for a string X by tracing down from the root the path indicated by the characters in X. If this path can be traced and terminates at a leaf node (or if we use the alternative implementation, an “end of work” field), then we know X is a string in S.

For example, in the trie in the previous figure, tracing the path for “bull” ends up at a leaf, so $X \in S$. If the path cannot be traced or the path can be traced but terminates at an internal node, then X is not a string in S. The path for “bet” cannot be traced and the path for “be” ends at an internal node. Neither such word “be” or “bet” is in the set S.

It is easy to see that the running time of the search for a string of length m is $O(m \cdot |\Sigma|)$, because we visit at most $m+1$ nodes of T and we spend $O(|\Sigma|)$ time at each node determining the child having the subsequent character as a label. The $O(|\Sigma|)$ upper bound on the time to locate a child with a given label is achievable, even if the children of a node are unordered, since there are at most $|\Sigma|$ children. We can improve the time spent at a node to be $O(\log|\Sigma|)$ or expected $O(1)$, by mapping characters to children using a secondary search table or hash table at each node, or by using a direct lookup table of size $|\Sigma|$ at each node, if $|\Sigma|$ is sufficiently small (as is the case for DNA strings). For these reasons, we typically expect a search for a string of length m to run in $O(m)$ time.

From the discussion above, it follows that we can use a trie to perform a special type of pattern matching, called word matching, where we want to determine whether a given pattern matches one of the words of the text exactly. Word matching differs from standard pattern matching because the pattern cannot match an arbitrary substring of the text—only one of its words. To accomplish this, each word of the original document must be added to the trie. A simple extension of this scheme supports prefix-matching queries. However, arbitrary occurrences of the pattern in the text (for example, the pattern is a proper suffix of a word or spans two words) cannot be efficiently performed.

To construct a standard trie for a set S of strings, we can use an incremental algorithm that inserts the strings one at a time. Recall the assumption that no string of S is a prefix of another string. To insert a string X into the current trie T, we trace the path associated with X in T, creating a new chain of nodes to store the remaining characters of X when we get stuck. The running time to insert X with length m is similar to a search, with worst-case $O(m \cdot |\Sigma|)$ performance, or expected $O(m)$ if using secondary hash tables at each node. Thus, constructing the entire trie for set S takes expected $O(n)$ time, where n is the total length of the strings of S.

There is a potential space inefficiency in the standard trie that has prompted the development of the compressed trie, which is also known (for historical reasons) as the Patricia trie. Namely, there are potentially a lot of nodes in the standard trie that have only one child, and the existence of such nodes is a waste. We discuss the compressed trie next.

24.3 Compressed Tries

A compressed trie is similar to a standard trie but it ensures that each internal node in the trie has at least two children. It enforces this rule by compressing chains of single-child nodes into individual edges. (See figure below). Let T be a standard trie. We say that an internal node v of T is redundant if v has one child and is not the root. For example, the standard trie in the first figure of these notes has eight redundant nodes. Let us also say that a chain of $k \geq 2$ edges,

$$(v_0, v_1)(v_1, v_2) \cdots (v_{k-1}, v_k),$$

is redundant if:

- ▶ v_i is redundant for $i = 1, ..., k-1$
- ▶ v_0 and v_k are not redundant

We can transform T into a compressed trie by replacing each redundant chain $(v_0, v_1) \cdots (v_{k-1}, v_k)$ of $k \geq 2$ edges into a single edge (v_0, v_k), relabeling v_k with the concatenation of the labels of nodes $v_1, ..., v_k$.

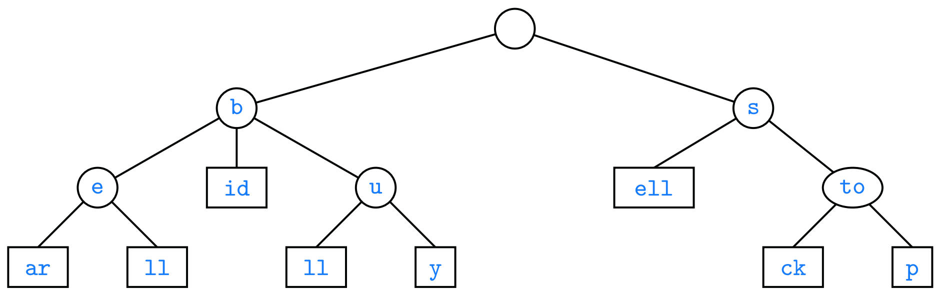


Figure 24.2: Compressed trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}. Notice that, in addition to compression at the leaves, the internal node with label "to" is shared by words "stock" and "stop".

Thus, nodes in a compressed trie are labeled with strings, which are substrings of strings in the collection, rather than with individual characters. The advantage of a compressed trie over a standard trie is that the number of nodes of the compressed trie is proportional to the number of strings and not to their total length, as shown in the following proposition (compare with Proposition 1).

Proposition 2. *A compressed trie storing a collection S of s strings from an alphabet of size d has the following properties:*

- *Every internal node of T has at least two children and most d children.*
- *T has s leaves nodes*
- *The number of nodes of T is $O(s)$*

The attentive reader may wonder whether the compression of paths provides any significant advantage, since it is offset by a corresponding expansion of the node labels. Indeed, a compressed trie is truly advantageous only when it is used as an auxiliary index structure over a collection of strings already stored in a primary structure, and is not required to actually store all the characters of the strings in the collection.

Suppose, for example, that the collection S of strings is an array of strings $S[0], S[1], ..., S[s-1]$. Instead of storing the label X of a node explicitly, we represent it implicitly by a combination of three integers (i, j, k), such that $X = S[i][j..k]$; that is, X is the substring of $S[i]$ consisting of the characters from the jth to the kth inclusive.

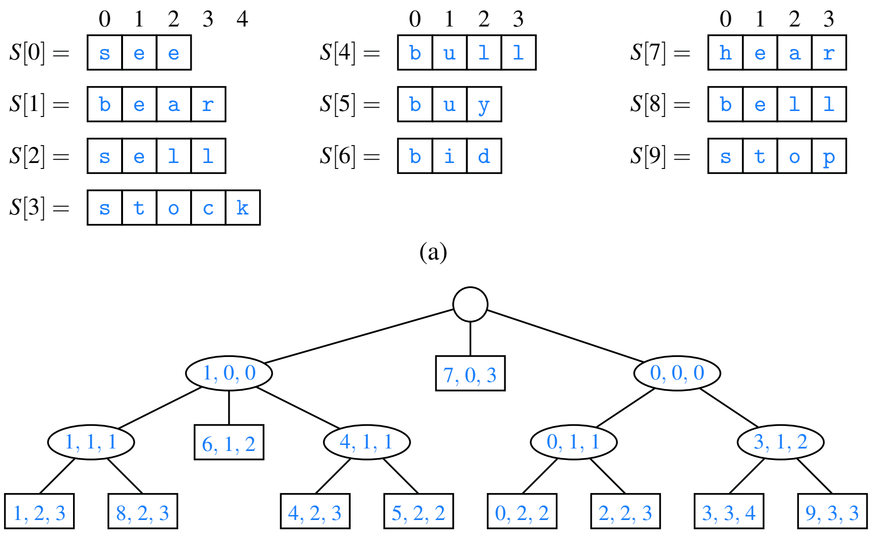


Figure 24.3: (a) Collection S of strings stored in an array. (b) Compact representation of the compressed trie for S.

This additional compression scheme allows us to reduce the total space for the trie itself from $O(n)$ for the standard trie to $O(s)$ for the compressed trie, where n is the total length of the strings in S and s is the number of strings in S. We must still store the different strings in S, of course, but we nevertheless reduce the space for the trie.

Searching in a compressed trie is not necessarily faster than in a standard tree, since there is still need to compare every character of the desired pattern with the potentially multicharacter labels while traversing paths in the trie.

24.4 Suffix Tries

One of the primary applications for tries is for the case when the strings in the collection S are all the suffixes of a string X. Such a trie is called the suffix trie (also known as a suffix tree or position tree) of string X. For example, the figure (a) below shows the suffix trie for the eight suffixes of string "minimize". For a suffix trie, the compact representation presented in the previous section can be further simplified. Namely, the label of each vertex is a pair "$j..k$" indicating the string $X[j..k]$. (See figure (b) below.) To satisfy the rule that no suffix of X is a prefix of another suffix, we can add a special character, denoted with \$, that is not in the original alphabet Σ at the end of X (and thus to every suffix). That is, if string X has length n, we build a trie for the set of n strings $X[j..n-1]\$$, for $j = 0, ..., n-1$.

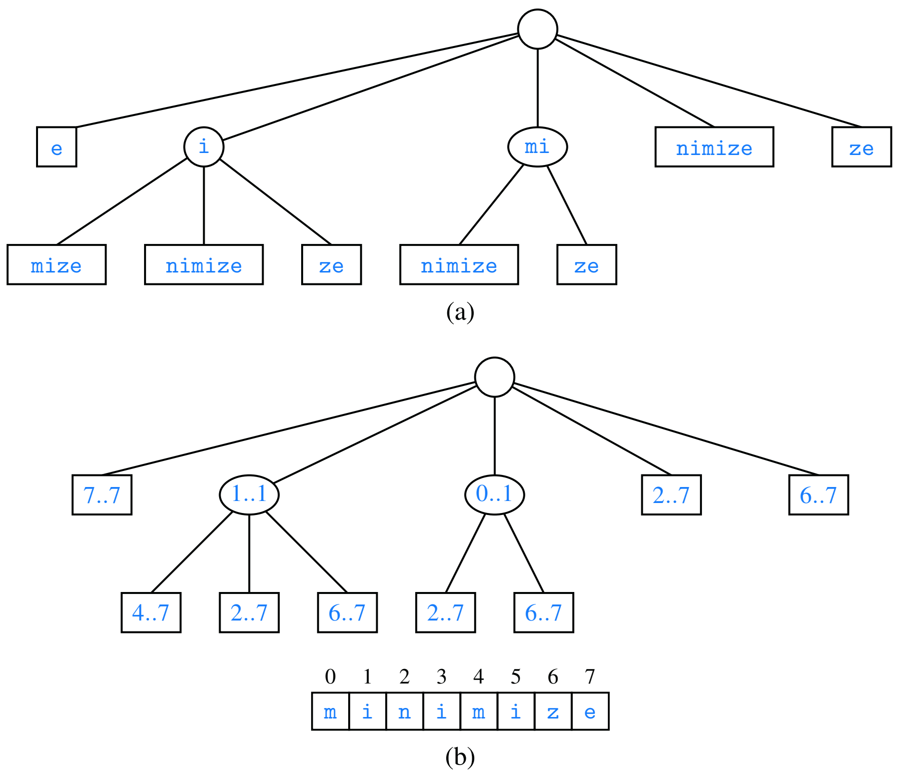


Figure 24.4: (a) Suffix trie T for the string X = "minimize". (b) Compact representation of T, where pair $j..k$ denotes the substring $X[j..k]$ in the reference string.

Saving Space

Using a suffix trie allows us to save space over a standard trie by using several space compression techniques, including those used for the compressed trie.

The advantage of the compact representation of tries now becomes apparent for suffix tries. Since the total length of the suffixes of a string X of length n is

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

storing all the suffixes of X explicitly would take $O(n^2)$ space. Even so, the suffix trie represents these strings implicitly in $O(n)$ space, as formally stated in the following proposition.

Proposition 3. *The compact representation of a suffix trie T for a string X of length n uses $O(n)$ space.*

Construction

We can construct the suffix trie for a string of length n with an incremental algorithm like the one presented earlier for standard tries. This construction takes $O(|\Sigma| \cdot n^2)$ time because the total length of the suffixes is quadratic in n. However, the (compact) suffix trie for a string of length n can be constructed in $O(n)$ time with a specialized algorithm, different from the one for general tries. This linear-time construction algorithm is fairly complex, however, and is not covered here. You can find a description of Ukkonen's Algorithm on the internet if you'd like – it is beyond the scope of this course. Still, we can take advantage of the existence of this fast construction algorithm when we want to use a suffix trie to solve other problems.

Using a Suffix Trie

The suffix trie T for a string X can be used to efficiently perform pattern-matching queries on text X. Namely, we can determine whether a pattern is a substring of X by trying to trace a path associated with P in T. P is a substring of X if and only if such a path can be traced. The search down the trie T assumes that nodes in T store some additional information, with respect to the compact representation of the suffix trie:

> If node v has label $j..k$ and Y is the string of length y associated with the path from the root to v (included), then $X[k - y + 1..k] = Y$.

This property ensures that we can compute the start index of the pattern in the text when a match occurs in $O(m)$ time.

Balanced BSTs: AVL Trees 25

25.1 Review: Binary Search Tree

A binary search tree is organized, as the name suggests, in a binary tree. The keys in a binary search tree are always stored in such a way as to satisfy the *binary search tree property.*

Definition. The binary search tree property is as follows: Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

Please review your CIS 120 notes for tree traversals, and how INSERT, SEARCH, and DELETE work.

All of the basic operations on a binary search trees run in $O(h)$ time, where h is the height of the tree. The height of a binary search tree varies, however, as items are inserted and deleted. If, for example, the n items are inserted in strictly increasing order, the tree will be a chain with height $n-1$, making all operations on that tree $O(n)$.

Therefore, our goal is to make the tree's height $O(\lg n)$ so that basic operations on a BST take worst case $O(\lg n)$ time.

25.2 Definition of an AVL Tree

The simple correction to the worst-case linear runtime for basic BST operations is to add a rule to the binary search tree definition that maintains a logarithmic height for the tree. The rule we consider in this section is the following *height-balance property*, which characterizes the structure of a binary search tree T in terms of the heights of its internal nodes (recall that the height of a node v in a tree is the length of the longest path from v to an external node).

Definition. Height balance property: For every internal node v of T, the heights of the children of v differ by at most 1.

Any binary search tree T that satisfies the height-balance property is said to be an AVL tree, named after the initials of its inventors Adel'son-Vel'skii and Landis.

These notes were adapted from Goodrich and Tamassia's *Data Structures and Algorithms in Java, 4th edition* Chapter 10.2

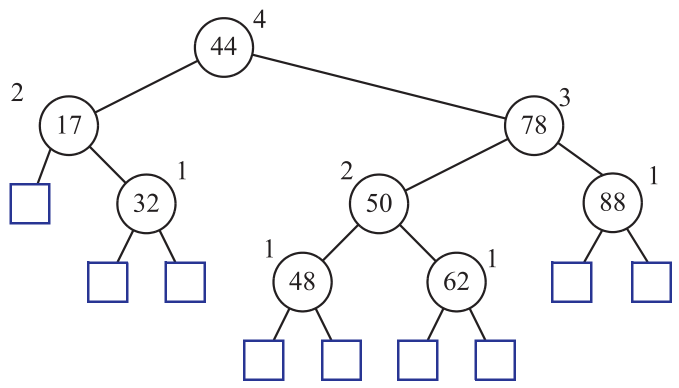


Figure 25.1: An example of an AVL tree. The keys of the entries are shown inside the nodes, and the heights of the nodes are shown next to the nodes.

An immediate consequence of the height-balance property is that a subtree of an AVL tree is itself an AVL tree. The height-balance property has also the important consequence of keeping the height small, as shown in the following proposition.

Proposition 1. *The height of an AVL tree storing n entries is $O(\lg n)$.*

Proof. Instead of trying to find an upper bound on the height of an AVL tree directly, it turns out to be easier to work on the "inverse problem" of finding a lower bound on the minimum number of internal nodes $n(h)$ of an AVL tree with height h. We show that $n(h)$ grows at least exponentially. From this, it is an easy step to derive that the height of an AVL tree storing n entries is $O(\lg n)$.

To start with, notice that $n(1) = 1$ and $n(2) = 2$, because an AVL tree of height 1 must have at least one internal node and an AVL tree of height 2 must have at least two internal nodes. Now, for $h \geq 3$, an AVL tree with height h and the minimum number of nodes is such that both its subtrees are AVL trees with the minimum number of nodes: one with height $h - 1$ and the other with height $h - 2$. Taking the root into account, we obtain the following formula that relates $n(h)$ to $n(h - 1)$ and $n(h - 2)$, for $h \geq 3$:

$$n(h) = 1 + n(h - 1) + n(h - 2)$$

You may see that this recurrence relation is that of the Fibonacci sequence, and can see that $n(h)$ is indeed exponential. If you aren't familiar with that explanation, we'll prove it below.

The recurrence relation implies that $n(h)$ is a strictly increasing function of h. Thus, we know that $n(h - 1) > n(h - 2)$. Replacing $n(h - 1)$ with $n(h - 2)$ in the recurrence relation and dropping the 1, we get, for $h \geq 3$,

$$n(h) > 2 \cdot n(h - 2)$$

This implies that $n(h)$ at least doubles each time h increases by 2, which intuitively means that $n(h)$ grows exponentially. To show this fact in a formal way, we apply this formula repeatedly, yielding the inequalities:

$$\begin{aligned} n(h) &> 2 \cdot n(h - 2) \\ &> 4 \cdot n(h - 4) \\ &> 8 \cdot n(h - 6) \\ &\dots \\ &> 2^i \cdot n(h - 2i) \end{aligned}$$

That is, $n(h) > 2^i \cdot n(h-2i)$, for any integer i, such that $h - 2i \geq 1$. Since we already know the values of $n(1)$ and $n(2)$, we pick i so that $h - 2i = 1$ or $h - 2i = 2$. That is, we pick

$$i = \left\lceil \frac{h}{2} \right\rceil - 1$$

By substituting the above value of i into the expanded inequality, we get for $h \geq 3$:

$$\begin{aligned} n(h) &> 2^{\lceil \frac{h}{2} \rceil - 1} \cdot n\left(h - 2\left\lceil \frac{h}{2} \right\rceil + 2\right) \\ &\geq 2^{\lceil \frac{h}{2} \rceil - 1} \cdot n(1) \\ &> 2^{\frac{h}{2} - 1} && \text{(since } n(1) = 1\text{)} \end{aligned}$$

By taking logarithms of both sides, we get

$$\lg n(h) > \frac{h}{2} - 1$$

from which we get

$$h < 2 \lg n(h) + 2$$

which implies that an AVL tree storing n entries has height at most $2 \lg n + 2$. □

With this knowledge, we can see that the SEARCH operation runs in $O(\lg n)$ time. Of course, we still have to show how to maintain the height-balance property after an insertion or removal.

25.3 Update Operations: Insertion and Deletion

The insertion and deletion operations for AVL trees are similar to those for binary search trees, but with AVL trees we must perform additional computations.

Insertion

An insertion in an AVL tree T begins as in an INSERT operation for a binary search tree. Recall that this operation always inserts the new entry at a node w in T that was previously an external node, and it makes w become an internal node. That is, it adds two external node children to w. This action may violate the height-balance property, however, for some nodes increase their heights by one. In particular, node w, and possibly some of its ancestors, increase their heights by one. Therefore, let us describe how to restructure T to restore its height balance.

Definition. Given a binary search tree T, we say that an internal node v of T is balanced if the absolute value of the difference between the heights of the children of v is at most 1, and we say that it is unbalanced otherwise. Thus, the height-balance property characterizing AVL trees is equivalent to saying that every internal node is balanced.

Suppose that T satisfies the height-balance property, and hence is an AVL tree, prior to our inserting the new entry. As we have mentioned, after performing the INSERT operation on T, the heights of some nodes of T, including w, increase. All such nodes are on the path of T from w to the root of T, and these are the only nodes of T that may have just become unbalanced. See the figure (a) below. Of course, if this happens, then T is no longer an AVL tree; hence, we need a mechanism to fix the "unbalance" that we have just caused.

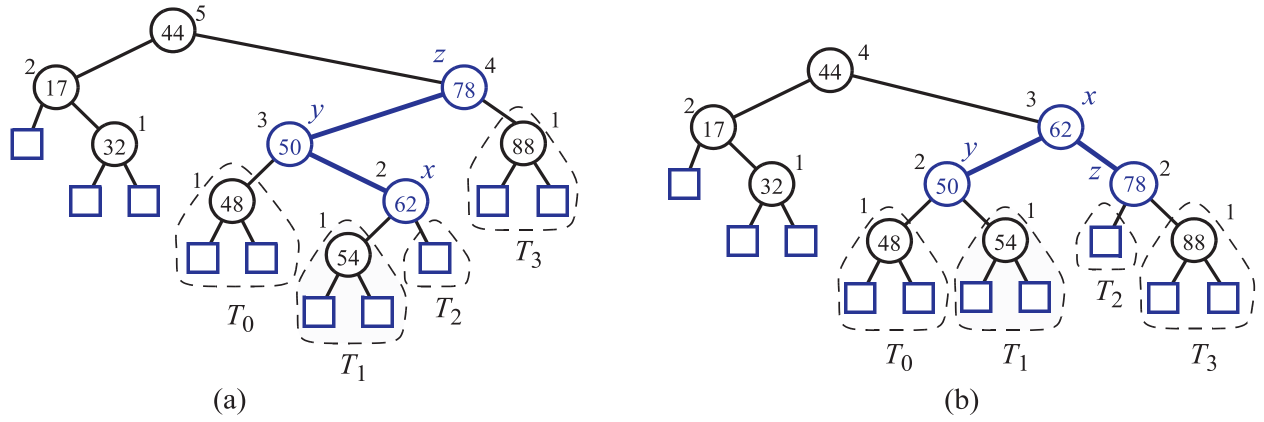


Figure 25.2: An example insertion of an entry with key 54 in the AVL tree of Figure 1: (a) after adding a new node for key 54, the nodes storing keys 78 and 44 become unbalanced; (b) a trinode restructuring restores the height-balance property. We show the heights of nodes next to them, and we identify the nodes x, y, and z participating in the trinode restructuring.

We restore the balance of the nodes in the binary search tree T by a simple "search-and-repair" strategy. In particular, let z be the first node we encounter in going up from w toward the root of T such that z is unbalanced. (See figure (a) above.) Also, let y denote the child of z with higher height (and note that node y must be an ancestor of w). Finally, let x be the child of y with higher height (there cannot be a tie and node x must be an ancestor of w). Also, node x is a grandchild of z and could be equal to w. Since z became unbalanced because of an insertion in the subtree rooted at its child y, the height of y is 2 greater than its sibling.

We now rebalance the subtree rooted at z by calling the trinode restructuring function*, RESTRUCTURE(x) as shown in the figures above and below. A trinode restructuring temporarily renames the nodes x, y, and z as a, b, and c, so that a precedes b and b precedes c in an inorder traversal of T. There are four possible ways of mapping x, y, and z to a, b, and c, as shown in the figure below, which are unified into one case by our relabeling. The trinode restructuring then replaces z with the node called b, makes the children of this node be a and c, and makes the children of a and c be the four previous children of x, y, and z (other than x and y) while maintaining the inorder relationships of all the nodes in T.

The modification of a tree T caused by a trinode restructuring operation is often called a rotation, because of the geometric way we can visualize the way it changes T. If $b = y$, the trinode restructuring method is called a single rotation, for it can be visualized as "rotating" y over z. (See figure (a) and (b).) Otherwise, if $b = x$, the trinode restructuring operation is called a double rotation, for it can be visualized as first "rotating" x over y and then over z. (See figure (c) and (d).) Some treat these two kinds of rotations as separate methods, each with two symmetric types. We have chosen, however, to unify these four types of rotations into a single trinode restructuring operation. No matter how we view it, though, the trinode restructuring method modifies parent-child relationships of $O(1)$ nodes in T, while preserving the inorder traversal ordering of all the nodes in T.

* Pseudocode is in the pages that follow.

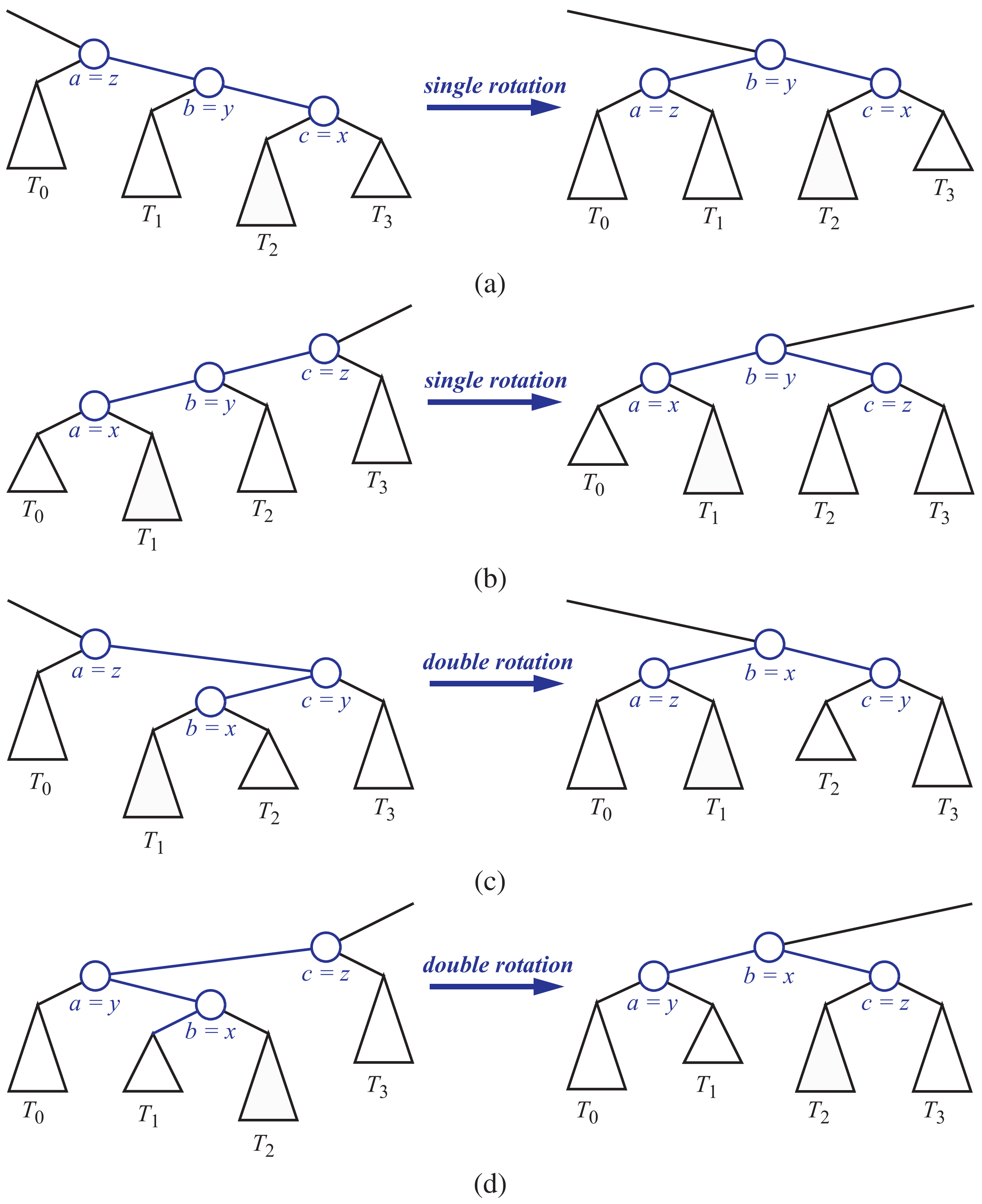


Figure 25.3: Schematic illustration of a trinode restructuring operation: (a) and (b) a single rotation; (c) and (d) a double rotation.

RESTRUCTURE(x) – **Trinode Restructuring Algorithm**

Input: A node x of a binary search tree T that has both a parent y and a grandparent z.

Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving nodes x, y, and z

```
1. Let $(a,b,c)$ be a left-to-right (inorder) listing of the nodes $x$, $y$, and
   $z$, and let $(T_0,T_1,T_2,T_3)$ be a left-to-right (inorder) listing of the
   four subtrees of $x$, $y$, and $z$ not rooted at $x$, $y$, or $z$.

2. Replace the subtree rooted at $z$ with a new subtree rooted at $b$.

3. Let $a$ be the left child of $b$ and let $T_0$ and $T_1$ be the left and right
   subtrees of $a$, respectively.

4. Let $c$ be the right child of $b$ and let $T_2$ and $T_3$ be the left and right
   subtrees of $c$, respectively.
```

In addition to its order-preserving property, a trinode restructuring changes the heights of several nodes in T, so as to restore balance. Recall that we execute the function RESTRUCTURE(x) because z, the grandparent of x, is unbalanced. Moreover, this unbalance is due to one of the children of x now having too large a height relative to the height of z's other child. As a result of a rotation, we move up the "tall" child of x while pushing down the "short" child of z. Thus, after performing RESTRUCTURE(x), all the nodes in the subtree now rooted at the node we called b are balanced. (See figure 3.) Thus, we restore the height-balance property locally at the nodes x, y, and z. In addition, since after performing the new entry insertion the subtree rooted at b replaces the one formerly rooted at z, which was taller by one unit, all the ancestors of z that were formerly unbalanced become balanced. (See figure 2) Therefore, this one restructuring also restores the height-balance property globally.

Deletion

As was the case for the INSERT operation, we begin the implementation of the DELETE operation on an AVL tree T by using the algorithm for performing this operation on a regular binary search tree. The added difficulty in using this approach with an AVL tree is that it may violate the height-balance property. In particular, after removing an internal node and elevating one of its children into its place, there may be an unbalanced node in T on the path from the parent w of the previously removed node to the root of T. (See figure (a) below.) In fact, there can be one such unbalanced node at most.

As with insertion, we use trinode restructuring to restore balance in the tree T. In particular, let z be the first unbalanced node encountered going up from w toward the root of T. Also, let y be the child of z with larger height (note that node y is the child of z that is not an ancestor of w), and let x be the child of y defined as follows: if one of the children of y is taller than the other, let x be the taller child of y; else (both children of y have the same height), let x be the child of y on the same side as y (that is, if y is a left child, let x be the left child of y, else let x be the right child of y). In any case, we then perform a RESTRUCTURE(x) operation, which restores the height-balance property locally, at the subtree that was formerly rooted at z and is now rooted at the node we temporarily called b. (See figure (b) below.)

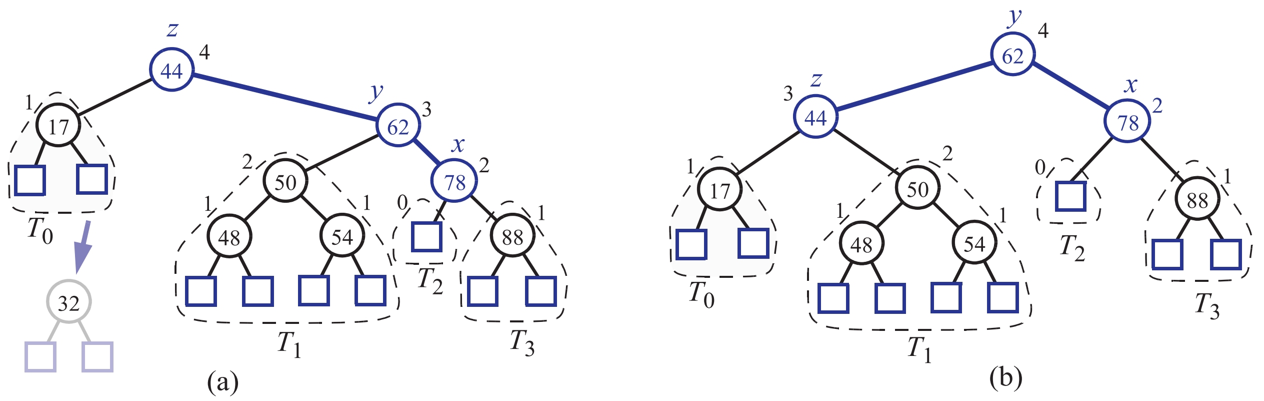


Figure 25.4: Removal of the entry with key 32 from the AVL tree of Figure 1: (a) after removing the node storing key 32, the root becomes unbalanced; (b) a (single) rotation restores the height-balance property.

Unfortunately, this trinode restructuring may reduce the height of the subtree rooted at b by 1, which may cause an ancestor of b to become unbalanced. So, after rebalancing z, we continue walking up T looking for unbalanced nodes. If we find another, we perform a restructure operation to restore its balance, and continue marching up T looking for more, all the way to the root. Still, since the height of T is $O(\lg n)$, where n is the number of entries, $O(\lg n)$ trinode restructurings are sufficient to restore the height-balance property.

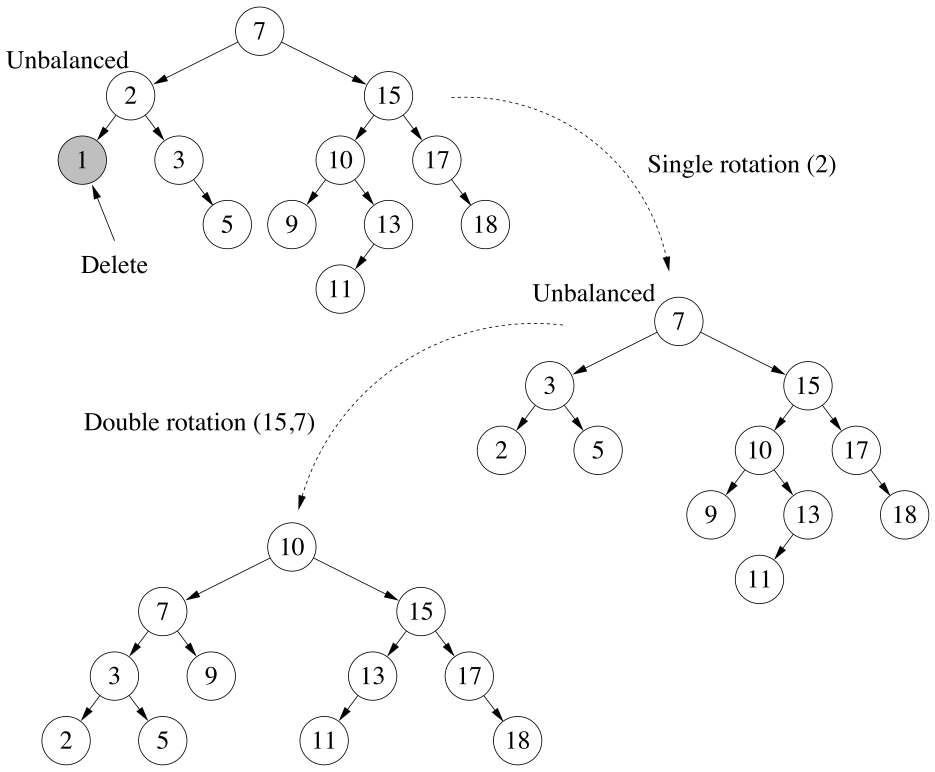


Figure 25.5: Additional AVL deletion example.

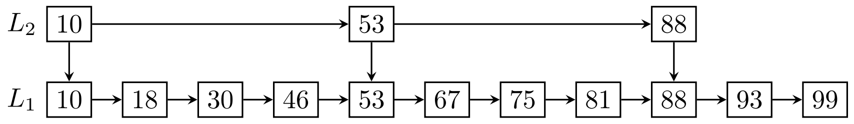
Advanced Topics

Skip Lists 26

26.1 Skip Lists

A skip list[*] (due to Bill Pugh in 1990) is a randomized data structure that can be thought of as a generalization of sorted linked lists. They have most of the desirable properties of balanced binary search trees and the simplicity of linked lists. Skip lists support the standard operations for managing a dynamic data set – INSERT, DELETE, and SEARCH.

In a sorted linked list with n elements, searching an element takes $\Theta(n)$ time. What if we had two-level sorted linked lists structure as shown in the figure below? The bottom list L_1 contains all the elements and another list L_2 contains only a subset of elements of L_1.



Note that to minimize the worst case search time, the nodes in L_2 should be evenly distributed, i.e., the number of nodes in L_1 that are "skipped" between any two consecutive nodes in L_2 should be the same. The search time in such a 2-level structure is given by

$$|L_2| + \frac{|L_1|}{|L_2|} = |L_2| + \frac{n}{|L_2|}$$

To minimize the search time, the two terms in the above expression must be equal. Thus solving

$$|L_2| = \frac{n}{|L_2|}$$

we get $|L_2| = \sqrt{n}$ and hence the total search time is at most $2\sqrt{n}$. Generalizing it a step further, consider a 3-level structure – list L_1 containing all the elements, list L_2 containing elements that are evenly distributed over elements in L_1, and list L_3 containing elements that are evenly distributed over elements in L_2. The search time in a 3-level structure is given by

$$|L_3| + \frac{|L_2|}{|L_3|} + \frac{|L_1|}{|L_2|} = |L_3| + \frac{|L_2|}{|L_3|} + \frac{n}{|L_2|}$$

The above expression is minimized when the three terms are equal. If we solve the following two equations:

$$|L_3| = \frac{|L_2|}{|L_3|} \qquad \text{and} \qquad \frac{|L_2|}{|L_3|} = \frac{n}{|L_2|},$$

we get that $|L_3| = \sqrt[3]{n}$ and hence the search time is at most $3\sqrt[3]{n}$. Generalizing this to a k-level structure, we get the total search time to be at most $k\sqrt[k]{n}$. Setting $k = \lg n$, we get the search time to be at most

$$\lg n (n^{\frac{1}{\lg n}}) = \lg n (2^{\lg n})^{\frac{1}{\lg n}} = 2 \lg n$$

[*] You can read the full research paper here: https://epaperpress.com/sortsearch/download/skiplist.pdf

Thus, in a skip list with $\lg n$ levels, the bottom list L_1 contains all of the n elements, L_2 contains $n/2$ elements (every other element of L_1), L_3 contains $n/4$ elements, and so on. Note that this structure is like a perfectly balanced binary search tree. The insertion or deletion of a node could disrupt this structure, however just as in AVL trees, we do not need a perfectly balanced structure – a little imbalance still allows for fast search time. In skip lists the balancing is done using randomization – for each element in L_i we toss a fair coin to decide whether the element should also belong to L_{i+1} or not. In expectation, we would expect half the elements to also be part of L_{i+1}. Note that the randomization in constructing this data structure does not arise because of any assumption on the distribution of the keys. The randomization only depends on a random number generator. Thus an adversary cannot pick a "bad" input for this data structure, i.e., a sequence of keys that will result in poor performance of this data structure.

The three operations Search, Insert, and Delete can be implemented as follows. Let ℓ be the largest index of a linked list. We add a sentinel nodes $-\infty$ to each list.

The pseudo-code for the Search operation is as follows.

```
Search($x$):
  $v_\ell \leftarrow$ element with key $-\infty$ in $L_\ell$.
  for $i \leftarrow \ell$ down to 2 do
      Follow the down-link from $v_i$ to $v_{i-1}$.
      Follow the right-links starting from $v_{i-1}$ until the key of an element is larger than $x$.
      $v_{i-1} \leftarrow$ largest element in $L_{i-1}$ that is at most $x$.
  return $v_1$
```

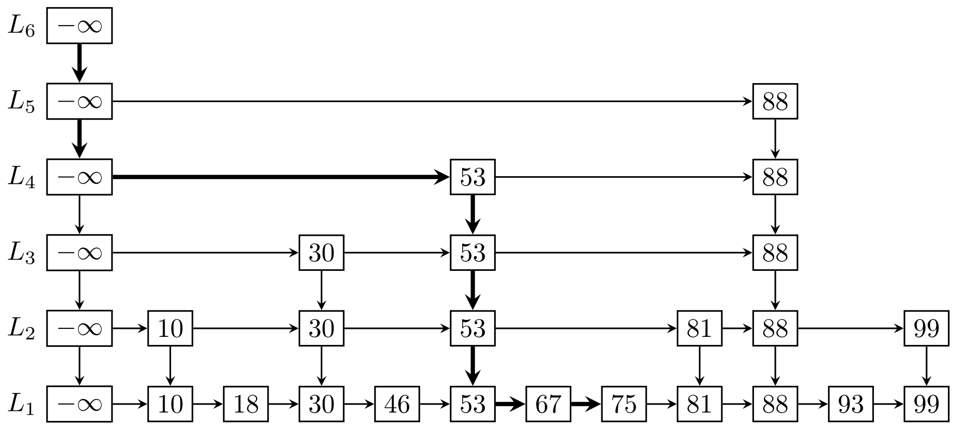
The pseudo-code for the Insert operation is given below.

```
Insert($x$):
  if Search($x$) $= x$ then
      return
  $(v_\ell, v_{\ell-1}, \ldots, v_1) \leftarrow$ elements in $L_\ell, L_{\ell-1}, \ldots, L_1$ with key values at most $x$
  Flip a fair coin until we get tails. Let $f$ be the number of coin flips.
  for $i \leftarrow 1$ to $\min(\ell, f)$ do
      Insert $x$ in $L_i$
  if $f > \ell$ then
      Create lists $L_{\ell+1}, L_{\ell+2}, \ldots, L_f$.
      Add elements $\{-\infty, x\}$ to $L_\ell, L_{\ell+1}, \ldots, L_f$.
      Create $L_{f+1}$ containing $-\infty$.
  $\ell \leftarrow \max(\ell, f+1)$
```

The pseudo-code for the Delete operation is as follows.

```
Delete($x$):
  if Search($x$) $\neq x$ then
      return
  $(v_\ell, v_{\ell-1}, \ldots, v_1) \leftarrow$ elements in $L_\ell, L_{\ell-1}, \ldots, L_1$ with key values at most $x$
  for $i \leftarrow 1$ to $\ell$ do
      if $v_i = x$ then
          Delete $x$ in $L_i$
  while $\ell \geq 2$ and $L_{\ell-1}$ contains only the element $-\infty$ do
      $\ell \leftarrow \ell - 1$
```

An example of a skip list on the set $\{30, 81, 10, 46, 53, 75, 99, 88, 93, 67, 18\}$ is as shown in the figure below. The search path for element with key 75 is highlighted.



26.2 Analysis

In the analysis below we will show that some events happen with a high probability (w.h.p). This means that the probability with which the event occurs is at least $1 - \frac{1}{n^c}$, for some constant $c \geq 1$.

Let $x_1, x_2, \ldots, x_n$ be the elements inserted into the skip list. Let H_i denote the number of lists that element x_i belongs to. Note that H_i is exactly equal to the total number of flips of a fair coin until the first "success". This is a geometric random variable with parameter 1/2 and hence $\mathbf{E}[H_i] = 2$. Let $H = \max\{H_1, H_2, \ldots, H_n\}$.

Theorem 1. *The space requirement for a skip list with n elements is $O(n)$ in expectation.*

Proof. Let S be the random variable denoting the total amount of space required by a skip list with n elements. Taking into account the space overhead of the sentinel keys ($-\infty$), we have

$$\begin{aligned} S &= O\left(\sum_{i=1}^{n} H_i\right) \\ \mathbf{E}[S] &= O\left(\sum_{i=1}^{n} \mathbf{E}[H_i]\right) \\ &= O(n) \end{aligned}$$

□

Lemma 1. $\Pr[H \geq h] \leq \frac{n}{2^{h-1}}$

Proof. We have

$$\begin{aligned}
\Pr[H \geq h] &= \Pr[\bigcup_{i=1}^{n} H_i \geq h] \\
&\leq \sum_{i=1}^{n} \Pr[H_i \geq h] && \text{(using the union-bound)} \\
&= \frac{n}{2^{h-1}}
\end{aligned}$$

□

Lemma 2. *The expected number of levels in a skip list with n elements is at most $\lceil \lg n \rceil + 3$.*

Proof. The number of levels in a skiplist of n elements is $H + 1$. The second term is due to the list with the highest index that contains only the key $-\infty$. Since H is a non-negative random variable, we have

$$\begin{aligned}
\mathbf{E}[H] &= \sum_{h=1}^{\infty} \Pr[H \geq h] \\
&= \sum_{h=1}^{\lceil \lg n \rceil} \Pr[H \geq h] + \sum_{h \geq \lceil \lg n \rceil + 1} \Pr[H \geq h] \\
&\leq \sum_{h=1}^{\lceil \lg n \rceil} 1 + \sum_{h \geq \lceil \lg n \rceil + 1} \frac{n}{2^{h-1}} \\
&\leq \lceil \lg n \rceil + \sum_{h \geq \lg n + 1} \frac{n}{2^{h-1}} \\
&= \lceil \lg n \rceil + \frac{n}{2^{\lg n}} \sum_{i=0}^{\infty} \frac{1}{2^i} \\
&= \lceil \lg n \rceil + 2
\end{aligned}$$

Thus the expected number of levels is $\mathbf{E}[H] + 1 \leq \lceil \lg n \rceil + 3$. □

Lemma 3. *With a high probability, a skip list with n elements has $O(\log n)$ levels.*

Proof. We will show that w.h.p, the number of levels in a skip list with n elements is at most $2 \lg n + 1$. This is equivalent to showing that w.h.p, $H \leq 2 \lg n$. From Lemma 1, we have

$$\begin{aligned}
\Pr[H \geq 2 \lg n + 1] &\leq \frac{n}{2^{2 \lg n + 1 - 1}} \\
&= \frac{1}{n}
\end{aligned}$$

□

Lemma 4. *For any r such that $0 < r \le n$,*

$$\sum_{i=0}^{r} \binom{n}{i} \le \left(\frac{ne}{r}\right)^r$$

Proof. Rewriting the left hand side in a more convenient form we have

$$\begin{aligned}
\sum_{i=0}^{r} \binom{n}{i} &= \left(\frac{n}{r}\right)^r \sum_{i=0}^{r} \binom{n}{i} \left(\frac{r}{n}\right)^r \\
&\le \left(\frac{n}{r}\right)^r \sum_{i=0}^{r} \binom{n}{i} \left(\frac{r}{n}\right)^i \\
&\le \left(\frac{n}{r}\right)^r \sum_{i=0}^{n} \binom{n}{i} \left(\frac{r}{n}\right)^i \\
&= \left(\frac{n}{r}\right)^r \left(1 + \frac{r}{n}\right)^n && \text{(using the Binomial Theorem)} \\
&\le \left(\frac{n}{r}\right)^r e^{\left(\frac{r}{n}\right)n} && \text{(using } 1 + x \le e^x \text{ for all } x\text{)} \\
&= \left(\frac{ne}{r}\right)^r
\end{aligned}$$

□

Theorem 2. *With a high probability, in a skip list of n elements, every search takes $O(\log n)$ time.*

Proof. When searching an element with key x, we begin at the first element in the top list and then move down to the next lower indexed list or move right along the same list until we reach the element v_1 (the element with the largest key in L_1 that is at most x). We want to bound the total number of steps ("down" steps plus "right" steps) with a high probability. Instead of counting the total number of down steps plus the total number of right steps, we will start at the element v_1 and trace the search path in reverse, i.e., either go "left" or go "up". Note that we go left at key x_j in list L_i because x_j was not promoted to the next higher list L_{i+1}, i.e., the ith flip of the coin must have resulted in tails when inserting x_j in the skip list. From Lemma 3, we know that w.h.p., the number of up moves are at most $2 \lg n$. We will show that the total number of up moves plus the total number of left moves (total number of coin flips) is at most $20 \lg n$ with a high probability.

We consider the following events:

E: event that the total number of coin flips is $20 \lg n$ and our search has not ended.
U: event that the total number of up-moves (heads) is greater than $2 \lg n$.

We want to upper bound $\Pr[E]$. By Lemma 3 we know that $\Pr[U] = \Pr[H > 2 \lg n] \le 1/n$.

$$\begin{aligned}
\Pr[E] &= \Pr[E \cap U] + \Pr[E \cap \overline{U}] \\
&\le \Pr[U] + \Pr[E] \Pr[\overline{U} \mid E] \\
&\le \frac{1}{n} + \Pr[\overline{U} \mid E]
\end{aligned}$$

Let X be the random variable denoting the number of heads.

$$\begin{aligned}
\Pr[\overline{U} \mid E] &= \sum_{i=0}^{2\lg n} \Pr[X = i] \\
&= \sum_{i=0}^{2\lg n} \binom{20\lg n}{i} \left(\frac{1}{2}\right)^{i} \left(\frac{1}{2}\right)^{20\lg n - i} \\
&= \left(\frac{1}{2}\right)^{20\lg n} \sum_{i=0}^{2\lg n} \binom{20\lg n}{i} \\
&\le \left(\frac{1}{2}\right)^{20\lg n} \left(\frac{e \cdot 20\lg n}{2\lg n}\right)^{2\lg n} && \text{(using Lemma 4)} \\
&= \left(\frac{1}{2}\right)^{20\lg n} (10e)^{2\lg n} \\
&= 2^{\lg(10e)\cdot 2\lg n} \cdot 2^{-20\lg n} \\
&= 2^{2\lg n(\lg(10e) - 10)} \\
&\le \frac{1}{2^{2\lg n}} \\
&= \frac{1}{n^2}
\end{aligned}$$

Plugging this bound in to the $\Pr[E] = \frac{1}{n} + \Pr[\overline{U} \mid E]$ equation we found earlier, we get

$$\begin{aligned}
\Pr[E] &\le \frac{1}{n} + \frac{1}{n^2} \\
&\le \frac{1}{n} + \frac{1}{n} \\
&= \frac{2}{n}
\end{aligned}$$

□

Bloom Filters 27

27.1 Bloom Filters

Bloom Filter, introduced by Bloom in 1970, is a space-efficient randomized data structure for representing a set in order to support membership queries. The set membership problem is as follows. We have a very large $\mathcal{U}$, with $|\mathcal{U}| = u$. Let $S = \{x_1, x_2, \dots, x_n\}$ be a subset of $\mathcal{U}$ such that $u >> n$. We want a data structure to maintain S that supports membership queries: "Given $x \in \mathcal{U}$, is $x \in S$?" The supported operations are

- $\textsc{Insert}(x)$: $S \leftarrow S \cup \{x\}$
- $\textsc{Query}(x)$: is $x \in S$?

Initial Idea: We will use a bit vector B of $m = 2n$ bits. Let h be a hash function such that maps each element of $\mathcal{U}$ to a random number uniform over the range $\{1, 2, \dots, m\}$. The above functions are implemented as follows.

```
Insert($x$):
$B[h(x)] \leftarrow 1$
```

```
Query($x$):
return $B[h(x)]$
```

Clearly, both operations can be performed in constant time. Note that if $x \in S$ then $\Pr[\textsc{Query}(x)] = 1$ is 1, i.e., our algorithm always gives the correct answer. If $x \notin S$ then we may get a *false positive* as shown below.

$$\Pr[\text{we output that } x \in S] = \Pr[\text{there is a } y \text{ s.t. } h[y] = h[x]] \leq \sum_{y \in S} \Pr[h(y) = h(x)] = \frac{n}{m} \leq \frac{1}{2}$$

Reducing the probability of error. Suppose we want to reduce the probability of error to ϵ, that is, when $x \notin S$, we want $\Pr[\textsc{Query}(x)] = 1] \leq \epsilon$. To achieve this we will have $k > 1$ tables, each of size $2n$ and each with its own hash function. The two operations now can be implemented as follows.

```
Insert($x$):
for $i \leftarrow 1$ to $k$ do
    $B_i[h_i(x)] \leftarrow 1$
```

```
Query($x$):
for $i \leftarrow 1$ to $k$ do
    if $B_i[h_i(x)] = 0$ then
        return 0
return 1
```

Note that if $x \in S$ then $\Pr[\textsc{Query}(x)] = 1$ is 1, i.e., our algorithm always gives the correct answer. If $x \notin S$ then

$$\begin{aligned}
\Pr[\text{ we output that } x \in S\ ] &= \Pr[\ \forall 1 \leq i \leq k, \text{there is a } y \text{ s.t. } h_i[y] = h_i[x]\ ] \\
&\leq \prod_{i=1}^{k} \left[\sum_{y \in S} \Pr[h_i(y) = h_i(x)] \right] \\
&= \left(\frac{n}{m} \right)^k \\
&\leq \left(\frac{1}{2} \right)^k
\end{aligned}$$

Since we want the error to be at most ϵ, solving for $\left(\frac{1}{2}\right)^k \leq \epsilon$ yields $k \geq \lg(1/\epsilon)$. Thus the time for each operation is $O(\lg(1/\epsilon))$ and the amount of space is $O(n \lg(1/\epsilon))$.

Sometimes Bloom filters are described slightly differently: instead of having different hash tables, there is one array of size m that is shared among all k hash functions.

Balanced BSTs: Red-Black Trees 28

Similar to AVL trees, our goal is to design a self-balancing BST so that basic operations on the tree take worst case $O(\lg n)$ time. Here, we will look an introductory look at a special type of BBSTs: red-black trees.

28.1 Properties of Red-Black Trees

The red-black tree is a binary search tree with one modification: each node has a color, either RED or BLACK, using a single bit of storage. By constraining the node colors on any path from the root to a leaf, red-black trees guarantee that no such path is more than twice as long as any other, so the tree is approximately balanced. If the child or parent of a node does not exist, the corresponding pointer contains the value NIL. We regard the NIL nodes as external nodes and the normal, key-bearing nodes as internal nodes.

A red-black tree satisfies the following properties, known as red-black properties:

1. Every node is either RED or BLACK.
2. The root is BLACK.
3. Every leaf (NIL) is BLACK.
4. If a node is RED, its children are BLACK.
5. For each node, all paths from the node to descendant leaves contain the same number of BLACK nodes.

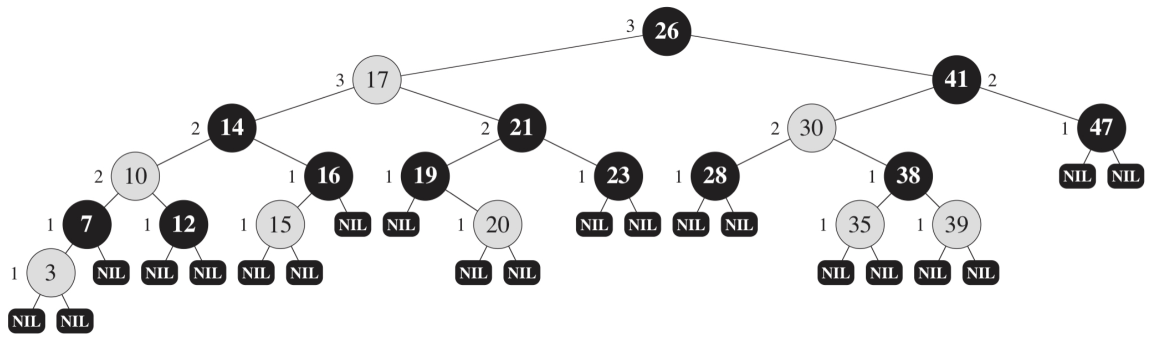


Figure 28.1: From CLRS, an example of a red-black tree with shaded BLACK nodes and non-shaded RED nodes. Leaf (NIL) nodes are colored BLACK. Note that all simple paths from root to leaves have the same number of BLACK nodes.

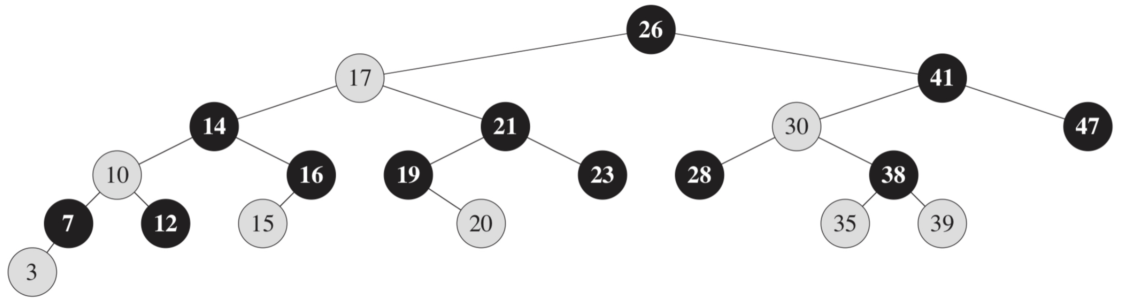


Figure 28.2: From CLRS, a more common representation of a red-black tree with NIL nodes omitted.

These notes were adapted from CLRS Chapter 13.1

We will show that the height of red-black trees is upper-bounded by $O(\lg n)$ in the following series of proofs. Denote the number of Black nodes on any simple path from, but not including, a node x to a leaf as the black-height of the node, or $bh(x)$.

Lemma 1. *The subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes.*

Proof. We will prove this claim using strong induction on the height of x.

Base Case: Height of x is 0. In this case, x must be a leaf, so the subtree rooted at x indeed has at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.

Inductive Hypothesis: Assume that the claim holds true for all nodes of height $0 \leq j \leq k$.

Inductive Step: Consider a node x of height $k+1$ such that x is an internal node with two children. By construction of the red-black tree, each child has black-height of either $bh(x)$ or $bh(x) - 1$, depending on whether the child is Red or Black, respectively. Since each child has height less than that of x, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1$ internal nodes in their rooted subtrees. Therefore, the subtree rooted at x has at least $2 \cdot (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes. □

Theorem 1. *A red-black tree with n internal nodes has height at most $2\lg(n+1)$.*

Proof. Let h be the height of the red-black tree. By Property 4, at least half the nodes of any simple path from the root to a leaf, not counting the root itself, must be Black. Consequently, the black-height of the root must be at least $\frac{h}{2}$. By **Lemma 1**,

$$n \geq 2^{\frac{h}{2}} - 1$$

Rearranging the inequality and applying a logarithm on both sides yields

$$h \leq 2\lg(n+1)$$

□

From **Theorem 1**, the height of a red-black tree is upper-bounded by $O(\lg n)$. Thus, basic operations on red-black trees can be performed in worst case logarithmic time with some overhead to maintain red-black properties. Like AVL trees, red-black trees have their own implementations of Insertion, Deletion, and Rotation to maintain their red-black property invariants. *

* These are covered in CLRS Chapters 13.2 - 13.4

Minimum Cut 29

29.1 The Minimum Cut Problem

Recall that a cut is a partition of vertices into two disjoint proper subsets, and a cut-set is the set of edges that cross such a cut. In this chapter, we introduce a fast randomized algorithm that finds the *minimum cut* (or "min-cut"), which is a minimum cardinality cut-set of a graph.

Definition. A minimum cut is the smallest set of edges in an undirected graph such that the removal of these edges breaks the graph into two or more connected components.

Minimum cut problems frequently arise in the study of network reliability, where nodes correspond to machines in the network and edges correspond to connections between machines. The min-cut of a network identifies the smallest number of connections that can fail before some pair of machines are unable to communicate. Minimum cuts also arise in clustering problems: if nodes represent Web pages and an edge between two nodes represent a corresponding hyperlink, then min-cuts divide the graph into clusters of related documents.

29.2 A Randomized Approach

A brute force approach to solve this problem is to calculate all $O(2^n)$ possible cuts and return the smallest cut found. As with most algorithms, this exponential blow-up time is not ideal for any practical application of this problem, so we set it as a benchmark to beat.

We propose a simple randomized algorithm for solving the min-cut problem with high probability in only polynomial time. The crux of this algorithm involves repeated calls to an operation known as an *edge contraction* until only one cut remains. In contracting an edge (u, v), we merge two vertices u and v at random into one vertex and eliminate all edges connecting u and v. This updated graph retains all other edges in the graph and permits parallel edges but no self-loops.

This algorithm is known as Karger's Algorithm (attributed to David Karger in 1993) and is formalized below.

Karger's Algorithm for the Minimum Cut Problem

Input: An undirected graph $G = (V, E)$.

Output: A set of edges in G that represent the minimum size cut-set (min-cut) in G.

```
Min-Cut(G)
    for i ← 1 to $n-2$ do
        Pick a remaining edge (u, v) in $G$ uniformly at random.
        Contract(u, v)

    Return the edges between the remaining two vertices as the min-cut.
```

These notes were adapted from Mitzenmacher and Upfal's *Probability and Computing*, Chapter 1.4

The algorithm runs until only two vertices (and one cut) remains. In each iteration, we pick an edge uniformly at random from the existing edges and contract that edge. After $n-2$ iterations, the graph consists of two vertices and outputs the set of edges connecting the two remaining vertices as our min-cut.

Examples of **a)** a successful run and **b)** an unsuccessful run of the MIN-CUT algorithm are shown below.

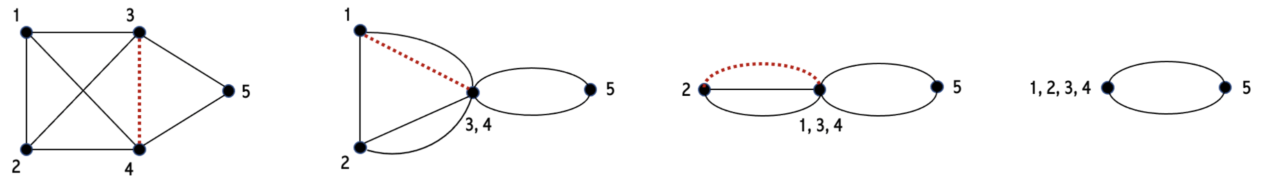


Figure 29.1: A successful run of MIN-CUT on a graph with min-cut of size 2.

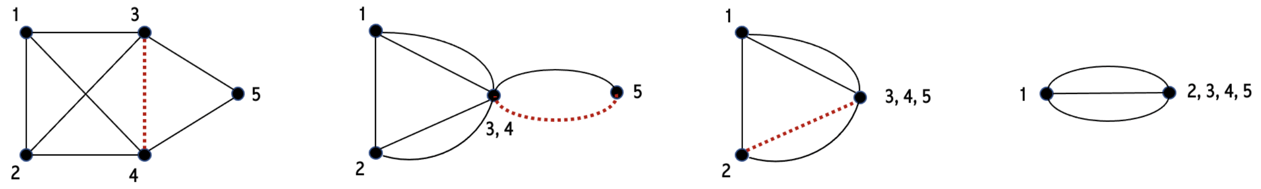


Figure 29.2: An unsuccessful run of MIN-CUT on the same graph. Note that this execution incorrectly returns a cut-set of size 3.

Note that any cut-set of the current graph in any immediate iteration of the algorithm is also a cut-set of the original graph. On the other hand, a particular cut-set of the original graph may not exist in the current graph if an edge of this cut-set was previously contracted. Therefore, the output of MIN-CUT will always be a cut-set of the original graph but not necessarily the minimum size cut-set (see **Figure 29.2**). In the next sections, we will show how MIN-CUT can be used to find the min-cut while mitigating the probability of an incorrect result.

29.3 Probabilistic Analysis

We will now establish a lower bound on the probability that an execution of MIN-CUT on a graph G with n vertices returns a correct output.

Theorem 1. *The* MIN-CUT *algorithm outputs a min-cut set with probability of at least* $\frac{2}{n(n-1)}$.

Proof: Let k be the size of a particular min-cut set C that partitions G into two sets, S and $V \setminus S$. There may be multiple min-cut sets, but we will only focus on a specific set C in our lower bounding. Notice that C only survives an iteration of MIN-CUT only if the contracted edge is **not** a part of C. Therefore, the algorithm correctly returns C as the minimum cut-set if the algorithm never chooses an edge in C to contract.

Let E_i be the event that the edge contracted in iteration i is not in C. Let $F_i = \cap_{j=1}^{i} E_j$ be the event that no edge of C was contracted within the first i iterations. Our goal is to compute $Pr[F_{n-2}]$.

We start by computing $Pr[E_1] = Pr[F_1]$. Since the minimum cut-set has k edges, every vertex in the graph must have degree k or larger ($\delta(G) \geq k$). By Handshaking Lemma, the graph must have at least $\frac{nk}{2}$ edges.

With at least $\frac{nk}{2}$ edges to choose from uniformly at random, the probability that we do not choose one of the k edges in C in the first iteration is as follows:

$$Pr[E_1] = Pr[F_1] \geq 1 - \frac{k}{nk/2} = 1 - \frac{2}{n}$$

For the sake of analysis, let's assume that the first contraction did not eliminate an edge in C. Conditioning on such an event F_1, we are left with $n-1$ nodes but still a minimum cut-set size k and thus still at least $\frac{k(n-1)}{2}$ edges. So,

$$Pr[E_2|F_1] \geq 1 - \frac{k}{(n-1)k/2} = 1 - \frac{2}{n-1}$$

We generalize this conditional probability in a similar manner.

$$Pr[E_i|F_{i-1}] \geq 1 - \frac{k}{(n-i+1)k/2} = 1 - \frac{2}{n-i+1}$$

All that remains is to compute a lower bound for $Pr[F_{n-2}]$.

$$\begin{aligned}
Pr[F_{n-2}] &= Pr[E_{n-2} \cap F_{n-3}] = Pr[E_{n-2}|F_{n-3}] \cdot Pr[F_{n-3}] \\
&= Pr[E_{n-2}|F_{n-3}] \cdot Pr[E_{n-3}|F_{n-4}] \cdot ... \cdot Pr[E_2|F_1] \cdot Pr[F_1] \\
&\geq \prod_{i=1}^{n-2}(1 - \frac{2}{n-i+1}) = \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1} \\
&= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot ... \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} \\
&= \frac{2}{n(n-1)}
\end{aligned}$$

Reducing the probability of error. Although one execution of MIN-CUT likely may not produce the correct output, we can reduce the error probability by repeating the algorithm. For reasons that will soon be clear, let's assume that MIN-CUT is independently run $n(n-1)\ln n$ times and the minimum size cut-set found over all runs is outputted.

Theorem 2. *The probability of an incorrect output over $n(n-1)\ln n$ calls to* MIN-CUT *is at most $\frac{1}{n^2}$*

The probability of an non-optimal output is bounded by

$$(1 - \frac{2}{n(n-1)})^{n(n-1)\ln n} \leq e^{-2\ln n} = \frac{1}{n^2} \qquad \text{(using } 1 - x \leq e^{-x}\text{)}.$$

Monte Carlo Algorithms. Karger's MIN-CUT Algorithm belongs to a class of algorithms known as Monte Carlo algorithms, which are finite randomized algorithms that may produce an incorrect output within a certain probability. By repeatedly sampling executions of these algorithms enough times, the probability of an incorrect solution becomes *almost neglibile*, demonstrating the power of randomization.

2-SAT 30

30.1 Introduction to the 2-SAT Problem

The Boolean satisfiability problem (abbrev. SAT) takes in a Boolean formula consisting of a conjunction (AND, denoted as $\wedge$) of a set of **clauses**, where each clause is a disjunction (OR, denoted as $\vee$) of literals. A **literal** is a Boolean variable (i.e. x_1) or the negation of a boolean variable (i.e. $\neg x_1$). A solution to a SAT instance is an assignment of TRUE and FALSE values to each variable x_1 ... x_n in a SAT formula such that all clauses return TRUE.

In the k-satisfiability (k-SAT) problem, each SAT clause is restricted to exactly k literals. The 2-SAT problem is a special case of k-SAT where each clause has exactly 2 variables. These types of boolean formulas are known as 2-CNF or Krom formulas. While generalized k-SAT belongs to a class of problems known as NP-complete problems, 2-SAT can actually be solved in linear time (using KOSARAJU'S, interestingly enough).

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2) \wedge (x_4 \vee \neg x_3) \wedge (x_4 \vee \neg x_1)$$

The above shows an instance of a 2-SAT formula. A truth assignment that solves this input is the following: $x_1 =$ TRUE, $x_2 =$ FALSE, $x_3 =$ FALSE, $x_4 =$ TRUE. You should trace through this assignment as an exercise to gauge your understanding of 2-SAT.

In this chapter, we examine a randomized algorithm to solve the 2-SAT problem with high probability.

30.2 Randomized 2-SAT Algorithm

Rather than trying all 2^n possible truth assignments for a given SAT formula, let's consider the following approach: start with an assignment, find an unsatisfied clause, and tweak the assignment so that the clause is satisfied. Our randomized 2-SAT algorithm repeatedly performs this step by choosing at random which variable's value in the clause to toggle in order to satisfy this clause. We also consider an integer parameter m that determines the probability of a correct answer, which will be soon made clear in the analysis.

Randomized 2-SAT Algorithm

Input: A 2-CNF Boolean formula Φ_n with n variables, and an integer parameter m.

Output: An assignment that solves the 2-SAT formula, or outputs that the formula is unsatisfiable.

```
2-SAT($\Phi_n$, m)
    Start with an arbitrary truth assignment.
    for i ← 1 to $2mn^2$ do
        Choose an arbitrary unsatisfied clause.
        Choose uniformly at random one of the literals in this clause and
            switch the value of its variable.
    If a valid truth assignment was ever found, return it.
```

These notes were adapted from Mitzenmacher and Upfal's *Probability and Computing*, Chapter 7.1

```
    Otherwise, return that the formula is unsatisfiable.
```

We can take the example 2-SAT formula above to demonstrate an execution of 2-SAT. Starting with an arbitrary truth assignment of all FALSE, we find that the clause $(x_1 \vee x_2)$ is unsatisfied. Choosing the literal x_1 randomly, we toggle the value of x_1 to be TRUE. Next, we pick $(x_4 \vee \neg x_3)$ as our unsatisfied clause, choose x_4 at random, and toggle x_4 to be TRUE. By now, we have reached a satisfying assignment which will be correctly outputted by the algorithm.

Note that we were extremely "lucky" in picking the right values to switch. In other random executions, it may feel like we take one step forward toward an optimal assignment during one iteration, but one step backward during another. This is precisely the correct intuition and will be explored in the following analysis.

30.3 Probabilistic Analysis

Let S be an optimal satisfying assignment of Φ_n for variables x_1 .. x_n, and let A_i be the assignment of variables after the i^{th} step of the algorithm. Let X_i denote the number of variables for which the truth assignments of A_i and S match. When $X_i = n$, then all variables in the assignment of A_n have the same values as that of S so the algorithm terminates with a satisfying assignment. Note that the algorithm may have found another satisfying assignment $\neq S$ and terminated early, but for our worst-case analysis we assume that the algorithm stops only when $X_i = n$.

Lemma 1. *Assume that Φ_n has a satisfying assignment and that the* 2-SAT *algorithm is allowed to run until it finds a satisfying assignment. Then the expected number of steps until the algorithm finds an assignment is at most n^2.*

We case on the value of the X_i, which is the number of matching values or "distance" toward from the optimal assignment S, in formalizing our conditional probabilities.

Case I: $X_i = 0$. For any change in a variable's value, we can only move one step closer to S.

$$Pr[X_{i+1} = 1 | X_i = 0] = 1$$

Case II: $1 \leq X_i \leq n-1$. At step i we choose an unsatisfied clause, at which point A_i and S disagree on the value of at least one of the two variables in this clause. If we mismatch on *both* variables, then flipping any of the variables' values would bring us one step closer to S. If we match on *one* of the variables, then we flip the mismatched value (and thus increase the number of matches) with a probability of $\frac{1}{2}$. On the other hand, we may flip a matched value and move one step away from S. Hence, for $1 \leq j \leq n-1$,

$$Pr[X_{i+1} = j+1 | X_i = j] \geq \frac{1}{2}$$
$$Pr[X_{i+1} = j-1 | X_i = j] \leq \frac{1}{2}$$

For the sake of analysis, we will only consider the pessimistic version of the above stochastic process.

$$
\begin{aligned}
Pr[X_{i+1} = 1 | X_i = 0] &= 1 \\
Pr[X_{i+1} = j+1 | X_i = j] &= \frac{1}{2} && \text{for } 1 \leq j \leq n-1 \\
Pr[X_{i+1} = j-1 | X_i = j] &= \frac{1}{2} && \text{for } 1 \leq j \leq n-1
\end{aligned}
$$

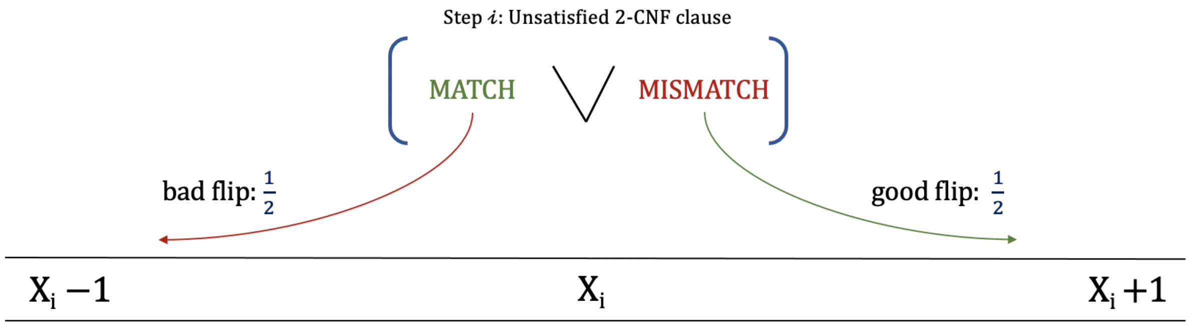


Figure 30.1: Illustration of the probabilistic effects of flipping a clause's value at random for $X_i \geq 1$.

Note that this pessimistic assumption can only increase the number of expected steps and frees us from any dependence on previously considered clauses. This is known as a **Markov chain**, which maps probabilistic transitions from one state to another. Here, this Markov chain models a random walk on a graph, where the states/vertices are the integers 0 ... n with edges connecting consecutive integers.

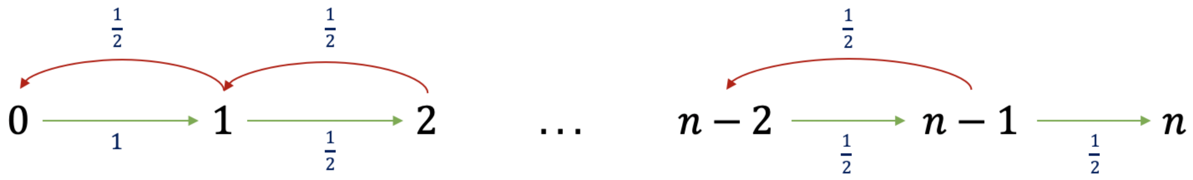


Figure 30.2: A probabilistic transition model known as a Markov chain. Each state is a possible value for X_i at step i, and each transition represents the probability of changing between states. S is discovered when n is reached.

Let T_j be the expected number of steps to reach n from state j. Clearly, $T_n = 0$ and $T_0 = T_1 + 1$. For $1 \leq j \leq n-1$, we move to the next state $j+1$ with probability $\frac{1}{2}$ and move to the previous state $j-1$ with probability $\frac{1}{2}$. We therefore have the following equations:

$$
\begin{aligned}
T_0 &= T_1 + 1 \\
T_n &= 0 \\
T_j &= \frac{1}{2}(1 + T_{j-1}) + \frac{1}{2}(1 + T_{j+1}) && \text{(Linearity of Expectation)} \\
&= 1 + \frac{T_{j-1}}{2} + \frac{T_{j+1}}{2}, && 1 \leq j \leq n-1
\end{aligned}
$$

Rearranging the last equation, we get:

$$\begin{aligned}
2T_j &= 2 + T_{j-1} + T_{j+1} \\
T_j - T_{j+1} &= 2 + T_{j-1} - T_j \\
&= 2 + 2 + T_{j-2} - T_{j-1} \\
&= 2 + 2 + 2 + T_{j-3} - T_{j-2} \\
&\vdots \\
&= 2 + 2 + 2 + ... + 2 + T_0 - T_1 \\
&= 2j + 1 && (T_0 - T_1 = 1)
\end{aligned}$$

Almost there! It follows to calculate T_0, the upper bound on the expected number of steps to reach S:

$$\begin{aligned}
T_0 &= (T_0 - T_1) + (T_1 - T_2) + ... + (T_{n-1} - T_n) \\
&= \sum_{j=0}^{n-1} T_j - T_{j+1} \\
&= \sum_{j=0}^{n-1} 2j + 1 \\
&= n^2
\end{aligned}$$

Theorem 1. *The* 2-SAT *algorithm always returns a correct answer if the formula is unsatisfiable, and returns a correct answer with probability of at least* $1 - \frac{1}{2^m}$ *if the formula is satisfiable.*

Proof: Clearly, if there is no satisfying assignment then the algorithm returns that the formula is unsatisfiable. Suppose the formula is satisfiable. Divide the execution of 2-SAT into m disjoint segments of $2n^2$ steps each. The expected number of steps that the algorithm takes to find a satisfying assignment for each segment is bound by n^2 by **Lemma 1**, regardless of the segment's starting position.

Let Z be the number of steps from the start of a segment until the algorithm finds a satisfying assignment. Applying Markov's Inequality,

$$Pr[Z > 2n^2] \leq \frac{n^2}{2n^2} = \frac{1}{2}$$

The probability that the algorithm succeeds in finding a satisfying assignment after m segments is thus at least $1 - (\frac{1}{2})^m$.

Appendix

Common Running Times A

For your convenience, we've compiled a list of running times of commonly used operations and data structures in Java. This list is not meant to be exhaustive.

All runtimes below assume Java 8 implementation.

`Array`

- `Initialization of size s`: worst case $O(s)$
- `Accessing an index`: worst case $O(1)$

`java.util.ArrayList` [JavaDoc] [Implementation]:

- `add(e)`: amortized $O(1)$, worst case $O(n)$
- `add(i, e)`: worst case $O(n)$
- `contains(e)`: worst case $O(n)$
- `get(i)`: worst case$O(1)$
- `indexOf(e)` : worst case $O(n)$
- `isEmpty()`: worst case $O(1)$
- `remove(i)`: worst case $O(n)$
- `set(i, e)`: worst case $O(1)$
- `size()`: worst case $O(1)$

Note: Adding/Removing an element to/from index $n - c$ for some constant c takes amortized $O(1)$ time.

`java.util.LinkedList` [JavaDoc][Implementation]:

- `add(e)`: worst case $O(1)$
- `add(i, e)`: worst case $O(n)$
- `contains(e)`: worst case $O(n)$
- `get(i)`: worst case $O(i)$
- `indexOf(e)` : worst case $O(n)$
- `isEmpty()`: worst case $O(1)$
- `remove()`: worst case $O(1)$
- `remove(i)`: worst case $O(n)$
- `set(i, e)`: worst case $O(i)$
- `size()`: worst case $O(1)$

Note: Adding/Removing an element to/from index $n - c$ for some constant c takes $O(1)$ time.

`java.util.HashMap` [JavaDoc] [Implementation]:

- `containsKey(k)`: expected $O(1)$, worst case $O(\lg n)$
- `containsValue(v)`: worst case $O(n)$
- `entrySet()`: worst case $O(1)$
- `get(k)`: expected $O(1)$, worst case $O(\lg n)$
- `isEmpty()`: worst case $O(1)$
- `keySet()`: worst case $O(1)$
- `put(k, v)`: expected $O(1)$, worst case $O(n)$
- `remove(k)`: expected $O(1)$, worst case $O(\lg n)$
- `size()`: worst case $O(1)$

- `values()`: worst case $O(1)$

`java.util.HashSet` [JavaDoc] [Implementation]

- `add(e)`: expected $O(1)$, worst case $O(n)$
- `contains(e)` : expected $O(1)$, worst case $O(\lg n)$
- `isEmpty()`: worst case $O(1)$
- `remove(e)`: expected $O(1)$, worst case $O(n)$
- `size()`: worst case $O(1)$

`java.lang.String` [Javadoc] [Implementation]

- `charAt(i)`: worst case $O(1)$
- `compareTo(s)`: worst case $O(n)$
- `concat(s)`: worst case $O(n + s.length())$
- `contains(s)`: worst case $O(n \cdot s.length())$
- `equals(s)`: worst case $O(n)$
- `indexOf(c)`: worst case $O(n)$
- `indexOf(s)`: worst case $O(n \cdot s.length())$
- `length()`: worst case $O(1)$
- `substring(i, j)`: worst case $O(j - i)$