Contents

Lecture 1

Introduction & Median Finding

Supplemental reading in CLRS: Section 9.3; Chapter 1; Sections 4.3 and 4.5

1.1 The Course

Hello, and welcome to 6.046 Design and Analysis of Algorithms. The prerequisites for this course are

1. *6.006 Introduction to Algorithms.* This course is designed to build on the material of 6.006. The algorithms in 6.046 are generally more advanced. Unlike 6.006, this course will not require students to implement the algorithms; instead, we make frequent use of pseudocode.

2. *Either 6.042 / 18.062J Mathematics for Computer Science or 18.310 Principles of Applied Mathematics.* Students must be familiar with beginning undergraduate mathematics and able to write mathematical proofs. After taking this course, students should be able not just to describe the algorithms they have learned, but also to prove their correctness (where applicable) and rigorously establish their asymptotic running times.

The course topics are as follows:

- Divide and Conquer
- Dynamic Programming
- Greedy Algorithms
- Graph Algorithms
- Randomized Algorithms
- Data Structures
- Approximation Algorithms.

The course textbook is *Introduction to Algorithms 3e.*, by Cormen, Leiserson, Rivest and Stein. We will usually refer to the textbook (or the authors) as "CLRS." In addition, Profs. Bruce Tidor and Dana Moshkovitz, who taught this course in spring of 2012, prepared hand-written notes for each lecture. What you are reading is an electronic, fleshed-out version of those notes, developed by Ben Zinberg with oversight by Prof. Moshkovitz under the commission of MIT OpenCourseWare.

1.2 Algorithms

Before our first example, let's take a moment to ask:

What is the study of algorithms, and why is it important?

CLRS offer the following answer: An algorithm is “any well-defined computational procedure that takes some [. . .] input and produces some [. . .] output.” They point out that algorithms are an essential component not just of inherently electronic domains such as electronic commerce or the infrastructure of the Internet, but also of science and industry, as exemplified by the Human Genome Project or an airline that wishes to minimize costs by optimizing crew assignments.

Studying algorithms allows us both to understand (and put to use) the capabilities of computers, and to communicate our ideas about computation to others. Algorithms are a basis for design: they serve as building blocks for new technology, and provide a language in which to discuss aspects of that technology. The question of whether a certain algorithm is effective depends on the context in which it is to be used. Often relevant are the following two concerns:

- *Correctness.* An algorithm is said to be **correct** if, for every possible input, the algorithm halts with the desired output. Usually we will want our algorithms to be correct, though there are occasions in which incorrect algorithms are useful, if their rate of error can be controlled.

- *Efficiency.* The best algorithms are ones which not just accomplish the desired task, but use minimal resources while doing so. The resources of interest may include time, money, space in memory, or any number of other “costs.” In this course we will mostly be concerned with time costs. In particular, we will try to give asymptotic bounds on the “number of steps” required to perform a computation as a function of the size of the input. The notion of a “step” is an imprecise (until it has been given a precise definition) abstraction of what on current computers could be processor cycles, memory operations, disk operations, etc. Of course, these different kinds of steps take different amounts of time, and even two different instructions completed by the same processor can take different amounts of time. In this course, we do not have the time to study these subtleties in much detail. More advanced courses, such as 6.854 Advanced Algorithms, take up the subject in further detail. Fortunately, the course material of 6.046 is extremely useful despite its limitations.

1.3 Order Statistics (“Median Finding”)

Median finding is important in many situations, including database tasks. A precise statement of the problem is as follows:

Problem 1.1. Given an array $A = A[1,\ldots,n]$ of n numbers and an index i ($1 \le i \le n$), find the ith-smallest element of A.

For example, if $i = 1$ then this amounts to finding the minimum element, and if $i = n$ then we are looking for the maximum element. If n is odd, then setting $i = \frac{n+1}{2}$ gives the median of A. (If n is even, then the median will probably be some average of the cases $i = \left\lfloor \frac{n+1}{2} \right\rfloor$ and $i = \left\lceil \frac{n+1}{2} \right\rceil$, depending on your convention.)

Intuitive Approach. We could simply sort the entire array A—the ith element of the resulting array would be our answer. If we use MERGE-SORT to sort A in place and we assume that jumping to the ith element of an array takes $O(1)$ time, then the running time of our algorithm is

$$\begin{array}{rrl} \text{(Merge-Sort)} & O(n\lg n) & \\ \text{(jump to the } i\text{th element)} & + \qquad O(1) & \\ \cline{2-2} & O(n\lg n) & \text{(worst case).} \end{array}$$

Can we expect to do better than this initial approach? If sorting all of A were a necessary step in computing any individual median, then the answer would be no. But as it stands, sorting all of A does much more work than we are asking for—it finds all n medians, whereas we only wanted one. So, even though the best sorting algorithms take $\Theta(n\lg n)$ time, it does not necessarily follow that median finding cannot be done faster.

It turns out that median finding can be done in $O(n)$ time. In practice, this is usually accomplished by a randomized algorithm with linear expected running time, but there also exists a deterministic algorithm with linear worst-case time. In a 1973 paper, Blum, Floyd, Pratt, Rivest and Tarjan proposed the so-called "median-of-medians" algorithm, which we present below.

For this algorithm, we will assume that the elements of A are distinct. This is not a very strong assumption, and it is easy to generalize to the case where A is allowed to have duplicate elements.[1]

Algorithm: SELECT(A, i)

1. Divide the n items into groups of 5 (plus any remainder).
2. Find the median of each group of 5 (by rote). (If the remainder group has an even number of elements, then break ties arbitrarily, for example by choosing the lower median.)
3. Use SELECT recursively to find the median (call it x) of these $\lceil n/5 \rceil$ medians.
4. Partition around x.[*] Let $k = \operatorname{rank}(x)$.[†]

$$\underbrace{\boxed{\quad A_i < x \quad}}_{k-1\text{ elements}}\underset{\substack{\uparrow\\ A_k}}{\boxed{x}}\underbrace{\boxed{\quad A_i > x \quad}}_{n-k\text{ elements}}$$

5.
   - If $i = k$, then return x.
   - Else, if $i < k$, use SELECT recursively by calling SELECT$(A[1,\dots,k-1],\ i)$.[‡]
   - Else, if $i > k$, use SELECT recursively by calling SELECT$(A[k+1,\dots,i],\ i-k)$.

[*] By "partition around x," we mean reorder the elements of A in place so that all elements prior to x are less than x (and consequently all elements after x are $\geq x$—in our case actually $> x$ since there are no duplicate elements). Partitioning can be done in linear time. We won't show the details here, but they can be found in §7.1 of CLRS.

[†] For a set S of distinct numbers, we define the **rank** of an element $x \in S$ to be the number k such that x is the kth-smallest element of S. Thus, in the set $\{5,8,2,3\}$, the rank of 5 is 3.

[‡] The array $A[1,\dots,k-1]$ should be passed by reference—there is certainly no need to waste time and memory by making a new copy.

[1] One example of such a way is to equip each element of A with satellite data. Replace each element $A[j]$ $(1 \leq j \leq n)$ with the pair $\langle A[j], j\rangle$, thus turning A into an array of ordered pairs. Then we can define an order on A by saying $\langle x_1, j_1\rangle < \langle x_2, j_2\rangle$ iff either $x_1 < x_2$, or $x_1 = x_2$ and $j_1 < j_2$. (This is the so-called "lexicographic order.") Since this order is strict, it is safe to feed into our algorithm which assumes no duplicate elements. On the other hand, once A is sorted in the lexicographic order, it will also be sorted in the regular order.

Note a few things about this algorithm. First of all, it is extremely non-obvious. Second, it is recursive, i.e., it calls itself. Third, it appears to do "less work" than the intuitive approach, since it doesn't sort all of A. Thus, we might suspect that it is more efficient than the intuitive approach.

1.3.1 Proof of Correctness

The strategy to prove correctness of an algorithm is a mix of knowing standard techniques and inventing new ones when the standard techniques don't apply. To prove correctness of our median-of-medians algorithm, we will use a very common technique called a loop invariant. A **loop invariant** is a set of conditions that are true when the loop is initialized, maintained in each pass through the loop, and true at termination. Loop invariant arguments are analogous to mathematical induction: they use a base case along with an inductive step to show that the desired proposition is true in all cases. One feasible loop invariant for this algorithm would be the following:

> At each iteration, the "active subarray" (call it A') consists of all elements of A which are between $\min(A')$ and $\max(A')$, and the current index (call it i') has the property that the i'th-smallest element of A' is the ith-smallest element of A.

In order to prove that this is a loop invariant, three things must be shown:

1. *True at initialization.* Initially, the active subarray is A and the index is i, so the proposition is obviously true.

2. *Maintained in each pass through the loop.* If the proposition is true at the beginning of an iteration (say with active subarray $A' = A'[1,\dots,n']$ and active index i'), then the $(k-1)$- and $(n'-k)$-element subarrays depicted in step 4 clearly also have the contiguity property that we desire. Then, if step 5 calls $\textsc{Select}(A'', i'')$, it is easy to see by casework that the i''th-smallest element of A'' equals the i'th-smallest element of A', which by hypothesis equals the ith-smallest element of A.

3. *True at termination.* Since there is nothing special about the termination step (it is just whichever step happens to be interrupted by the returned value), the proof of maintenance in each pass though the loop is sufficient to show maintenance at termination.

Now that we have established the loop invariant, it is easy to see that our algorithm is correct. If the final recursive iteration of $\textsc{Select}$ has arguments A' and i', then step 5 returns the i'th-smallest element of A', which by the loop invariant equals the ith-smallest element of A. Thus we have proven that, if our algorithm terminates at all, then it terminates with the correct output. Moreover, the algorithm must terminate eventually, since the size of the active subarray shrinks by at least 1 with each recursive call and since $\textsc{Select}(A', i')$ terminates immediately if A' has one element.

1.3.2 Running Time

There is no single magic bullet for writing proofs. Instead, you should use nonrigorous heuristics to guide your thinking until it becomes apparent how to write down a rigorous proof.

In order to prove that our algorithm is efficient, it would help to know that the active subarray in each successive recursive call is much smaller than the previous subarray. That way, we are guaranteed that only relatively few recursive calls are necessary. Therefore, let's determine an upper bound on the size of A', the new active subarray, given that the old subarray, A, had size n.

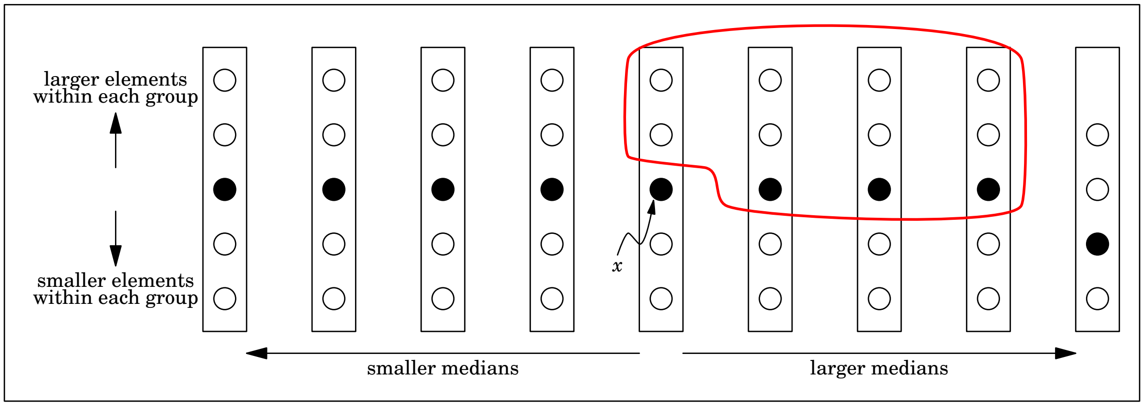

larger elements within each group
smaller elements within each group
x
smaller medians
larger medians

Figure 1.1. Imagine laying out the groups side by side, ordered with respect to their medians. Also imagine that each group is sorted from greatest to least, top to bottom. Then each of the elements inside the red curve is guaranteed to be greater than x.

As in step 3, let x be the median of medians. Then either A' doesn't contain any items greater than x, or A' doesn't contain any items less than x. However, as we will see, there are lots of elements greater than x and there are lots of elements less than x. In particular, each (non-remainder) group whose median is less than x contains at least three elements less than x, and each (non-remainder) group whose median is greater than x contains at least three elements greater than x. Moreover, there are at least $\lceil n/5 \rceil - 1$ non-remainder groups, of which at least $\left\lfloor \frac{1}{2} \lceil n/5 \rceil - 2 \right\rfloor$ have median less than x and at least $\left\lfloor \frac{1}{2} \lceil n/5 \rceil - 2 \right\rfloor$ have mean greater than x. Finally, the group with x as its median contains two elements greater than x and two elements less than x (unless $n < 5$, but you can check the following inequalities in that case separately if you like). Thus

$$\left(\#\text{ elts of } A \text{ less than } x\right) \geq 3\left(\left\lfloor \tfrac{1}{2} \lceil n/5 \rceil - 2 \right\rfloor\right) + 2 \geq 3\left(\tfrac{1}{2} \lceil n/5 \rceil - 2 - \tfrac{1}{2}\right) + 2 > \tfrac{3}{10} n - 6.$$

Similarly,

$$\left(\#\text{ elts of } A \text{ greater than } x\right) \geq \tfrac{3}{10} n - 6.$$

Therefore A' must exclude at least $\frac{3}{10} n - 6$ elements of A, which means that A' contains at most $n - \left(\frac{3}{10} n - 6\right) = \frac{7}{10} n + 6$ elements. See Figure 1.1 for an illustration.

Let's now put together all the information we know about the running time of SELECT. Let $T(n)$ denote the worst-case running time of SELECT on an array of size n. Then:

- Step 1 clearly takes $O(n)$ time.
- Step 2 takes $O(n)$ time since it takes $O(1)$ time to find the median of each 5-element group.
- Step 3 takes $T(\lceil n/5 \rceil)$ time since there are $\lceil n/5 \rceil$ submedians.
- Step 4 takes $O(n)$ time, as explained in §7.1 of CLRS.
- Step 5 takes at most $T\left(\frac{7}{10} n + 6\right)$ since the new subarray has at most $\frac{7}{10} n + 6$ elements.[2]

[2] Technically, we haven't proved that $T(n)$ is an increasing function of n. However, if we let $T'(n) = \max\{T(m) : m \leq n\}$, then $T'(n)$ is an increasing function of n, and any asymptotic upper bound on $T'(n)$ will also be an asymptotic upper bound on $T(n)$.

Thus

$$\boxed{T(n) \le T(\lceil n/5 \rceil) + T\left(\tfrac{7}{10}n + 6\right) + O(n).} \tag{1.1}$$

1.3.3 Solving the Recurrence

How does one go about solving a recurrence like (1.1)? Possible methods include:

- Substitution Method: make a guess and prove it by induction.
- Recursion Tree Method: branching analysis.
- Master Method: read off the answer from a laundry list of already-solved recurrences.[3]

We will use the substitution method this time. Our guess will be that $T(n)$ is $O(n)$, i.e., that there exists a positive constant c such that $T(n) \le cn$ for all sufficiently large n. We won't choose c just yet; later it will become apparent which value of c we should choose.

In order to deal with the base case of an inductive argument, it is often useful to separate out the small values of n. (Later in the argument we will see how this helps.) Since SELECT always terminates, we can write

$$T(n) \le \begin{cases} O(1), & n < 140 \\ T(\lceil n/5 \rceil) + T\left(\tfrac{7}{10}n + 6\right) + O(n), & \text{for all } n. \end{cases}$$

In other words, there exist positive constants a, b such that $T(n) \le b$ for $n < 140$ (for example, $b = \max\{T(m) : m < 140\}$) and $T(n) \le T(\lceil n/5 \rceil) + T\left(\tfrac{7}{10}n + 6\right) + an$ for all n. Once this is done, we can start our inductive argument with the base case $n = 140$:

Base case.

$$\begin{aligned} T(140) &\le T(\lceil 140/5 \rceil) + T\left(\tfrac{7}{10}140 + 6\right) + an \\ &\le b + b + an \\ &= 2b + an \\ &= 2b + 140a. \end{aligned}$$

Thus, the base case holds if and only if $2b + 140 \le c(140)$, or equivalently $c \ge a + \frac{b}{70}$.

Inductive step. Suppose $n \ge 140$. Then the inductive hypothesis gives

$$\begin{aligned} T(n) &\le c(\lceil n/5 \rceil) + c\left(\tfrac{7}{10}n + 6\right) + an \\ &\le c\left(\tfrac{n}{5} + 1\right) + c\left(\tfrac{7}{10}n + 6\right) + an \\ &= \tfrac{9}{10}cn + 7c + an \\ &= cn + \underbrace{\left(\frac{-cn}{10} + 7c + an\right)}. \end{aligned}$$

Thus all we need is for the part with the brace to be nonpositive. This will happen if and only if $c \ge \frac{10an}{n-70}$. On the other hand, since $n \ge 140$, we know $\frac{n}{n-70} \le 2$. (This is why we wanted a base case of $n = 140$. Though, any base case bigger than 70 would have worked.) Thus the part with the brace will be nonpositive whenever $c \ge 20a$.

[3] The master method is a special case of a general technique known as *literature search*, which is of fundamental importance in not just computer science but also physics, biology, and even sociology.

Putting these two together, we find that the base case and the inductive step will both be satisfied as long as c is greater than both $a + \frac{b}{70}$ and $20a$. So if we set $c = \max\left\{20a, \frac{b}{70}\right\} + 1$, then we have $T(n) \le cn$ for all $n \ge 140$. Thus $T(n)$ is $O(n)$.

Lecture 2

Recap & Interval Scheduling

Supplemental reading in CLRS: Section 16.1; Section 4.4

2.1 Recap of Median Finding

Like MERGE-SORT, the median-of-medians algorithm SELECT calls itself recursively, with the argument to each recursive call being smaller than the original array A. However, while MERGE-SORT runs in $O(n \lg n)$ time, SELECT is able to run in linear time. Why is this? There are two key observations to make:

Observation 1. Given an element x of A, we can partition around x in linear time. Thus, all the steps in SELECT other than the recursive calls take a total of only $O(n)$ time.

Observation 2. Finding the median of n elements can be reduced to finding the median of $n/5$ elements and then finding the median of at most $\frac{7}{10}n$ elements. (The true figures in our case are $\lceil n/5 \rceil$ and $\frac{7}{10}n+6$, but that's not important for the analysis that follows. For a refresher on where this observation comes from, see the proof of running time and Figure 1.1.)

To see why the second observation is important, we'll do a branching analysis. This will also show how we might have figured out the running time of our algorithm without guessing. Observation 2 is just a translation into English of the recurrence relation

$$T(n) \le T\left(\tfrac{1}{5}n\right) + T\left(\tfrac{7}{10}n\right) + O(n). \tag{2.1}$$

Pick a constant c such that $T(n) \le T\left(\frac{1}{5}n\right) + T\left(\frac{7}{10}n\right) + cn$ for all sufficiently large n. (Actually, for simplicity, let's assume the relation holds for all n.[1]) Then we can apply (2.1) to each of the terms $T\left(\frac{1}{5}n\right)$ and $T\left(\frac{7}{10}n\right)$, obtaining

$$\begin{aligned}
T(n) &\le T\left(\tfrac{1}{5}n\right) + T\left(\tfrac{7}{10}n\right) + cn \\
&\le \underbrace{\left[T\left(\tfrac{1}{5}\cdot\tfrac{1}{5}n\right) + T\left(\tfrac{7}{10}\cdot\tfrac{1}{5}n\right) + c\left(\tfrac{1}{5}n\right)\right]}_{} + \underbrace{\left[T\left(\tfrac{1}{5}\cdot\tfrac{7}{10}n\right) + T\left(\tfrac{7}{10}\cdot\tfrac{7}{10}n\right) + c\left(\tfrac{7}{10}n\right)\right]}_{} + cn \\
&\le \cdots \\
&\le cn + c\left(\tfrac{1}{5} + \tfrac{7}{10}\right)n + c\left(\tfrac{1}{5}\cdot\tfrac{1}{5} + \tfrac{1}{5}\cdot\tfrac{7}{10} + \tfrac{7}{10}\cdot\tfrac{1}{5} + \tfrac{7}{10}\cdot\tfrac{7}{10}\right)n + \cdots
\end{aligned}$$

[1] Make sure you see why we are allowed to do this.

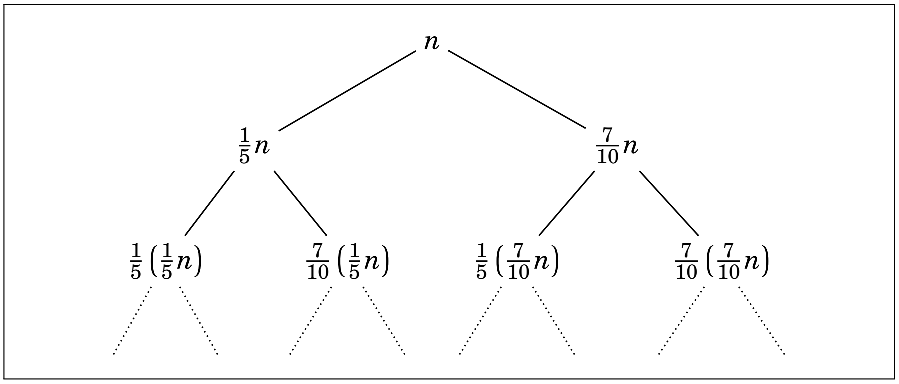

n
$\frac{1}{5}n$
$\frac{7}{10}n$
$\frac{1}{5}\left(\frac{1}{5}n\right)$
$\frac{7}{10}\left(\frac{1}{5}n\right)$
$\frac{1}{5}\left(\frac{7}{10}n\right)$
$\frac{7}{10}\left(\frac{7}{10}n\right)$

Figure 2.1. A recursion tree for SELECT. Running SELECT on an array of n elements gives rise to two daughter processes, with arguments of sizes $\frac{1}{5}n$ and $\frac{7}{10}n$. Each of these daughters produces two of its own daughters, with arguments of sizes $\frac{1}{5}\left(\frac{1}{5}n\right)$, $\frac{7}{10}\left(\frac{1}{5}n\right)$, etc.

$$= cn + c\left(\tfrac{1}{5}+\tfrac{7}{10}\right)n + c\left(\tfrac{1}{5}+\tfrac{7}{10}\right)^2 n + c\left(\tfrac{1}{5}+\tfrac{7}{10}\right)^3 n + \cdots$$

$$= c\left[\sum_{j=0}^{\infty}\left(\tfrac{1}{5}+\tfrac{7}{10}\right)^j\right] n \qquad \text{(a geometric sum)}$$

$$= 10cn.$$

This geometric sum makes it clear that it is very important for $\frac{1}{5}+\frac{7}{10}$ to be less than 1—otherwise, the geometric sum would diverge. If the number $\frac{1}{5}$ in our recurrence had been replaced by $\frac{2}{5}$, the resulting recurrence would have non-linear solutions and the running time would be $T(n) = \Theta(n^2)$. On an intuitive level, this makes sense: reducing an input of n entries to an input of $\left(\frac{1}{5}+\frac{7}{10}\right)n = \frac{9}{10}n$ entries is a good time-saver, but "reducing" an input of n entries to an input of $\left(\frac{2}{5}+\frac{7}{10}\right)n = \frac{11}{10}n$ entries is not.

Finally, let's make an observation that one is not usually able to make after writing an algorithm:

> "SELECT achieves the best possible asymptotic running time for an algorithm that solves the median finding problem."

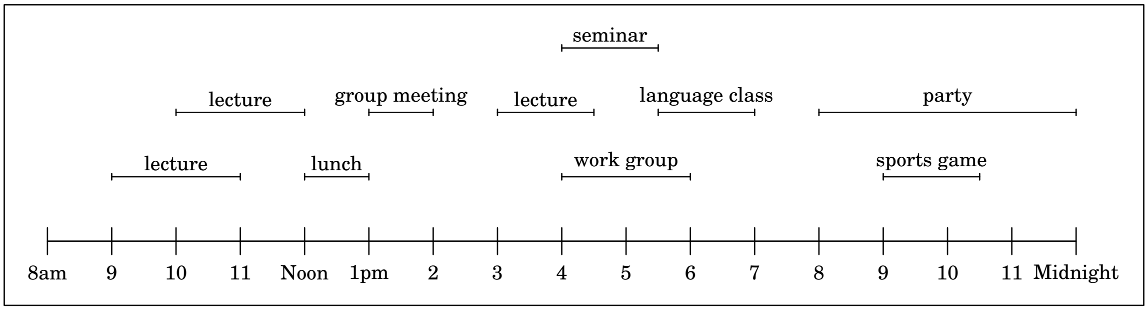
This statement must be interpreted carefully. What we mean is that any correct algorithm which solves the median finding problem must be $\Omega(n)$, since such an algorithm must at least look at each of the n entries of A. However, there are several other considerations that prevent us from being able to conclude that SELECT is the "best" median finding algorithm:

- It is only the *asymptotic* running time of SELECT that is optimal. The overhead could be very large. There may exist algorithms with a larger asymptotic running time that are still much faster than SELECT for all practical purposes because the asymptotic behavior only becomes significant when n is inhumanly large.
- There may also exist algorithms with the same asymptotic running time as SELECT but whose implementations run faster than those of SELECT because they have smaller overhead (i.e., the constants implicit in the big-O notation are smaller).
- There may exist randomized algorithms which run in sublinear time and return the correct answer with high probability, or deterministic algorithms which run in sublinear time and

return approximately the correct answer.[2]

2.2 Interval Scheduling

Does your planner look like this?



The first step to getting the most out of your day is getting enough sleep and eating right. After that, you'll want to make a schedule that is in some sense "maximal." There are many possible precise formulations of this problem; one of the simpler ones is this:

Input: A list of pairs $L = \langle (s_1, f_1), \dots, (s_n, f_n) \rangle$, representing n events with start time s_i and end time f_i ($s_i < f_i$).

Output: A list of pairs $S = \langle (s_{i_1}, f_{i_1}), \dots, (s_{i_k}, f_{i_k}) \rangle$ representing a schedule, such that

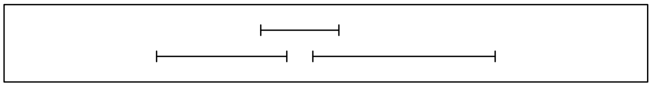
$$s_{i_1} < f_{i_1} \le s_{i_2} < f_{i_2} \le \cdots \le s_{i_k} < f_{i_k}$$

(i.e., no scheduled events overlap), where k is maximal.

This formulation corresponds to wanting to attend as many events as possible (not allowing late arrivals or early departures, without any regard for how long the events are, without any preference of one event over another, and assuming it takes no time to travel between events).

While all of us face problems like this in everyday life, probably very few of us have tried to mathematically rigorize the heuristics we use to decide what to do each day. Often we employ greedy strategies. A **greedy strategy** can be used in situations where it is easy to tell what is "locally optimal." The hope is that by making locally optimal decisions at each point, we will arrive at a globally optimal solution.[3] A couple of reasonable-sounding greedy strategies are:

- Pick the activities that take the least time (minimize $f_i - s_i$).

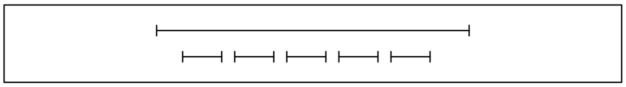


[2] While such a sublinear algorithm is probably out of the question in our case (at least if we have no prior information about A), there are lots of problems in which this does occur. For example, if you believe P $\neq$ NP, we will see later in the course that there exist polynomial-time approximations to some NP-hard problems.

[3] An analogous situation is this: Imagine that you live in the Himalayan village of Nagarkot and are trying to find the highest point in the Himalayan mountain range. A greedy strategy would be to hike straight up. Since this strategy is locally optimal, you will eventually arrive at a peak from which it is impossible to go up without first going down. However, your peak will probably not be the global optimum, since Nagarkot is in central Nepal and Mt. Everest is on the Chinese border.

This strategy fails because attending one short activity may prevent you from attending two other activities.

- Each time you find yourself doing nothing, go to the activity that starts the soonest (locally minimize s_i).



This strategy fails because it is possible for the earliest activity to preclude lots of other activities.

Somewhat surprisingly, there is a greedy strategy that is optimal:

- Each time you find yourself doing nothing, go to the activity that *ends* the soonest (locally minimize f_i). In pseudocode,

```
while there are still activities to consider do
    Pick $(s_i, f_i)$ with the smallest $f_i$
    Remove all activities that intersect $(s_i, f_i)$
```

Claim. The above algorithm outputs a list of pairs $\langle (s_{i_1}, f_{i_1}), \ldots, (s_{i_k}, f_{i_k}) \rangle$ such that

$$s_{i_1} < f_{i_1} \leq s_{i_2} < f_{i_2} \leq \cdots \leq s_{i_k} < f_{i_k}.$$

Proof. By definition, $s_i < f_i$. Now, if $s_{i_2} < f_{i_1}$, then (s_{i_2}, f_{i_2}) overlaps (s_{i_1}, f_{i_1}). Thus (s_{i_2}, f_{i_2}) must have been removed from consideration after we chose (s_{i_1}, f_{i_1}), a contradiction. The same reasoning with 1 and 2 replaced by j and $j+1$ for arbitrary j completes the proof. □

Claim. Suppose given a list of activities L. If an optimal schedule has k^* activities, then the above algorithm outputs a schedule with k^* activities.

Proof. By induction on k^*. If the optimal schedule has only one activity, then obviously the claim holds. Next, suppose the claim holds for k^* and we are given a set of events whose optimal schedule has k^*+1 activities. Let $S^* = S^*[1, \ldots, k^*+1] = \langle (s_{\ell_1}, f_{\ell_1}), \ldots, (s_{\ell_{k^*+1}}, f_{\ell_{k^*+1}}) \rangle$ be an optimal schedule, and let $S = S[1, \ldots, k] = \langle (s_{i_1}, f_{i_1}), \ldots, (s_{i_k}, f_{i_k}) \rangle$ be the schedule that our algorithm gives. By construction, $f_{i_1} \leq f_{\ell_1}$. Thus the schedule $S^{**} = \langle (s_{i_1}, f_{i_1}), (s_{\ell_2}, f_{\ell_2}), \ldots, (s_{\ell_{k^*+1}}, f_{\ell_{k^*+1}}) \rangle$ has no overlap, and is also optimal since it has k^*+1 activities. Let L' be the set of activities with $s_i \geq f_{i_1}$. The fact that S^{**} is optimal for L implies that $S^{**}[2, \ldots, k^*+1]$ is optimal for L'. Thus an optimal schedule for L' has k^* activities. So by the inductive hypothesis, running our algorithm on L' produces an optimal schedule. But by construction, running our algorithm on L' just gives $S[2, \ldots, k]$. Since any two optimal schedules have the same length, we have $k = k^*+1$. So S is an optimal schedule, completing the induction. □

This algorithm can be implemented with $\Theta(n \lg n)$ running time as follows:

```
Algorithm: UNWEIGHTED-SCHEDULE($L$)
Sort $L$ according to finish time using MERGE-SORT
$S \leftarrow \langle\rangle$
$curf \leftarrow -\infty$    ▷ the finish time of our current schedule
for $(s, f)$ in $L$ do
    if $s \geq curf$ then
        $S.append[(s, f)]$
        $curf \leftarrow f$
return $S$
```

This implementation makes one pass through the list of activities, ignoring those which overlap with the current schedule. Notice that the only part of this algorithm that requires $\Theta(n \lg n)$ time is the sorting—once L is sorted, the remainder of the algorithm takes $\Theta(n)$ time.

2.2.1 Weighted Interval Scheduling

Let's try a more sophisticated formulation of the scheduling problem. Suppose that rather than maximize the total number of events we attend, we want to maximize the total amount of time we are busy. More generally still, suppose each event in L has a weight $w > 0$ representing how much we prioritize that event.

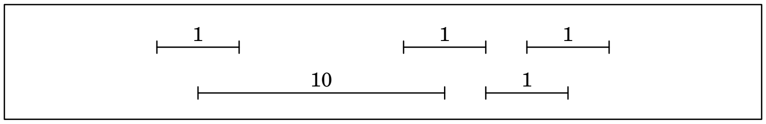
Input: A list of triples $L = \langle (s_1, f_1, w_1), \ldots, (s_n, f_n, w_n) \rangle$, representing n events with start time s_i, end time f_i ($s_i < f_i$), and weight $w_i > 0$.

Output: A list of triples $S = \langle (s_{i_1}, f_{i_1}, w_{i_1}), \ldots, (s_{i_k}, f_{i_k}, w_{i_k}) \rangle$ representing a schedule, such that

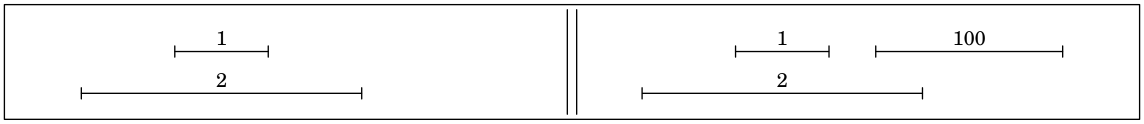
$$s_{i_1} < f_{i_1} \leq s_{i_2} < f_{i_2} \leq \cdots \leq s_{i_k} < f_{i_k}$$

(i.e., no scheduled events overlap), where $\sum_{j=1}^{k} w_{i_j}$ is maximal.

A greedy algorithm of the sort above will not work for this problem.



Which is better: attending a short event with low weight (so that you have more free time for the heavy events), or attending a slightly longer event with slightly higher weight? It depends on whether there are exciting opportunities in the future.



Here is a sketch of an efficient solution to the weighted scheduling problem:

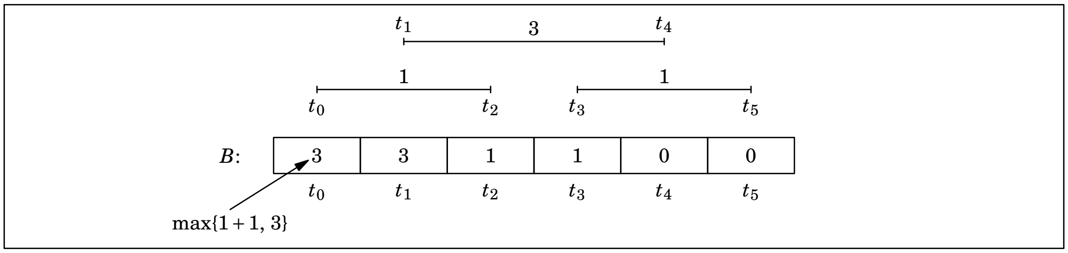


Figure 2.2. An illustration of WEIGHTED-SCHEDULE.

Algorithm: WEIGHTED-SCHEDULE(L)
Put all the start and finish times into an array $A = \langle s_1, f_1, \ldots, s_n, f_n \rangle$. Sort A, keeping track of which activity each number belongs to (that is, the index i). Make a new array B with one entry for each element of A. The idea is that the value of $B[t]$ will be the maximal weight of a schedule that starts at point t, along with a list of event indices that gives such a schedule. Thus $B[t_0]$ (assuming your day starts at $t = t_0$) will give you an optimal schedule for your day.

To construct B, first initialize all entries to zero. Then traverse backwards through A. The first time you hit a start point $s_{i_{\text{first}}}$, set $B[s_{i_{\text{first}}}] \leftarrow w_{i_{\text{first}}}$. From then on, keep track of the most recent start point you have seen in a new variable i_{cur}. Each time you hit an endpoint f_i, set $B[f_i] \leftarrow B[s_{i_{\text{cur}}}]$. Each time you hit a new start point s_i, set

$$B[s_i] \leftarrow \max\left\{B[s_{i_{\text{cur}}}],\ w_i + B[f_i]\right\}.$$

Once you have finished, $B[t_0]$ (assuming your day starts at $t = t_0$) will be the weight of an optimal schedule.

For ease of exposition, I didn't say explicitly how to keep track of the actual schedule you are constructing (which you will usually want to do—what use is the maximal weight if you don't know what schedule gives that weight?). That is an easy implementation detail that you can figure out for yourself. A more subtle issue that I didn't address is the possibility of duplicate elements of A. One way to deal with this is to, for each new start point s, keep a cache of possible weights of schedules starting at s (using a new variable to keep track of the heaviest schedule in cache). The cache should only be cleared when you hit a start point whose value is strictly less than s (or run out of elements of A), at which point you will know what value to assign to $B[s]$.

Again, it is easy to see that this algorithm runs in linear time after sorting, for a total running time of $O(n \lg n)$.

This example introduces the algorithmic paradigm of **dynamic programming**. "Dynamic programming" refers to a strategy in which the solution to a problem is computed by combining the solutions to subproblems which are essentially smaller versions of the same problem. In our example, we started with an optimal schedule for the empty set of events and then added events one at a time (in a strategic order), arriving at an optimal schedule for the full set L of events. Each time we wanted to solve a subproblem, we used the answers to the previous subproblems.

2.2.2 Conclusions

This lecture showed that the way in which a problem is formulated is important. A given real-life problem can be formulated as a precise algorithmic problem in many different, non-equivalent ways, with different solutions.

There are several standard approaches that you should have in your toolbox. So far we have seen greedy algorithms and dynamic programming; more will come up later in the course.

Lecture 3

Minimum Spanning Trees I

Supplemental reading in CLRS: Chapter 4; Appendix B.4, B.5; Section 16.2

3.1 Greedy Algorithms

As we said above, a **greedy algorithm** is an algorithm which attempts to solve an optimization problem in multiple stages by making a "locally optimal" decision at each stage.

Example. We wish to make 99¢ in change using the minimal number of coins. Most people instinctively use a greedy algorithm:

$$\begin{array}{rcllll}
99\text{¢} & = & (25\text{¢})\times 3+ & & & \\
-75\text{¢} & & & & & \\
\cline{1-1}
24\text{¢} & = & & (10\text{¢})\times 2\ + & & \\
-20\text{¢} & & & & & \\
\cline{1-1}
4\text{¢} & = & & & (1\text{¢})\times 4\ \ \text{(no nickels)} & \\
-4\text{¢} & & & & & \\
\cline{1-1}
0\text{¢} & & & & & \\
\hline
 & & \text{3 quarters +} & \text{2 dimes +} & \text{4 pennies =} & \text{9 coins.}
\end{array}$$

This greedy algorithm is correct: starting with the coin of largest value, take as many as possible without allowing your total to exceed 99¢. However, plenty of greedy algorithms are not correct. For example, suppose we started with the smallest coin rather than the largest coin:

$$99\text{¢} = (1\text{¢})\times 99 \Longrightarrow 99 \text{ pennies.}$$

Or, imagine trying to make 15¢ of change if the dime were replaced by an 11¢ piece:

$$\begin{array}{rlcl}
\text{Greedy:} & 15\text{¢} = (11\text{¢})\times 1 + (5\text{¢})\times 0 + (1\text{¢})\times 4 & \Longrightarrow & 5 \text{ coins} \\
\text{Optimal:} & 15\text{¢} = (11\text{¢})\times 0 + (5\text{¢})\times 3 + (1\text{¢})\times 0 & \Longrightarrow & 3 \text{ coins.}
\end{array}$$

Remark. Greedy algorithms sometimes give a correct solution (are globally optimal). But even when they're not correct, greedy algorithms can be useful because they provide "good solutions" efficiently. For example, if you worked as a cashier in a country with the 11¢ piece, it would be perfectly reasonable to use a greedy algorithm to choose which coins to use. Even though you won't always use the smallest possible number of coins, you will still always make the correct amount of change, and the number of coins used will always be close to optimal.

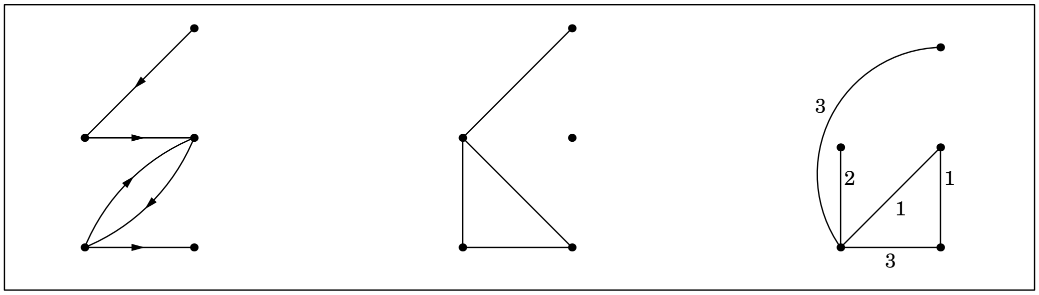


Figure 3.1. From left to right: A directed graph, an undirected graph, and a weighted undirected graph.

The above example shows that the performance of a greedy algorithm (in terms of both correctness and efficiency) depends on both

- the structure of the algorithm *(starting with big coins vs. small coins)*
- the structure of the problem *(coin values)*.

3.2 Graphs

We assume that you have seen some graph theory before. We will give a quick review here; if you are shaky on graph theory then you should review §B.4 of CLRS. A **graph** is a mathematical object which consists of vertices and edges. Thus, we may say that a graph is a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. Or, if the vertex set V has been fixed ahead of time, we may simply say that "E is a graph on V." In an **undirected graph**, the edges are unordered pairs $e = \{u, v\}$ with $u, v \in V$; whereas in a **directed graph**, the edges are ordered pairs $e = (u, v)$. A directed edge $e = (u, v)$ is said to emanate "from u to v," and we may use the notation $e = (u \to v)$.[1] Our convention (which is by no means standard convention, since there is no standard) will be that all graphs are assumed finite, and are allowed to have loops (edges $u \to u$) but may not have multiple edges between the same pair of vertices. Thus E is just a set, not a multiset (though in a directed graph, $u \to v$ and $v \to u$ count as different edges).

It is often useful to consider graphs with satellite data attached to the edges or vertices (or both). One common instance of this is a **weighted undirected graph**, which is an undirected graph equipped with a weight function $w : E \to \mathbb{R}$. We say that the edge $e = \{u, v\}$ has *weight* $w(e)$, or sometimes w_{uv}. For example, let V be the set of cities in the United States, and let E be a **complete graph** on V, meaning that E contains every pair of vertices $\{u, v\}$ with $u \neq v$ (but no loops). Let $w(u, v)$ be the distance in miles between u and v. The graph $G = (V, E, w)$ is obviously of great importance to airline booking companies, shipping companies, and presidential candidates in the last few months of compaigning.

Consider a fixed set V of vertices. A **tree** T on V is a graph which is acyclic and connected.[2] (See Figure 3.2.) Note that acyclic is a smallness condition, while connectivity is a largeness condition. Thus, trees are "just the right size." According to the following proposition, that size is $|V| - 1$.

[1] In fact, we will probably use several other notations, depending on our whim; in particular, we will sometimes use the notation of directed graphs in discussions of undirected graphs. It is the author's job to keep clear the meaning of the notation—you should speak up if you are confused by any notation, since it is more likely the author's fault than yours.

[2] A **path** in a graph (V, E) is a sequence of vertices $\langle v_0, v_1, \ldots, v_n \rangle$, where $(v_j \to v_{j+1}) \in E$ for $j = 0, \ldots, n-1$. The number n is called the *length* of the path. If $v_0 = v_n$, then the path is called a **cycle**. A graph is called **acyclic** if it contains no cycles, and is said to be **connected** if, for every pair u, v of vertices, there exists a path from u to v.

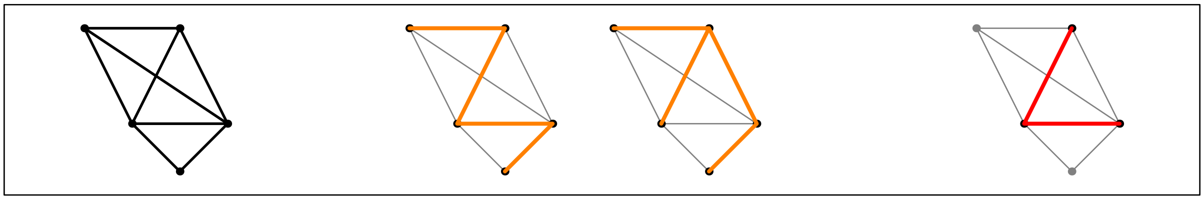


Figure 3.2. Left: An undirected graph G. Center: Two different spanning trees of G. Right: A subgraph of G which is a tree but not a spanning tree. The gray vertices and edges are just visual aids; they are not part of the graphs.

Proposition 3.1 (Theorem B.2 of CLRS, slightly modified)**.** *Let V be a finite set of vertices, and let $G = (V, T)$ be an undirected graph on V. The following conditions are equivalent:*

(i) G is acyclic and connected.
(ii) G is acyclic and has at least $|V| - 1$ edges.
(iii) G is connected and has at most $|V| - 1$ edges.
(iv) G is acyclic, but if any edge is added to T, the resulting graph contains a cycle.
(v) G is connected, but if any edge is removed from T, the resulting graph is not connected.
(vi) For any two vertices $u, v \in V$, there exists a unique simple path from u to v.[3]

Any one of these six conditions could be taken as the definition of a tree. If G is a tree, then G has exactly $|V| - 1$ edges.

We will not prove the proposition here, but you would do yourself a service to write the proof yourself (or look it up in CLRS), especially if you have not seen proofs in graph theory before. There are certain types of arguments that occur again and again in graph theory, and you will definitely want to know them for the remainder of the course.

Definition. Let $G = (V, E)$ be an undirected graph. A **spanning tree** of G is a subset $T \subseteq E$ such that T is a tree on V.

The reason for the word "spanning" is that T must be a tree on all of V. There are plenty of subgraphs of G that are trees but not spanning trees: they are graphs of the form $G' = (V', T')$ where $V' \subsetneq V$ and T' is a tree on V' (but not on V because it does not touch all the vertices).

Proposition 3.2. *Let $G = (V, E)$ be an undirected graph. Then G has a spanning tree if and only if G is connected. Thus, every connected graph on V has at least $|V| - 1$ edges.*

Proof. If G has a spanning tree T, then for any pair of vertices u, v there is a path from u to v in T. That path also lies in E since $T \subseteq E$. Thus G is connected.

Conversely, suppose G is connected but is not a tree. Then, by Proposition 3.1(v), we can remove an edge from E, obtaining a connected subgraph E_1. If E_1 is not a tree, then we can repeat the process again. Eventually we will come upon a connected subgraph (V, E_k) of G such that it is impossible to remove an edge from E_k, either because we have run out of edges, or because the resulting graph will be disconnected. In the former case, V must have only one vertex, so the empty set of edges is a spanning tree. In the latter case, E_k is a spanning tree by Proposition 3.1(v). It follows from Proposition 3.1 that G has at least $|V| - 1$ edges, since any spanning tree has exactly $|V| - 1$ edges. □

[3] A path is said to be **simple** if all vertices in the path are distinct. A cycle is said to be **simple** if all vertices are distinct except for the common start/end vertex. Thus, strictly speaking, a simple cycle is not a simple path; extra clarifications will be made in any situation where there might be confusion.

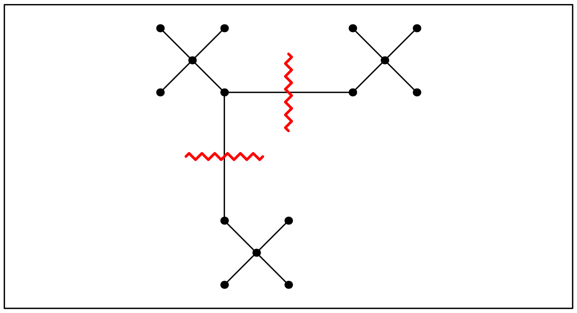


Figure 3.3. Clustering using an MST.

3.3 Minimum Spanning Trees

Given a weighted undirected graph $G = (V, E, w)$, one often wants to find a **minimum spanning tree** (MST) of G: a spanning tree T for which the total weight $w(T) = \sum_{(u,v)\in T} w(u,v)$ is minimal.

Input: A connected, undirected weighted graph $G = (V, E, w)$

Output: A spanning tree T such that the total weight

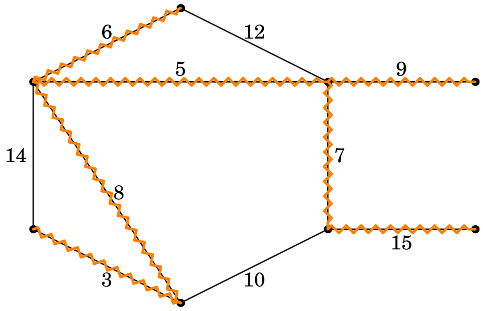
$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

is minimal.

For example, if V is the set of buildings in a city and E is the complete graph on V, and if $w(u,v)$ is the distance between u and v, then a minimum spanning tree would be very useful in building a minimum-length fiber-optic network, pipeline network, or other infrastructure for the city.

There are many other, less obvious applications of minimum spanning trees. Given a set of data points on which we have defined a metric (i.e., some way of quantifying how close together or similar a pair of vertices are), we can use an MST to cluster the data by starting with an MST for the distance graph and then deleting the heaviest edges. (See Figure 3.3 and Lecture 21.) If V is a set of data fields and the distance is mutual information, then this clustering can be used to find higher-order correlative relationships in a large data set.

How would we go about finding an MST for this graph?



There are several sensible heuristics:

- *Avoid large weights.* We would like to avoid the 14 and 15 edges if at all possible.

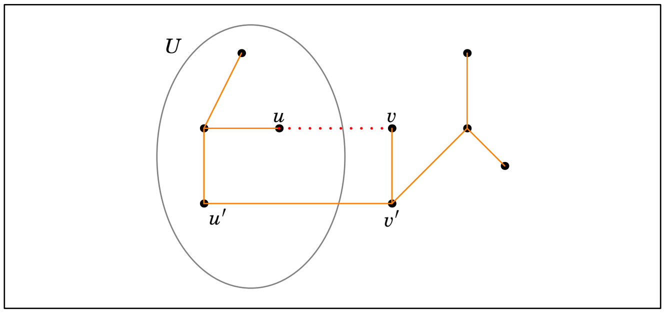


Figure 3.4. Illustration of the MST property.

- *Include small weights.* If the 3 and 5 edges provide any benefit to us, we will probably want to include them.
- *Some edges are inevitable.* The 9 and 15 edges must be included in order for the graph to be connected.
- *Avoid cycles*, since an MST is not allowed to have cycles.

Some more thoughts:

- Would a greedy algorithm be likely to work here? Is there something about the structure of the problem which allows locally optimal decisions to be informed enough about the global solution?
- Should we start with all the edges present and then remove some of them, or should we start with no edges present and then add them?
- Is the MST unique? Or are there multiple solutions?

A key observation to make is the following: If we add an edge to an MST, the resulting graph will have a cycle, by Proposition 3.1(iv). If we then remove one of the edges in this cycle, the resulting graph will be connected and therefore (by Proposition 3.1(iii)) will be a spanning tree. If we have some way of knowing that the edge we removed is at least as heavy as the edge we added, then it follows that we will have a minimum spanning tree. This is the idea underlying the so-called "MST property."

Theorem 3.3 (MST Property)**.** *Let $G = (V,E,w)$ be a connected, weighted, undirected graph. Let U be a proper nonempty subset of V.*[4] *Let S be the set of edges (x,y) with $x \in U$ and $y \in V \setminus U$. Suppose (u,v) is the lightest edge (or one of the lightest edges) in S. Then there exists an MST containing (u,v).*

This partition of V into U and $V \setminus U$ is called a "**cut**." An edge is said to "cross" the cut if one endpoint is in U and the other endpoint is in $V \setminus U$. Otherwise the edge is said to "respect" the cut. So S is the set of edges which cross the cut and $E \setminus S$ is the set of edges that respect the cut. An edge in S of minimal weight is called a "light edge" for the cut.

[4] That is, $\emptyset \subsetneqq U \subsetneqq V$.

Proof. Let $T \subseteq E$ be an MST for G, and suppose $(u,v) \notin T$. If we add the edge (u,v) to T, the resulting graph T' has a cycle. This cycle must cross the cut $(U, V \setminus U)$ in an even number of places, so in addition to (u,v), there must be some other edge (u',v') in the cycle, such that (u',v') crosses the cut. Remove (u',v') from T' and call the resulting graph T''. We showed above that T'' is a spanning tree. Also, since (u,v) is a light edge, we have $w(u,v) \leq w(u',v')$. Thus

$$w(T'') = w(T) + w(u,v) - w(u',v') \leq w(T).$$

Since T is a minimum spanning tree and T'' is a spanning tree, we also have $w(T) \leq w(T'')$. Therefore $w(T) = w(T'')$ and T'' is a minimum spanning tree which contains (u,v). □

Corollary 3.4. *Preserve the setup of the MST property. Let T be any MST. Then there exists an edge $(u',v') \in T$ such that $u' \in U$ and $v' \in V \setminus U$ and $\left(T \setminus \{(u',v')\}\right) \cup \{(u,v)\}$ is an MST.*

Proof. In the proof of the MST property, $T'' = \left(T \setminus \{(u',v')\}\right) \cup \{(u,v)\}$. □

Corollary 3.5. *Let $G = (V,E,w)$ be a connected, weighted, undirected graph. Let T be any MST and let $(U, V \setminus U)$ be any cut. Then T contains a light edge for the cut. If the edge weights of G are distinct, then G has a unique MST.*

Proof. If T does not contain a light edge for the cut, then the graph T'' constructed in the proof of the MST property weighs strictly less than T, which is impossible.

Suppose the edge weights in G are distinct. Let M be an MST. For each edge $(u,v) \in M$, consider the graph $(V, M \setminus \{(u,v)\})$. It has two connected components[5]; let U be one of them. The only edge in M that crosses the cut $(U, V \setminus U)$ is (u,v). Since M must contain a light edge, it follows that (u,v) is a light edge for this cut. Since G has distinct edge weights, (u,v) is the unique light edge for the cut, and every MST must contain (u,v). Letting (u,v) vary over all edges of M, we find that every MST must contain M. Thus, every MST must equal M. □

Now do you imagine that a greedy algorithm might work? The MST property is the crucial idea that allows us to use local information to decide which edges belong in an MST. Below is **Kruskal's algorithm**, which solves the MST problem.

Algorithm: KRUSKAL-MST(V,E,w)

Initially, let $T \leftarrow \emptyset$ be the empty graph on V.

Examine the edges in E in increasing order of weight (break ties arbitrarily).

- If an edge connects two unconnected components of T, then add the edge to T.*
- Else, discard the edge and continue.

Terminate when there is only one connected component. (Or, you can continue through all the edges.)

* By "two unconnected components," we mean two distinct connected components. Please forgive me for this egregious abuse of language.

[5] Given a graph G, we say a vertex v is **reachable** from u if there exists a path from u to v. For an undirected graph, reachability is an equivalence relation on the vertices. The restrictions of G to each equivalence class are called the **connected components** of G. (The **restriction** of a graph $G = (V,E)$ to a subset $V' \subseteq V$ is the subgraph (V',E'), where E' is the set of edges whose endpoints are both in V'.) Thus each connected component is a connected subgraph, and a connected undirected graph has only one connected component.

Try performing Kruskal's algorithm by hand on our example graph.

Proof of correctness for Kruskal's algorithm. We will use the following loop invariant:

> Prior to each iteration, every edge in T is a subset of an MST.

- *Initialization.* T has no edges, so of course it is a subset of an MST.
- *Maintenance.* Suppose $T \subseteq T^*$ where T^* is an MST, and suppose the edge (u,v) gets added to T. Then, as we will show in the next paragraph, (u,v) is a light edge for the cut $(U, V \setminus U)$, where U is one of the connected components of T. Therefore, by Corollary 3.4, there exist vertices $u' \in U$ and $v' \in V \setminus U$ with $(u',v') \in T^*$ such that T' is an MST, where $T' = \left(T^* \setminus \{(u',v')\}\right) \cup \{(u,v)\}$. But $(u',v') \notin T$, so $T \cup \{(u,v)\}$ is a subset of T'.

  As promised, we show that (u,v) is a light edge for the cut $(U, V \setminus U)$, where U is one of the connected components of T. The reasoning is this: If (u,v) is added, then it joins two unconnected components of T. Let U be one of those components. Now suppose that $(x,y) \in E$ were some lighter edge crossing the cut $(U, V \setminus U)$, say $x \in U$ and $y \in V \setminus U$. Since x and y lie in different connected components of T, the edge (x,y) is not in T. Thus, when we examined (x,y), we decided not to add it to T. This means that, when we examined the edge (x,y), x and y lay in the same connected component of T_{prev}, where T_{prev} is what T used to be at that point in time. But $T_{\text{prev}} \subseteq T$, which means that x and y lie in the same connected component of T. This contradicts the fact that $x \in U$ and $y \in V \setminus U$.
- *Termination.* At termination, T must be connected: If T were disconnected, then there would exist an edge of E which joins two components of T. But that edge would have been added to T when we examined it, a contradiction. Now, since T is connected and is a subset of an MST, it follows that T is an MST. □

To implement Kruskal's algorithm and analyze its running time, we will first need to think about the data structures that will be used and the running time of operations on them.

Exercise 3.1. *What if we wanted a maximum spanning tree rather than a minimum spanning tree?*

Lecture 4

Minimum Spanning Trees II

Supplemental reading in CLRS: Chapter 23; Section 16.2; Chapters 6 and 21

4.1 Implementing Kruskal's Algorithm

In the previous lecture, we outlined Kruskal's algorithm for finding an MST in a connected, weighted undirected graph $G = (V, E, w)$:

> Initially, let $T \leftarrow \emptyset$ be the empty graph on V.
> Examine the edges in E in increasing order of weight (break ties arbitrarily).
> - If an edge connects two unconnected components of T, then add the edge to T.
> - Else, discard the edge and continue.
>
> Terminate when there is only one connected component. (Or, you can continue through all the edges.)

Before we can write a pseudocode implementation of the algorithm, we will need to think about the data structures involved. When building up the subgraph T, we need to somehow keep track of the connected components of T. For our purposes it suffices to know which vertices are in each connected component, so the relevant information is a partition of V. Each time a new edge is added to T, two of the connected components merge. What we need is a disjoint-set data structure.

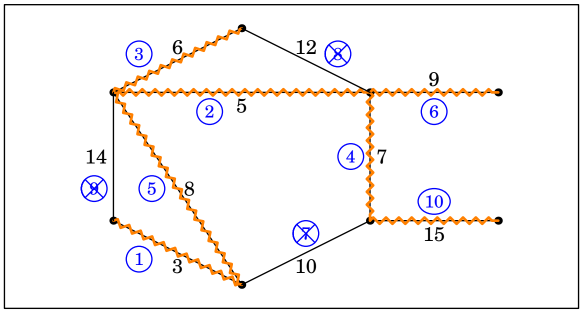


Figure 4.1. Illustration of Kruskal's algorithm.

4.1.1 Disjoint-Set Data Structure

A **disjoint-set data structure** maintains a dynamic collection of pairwise disjoint sets $\mathbf{S} = \{S_1, \ldots, S_r\}$ in which each set S_i has one representative element, rep$[S_i]$. Its supported operations are

- MAKE-SET(u): Create new set containing the single element u.
  - u must not belong to any already existing set
  - of course, u will be the representative element initially
- FIND-SET(u): Return the representative rep$[S_u]$ of the set S_u containing u.
- UNION(u,v): Replace S_u and S_v with $S_u \cup S_v$ in $\mathbf{S}$. Update the representative element.

4.1.2 Implementation of Kruskal's Algorithm

Equipped with a disjoint set data structure, we can implement Kruskal's algorithm as follows:

```
Algorithm: KRUSKAL-MST($V,E,w$)
1  ▷ Initialization and setup
2  $T \leftarrow \emptyset$
3  for each vertex $v \in V$ do
4      MAKE-SET($v$)
5  Sort the edges in $E$ into non-decreasing order of weight
6  ▷ Main loop
7  for each edge $(u,v) \in E$ in non-decreasing order of weight do
8      if FIND-SET($u$) $\neq$ FIND-SET($v$) then
9          $T \leftarrow T \cup \{(u,v)\}$
10         UNION($u,v$)
11 return $T$
```

The running time of this algorithm depends on the implementation of the disjoint set data structure we use. If the disjoint set operations have running times $T_{\text{MAKE-SET}}$, T_{UNION} and $T_{\text{FIND-SET}}$, and if we use a good $O(n \lg n)$ sorting algorithm to sort E, then the running time is

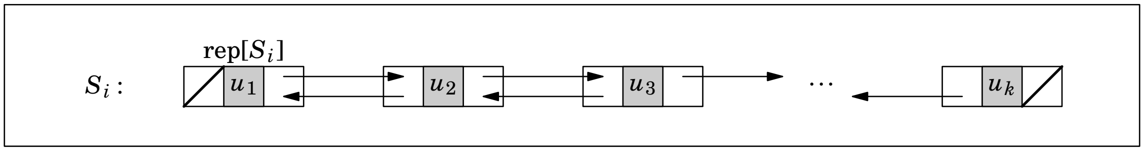
$$O(1) + V T_{\text{MAKE-SET}} + O(E \lg E) + 2E T_{\text{FIND-SET}} + O(E) + E T_{\text{UNION}}.$$[1]

4.1.3 Implementations of Disjoint-Set Data Structure

The two most common implementations of the disjoint-set data structure are (1) a collection of doubly linked lists and (2) a forest of balanced trees. In what follows, n denotes the total number of elements, i.e., $n = |S_1| + \cdots + |S_r|$.

Solution 1: Doubly-linked lists. Represent each set S_i as a doubly-linked list, where each element is equipped with a pointer to its two neighbors, except for the leftmost element which has a "stop" marker on the left and the rightmost element which has a "stop" marker on the right. We'll take the leftmost element of S_i as its representative.

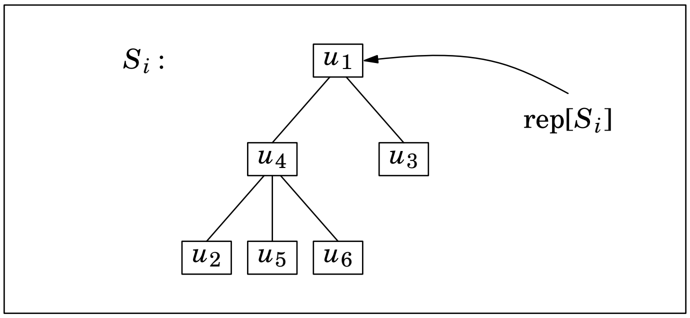
[1] Actually, we can do better. In line 10, since we have already computed rep$[S_u]$ and rep$[S_v]$, we do not need to call UNION; we need only call WEAK-UNION, an operation which merges two sets assuming that it has been given the correct representative of each set. So we can replace the $E T_{\text{UNION}}$ term with $E T_{\text{WEAK-UNION}}$.



MAKE-SET(u)	–	initialize as a lone node	$\Theta(1)$
FIND-SET(u)	–	walk left from u until you reach the head	$\Theta(n)$ worst-case
UNION(u,v)	–	walk right from u and left from v. Reassign pointers so that the tail of S_u and the head of S_v become neighbors. The representative is updated automatically.	$\Theta(n)$ worst-case

These can be improved upon—there exist better doubly-linked list implementations of the disjoint set data structure.

Solution 2: Forest of balanced trees.[2]



MAKE-SET(u)	–	initialize new tree with root node u	$\Theta(1)$
FIND-SET(u)	–	walk up tree from u to root	$\Theta(\text{height}) = \Theta(\lg n)$ best-case
UNION(u,v)	–	change rep[S_v]'s parent to rep[S_u]	$O(1) + 2T_{\text{FIND-SET}}$

The forest of balanced trees will be our implementation of choice. With a couple of clever tricks[3], the running times of the operations can be greatly improved: In the worst case, the improved structure has an amortized (average) running time of $\Theta(\alpha(n))$ per operation[4], where $\alpha(n)$ is the inverse Ackermann function, which is technically unbounded but for all practical purposes should be considered bounded.[5] So in essence, each disjoint set operation takes constant time, on average.

[2] A **rooted tree** is a tree with one distinguished vertex u, called the root. By Proposition 3.1(vi), for each vertex v there exists a unique simple path from u to v. The length of that path is called the **depth** of v. It is common to draw rooted trees with the vertices arranged in rows, with the root on top, all vertices of depth 1 on the row below that, etc.

[3] The tricks are called union-by-rank and path compression. For more information, see Lecture 16.

[4] In a 1989 paper, Fredman and Saks proved that $\Theta(\alpha(n))$ is the optimal amortized running time.

[5] For example, if n is small enough that it could be written down by a collective effort of the entire human population before the sun became a red giant star and swallowed the earth, then $\alpha(n) \leq 4$.

In the analysis that follows, we will not use these optimizations. Instead, we will assume that Find-Set and Union both run in $\Theta(\lg n)$ time. The asymptotic running time of Kruskal-MST is not affected.

As we saw above, the running time of Kruskal-MST is

$$\begin{array}{ll} \text{Initialize:} & O(1) + V\,\overbrace{T_{\text{Make-Set}}}^{O(1)} + O(E\lg E) \\ \text{Loop:} & 2E\,\underbrace{T_{\text{Find-Set}}}_{O(\lg V)} + O(E) + E\,\underbrace{T_{\text{Union}}}_{O(\lg V)} \\ \hline & O(E\lg E) + 2O(E\lg V). \end{array}$$

Since there can only be at most V^2 edges, we have $\lg E \le 2\lg V$. Thus the running time of Kruskal's algorithm is $O(E\lg V)$, the same amount of time it would take just to sort the edges.

4.1.4 Safe Choices

Let's philosophize about Kruskal's algorithm a bit. When adding edges to T, we do not worry about whether T is connected until the end. Instead, we worry about making "safe choices." A **safe choice** is a greedy choice which, in addition to being locally optimal, is also part of some globally optimal solution. In our case, we took great care to make sure that at every stage, there existed some MST T^* such that $T \subseteq T^*$. If T is safe and $T \cup \{(u,v)\}$ is also safe, then we call (u,v) a "safe edge" for T. We have already done the heavy lifting with regard to safe edge choices; the following theorem serves as a partial recap.

Proposition 4.1 (CLRS Theorem 23.1)**.** *Let $G = (V,E,w)$ be a connected, weighted, undirected graph. Suppose A is a subset of some MST T. Suppose $(U, V \setminus U)$ is a cut of G that is respected by A, and that (u,v) is a light edge for this cut. Then (u,v) is a safe edge for A.*

Proof. In the notation of Corollary 3.4, the edge (u',v') does not lie in A because A respects the cut $(U, V \setminus U)$. Therefore $A \cup \{(u,v)\}$ is a subset of the MST $\big(T \setminus \{(u',v')\}\big) \cup \{(u,v)\}$. □

4.2 Prim's Algorithm

We now present a second MST algorithm: **Prim's algorithm**. Like Kruskal's algorithm, Prim's algorithm depends on a method of determining which greedy choices are safe. The method is to continually enlarge a single connected component by adjoining edges emanating from isolated vertices.[6]

Algorithm: Prim-MST(V,E,w)

```
Choose an arbitrary start vertex $s$
$C \leftarrow \{s\}$
$T \leftarrow \emptyset$
while $C$ is not the only connected component of $T$ do
    Select a light edge $(u,v)$ connecting $C$ to an isolated vertex $v$
    $T \leftarrow T \cup \{(u,v)\}$
    $C \leftarrow C \cup \{v\}$
return $T$
```

[6] An **isolated** vertex is a vertex which is not connected to any other vertices. Thus, an isolated vertex is the only vertex in its connected component.

Proof of correctness for Prim's algorithm. Again, we use a loop invariant:

> Prior to each iteration, T is a subset of an MST.

- *Initialization.* T has no edges, so trivially it is a subset of an MST.
- *Maintenance.* Suppose $T \subseteq T^*$ where T^* is an MST, and suppose the edge (u,v) gets added to T, where $u \in C$ and v is an isolated vertex. Since (u,v) is a light edge for the cut $(C, V \setminus C)$ which is respected by T, it follows by Proposition 4.1 that (u,v) is a safe edge for T. Thus $T \cup \{(u,v)\}$ is a subset of an MST.
- *Termination.* At termination, C is the only connected component of T, so by Proposition 3.1(v), T has at least $|V|-1$ edges. Since T is also a subset of an MST, it follows that T has exactly $|V|-1$ edges and is an MST. □

The tricky part of Kruskal's algorithm was keeping track of the connected components of T. In Prim's algorithm this is easy: except for the special component C, all components are isolated vertices. The tricky part of Prim's algorithm is efficiently keeping track of which edge is lightest among those which join C to a new isolated vertex. This task is typically accomplished with a data structure called a min-priority queue.

4.2.1 Min-Priority Queue

A **min-priority queue** is a data structure representing a collection of elements which supports the following operations:

> INSERT(Q,x) – Inserts element x into the set of elements Q
>
> MINIMUM(Q) – Returns the element of Q with the smallest key
>
> EXTRACT-MIN(Q) – Removes and returns the element with the smallest key
>
> DECREASE-KEY(Q,x,k) – Decreases the value of x's key to new value k.

With this data structure, Prim's algorithm could be implemented as follows:

Algorithm: PRIM-MST(G,s)

```
1  $T \leftarrow \emptyset$
2  **for each** $u \in G.V$ **do**
3      $u.key \leftarrow \infty$ ▷ initialize all edges to “very heavy”
4      ▷ The component $C$ will be a tree rooted at $s$. Once a vertex $u$ gets added to $C$, $u.\pi$
       will be a pointer to its parent in the tree.
5      $u.\pi \leftarrow$ NIL
6  $s.key \leftarrow 0$ ▷ this ensures that $s$ will be the first vertex we pick
7  Let $Q \leftarrow G.V$ be a min-priority queue
8  **while** $Q \neq \emptyset$ **do**
9      $u \leftarrow$ EXTRACT-MIN$(Q)$
10     **if** $u.\pi \neq$ NIL **then**
11         $T \leftarrow T \cup \{(u, u.\pi)\}$
12     ▷ We assume that $G$ is presented in the adjacency-list format*
13     **for each** $v \in G.adj[u]$ **do**
14         **if** $v \in Q$ and $w(u,v) < v.key$ **then**
15             $v.\pi \leftarrow u$
16             $v.key \leftarrow w(u,v)$ ▷ using DECREASE-KEY
17 **return** $T$
```

*For more information about ways to represent graphs on computers, see §22.1 of CLRS.

4.2.2 Running Time of Prim’s Algorithm

Lines 1 through 6 clearly take $O(V)$ time. Line 7 takes $T_{\text{BUILD-QUEUE}}(V)$, where $T_{\text{BUILD-QUEUE}}(n)$ is the amount of time required to build a min-priority queue from an array of n elements. Within the “while” loop of line 8, EXTRACT-MIN gets called $|V|$ times, and the instructions in lines 14–16 are run a total of $O(E)$ times. Thus the running time of Prim’s algorithm is

$$O(V) + T_{\text{BUILD-QUEUE}}(V) + V\,T_{\text{EXTRACT-MIN}} + O(E)\,T_{\text{DECREASE-KEY}}.$$

Exercise 4.1. *How can we structure our implementation so that line 14 runs in $O(1)$ time?*

Let’s take a look at the performance of PRIM-MST under some implementations of the min-priority queue, in increasing order of efficiency:

Q	$T_{\text{BUILD-QUEUE}}(n)$	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Runtime of PRIM-MST
Array	$O(n)$	$O(n)$	$O(1)$	$O(V^2)$
Binary Heap	$O(n)$	$O(\lg n)$	$O(\lg n)$	$O(E \lg V)$
Fibonacci Heap	$O(n)$	$\underbrace{O(\lg n)}_{\text{amortized}}$	$\underbrace{O(1)}_{\text{amortized}}$	$O(E + V \lg V)$

Thus, for a dense graph (a graph with $\Theta(V^2)$ edges), Prim’s algorithm with a Fibonacci heap outperforms Kruskal’s algorithm. The best known MST algorithm to date is a randomized algorithm with $\Theta(V+E)$ expected running time, introduced by Karger, Klein and Tarjan in 1995.

4.3 Greedy Strategies

General approach:

1. Structure the problem so that we make a choice and are left with one subproblem to solve.

2. Make a greedy choice and then prove that there exists an optimal solution to the original problem which makes the same greedy choice (“safe choice”).

3. Demonstrate optimal substructure.

    - After making the greedy choice, combine with the optimal solution of the remaining subproblem, giving an optimal solution to the original problem.

Note that this sounds a lot like dynamic programming. Let’s now examine the key properties of greedy algorithms in comparison to those of dynamic programming.

1) *Greedy Choice Property – Locally optimal solution leads to globally optimal solution.* Each greedy local choice is made independently of the solution to the subproblem. (E.g.: Kruskal’s and Prim’s algorithms can choose a safe edge without having examined the full problem.) In dynamic programming, the local choice depends on the solution to the subproblem—it is a bottom-up solution.

2) *Optimal Substructure – The optimal solution to a problem contains optimal solutions to subproblems.* Both greedy strategies and dynamic programming exploit (or rely on) optimal substructures. Prim’s algorithm produces (optimal) MSTs on subsets of V on the way to finding the full MST.

Lecture 5

Fast Fourier Transform

Supplemental reading in CLRS: Chapter 30

The algorithm in this lecture, known since the time of Gauss but popularized mainly by Cooley and Tukey in the 1960s, is an example of the **divide-and-conquer** paradigm. Actually, the main uses of the fast Fourier transform are much more ingenious than an ordinary divide-and-conquer strategy—there is genuinely novel mathematics happening in the background. Ultimately, the FFT will allow us to do n computations, each of which would take $\Omega(n)$ time individually, in a total of $\Theta(n \lg n)$ time.

5.1 Multiplication

To motivate the fast Fourier transform, let's start with a very basic question:

> How can we efficiently multiply two large numbers or polynomials?

As you probably learned in high school, one can use essentially the same method for both:

$$\begin{array}{r} 385 \\ \times 426 \\ \hline 2310 \\ 770 \\ +1540 \\ \hline 164010 \end{array} \qquad\qquad \begin{array}{r} (3x^2 + 8x + 5) \\ \times\ (4x^2 + 2x + 6) \\ \hline 18x^2 + 48x + 30 \\ 6x^3 + 16x^2 + 10x \phantom{{}+30} \\ 12x^4 + 32x^3 + 20x^2 \phantom{{}+10x+30} \\ \hline 12x^4 + 38x^3 + 54x^2 + 58x + 30 \end{array}$$

Of these two, polynomials are actually the easier ones to work with. One reason for this is that multiplication of integers requires carrying, while multiplication of polynomials does not. To make a full analysis of this extra cost, we would have to study the details of how large integers can be stored and manipulated in memory, which is somewhat complicated.[1] So instead, let's just consider multiplication of polynomials.

Suppose we are trying to multiply two polynomials p, q of degree at most n with complex coefficients. In the high-school multiplication algorithm (see Figure 5.1), each row of the diagram is

[1] More ideas are required to implement efficient multiplication of n-bit integers. In a 1971 paper, Schönhage and Strassen exhibited integer multiplication in $O(n \lg n \cdot \lg\lg n)$ time; in 2007, Fürer exhibited $O\left(n \lg n \cdot 2^{O(\lg^* n)}\right)$ time. The iterated logarithm $\lg^* n$ is the smallest k such that $\lg\lg\cdots\lg n$ (k times) ≤ 1. It is not known whether integer multiplication can be achieved in $O(n \lg n)$ time, whereas by the end of this lecture we will achieve multiplication of degree-n polynomials in $O(n \lg n)$ time.

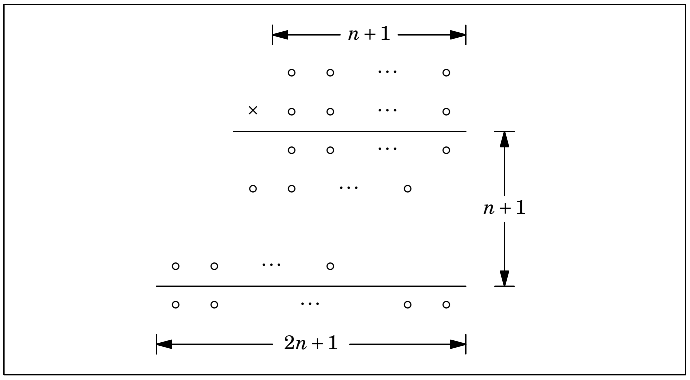


Figure 5.1. The high-school polynomial multiplication algorithm.

obtained by multiplying p by a monomial. Thus any single row would take $\Omega(n)$ time to compute individually. There are $n+1$ rows, so the computation takes $\Omega(n^2)$ time.

The key to improving this time is to consider alternative ways of representing p and q. Fix some $N > n$. We'll choose N later, but for now all that matters is that $N = O(n)$. A polynomial of degree at most $N-1$ is uniquely determined by its values at N points. So, instead of storing the coefficients of p and q, we could represent p and q as the lists

$$p_z = \big\langle p(z_0), \ldots, p(z_{N-1})\big\rangle \quad \text{and} \quad q_z \big\langle p(z_0), \ldots, p(z_{N-1})\big\rangle$$

for any distinct complex numbers $z_0, \ldots, z_{N-1}$. In this representation, computing pq is very cheap—the list

$$(pq)_z = \big\langle p(z_0)q(z_0), \ldots, p(z_{N-1})q(z_{N-1})\big\rangle$$

can be computed in $O(n)$ time. If $N > \deg pq$, then pq is the unique polynomial whose FFT is $(pq)_z$.

```
Algorithm: MULTIPLY($p, q$)
  1. Fix some $N = O(\deg p + \deg q)$ such that $N > \deg p + \deg q$.
  2. Compute the sample vectors $p_z = \langle p(z_0), \ldots, p(z_{N-1})\rangle$ and $q_z = \langle q(z_0), \ldots, q(z_{N-1})\rangle$.
  3. Compute the unique polynomial $r$ such that $r_z = p_z \cdot q_z$, where $\cdot$ denotes entrywise multiplication.*
  4. Return $r$.
```

* For example, $\langle 1,2,3\rangle \cdot \langle 4,5,6\rangle = \langle 4,10,18\rangle$.

Thus, the cost of finding the coefficients of pq is equal to the cost of converting back and forth between the coefficients representation and the sample-values representation (see Figure 5.2). We are free to choose values of $z_0, \ldots, z_{N-1}$ that make this conversion as efficient as possible.

5.1.1 Efficiently Computing the FFT

In the fast Fourier transform, we choose $z_0, \ldots, z_{N-1}$ to be the Nth roots of unity, $1, \omega_N, \omega_N^2, \ldots, \omega_N^{N-1}$, where $\omega_N = e^{2\pi i/N}$. We make this choice because roots of unity enjoy the useful property that

$$\omega_N^2 = \omega_{N/2}.$$

$$\begin{tikzcd} P_N \times P_N \arrow[r, squiggly, "\text{FFT}"] \arrow[d, "\substack{\text{Polynomial}\\ \text{mult. (slow)}}"'] & \mathbb{C}^N \times \mathbb{C}^N \arrow[d, "\substack{\text{Entrywise}\\ \text{mult. (fast)}}"] \\ P_N & \mathbb{C}^N \arrow[l, squiggly, "\text{FFT}^{-1}"'] \end{tikzcd}$$

Figure 5.2. Commutative diagram showing the cost of multiplication on either side of a fast Fourier transform. As we will see, the fastest way to get from the top-left to the bottom-left is through the FFT.

(We can choose N to be a power of 2.) This allows us to *divide* the task of computing an FFT of size N into computing two FFTs of size $N/2$, and then combining them in a certain way.[2]

```
Algorithm: FFT$\left(N,\ p(x) = a_{N-1}x^{N-1} + a_{N-2}x^{N-2} + \cdots + a_1 x + a_0\right)$
▷ As part of the specification, we assume $N$ is a power of 2
if $N = 1$ then
    return $a_0$
Let
    $p^{\text{even}}(y) \leftarrow a_0 + a_2 y + \cdots + a_{N-2} y^{N/2-1},$
    $p^{\text{odd}}(y) \leftarrow a_1 + a_3 y + \cdots + a_{N-1} y^{N/2-1}$
be the even and odd parts of $p$
▷ Thus
    $p(x) = p^{\text{even}}\left(x^2\right) + x p^{\text{odd}}\left(x^2\right)$    (5.1)
$p^{\text{even}}_{\omega_{N/2}} \leftarrow \text{FFT}(N/2,\ p^{\text{even}})$
$p^{\text{odd}}_{\omega_{N/2}} \leftarrow \text{FFT}\left(N/2,\ p^{\text{odd}}\right)$
▷ That is,
    $p^{\text{even}}_{\omega_{N/2}} = \left\langle p^{\text{even}}(1),\ p^{\text{even}}(\omega_{N/2}),\ \ldots,\ p^{\text{even}}\left(\omega_{N/2}^{N/2-1}\right)\right\rangle$
    $p^{\text{odd}}_{\omega_{N/2}} = \left\langle p^{\text{odd}}(1),\ p^{\text{odd}}(\omega_{N/2}),\ \ldots,\ p^{\text{odd}}\left(\omega_{N/2}^{N/2-1}\right)\right\rangle.$
▷ Because $\omega_{N/2} = \omega_N^2$, we can calculate the vector $p_{\omega_N} = \left\langle p(1),\ p(\omega_N),\ \ldots,\ p\left(\omega_N^{N-1}\right)\right\rangle$
  very quickly using (5.1):
$\boldsymbol{\omega} \leftarrow \left\langle 1,\ \omega_N,\ \ldots,\ \omega_N^{N-1}\right\rangle$    ▷ the left side is a bold omega
return $p^{\text{even}}_{\omega_{N/2}} + \boldsymbol{\omega} \cdot p^{\text{odd}}_{\omega_{N/2}}$, where $\cdot$ denotes entrywise multiplication
```

[2] I suppose the “conquer” stage is when we recursively compute the smaller FFTs (but of course, each of these smaller FFTs begins with its own “divide” stage, and so on). After the “conquer” stage, the answers to the smaller problems are combined into a solution to the original problem.

Above, we compute p_{ω_N} by computing $p^{\text{odd}}_{\omega_{N/2}}$ and $p^{\text{even}}_{\omega_{N/2}}$ and combining them in $\Theta(N)$ time. Thus, the running time of an FFT of size N satisfies the recurrence

$$T(N) = 2T(N/2) + \Theta(N).$$

This recurrence is solved in CLRS as part of the Master Theorem in §4.5. The solution is

$$T(N) = \Theta(N \lg N).$$

5.1.2 Computing the Inverse FFT

Somewhat surprisingly, the inverse FFT can be computed in almost exactly the same way as the FFT. In this section we will see the relation between the two transforms. If you don't have a background in linear algebra, you can take the math on faith.

Let P_N be the vector space of polynomials of degree at most $N-1$ with complex coefficients. Then the FFT is a bijective linear map $P_N \to \mathbb{C}^N$. If we use the ordered basis $1, x, \ldots, x^{N-1}$ for P_N and the standard basis for $\mathbb{C}^N$, then the matrix of the FFT is

$$A = \left(\omega_N^{ij}\right)_{0 \le i,j \le N-1} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_N & \omega_N^2 & \cdots & \omega_N^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_N^{N-1} & \omega_N^{2(N-1)} & \cdots & \omega_N^{(N-1)^2} \end{pmatrix}.$$

The jth column of A^{-1} contains the coefficients of the polynomial which, when evaluated on ω_N^j, gives 1 and, when evaluated on ω_N^i $(i \neq j)$, gives zero. This polynomial is

$$\frac{\prod_{i \neq j} \left(x - \omega_N^i\right)}{\prod_{i \neq j} \left(\omega_N^j - \omega_N^i\right)}.$$

We will not show the details here, but after about a page of calculation you can find that

$$A^{-1} = \tfrac{1}{N}\overline{A} = \frac{1}{N}\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_N^{-1} & \omega_N^{-2} & \cdots & \omega_N^{-(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_N^{-(N-1)} & \omega_N^{-2(N-1)} & \cdots & \omega_N^{-(N-1)^2} \end{pmatrix},$$

where the bar indicates entrywise complex conjugation. You can certainly check that the i,j entry of $\frac{1}{N}A\overline{A}$ is

$$\sum_{\ell=0}^{N-1} \omega_N^{i\ell} \cdot \tfrac{1}{N}\omega_N^{-j\ell} \quad = \quad \tfrac{1}{N}\sum_{\ell=0}^{N-1} \omega_N^{\ell(i-j)} \quad = \quad \begin{cases} 1, & i = j \\ 0, & i \neq j, \end{cases}$$

and similarly for $\frac{1}{N}\overline{A}A$. Thus

$$\boxed{\mathrm{FFT}^{-1}(\mathbf{v}) = \tfrac{1}{N}\overline{\mathrm{FFT}\left(\overline{\mathbf{v}}\right)}.} \tag{5.2}$$

Again, this is surprising. A priori, we would consider P_N and $\mathbb{C}^N$ to be "different worlds"—like two different implementations of the same abstract data structure, the coefficients representation and the sample-values representation of polynomials serve essentially the same role but have different procedures for performing operations. Yet, (5.2) tells us that inverting an FFT involves creating a new polynomial whose coefficients are the *sample values* of our original polynomial. This of course has everything to do with the fact that roots of unity are special; it would not have worked if we had not chosen $1, \omega_N, \ldots, \omega_N^{N-1}$ as our sample points.

```
Algorithm: INVERSE-FFT$(N,\ \mathbf{v} = \langle v_0, \ldots, v_{N-1} \rangle)$
$\overline{\mathbf{v}} \leftarrow \langle \overline{v_0}, \ldots, \overline{v_{N-1}} \rangle$
Let $p_{\overline{\mathbf{v}}}(x)$ be the polynomial $\overline{v_{N-1}} x^{N-1} + \cdots + \overline{v_1} x + \overline{v_0}$
return the value of
        $\frac{1}{N} \overline{\mathrm{FFT}(N,\ p_{\overline{\mathbf{v}}})}$
```

With all the details in place, the MULTIPLY algorithm looks like this:

```
Algorithm: FFT-MULTIPLY$(p, q)$
  1. Let $N$ be the smallest power of 2 such that $N - 1 \geq \deg p + \deg q$.
  2. Compute $\mathbf{v} = \mathrm{FFT}(N, p)$ and $\mathbf{w} = \mathrm{FFT}(N, q)$, and let $\mathbf{u} = \mathbf{v} \cdot \mathbf{w}$, where $\cdot$ denotes entry-
     wise multiplication.
  3. Compute and return INVERSE-FFT$(N,\ \mathbf{u})$.
```

It runs in $\Theta(N \lg N) = \Theta\big((\deg p + \deg q) \lg(\deg p + \deg q)\big)$ time.

5.2 Convolution

Let $p(x) = a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ and $q(x) = b_{n-1} x^{n-1} + \cdots + b_1 x + b_0$. The coefficient of x^k in pq is

$$\sum_{\ell \in \mathbb{Z}_{2n-1}} a_\ell b_{k-\ell},$$

where the subscripts are interpreted modulo $2n-1$. This sort of operation is known as **convolution**, and is written as $*$. Convolution can be applied quite generally: if f and g are any functions $\mathbb{Z}_{2n-1} \to \mathbb{C}$, then one common definition of the convolution operation is

$$(f * g)(k) = \frac{1}{2n-1} \sum_{\ell \in \mathbb{Z}_{2n-1}} f(\ell) g(k - \ell).$$

For each i, computing $(f * g)(i)$ requires linear time, but the fast Fourier transform allows us to compute $(f * g)(i)$ for all $i = 0, \ldots, 2n-2$ in a total of $\Theta(n \lg n)$ time.

Convolution appears frequently, which is part of the reason that the FFT is useful. The above notion of convolution can easily be generalized to allow f and g to be functions from any group G to any ring in which $|G|$ is a unit. There is also a continuous version of convolution which involves an integral. This continuous version is useful in signal compression. The idea is to let $f : \mathbb{R} \to \mathbb{R}$ be an arbitrary signal and let g be a smooth bump function which spikes up near the origin and is zero elsewhere. (See Figure 5.3.) Then the convolution of f and g is a new function $f * g$ defined by

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau)\, d\tau.$$

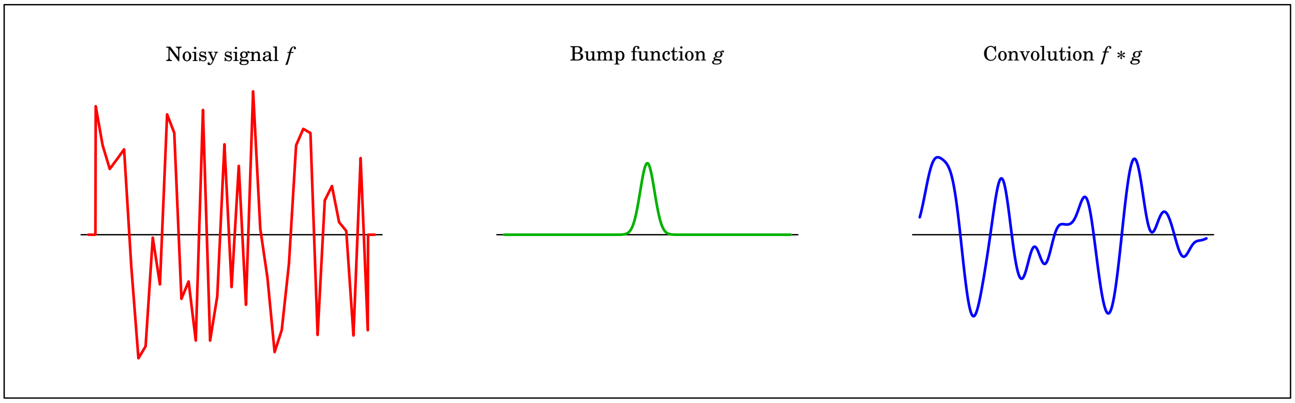


Figure 5.3. A complicated, noisy signal f, a bump function g, and the convolution $f * g$.

The value $(f * g)(t)$ is essentially an average of the values of f at points close to t. Thus, the funtion $f * g$ inherits smoothness from g while still carrying most of the information from f. As a bonus, if f comes with some unwanted random noise, then g will have much less noise. For example, f may be the raw output of a recording device, and computing $f * g$ may be the first step towards encoding f into a compressed audio format such as `MP3`. The point is that smooth functions are much easier to describe concisely, so creating a smooth version of f is useful in (lossy) data compression.

Lecture 6

All-Pairs Shortest Paths I

Supplemental reading in CLRS: Chapter 25 intro; Section 25.2

6.1 Dynamic Programming

Like the greedy and divide-and-conquer paradigms, **dynamic programming** is an algorithmic paradigm in which one solves a problem by combining the solutions to smaller subproblems. In dynamic programming, the subproblems overlap, and the solutions to "inner" problems are stored in memory (see Figure 6.1). This avoids the work of repeatedly solving the innermost problem. Dynamic programming is often used in optimization problems (e.g., finding the maximum or minimum solution to a problem).

- The key feature that a problem must have in order to be amenable to dynamic programming is that of **optimal substructure**: the optimal solution to the problem must contain optimal solutions to subproblems.
- Greedy algorithms are similar to dynamic programming, except that in greedy algorithms, the solution to an inner subproblem does not affect the way in which that solution is augmented to the solution of the full problem. In dynamic programming the combining process is more sophisticated, and depends on the solution to the inner problem.
- In divide-and-conquer, the subproblems are disjoint.

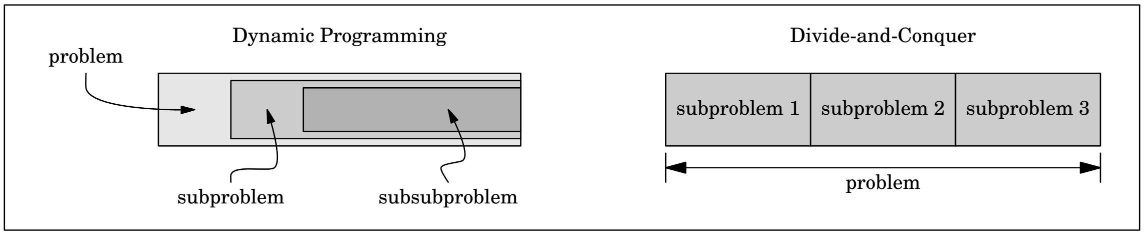


Figure 6.1. Schematic diagrams for dynammic programming and divide-and-conquer.

6.2 Shortest-Path Problems

Given a directed graph $G = (V, E, w)$ with real-valued edge weights, it is very natural to ask what the shortest (i.e., lightest) path between two vertices is.

Input: A weighted, directed graph $G = (V, E, w)$ with real-valued edge weights, a start vertex u, and an end vertex v

Output: The minimal weight of a path from u to v. That is, the value of

$$\delta(u,v) = \begin{cases} \min\left\{w(p) : \text{ paths } u \overset{p}{\rightsquigarrow} v\right\} & \text{if such a path exists} \\ \infty & \text{otherwise,} \end{cases}$$

where the weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

(Often we will also want an example of a path which achieves this minimal weight.)

Shortest paths exhibit an optimal-substructure property:

Proposition 6.1 (Theorem 24.1 of CLRS)**.** *Let $G = (V, E, w)$ be a weighted, directed graph with real-valued edge weights. Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from v_0 to v_k. Let i, j be indices with $0 \le i \le j \le k$, and let p_{ij} be the subpath $\langle v_i, v_{i+1}, \ldots, v_j \rangle$. Then p_{ij} is a shortest path from v_i to v_j.*

Proof. We are given that

$$p : v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

is a shortest path. If there were a shorter path from v_i to v_j, say $v_i \overset{p'_{ij}}{\rightsquigarrow} v_j$, then we could patch it in to obtain a shorter path from v_0 to v_k:

$$p' : v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k \ ,$$

which is impossible. □

6.2.1 All Pairs

In recitation, we saw Dijkstra's algorithm (a greedy algorithm) for finding all shortest paths from a single source, for graphs with nonnegative edge weights. In this lecture, we will solve the problem of finding the shortest paths between all pairs of vertices. This information is useful in many contexts, such as routing tables for courier services, airlines, navigation software, Internet traffic, etc.

The simplest way to solve the all-pairs shortest path problem is to run Dijkstra's algorithm $|V|$ times, once with each vertex as the source. This would take time $|V| \cdot T_{\text{DIJKSTRA}}$, which, depending on the implementation of the min-priority queue data structure, would be

Linear array:	$O\left(V^3 + VE\right) = O\left(V^3\right)$
Binary min-heap:	$O\left(VE \lg V\right)$
Fibonacci heap:	$O\left(V^2 \lg V + VE\right)$.

However, Dijkstra's algorithm only works if the edge weights are nonnegative. If we wanted to allow negative edge weights, we could instead use the slower Bellman–Ford algorithm once per vertex:

$$O\left(V^2 E\right) \xrightarrow{\text{dense graph}} O\left(V^4\right).$$

We had better be able to beat this!

6.2.2 Formulating the All-Pairs Problem

When we solved the single-source shortest paths problem, the shortest paths were represented on the actual graph, which was possible because subpaths of shortest paths are shortest paths and because there was only one source. This time, since we are considering all possible source vertices at once, it is difficult to conceive of a graphical representation. Instead, we will use matrices.

Let $n = |V|$, and arbitrarily label the vertices $1, 2, \ldots, n$. We will format the output of the all-pairs algorithm as an $n \times n$ distance matrix $D = \left(d_{ij}\right)$, where

$$d_{ij} = \delta(i,j) = \begin{cases} \text{weight of a shortest path from } i \text{ to } j & \text{if a path exists} \\ \infty & \text{if no path exists,} \end{cases}$$

together with an $n \times n$ predecessor matrix $\Pi = \left(\pi_{ij}\right)$, where

$$\pi_{ij} = \begin{cases} \text{NIL}, & \text{if } i = j \text{ or there is no path from } i \text{ to } j \\ \text{predecessor of } j \text{ in our shortest path } i \rightsquigarrow j & \text{otherwise.} \end{cases}$$

Thus, the ith row of Π represents the sinlge-source shortest paths starting from vertex i, and the ith row of D gives the weights of these paths. We can also define the weight matrix $W = \left(w_{ij}\right)$, where

$$w_{ij} = \begin{cases} \text{weight of edge } (i,j) & \text{if edge exists} \\ \infty & \text{otherwise.} \end{cases}$$

Because G is a directed graph, the matrices D, Π and W are not necessarily symmetric. The existence of a path from i to j does not tell us anything about the existence of a path from j to i.

6.3 The Floyd–Warshall Algorithm

The Floyd–Warshall algorithm solves the all-pairs shortest path problem in $\Theta(V^3)$ time. It allows negative edge weights, but assumes that there are no cycles with negative total weight.[1] The Floyd–Warshall algorithm uses dynamic programming based on the following subproblem:

> What are D and Π if we require all paths to have intermediate vertices[2] taken from the set $\{1, \ldots, k\}$, rather than the full $\{1, \ldots, n\}$?

[1] If there were a negative-weight cycle, then there would exist paths of arbitrarily low weight, so some vertex pairs would have a distance of $-\infty$. In particular, the distance from a vertex to itself would not necessarily be zero, so our algorithms would need some emendations.

[2] An *intermediate vertex* in a path $p = \langle v_0, \ldots, v_n \rangle$ is one of the vertices $v_1, \ldots, v_{n-1}$, i.e., not an endpoint.

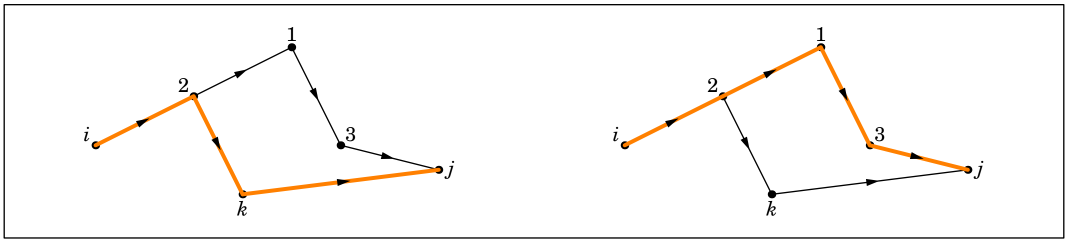


Figure 6.2. Either p passes through vertex k, or p doesn't pass through vertex k.

As k increases from 0 to n, we build up the solution to the all-pairs shortest path problem. The base case $k = 0$ (where no intermediate vertices are allowed) is easy:

$$D^{(0)} = \left(d_{ij}^{(0)}\right) \qquad \text{and} \qquad \Pi^{(0)} = \left(\pi_{ij}^{(0)}\right),$$

where

$$d_{ij}^{(0)} = \left\{\begin{matrix} 0 & \text{if } i = j \\ w_{ij} & \text{otherwise} \end{matrix}\right\} \qquad \text{and} \qquad \pi_{ij}^{(0)} = \left\{\begin{matrix} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty \\ i & \text{if } i \neq j \text{ and there is an edge from } i \text{ to } j \end{matrix}\right\}.$$

In general, let $D^{(k)}$ and $\Pi^{(k)}$ be the corresponding matrices when paths are required to have all their intermediate vertices taken from the set $\{1, \ldots, k\}$. (Thus $D = D^{(n)}$ and $\Pi = \Pi^{(n)}$.) What we need is a way of computing $D^{(k)}$ and $\Pi^{(k)}$ from $D^{(k-1)}$ and $\Pi^{(k-1)}$. The following observation provides just that.

Observation. Let p be a shortest path from i to j when we require all intermediate vertices to come from $\{1, \ldots, k\}$. Then either k is an intermediate vertex or k is not an intermediate vertex. Thus, one of the following:

1. k is not an intermediate vertex of p. Thus, p is also a shortest path from i to j when we require all intermediate vertices to come from $\{1, \ldots, k-1\}$.

2. k is an intermediate vertex for p. Thus, we can decompose p as

$$p : \ i \overset{p_{ik}}{\rightsquigarrow} k \overset{p_{kj}}{\rightsquigarrow} j\ ,$$

   where p_{ik} and p_{kj} are shortest paths when we require all intermediate vertices to come from $\{1, \ldots, k-1\}$, by (a slightly modified version of) Proposition 6.1.

This tells us precisely that

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1. \end{cases}$$

Thus we may compute D by the following procedure:

Algorithm: FLOYD–WARSHALL(W)

```
1  $n \leftarrow |W.rows|$
2  $D^{(0)} \leftarrow W$
3  $\Pi^{(0)} \leftarrow$ an $n \times n$ matrix with all entries initially NIL
4  for $k \leftarrow 1$ to $n$ do
5      Let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ and $\Pi^{(k)} = \left(\pi_{ij}^{(k)}\right)$ be new $n \times n$ matrices
6      for $i \leftarrow 1$ to $n$ do
7          for $j \leftarrow 1$ to $n$ do
8              $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
9              if $d_{ij}^{(k)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ then
10                 $\pi_{ij}^{(k)} \leftarrow \pi_{ij}^{(k-1)}$
11             else
12                 $\pi_{ij}^{(k)} \leftarrow \pi_{kj}^{(k-1)}$
```

Being in a triply nested loop, lines 8–12 are executed n^3 times. Thus, the algorithm runs in $\Theta\left(n^3\right) = \Theta\left(V^3\right)$ time. Note that while the algorithm as written requires $\Theta\left(V^3\right)$ space, reasonable implementations will only require $\Theta\left(V^2\right)$ space: we only ever need to memorize four matrices at a time, namely $D^{(k)}$, $D^{(k-1)}$, $\Pi^{(k)}$, $\Pi^{(k-1)}$. In fact, we can get away with memorizing even less—see Exercise 25.2-4 of CLRS.

Lecture 7

All-Pairs Shortest Paths II

Supplemental reading in CLRS: Section 25.3

7.1 Johnson's Algorithm

The Floyd–Warshall algorithm runs in $\Theta\left(V^3\right)$ time. Recall that, if all edge weights are nonnegative, then repeated application of Dijkstra's algorithm using a Fibonacci heap gives the all-pairs shortest paths in $\Theta\left(V^2 \lg V + VE\right)$ time. If G is not dense (i.e., if $|E|$ is not on the order of $|V|^2$), then Dijkstra's algorithm asymptotically outperforms the Floyd–Warshall algorithm. So our goal for the first half of this lecture will be to reduce the problem of finding all-pairs shortest paths on an arbitrary graph to that of finding all-pairs shortest paths on a graph with nonnegative edge weights.

7.1.1 Reweighting

How might we "eliminate" negative edge weights? The naïve way would be to add some constant N to all the edge weights, where N is chosen large enough to make all the edge weights nonnegative. Unfortunately, this transformation does not preserve shortest paths. Given a path $p : \langle u = v_0,\ v_1, \ldots, v_k = v \rangle$ from u to v with total weight $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$, the weight of p in the new graph would be $w(p) + kN$. Thus, a very light path in the original graph might not be so attractive in the new graph, if it contained lots of edges (see Figure 7.1).

We will need to do something smarter; the change in weight must vary from edge to edge. Our goal is to assign to each edge (u, v) a new weight $\widehat{w}(u, v)$, such that the new weight function $\widehat{w}$ has the following two properties:

1. For each pair of vertices $u, v \in V$ and each path $p : u \rightsquigarrow v$, the old weight $w(p)$ is minimal if and only if the new weight $\widehat{w}(p)$ is minimal (among paths from u to v).

2. For all edges (u, v), the new weight $\widehat{w}(u, v)$ is nonnegative.

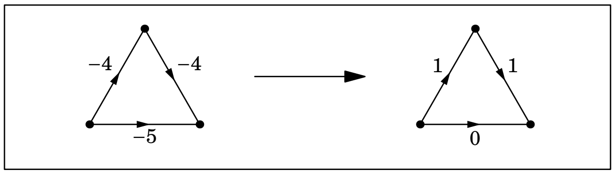


Figure 7.1. Adding a constant to the weight of every edge does not preserve shortest paths.

A clever, simple solution to this problem is as follows. Let $h : V \to \mathbb{R}$ be a scalar-valued function on the vertices. Assign

$$\widehat{w}(u,v) = w(u,v) + h(v) - h(u).$$

This weighting has the property that, for a path $p : \langle u = v_0,\ v_1, \ldots, v_k = v \rangle$, the new weight is

$$\begin{aligned}
\widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_i) - h(v_{i-1}) \\
&= w(p) + \sum_{i=1}^{k} h(v_i) - h(v_{i-1}) \\
&= w(p) + h(v_k) - h(v_0) \\
&= w(p) + h(v) - h(u),
\end{aligned}$$

by telescoping. Thus, the amount by which p changes depends only on the endpoints u and v, not on the intermediate vertices. Consequently, a path $p : u \rightsquigarrow v$ is minimal with respect to $\widehat{w}$ if and only if it is minimal with respect to w. Also, if $u = v$, then $\widehat{w}(p) = w(p)$, so the weight of cycles is unchanged. Thus, negative-weight cycles are preserved, so the distance from u to v with respect to $\widehat{w}$ is $-\infty$ if and only if the distance from u to v with respect to w is $-\infty$.

Our task now is to choose an appropriate function h so that the edge wieghts $\widehat{w}$ become nonnegative. There is a magical answer to this question: Let s be a new vertex, and construct an augmented graph $G' = (V', E')$ with $V' = V \cup \{s\}$. Take E' to consist of all the edges in E, plus a directed edge from s to each vertex in V (see Figure 7.2). Then let

$$h(v) = -\delta(s,v),$$

where $\delta(s,v)$ is the weight of the shortest path from s to v. Note that Proposition 6.1 implies the "triangle inequality"

$$\delta(s,v) \le \delta(s,u) + \delta(u,v)$$

(with equality if one of the shortest paths from s to v passes through u). Since $\delta(u,v) \le w(u,v)$, we have

$$\delta(s,v) \le \delta(s,u) + w(u,v).$$

Hence

$$\begin{aligned}
\widehat{w}(u,v) &= w(u,v) + h(v) - h(u) \\
&= w(u,v) - \delta(s,v) + \delta(s,u) \\
&\ge w(u,v) - \big(\delta(s,u) + w(u,v)\big) + \delta(s,u) \\
&= 0.
\end{aligned}$$

Applying the weights $\widehat{w}$ to G, we obtain the following algorithm, due to Johnson:

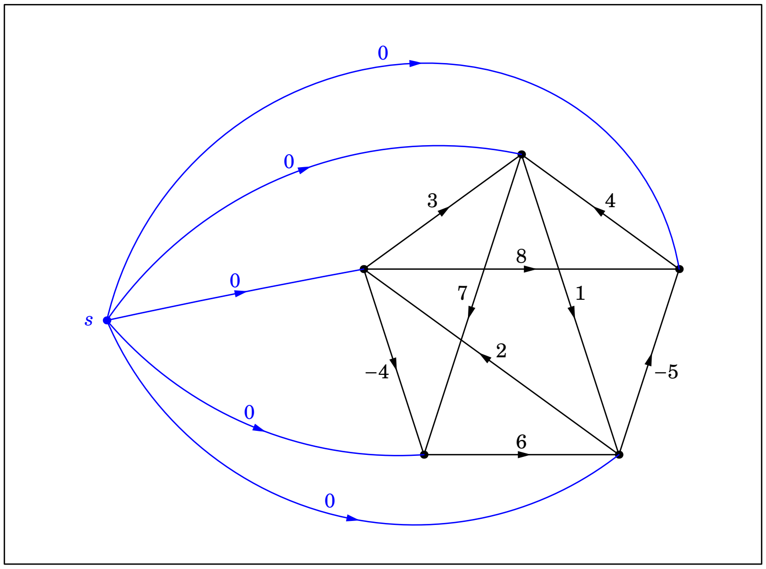


Figure 7.2. The augmented graph G' in Johnson's algorithm, consisting of G together with a special vertex s from which there emanates an edge of weight 0 to each other vertex.

```
Algorithm: JOHNSON($G$)
 1  Construct the augmented graph $G'$, where $G'.V = G.V \cup \{s\}$ and $G'.E = G.E \cup$
    $\{(s,v)$ with $G'.w(s,v) = 0$, for each $v \in G.V\}$
 2  if BELLMAN–FORD($G', s$) = FALSE then
 3          ▷ if BELLMAN–FORD complains about a negative-weight cycle
 4          error "The input graph contians a negative-weight cycle."
 5  else
 6          for each vertex $v \in G.V$ do
 7                  Set $h(v) \leftarrow -\delta(s,v)$, where $\delta(s,v)$ is as computed by BELLMAN–FORD
 8          for each edge $(u,v) \in G.E$ do
 9                  $\widehat{w}(u,v) \leftarrow G.w(u,v) + h(v) - h(u)$
10          Let $D = (d_{uv})$ and $\Pi = (\pi_{uv})$ be new $n \times n$ matrices
11          for each vertex $u \in G.V$ do
12                  Run DIJKSTRA($G, \widehat{w}, u$) to compute $\widehat{\delta}(u,v)$ for all $v \in G.V$
13                  for each vertex $v \in G.V$ do
14                          ▷ correct the shortest path weights
15                          $d_{uv} \leftarrow \widehat{\delta}(u,v) - (h(v) - h(u))$
16                          $\pi_{uv} \leftarrow$ the predecessor of $v$ as computed by DIJKSTRA($G, \widehat{w}, u$) above
17          return $D$
```

The only steps in this algorithm that take more than $O(E)$ time are the call to BELLMAN–FORD on line 2 (which takes $\Theta(VE)$ time) and the $|V|$ calls to DIJKSTRA on lines 11–16 (each of which takes $\Theta(V \lg V + E)$ time using a Fibonacci heap). Thus, the total running time of JOHNSON is $\Theta(V^2 \lg V + VE)$.

7.2 Linear Programming

Linear programming is a very general class of problem that arises frequently in diverse fields. The goal is to optimize a linear objective function subject to linear constraints (equalities and inequalities). That is, we want to maximize or minimize the function

$$f(x_1, x_2, \ldots, x_n) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = \sum_{j=1}^{n} a_j x_j$$

subject to the constraints

$$g^{(i)}(x_1, \ldots, x_n) = b_1^{(i)} x_1 + b_2^{(i)} x_2 + \cdots + b_n^{(i)} x_n = \sum_{j=1}^{n} b_j^{(i)} x_j = p^{(i)}$$

and

$$h^{(i)}(x_1, \ldots, x_n) = c_1^{(i)} x_1 + c_2^{(i)} x_2 + \cdots + c_n^{(i)} x_n = \sum_{j=1}^{n} c_j^{(i)} x_j \le q^{(i)}.$$

In vector form, we want to maximize or minimize

$$\mathbf{a}^T \mathbf{x}$$

subject to the constraints

$$B\mathbf{x} = \mathbf{p} \quad \text{and} \quad C\mathbf{x} \le \mathbf{q}$$

(meaning that the ith entry of $C\mathbf{x}$ is less than or equal to the ith entry of $\mathbf{q}$ for every i). Here B and C are matrices with n columns (and any number of rows).

7.2.1 Application: Bacterial Growth

One of the many applications of linear programming is the problem of maximizing bacterial growth subject to biochemical constraints. Consider the transport pathways of a bacterial cell, which we model as a graph embedded in three-dimensional space (see Figure 7.3). Let x_i be the flux through edge i.

- At steady-state, the net flux through each vertex of the graph is zero, as wastes are not building up along the cell pathways. This amounts to a linear constraint at each vertex, e.g., $x_1 - x_2 - x_3 = 0$ in the picture.

- Each edge has a minimum (namely zero) and a maximum rate at which molecules can pass through. This amounts to a linear constraint $0 \le x_i \le v_{\max}^{(i)}$ at each edge.

- The objective function is the sum of fluxes contributing to biomass production (e.g., synthesis of amino acids and proteins).

The availability of efficient solutions to linear programming problems has been valuable in the engineering of new bacterial strains with improved capabilities, e.g., to produce biofuels.

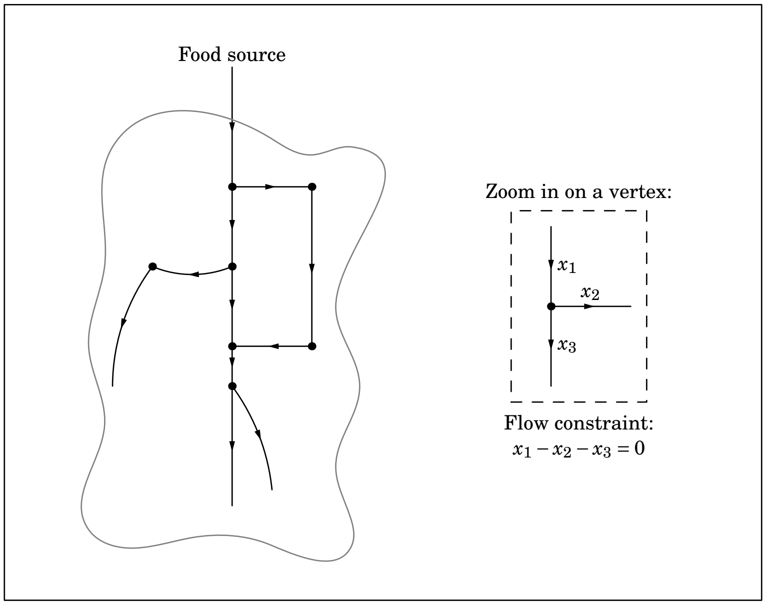


Figure 7.3. Bacterial cell, with a graph representing its transport pathways. At steady-state, the net flux through each vertex is zero.

7.2.2 Difference Constraints as Graphs

It turns out that shortest path problems are related to some forms of linear programming problems. Consider a system of linear constraints, each having the form

$$x_j - x_i \le b_k,$$

in other words, a system of the form

$$A\mathbf{x} \le \mathbf{b},$$

where A is an $m \times n$ matrix, $\mathbf{x}$ is an n-dimensional vector and $\mathbf{b}$ is an m-dimensional vector, and where each row of A contains one 1, one -1, and all other entries are zero. As a concrete example, let's consider the system

$$\begin{pmatrix} 1 & -1 & 0 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \le \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ -3 \end{pmatrix}.$$

This is equalivalent to the inequalities

$$\left\{ \begin{array}{l} x_1 - x_2 \le 0 \\ x_1 - x_4 \le -1 \\ x_2 - x_4 \le 1 \\ x_3 - x_1 \le 5 \\ x_4 - x_3 \le -3 \end{array} \right\}.$$

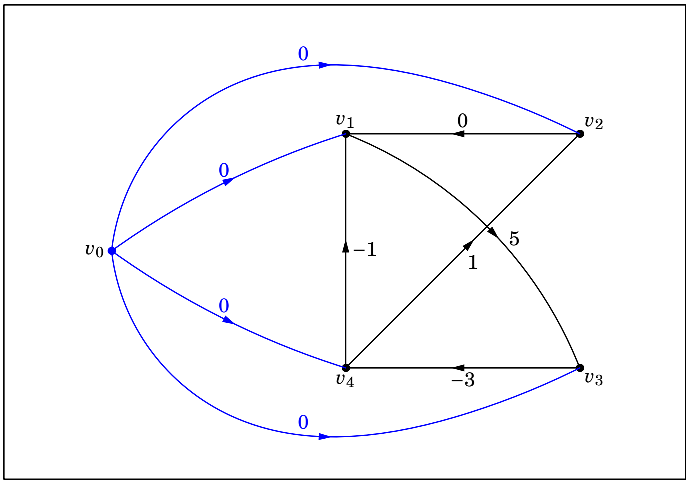


Figure 7.4. Constraint graph for the system of difference constraints $A\mathbf{x} \le \mathbf{b}$, where A and $\mathbf{b}$ are as in our example.

Two plausible solutions to this system are

$$\mathbf{x} = \begin{pmatrix} -5 \\ -3 \\ 0 \\ -4 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} 0 \\ 2 \\ 5 \\ 1 \end{pmatrix}.$$

In general, it's not hard to see that adding any multiple of $(1,1,1,1)^T$ to a solution produces another solution.

Let's now construct a graph G, called the *constraint graph* for this system (see Figure 7.4). The vertex set of G is $V = \{v_0, \ldots, v_n\}$. Draw a directed edge with weight 0 from v_0 to each other vertex. Then, for each difference constraint $x_j - x_i \le b$, draw an edge from v_i to v_j with weight b.

Theorem 7.1 (Theorem 24.9 of CLRS)**.** *Given a system $A\mathbf{x} \le \mathbf{b}$ of difference constraints, let $G = (V,E)$ be the corresponding constraint graph. If G contians no negative-weight cycles, then*

$$\mathbf{x} = \begin{pmatrix} \delta(v_0, v_1) \\ \delta(v_0, v_2) \\ \vdots \\ \delta(v_0, v_n) \end{pmatrix}$$

is a feasible solution. If G does contain negative-weight cycles, then there is no feasible solution.

Proof. Assume there are no negative-weight cycles. Since shortest paths satisfy the triangle inequality, we have

$$\underbrace{\delta(v_0, v_j)}_{x_j} \le \underbrace{\delta(v_0, v_i)}_{x_i} + \underbrace{w(v_i, v_j)}_{b}.$$

Rearranging, we obtain

$$x_j - x_i \le b.$$

Next, consider the case of a negative-weight cycle. Without loss of generality, assume the cycle is of the form $c = \langle v_1, v_2, \ldots, v_k, v_1 \rangle$. Then any solution $\mathbf{x}$ to the system of difference constraints must satisfy

$$\begin{array}{rrcll}
 & x_2 - x_1 & \leq & w(v_1, v_2) & \\
 & x_3 - x_2 & \leq & w(v_2, v_3) & \\
 & & \vdots & & \\
 & x_k - x_{k-1} & \leq & w(v_{k-1}, v_k) & \\
+ & x_1 - x_k & \leq & w(v_k, v_1) & \\
\hline
 & 0 & \leq & w(c) & < \quad 0,
\end{array}$$

a contradiction. Therefore there cannot be any solutions to the system of difference constraints. □

Lecture 8

Randomized Algorithms I

Supplemental reading in CLRS: Chapter 5; Section 9.2

Should we be allowed to write an algorithm whose behavior depends on the outcome of a coin flip? It turns out that allowing random choices can yield a tremendous improvement in algorithm performance. For example,

- One can find a minimum spanning tree of a graph $G = (V, E, w)$ in linear time $\Theta(V + E)$ using a randomized algorithm.
- Given two polynomials p, q of degree $n - 1$ and a polynomial r of degree $n - 2$, one can check whether $r = pq$ in linear time $\Theta(n)$ using a randomized algorithm.

No known deterministic algorithms can match these running times.

Randomized algorithms are generally useful when there are many possible choices, "most" of which are good. Surprisingly, even when most choices are good, it is not necessarily easy to find a good choice deterministically.

8.1 Randomized Median Finding

Let's now see how randomization can improve our median-finding algorithm from Lecture 1. Recall that the main challenge in devising the deterministic median-finding algorithm was this:

Problem 8.1. Given an unsorted array $A = A[1, \ldots, n]$ of n numbers, find an element whose rank is sufficiently close to $n/2$.

While the deterministic solution of Blum, Floyd, Pratt, Rivest and Tarjan is ingenious, a simple randomized algorithm will typically outperform it in practice.

Input: An array $A = A[1, \ldots, n]$ of n numbers

Output: An element $x \in A$ such that $\frac{1}{10}n \le \operatorname{rank}(x) \le \frac{9}{10}n$.

```
Algorithm: FIND-APPROXIMATE-MIDDLE($A$)
Pick an element from $A$ uniformly at random.
```

Note that the above algorithm is *not* correct. (Why not?) However, it does return a correct answer with probability $\frac{8}{10}$.

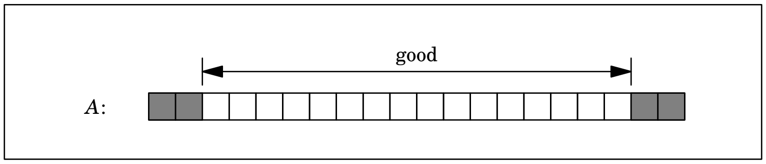


Figure 8.1. Wouldn't it be nice if we had some way of choosing an element that probably lies in the middle?

8.1.1 Randomized Median Finding

Let's put the elegant and simple algorithm FIND-APPROXIMATE-MIDDLE to use in a randomized version of the median-finding algorithm from Lecture 1:

```
Algorithm: RANDOMIZED-SELECT($A, i$)
  1. Pick an element $x \in A$ uniformly at random.
  2. Partition around $x$. Let $k = \text{rank}(x)$.
  3.   • If $i = k$, then return $x$.
       • If $i < k$, then recursively call RANDOMIZED-SELECT($A[1, \dots, k-1]$, $i$).
       • If $i > k$, then recursively call RANDOMIZED-SELECT($A[k+1, \dots, i]$, $i - k$).
```

Note two things about this algorithm:

- Unlike FIND-APPROXIMATE-MIDDLE, RANDOMIZED-SELECT makes several random choices: one in each recursive call.
- Unlike FIND-APPROXIMATE-MIDDLE, RANDOMIZED-SELECT *is* correct. The effect of bad random choices is to prolong the running time, not to generate an incorrect answer. For example, if we wanted to find the middle element ($i = n/2$), and if our random element x happened to always be the smallest element of A, then RANDOMIZED-SELECT would take $\Theta(n) \cdot T_{\text{partition}} = \Theta(n^2)$ time. (To prove this rigorously, a more detailed calculation is needed.)

The latter point brings to light the fact that, for randomized algorithms, the notion of "worst-case" running time is more subtle than it is for deterministic algorithms. We cannot expect randomized algorithms to work well for every possible input *and* every possible sequence of random choices (in that case, why even use randomization?). Instead, we do one of two things:

- Construct algorithms that always run quickly, and return the correct answer with high probability. These are called **Monte Carlo algorithms**. With some small probability, they may return an incorrect answer or give up on computing the answer.
- Construct algorithms that always return the correct answer, and have low *expected* running time. These are called **Las Vegas algorithms**. Note that the expected running time is an average over all possible sequences of random choices, but *not* over all possible inputs. An algorithm that runs in expected $\Theta(n^2)$ time on difficult inputs and expected $\Theta(n)$ time on easy inputs has worst-case expected running time $\Theta(n^2)$, even if most inputs are easy.

FIND-APPROXIMATE-MIDDLE is a Monte Carlo algorithm, and RANDOMIZED-SELECT is a Las Vegas algorithm.

8.1.2 Running Time

To begin our analysis of RANDOMIZED-SELECT, let's first suppose all the random choices happen to be good. For example, suppose that every recursive call of RANDOMIZED-SELECT(A,i) returns an element whose rank is between $\frac{1}{10}|A|$ and $\frac{9}{10}|A|$. Then, the argument to each recursive call of RANDOMIZED-SELECT is at most nine-tenths the size of the argument to the previous call. Hence, if there are a total of K recursive calls to RANDOMIZED-SELECT, and if n is the size of the argument to the original call, then

$$\left(\tfrac{9}{10}\right)^K n \ge 1 \quad\Longrightarrow\quad K \le \log_{10/9} n.$$

(Why?) Of course, in general we won't always get this lucky; things might go a little worse. However, it is quite unlikely that things will go *much* worse. For example, there is a 96% chance that (at least) one of our first 15 choices will be bad—quite high. But there is only a 13% chance that three of our first 15 choices will be bad, and only a 0.06% chance that nine of our first 15 choices will be bad.

Now let's begin the comprehensive analysis. Suppose the size of the original input is n. Then, let T_r be the number of recursive calls that occur after the size of A has dropped below $\left(\frac{9}{10}\right)^r n$, but before it has dropped below $\left(\frac{9}{10}\right)^{r+1} n$. For example, if it takes five recursive calls to shrink A down to size $\frac{9}{10}n$ and it takes eight recursive calls to shrink A down to size $\left(\frac{9}{10}\right)^2 n$, then $T_0 = 5$ and $T_1 = 3$. Of course, each T_i is a random variable, so we can't say anything for sure about what values it will take. However, we do know that the total running time of RANDOMIZED-SELECT$(A = A[1,\ldots,n], i)$ is

$$T = \sum_{r=0}^{\log_{10/9} n} \Theta\left(\left(\tfrac{9}{10}\right)^r n\right) \cdot T_r.$$

Again, T is a random variable, just as each T_r is. Moreover, each T_r has low expected value: Since each recursive call to RANDOMIZED-SELECT has a 4/5 chance of reducing the size of A by a factor of $\frac{9}{10}$, it follows that (for any r)

$$\Pr\left[T_r > s\right] \le \left(\tfrac{1}{5}\right)^s.$$

Thus, the expected value of T_r satisfies

$$\mathbb{E}[T_r] \le \sum_{s=0}^{\infty} s\left(\tfrac{1}{5}\right)^s = \frac{5}{16}.$$

(The actual value 5/16 is not important; just make sure you are able to see that the series converges quickly.) Thus, by the linearity of expectation,

$$\begin{aligned}
\mathbb{E}[T] &= \mathbb{E}\left[\sum_{r=0}^{\log_{10/9} n} \Theta\left(\left(\tfrac{9}{10}\right)^r n\right) \cdot T_r\right] \\
&= \sum_{r=0}^{\log_{10/9} n} \Theta\left(\left(\tfrac{9}{10}\right)^r n\right) \cdot \mathbb{E}[T_r] \\
&\le \frac{5}{16} \sum_{r=0}^{\log_{10/9} n} O\left(\left(\tfrac{9}{10}\right)^r n\right) \\
&= O\left(n \sum_{r=0}^{\log_{10/9} n} \left(\tfrac{9}{10}\right)^r\right)
\end{aligned}$$

$$\le O\left(n \sum_{r=0}^{\infty} \left(\tfrac{9}{10}\right)^r\right)$$

$$= O(n),$$

since the geometric sum $\sum_{r=0}^{\infty} \left(\frac{9}{10}\right)^r$ converges (to 10).

8.2 Another Example: Verifying Polynomial Multiplication

Suppose we are given two polynomials p, q of degree $n-1$ and a polynomial r of degree $2n-2$, and we wish to check whether $r = pq$. We could of course compute pq in $\Theta(n \lg n)$ time using the fast Fourier transform, but for some purposes the following Monte Carlo algorithm would be more practical:

```
Algorithm: VERIFY-MULTIPLICATION($p, q, r$)
    1. Choose $x$ uniformly at random from $\{0, 1, \ldots, 100n-1\}$.
    2. Return whether $p(x) \cdot q(x) = r(x)$.
```

This algorithm is not correct. It will never report no if the answer is yes, but it might report yes when the answer is no. This would happen if x were a root of the polynomial $pq - r$, which has degree at most $2n-2$. Since $pq - r$ can only have at most $2n-2$ roots, it follows that

$$\Pr\left[\text{wrong answer}\right] \le \frac{2n-2}{100n} < 0.02.$$

Often, the performance improvement offered by this randomized algorithm (which runs in $\Theta(n)$ time) is worth the small risk of (one-sided) error.

8.3 The Markov Bound

The Markov bound states that it is unlikely for a nonnegative random variable X to exceed its expected value by very much.

Theorem 8.2 (Markov bound). *Let X be a nonnegative random variable with positive expected value.*[1] *Then, for any constant $c > 0$, we have*

$$\Pr\left[X \ge c \cdot \mathbb{E}[X]\right] \le \frac{1}{c}.$$

Proof. We will assume X is a continuous random variable with probability density function f_X; the discrete case is the same except that the integral is replaced by a sum. Since X is nonnegative, so is $\mathbb{E}[X]$. Thus we have

$$\mathbb{E}[X] = \int_0^{\infty} x\, f_X(x)\, dx$$

$$\ge \int_{c\,\mathbb{E}[X]}^{\infty} x\, f_X(x)\, dx$$

[1] i.e., $\Pr\left[X > 0\right] > 0$.

$$\geq c\,\mathbb{E}[X]\int_{c\,\mathbb{E}[X]}^{\infty} f_X(x)\,dx$$

$$= c\,\mathbb{E}[X]\cdot \Pr\Big[X \geq c\,\mathbb{E}[X]\Big].$$

Dividing through by $c\,\mathbb{E}[X]$, we obtain

$$\frac{1}{c} \geq \Pr\Big[X \geq c\,\mathbb{E}[X]\Big]. \qquad \square$$

Corollary 8.3. *Given any constant $c > 0$ and any Las Vegas algorithm with expected running time T, we can create a Monte Carlo algorithm which always runs in time cT and has probability of error at most $1/c$.*

Proof. Run the Las Vegas algorithm for at most cT steps. By the Markov bound, the probability of not reaching termination is at most $1/c$. If termination is not reached, give up. □

Lecture 9

Randomized Algorithms II

Supplemental reading in CLRS: Appendix C; Section 7.3

After Lecture 8, several students asked whether it was fair to compare randomized algorithms to deterministic algorithms.

Determistic algorithm:

- always outputs the right answer
- always runs efficiently

Randomized algorithm:

- may sometimes not output the right answer
- may sometimes not run efficiently

Is this "fair"? Of course not. We demand less from randomized algorithms. But in exchange, we expect randomized algorithms to do more—to be more efficient, or to be simple and elegant.

Separately, we might ask whether it is *useful* to employ randomized algorithms. The answer here is that it depends:

- on the situation
- on the available alternatives
- on the probability of error or inefficiency.

Lots of real-life situations call for randomized algorithms. For example, Google Search and IBM's *Jeopardy!*-playing computer Watson both employ randomized algorithms.

9.1 The Central Limit Theorem and the Chernoff Bound

Ideally, a randomized algorithm will have the property that

$$\Pr\left[\text{bad things happening}\right] \xrightarrow{n\to\infty} 0.$$

Before further analyzing randomized algorithms, it will be useful to establish two basic facts about probability theory: the central limit theorem and the Chernoff bound. The central limit theorem states that the mean of a large number of independent copies of a random variable is approximately normal, as long as the variable has finite variance. The **normal distribution** (or **Gaussian distribution**) of mean 0 and variance 1, denoted $N(0,1)$, is defined by the probability density function

$$f_{N(0,1)}(x) = \frac{1}{\sqrt{2\pi}}\,\exp\left(-x^2/2\right).$$

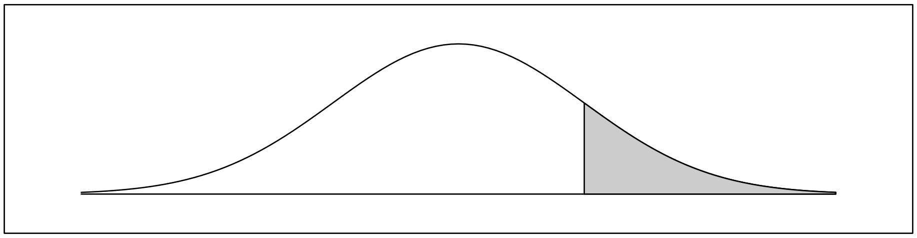


Figure 9.1. A tail of the normal distribution.

Taking an affine transform of this standard Gaussian, we obtain the normal distribution with arbitrary mean μ and arbitrary variance σ^2, denoted $N\left(\mu,\sigma^2\right)$. Its probability density function is

$$f_{N(\mu,\sigma^2)}(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\ \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

Due in part to the following theorem, the Gaussian distribution is extremely important in all of probability theory.

Theorem 9.1 (Central Limit Theorem)**.** *Suppose $X_1, X_2, \ldots$ are i.i.d.*[1] *random variables, each with finite mean μ and finite variance σ^2. Let $S_n = \frac{X_1+\ldots+X_n}{n}$, and let $Y_n = \sqrt{n}\left(S_n - \mu\right)$. Then the variables $Y_1, Y_2, \ldots$ converge to a normal distribution:*

$$Y_n \xrightarrow{d} N\left(0,\sigma^2\right).$$

The precise meaning of this theorem is somewhat technical. Again, in effect it means that the average of a large number of independent copies of a random variable X is approximately normally distributed. Thus, the probability that this average exceeds its expected value by r is approximately the probability that a normal random variable with variance σ^2 exceeds its expected value by r. While this heuristic reasoning does not rigorously prove any specific bound, there are a number of bounds that have been worked out for certain common distributions of X. One such bound is the Chernoff bound, one version of which is as follows:

Theorem 9.2 (Chernoff bound)**.** *Let $Y \sim B(n,p)$ be a random variable representing the total number of heads in a series of n independent coin flips, where each flip has probability p of coming up heads. Then, for all $r > 0$, we have*

$$\Pr\Big[Y \geq \mathbb{E}[Y] + r\Big] \leq \exp\left(-2r^2/n\right).$$

The distribution in Theorem 9.2 is called the **binomial distribution** with parameters (n,p). The probability of m heads is $\binom{n}{m}p^m(1-p)^{n-m}$.

There are several different versions of the Chernoff bound—some with better bounds for specific hypotheses, some more general. Section C.5 of CLRS gives a classic proof of the Chernoff bound, applying the Markov inequality to the random variable $\exp\left(\alpha(Y - \mathbb{E}[Y])\right)$ for a suitable constant α. We give a different proof here[2], due to Impagliazzo and Kabanets, 2010.

[1] independent and identically distributed

[2] Of a different theorem, actually. The version we prove here is (stronger than) Exercise C.5-6 of CLRS.

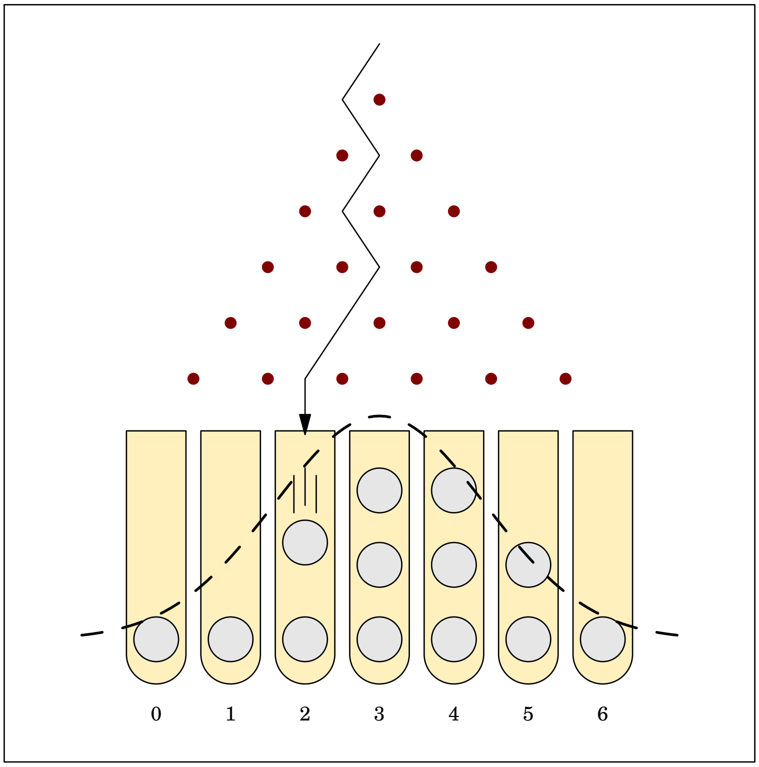


Figure 9.2. *Galton's board* is a toy in which, at each peg, the ball has probability $\frac{1}{2}$ of moving left and probability $\frac{1}{2}$ of moving right. The number of the bin into which it falls is a binomial random variable with parameters $(n, \frac{1}{2})$. If n is large and you drop a large number of balls into the board, the collection of balls in the bins underneath will start to look like a Gaussian probability density curve.

Proof of Theorem 9.2. Let

$$Y = X_1 + \cdots + X_n,$$

where $X_1, \ldots, X_n$ are independent $\{0,1\}$-valued random variables, each having probability p of being 1. Thus Y is a binomial random variable with parameters (n, p). Let S be a random subset of $\{1, \ldots, n\}$ such that each element of $\{1, \ldots, n\}$ independently has probability q of being included in S, where q is a parameter that will be determined later. The probability that $X_i = 1$ for all $i \in S$ is

$$\begin{aligned} \Pr\left[\bigwedge_{i\in S} X_i = 1\right] &= \Pr\left[\bigwedge_{i\in S} X_i = 1 \,\middle|\, Y \geq \mathbb{E}[Y] + r\right] \Pr\left[Y \geq \mathbb{E}[Y] + r\right] \\ &\qquad + \Pr\left[\bigwedge_{i\in S} \,\middle|\, Y < \mathbb{E}[Y] + r\right] \Pr\left[Y < \mathbb{E}[Y] + r\right] \\ &\geq \Pr\left[\bigwedge_{i\in S} X_i = 1 \,\middle|\, Y \geq \mathbb{E}[Y] + r\right] \Pr\left[Y \geq \mathbb{E}[Y] + r\right]. \end{aligned} \tag{9.1}$$

Meanwhile,

$$\Pr\left[\bigwedge_{i\in S} X_i = 1 \,\middle|\, Y \geq \mathbb{E}[Y] + r\right] \geq (1-q)^{n-(\mathbb{E}[Y]+r)}, \tag{9.2}$$

since the condition $\bigwedge_{i\in S} X_i = 1$ is equivalent to the condition that S doesn't include the indices of any zeros, and we are conditioning on the hypothesis that there are at most $n - (\mathbb{E}[Y] + r)$ zeros. (This is the "key idea" of the proof. Notice that we have switched from conditioning on S followed by the X_i's to conditioning on the X_i's followed by S.) Combining (9.1) and (9.2), we obtain

$$\Pr\left[\bigwedge_{i\in S} X_i = 1\right] \geq (1-q)^{n-(\mathbb{E}[Y]+r)} \Pr\left[Y \geq \mathbb{E}[Y] + r\right]. \tag{9.3}$$

We could also compute $\Pr\left[\bigwedge_{i\in S} X_i = 1\right]$ by conditioning on S in a different way:

$$\Pr\left[\bigwedge_{i\in S} X_i = 1\right] = \sum_{k=0}^{n} \underbrace{\left(\binom{n}{k} q^k (1-q)^{n-k}\right)}_{\Pr\left[|S|=k\right]} \cdot \underbrace{\left(\qquad p^k \qquad\right)}_{\Pr\left[\bigwedge_{i\in S} X_i = 1 \,\middle|\, |S|=k\right]}.$$

Recall the binomial formula, which states

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}.$$

In the present case, we have

$$\Pr\left[\bigwedge_{i\in S} X_i = 1\right] = \left(qp + 1 - q\right)^n. \tag{9.4}$$

Combining (9.3) and (9.4), we obtain

$$(1-q)^{n-(\mathbb{E}[Y]+r)} \Pr\left[Y \geq \mathbb{E}[Y] + r\right] \leq \left(qp + 1 - q\right)^n.$$

Rearranging and using the fact that $\mathbb{E}[Y] = np$, we have equivalently

$$\Pr\left[Y \geq \mathbb{E}[Y] + r\right] \leq \frac{\left(qp + 1 - q\right)^n}{(1-q)^{n-(np+r)}}. \tag{9.5}$$

Equation (9.5) holds for arbitrary q, so we might as well assign a value to q for which the right side is minimal. Let

$$q = \frac{r/n}{\left(p+\frac{r}{n}\right)(1-p)};$$

after some expansion, the right side of (9.5) becomes

$$\left[\left(\frac{p}{p+\frac{r}{n}}\right)^{p+\frac{r}{n}}\left(\frac{1-p}{1-\left(p+\frac{r}{n}\right)}\right)^{1-\left(p+\frac{r}{n}\right)}\right]^n = \exp\left(-nf\left(\tfrac{r}{n}\right)\right),$$

where[3]

$$f(x) = \left(p+x\right)\ln\left(1+\tfrac{x}{p}\right) + \left(1-(p+x)\right)\ln\left(1-\tfrac{x}{1-p}\right).$$

Below we will show that

$$f(x) \geq 2x^2 \quad \text{for all } x \in \mathbb{R} \text{ such that } f(x) \in \mathbb{R},$$

so that (9.5) becomes

$$\begin{aligned} \Pr\left[Y \geq \mathbb{E}[Y]+r\right] &\leq \exp\left(-nf\left(\tfrac{r}{n}\right)\right) \\ &\leq \exp\left(-2n\left(\tfrac{r}{n}\right)^2\right) \\ &= \exp\left(-\tfrac{2r^2}{n}\right). \end{aligned}$$

This will complete the proof.

All that remains is to show that $f(x) \geq 2x^2$ for all x such that $f(x) \in \mathbb{R}$—namely, for $-p < x < 1-p$, though we will only need the range $0 \leq x < 1-p$.[4] This can be shown by calculus: the first and second derivatives of f are

$$\begin{aligned} f'(x) &= \ln\left(1+\tfrac{x}{p}\right) - \ln\left(1-\tfrac{x}{1-p}\right), \\ f''(x) &= \frac{1}{(1-(p+x))(p+x)}. \end{aligned}$$

In particular, $f''(x)$ is minimized when $p+x = \frac{1}{2}$, at which point $f''(x) = 4$. Thus, the fundamental theorem of calculus gives

$$f'(x) \ = \ \underbrace{f'(0)}_{0} + \int_{t=0}^{x} f''(t)\,dt \ \geq \ \int_{t=0}^{x} 4\,dt \ = \ 4x.$$

Applying the fundamental theorem of calculus again, we have

$$f(x) \ = \ \underbrace{f(0)}_{0} + \int_{t=0}^{x} f'(t)\,dt \ \geq \ \int_{t=0}^{x} 4t\,dt \ = \ 2x^2. \qquad \square$$

[3] The definition of f may seem unmotivated here, but in fact $f(x)$ is properly interpreted as the *relative entropy* between two Bernoulli random variables with parameters p and $p+x$, respectively. Relative entropy is a measure of the "distance" between two probability distributions.

[4] Note that $x = \frac{r}{n}$ will always fall within this range, as we must have $\mathbb{E}[Y]+r \leq n$. (Or at least, the theorem becomes trivial when $\frac{r}{n} > 1-p$.)

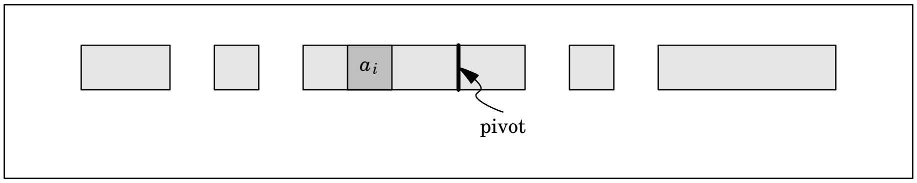


Figure 9.3. In a given iteration of QUICKSORT, there will be several working subarrays, each in the process of being sorted by its own recursive call to QUICKSORT.

9.2 Analysis of QUICKSORT

Recall the procedure QUICKSORT, which sorts an array $A = \langle a_1, \ldots, a_n \rangle$ in place (see Figure 9.3):

Algorithm: QUICKSORT(A)

1. Choose an element $x \in A$ uniformly at random. We will call x the "pivot."
2. Partition A around x.
3. Sort the elements less than x recursively.
4. Sort the elements greater than x recursively.

We saw in recitation that the expected running time of QUICKSORT is $\Theta(n \lg n)$. In this lecture we'll use the Chernoff bound to show that, for some constant $c \geq 1$, the probability that QUICKSORT takes more than $cn \lg n$ time on any given input is at most $\frac{1}{n}$. In fact, what we'll show is that after time $cn \lg n$, the probability that the working subarray containing any given element $a \in A$ consists of more than one element is at most $\frac{1}{n^2}$. Once this is established, the **union bound** shows that the probability that there exists *any* working subarray with more than one element is at most $\frac{1}{n}$. That is to say, let $A = \langle a_1, \ldots, a_n \rangle$ and let E_i be the event that after time $cn \lg n$, the working subarray containing a_i has more than one element. Then the union bound states that

$$\Pr\left[\begin{matrix}\exists \text{ working subarray with} \\ \text{more than one element}\end{matrix}\right] = \Pr\left[\bigcup_{i=1}^{n} E_i\right] \leq \sum_{i=1}^{n} \Pr\left[E_i\right] \leq n\left(\frac{1}{n^2}\right) = \frac{1}{n}.$$

(In general, the union bound says that the probability of a finite or countably infinite union of events E_i is at most the sum of their individual probabilities. The intuition is that equality is achieved when the E_i's are pairwise disjoint, and any overlap only diminishes the size of the union.)

All that remains is to find c such that, for any array A,

$$\Pr\left[E_i\right] \leq \frac{1}{n^2} \quad \text{for all } i = 1, \ldots, n.$$

As in Lecture 8, we define a "good" pivot x in an array A to be a pivot such that

$$\tfrac{1}{10}|A| \leq \operatorname{rank}(x) \leq \tfrac{9}{10}|A|.$$

Thus, a good pivot splits the current array into two subarrays each at most $\frac{9}{10}$ as big. Let $X_{i,k}$ be the indicator random variable

$$X_{i,k} = \begin{cases} 0 & \text{if, in the } k\text{th iteration, a good pivot was chosen for } a_i\text{'s subarray} \\ 1 & \text{otherwise.} \end{cases}$$

(By “the kth iteration,” we mean all calls to QUICKSORT at recursive depth k. Thus, the kth iteration will consist of 2^{k-1} recursive calls to QUICKSORT, assuming none of the subarrays have shrunk to size 1 yet.) Note two things:

- $\mathbb{E}\left[X_{i,k}\right] = 0.2$ for all i and all k. (To reduce the amount of bookkeeping we have to do, we will assume that pivots continue to be chosen even after an element's subarray is reduced to size 1, and that the probability of choosing a good pivot continues to be 0.8.)

- For a fixed k, the variables $X_{1,k}, X_{2,k}, \ldots, X_{n,k}$ are dependent on each other. However, for a fixed i, the variables $X_{i,1}, X_{i,2}, \ldots$ are independent of each other.

For a fixed i and a fixed K, we have

$$\mathbb{E}\left[\sum_{k=1}^{K} X_{i,k}\right] = 0.2K.$$

Thus, by the Chernoff bound (with $r = 0.1K$), we have

$$\Pr\left[\sum_{k=1}^{K} X_{i,k} > 0.2K + 0.1K\right] \le \exp\left(-\frac{2(0.1K)^2}{K}\right) = \exp(-0.02K).$$

Now let $K = 100 \ln n$. We obtain

$$\Pr\left[\sum_{k=1}^{K} X_{i,k} > 0.3K\right] \le \frac{1}{n^2}.$$

On the other hand, if $\sum_{k=1}^{K} X_{i,k} \le 0.3K$, then at least $0.7K$ of the first K pivots were good, and the size of the working subarray containing a_i after K steps is at most

$$n \cdot \left(\frac{9}{10}\right)^{0.7K} = n \cdot \left(\frac{9}{10}\right)^{70 \ln n} = n^{70 \ln(9/10)+1} \approx n^{-6.3} < 1$$

(i.e., the size is at most 1; the reason for the fractional number is that we are ignoring issues of rounding). Thus, after $K = 100 \ln n$ iterations, the probability that the working subarray containing a_i has more than one element is at most $\frac{1}{n^2}$. So by the union bound, the probability that QUICKSORT requires more than K iterations is at most $\frac{1}{n}$.

While the sizes of the inputs to each iteration and the total number of iterations required are random, the running time of the non-recursive part of QUICKSORT is deterministic: It takes $\Theta(A)$ time to pick a random element and partition around that element. Let's say it takes at most τA time to do this, where τ is some appropriately chosen constant. Now, if the sizes of the inputs to the kth iteration are $m_1, \ldots, m_\ell$, then the quantities $m_1, \ldots, m_\ell$ are random, but we know $\sum_{\lambda=1}^{\ell} m_\lambda = n$, so the total amount of time required by the kth iteration is at most $\sum_{\lambda=1}^{\ell} \tau m_\lambda = \tau n$. Thus, the probability that QUICKSORT takes more than $K\tau n = (100\tau) n \ln n$ time is at most $1/n$. This is what we set out to show in the beginning of this section, with $c = 100\tau$.

To recap:

> How can we argue about randomized algorithms?
>
> - Identify the random choices in the algorithm.
> - After fixing the random choices, we have a deterministic algorithm.

In this example, the random choices were the pivots. Fixing the sequence of choices of pivots, we were able to analyze $\left|A_{i,k}\right|$, the size of the subarray containing a_i after k iterations, for each i and k (see Figure 9.4).

Random choices at depth 1	$\left\lvert A_{1,1}\right\rvert = n$	$\left\lvert A_{2,1}\right\rvert = n$	$\cdots$
Random choices at depth 2	$\left\lvert A_{1,2}\right\rvert = 0.3n$	$\left\lvert A_{2,2}\right\rvert = 0.8n$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 9.4. The values $0.3n$ and $0.8n$ are just examples, and of course depend on the sequence of random choices. Once the sequence of random choices is fixed, we would like to know how many rows down we must go before all entries are 1.

9.3 Monte Carlo Sampling

Suppose that out of a large population U, there occurs a certain phenomenon in a subset $S \subseteq U$. Given an element x, we are able to test whether $x \in S$. How can we efficiently estimate $\frac{|S|}{|U|}$? For example, how can we efficiently estimate what fraction of the world's population is left-handed, or what percentage of voters vote Republican?

There is an elegant simple solution to this problem: Choose a k-element sample $A = \{x_1, \ldots, x_k\} \subseteq U$ uniformly at random. We then estimate $\frac{|S|}{|U|}$ by the quantity $\widehat{S} = \frac{|A \cap S|}{|A|} = \frac{1}{k}\left|\{i : x_i \in S\}\right|$, figuring that the chance that an element $a \in A$ belongs to S ought to be the same as the chance that a general element $x \in U$ belongs to S. Thus, we are relying heavily on our ability to pick a truly uniform sample.[5] Assuming this reliance is safe, $k\widehat{S} = |A \cap S|$ is a binomial random variable with parameters $\left(k, \frac{|S|}{|U|}\right)$. In particular, the expected value of $\widehat{S}$ is the exact correct answer $\frac{|S|}{|U|}$. Thus, the Chernoff bound states that for any $r > 0$,

$$\Pr\left[\widehat{S} \geq \frac{|S|}{|U|} + r\right] = \Pr\left[k\widehat{S} \geq \mathbb{E}\left[k\widehat{S}\right] + kr\right] \leq \exp\left(-2kr^2\right).$$

Exercise 9.1. *How can we use the Chernoff bound to give an upper bound on* $\Pr\left[\widehat{S} \leq \frac{|S|}{|U|} - r\right]$*? (Hint: replace* S *by its complement* $U \setminus S$*.)*

9.4 Amplification

Suppose we are given a Monte Carlo algorithm which always runs in time T and returns the correct answer with probability 2/3. (Assume there is a unique correct answer.) Then, for any $\epsilon > 0$, we can create a new randomized algorithm which always runs in time $O\left(T \lg \frac{1}{\epsilon}\right)$ and outputs an incorrect answer with probability at most ϵ. How?

The plan is to run the original algorithm k times, where $k = O\left(\lg \frac{1}{\epsilon}\right)$ is a quantity which we will determine later. The algorithm should then output whichever answer occurred most frequently out of the k computed answers. To bound the probability of error, let I_j be the indicator random variable which equals 1 if the jth run returns an incorrect answer, and let $I = \sum_{j=1}^{k} I_j$. Then I is a binomial random variable with parameters $\left(k, \frac{1}{3}\right)$. Thus, by the Chernoff bound,

$$\Pr\left[\begin{array}{c}\text{we ultimately return}\\ \text{an incorrect answer}\end{array}\right] \leq \Pr\left[I \geq \tfrac{1}{2}k\right] = \Pr\left[I \geq \mathbb{E}[I] + \tfrac{1}{6}k\right] \leq \exp\left(-\tfrac{1}{18}k\right).$$

[5] In practice, it could be quite hard to pick a uniform sample. For example, what if you wanted figure out what proportion of New Yorkers speak Chinese? How much of your random sampling should be done in Chinatown? Without the help of extensive census data, it can be hard to make unbiased choices.

Thus, the probability of error will be at most ϵ if we let $k = 18\ln\frac{1}{\epsilon}$.

Exercise 9.2. *Suppose that instead of returning the correct answer with probability 2/3, our Monte Carlo algorithm returned the correct answer with probability p. What conditions on p allow the above strategy to work? In terms of p and ϵ, how many times must we run the Monte Carlo algorithm?*

Lecture 10

Hashing and Amortization

Supplemental reading in CLRS: Chapter 11; Chapter 17 intro; Section 17.1

10.1 Arrays and Hashing

Arrays are very useful. The items in an array are statically addressed, so that inserting, deleting, and looking up an element each take $O(1)$ time. Thus, arrays are a terrific way to encode functions

$$\{1,\ldots,n\} \to T,$$

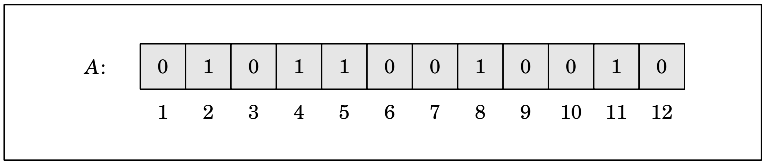
where T is some range of values and n is known ahead of time. For example, taking $T = \{0,1\}$, we find that an array A of n bits is a great way to store a subset of $\{1,\ldots,n\}$: we set $A[i] = 1$ if and only if i is in the set (see Figure 10.1). Or, interpreting the bits as binary digits, we can use an n-bit array to store an integer between 0 and $2^n - 1$. In this way, we will often identify the set $\{0,1\}^n$ with the set $\{0,\ldots,2^n - 1\}$.

What if we wanted to encode subsets of an arbitrary domain U, rather than just $\{1,\ldots,n\}$? Or to put things differently, what if we wanted a *keyed* (or *associative*) *array*, where the keys could be arbitrary strings? While the workings of such data structures (such as dictionaries in Python) are abstracted away in many programming languages, there is usually an array-based solution working behind the scenes. Implementing associative arrays amounts to finding a way to turn a key into an array index. Thus, we are looking for a suitable function $U \to \{1,\ldots,n\}$, called a **hash function**. Equipped with this function, we can perform key lookup:

$$U \xrightarrow{\text{hash function}} \{1,\ldots,n\} \xrightarrow{\text{array lookup}} T$$

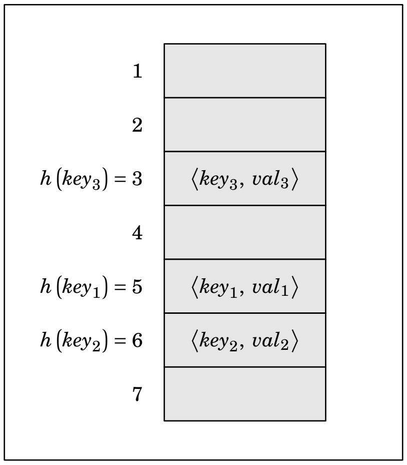
(see Figure 10.2). This particular implementation of associative arrays is called a **hash table**.

There is a problem, however. Typically, the domain U is much larger than $\{1,\ldots,n\}$. For any hash function $h : U \to \{1,\ldots,n\}$, there is some i such that at least $\frac{|U|}{n}$ elements are mapped to i. The set



A:	0	1	0	1	1	0	0	1	0	0	1	0
	1	2	3	4	5	6	7	8	9	10	11	12

Figure 10.1. This 12-bit array encodes the set $\{2,4,5,8,11\} \subseteq \{1,\ldots 12\}$.



1
2
$h\left(key_3\right) = 3$
$\langle key_3,\ val_3\rangle$
4
$h\left(key_1\right) = 5$
$\langle key_1,\ val_1\rangle$
$h\left(key_2\right) = 6$
$\langle key_2,\ val_2\rangle$
7

Figure 10.2. An associative array with keys in U and values in T can be implemented as a $(U \times T)$-valued array equipped with a hash function $h : U \to \{1,\ldots,n\}$.

$h^{-1}(i)$ of all elements mapped to i is called the *load* on i, and when this load contains more than one of the keys we are trying to store in our hash table we say there is a **collision** at i. Collisions are problem for us—if two keys map to the same index, then what should we store at that index? We have to store both values somehow. For now let's say we do this in the simplest way possible: storing at each index i in the array a linked list (or more abstractly, some sort of bucket-like object) consisting of all values whose keys are mapped to i. Thus, lookup takes $O\left(h^{-1}(i)\right)$ time, which may be poor if there are collisions at i. Rather than thinking about efficient ways to handle collisions,[1] let's try to reason about the probability of having collisions if we choose our hash functions well.

10.2 Hash Families

Without any prior information about which elements of U will occur as keys, the best we can do is to choose our hash function h at random from a suitable hash family. A **hash family** on U is a set $\mathscr{H}$ of functions $U \to \{1,\ldots,n\}$. Technically speaking, $\mathscr{H}$ should come equipped with a probability distribution, but usually we just take the uniform distribution on $\mathscr{H}$, so that each hash function is equally likely to be chosen.

If we want to avoid collisions, it is reasonable to hope that, for any fixed $x_1, x_2 \in U$ $(x_1 \neq x_2)$, the values $h(x_1)$ and $h(x_2)$ are completely uncorrelated as h ranges through the sample space $\mathscr{H}$. This leads to the following definition:

Definition. A hash family $\mathscr{H}$ on U is said to be **universal** if, for any $x_1, x_2 \in U$ $(x_1 \neq x_2)$, we have

$$\Pr_{h \in \mathscr{H}}\left[h(x_1) = h(x_2)\right] \leq \frac{1}{n}.$$

[1] If you are expecting lots of collisions, a more efficient way to handle things is to create a two-layered hash table, where each element of A is itself a hash table with its own, different hash function. In order to have collisions in a two-layer hash table, the same pair of keys must collide under two different hash functions. If the hash functions are chosen well (e.g., if the hash functions are chosen randomly), then this is extremely unlikely. Of course, if you want to be even more sure that collisions won't occur, you can make a three-layer hash table, and so on. There is a trade-off, though: introducing unnecessary layers of hashing comes with a time and space overhead which, while it may not show up in the big-O analysis, makes a difference in practical applications.

Similarly, $\mathscr{H}$ is said to be **ϵ-universal** if for any $x_1 \neq x_2$ we have

$$\Pr_{h \in \mathscr{H}} \left[h(x_1) = h(x_2) \right] \le \epsilon.$$

The consequences of the above hypotheses with regard to collisions are as follows:

Proposition 10.1. *Let $\mathscr{H}$ be a universal hash family on U. Fix some subset $S \subseteq U$ and some element $x \in U$. Pick $h \in \mathscr{H}$ at random. The expected number of elements of S that map to $h(x)$ is at most $1 + \frac{|S|}{n}$. In symbols,*

$$\mathop{\mathbb{E}}_{h \in \mathscr{H}} \left[\left| h^{-1}\left(h(x)\right) \right| \right] \le 1 + \frac{|S|}{n}.$$

If $\mathscr{H}$ is ϵ-universal rather than universal, then the same holds when $1 + \frac{|S|}{n}$ is replaced by $1 + \epsilon |S|$.

Proof. For a proposition φ with random parameters, let I_φ be the indicator random variable which equals 1 if φ is true and equals 0 otherwise. The fact that $\mathscr{H}$ is universal means that for each $x' \in U \setminus \{x\}$ we have

$$\mathop{\mathbb{E}}_{h \in \mathscr{H}} \left[I_{h(x) = h(x')} \right] \le \frac{1}{n}.$$

Thus by the linearity of expectation, we have

$$\begin{aligned}
\mathbb{E}\left[\left| h^{-1}\left(h(x)\right) \cap S \right| \right] &= I_{x \in S} + \mathop{\mathbb{E}}_{h \in \mathscr{H}} \left[\sum_{\substack{x' \in S \\ x' \neq x}} I_{h(x) = h(x')} \right] \\
&= I_{x \in S} + \sum_{\substack{x' \in S \\ x' \neq x}} \mathop{\mathbb{E}}_{h \in \mathscr{H}} \left[I_{h(x) = h(x')} \right] \\
&\le 1 + |S| \cdot \tfrac{1}{n}.
\end{aligned}$$

The reasoning is almost identical when $\mathscr{H}$ is ϵ-universal rather than universal. □

Corollary 10.2. *For a hash table in which the hash function is chosen from a universal family, insertion, deletion, and lookup have expected running time $O\left(1 + \frac{|S|}{n}\right)$, where $S \subseteq U$ is the set of keys which actually occur. If instead the hash family is ϵ-universal, then the operations have expected running time $O\left(1 + \epsilon |S|\right)$.*

Corollary 10.3. *Consider a hash table of size n with keys in U, whose hash function is chosen from a universal hash family. Let $S \subseteq U$ be the set of keys which actually occur. If $|S| = O(n)$, then insertion, deletion, and lookup have expected running time $O(1)$.*

Let $\mathscr{H}$ be a universal hash family on U. If $|S| = O(n)$, then the expected load on each index is $O(1)$. Does this mean that a typical hash table has $O(1)$ load at each index? Surprisingly, the answer is no, even when the hash function is chosen well. We'll see this below when we look at examples of universal hash families.

Examples 10.4.

1. The set of *all* functions $h : U \to \{1, \ldots, n\}$ is certainly universal. In fact, we could not hope to get any more balanced than this:

- For any $x \in U$, the random variable $h(x)$ (where h is chosen at random) is uniformly distributed on the set $\{1,\ldots,n\}$.
- For any pair $x_1 \neq x_2$, the random variables $h(x_1), h(x_2)$ are independent. In fact, for any finite subset $\{x_1,\ldots,x_k\} \subseteq U$, the tuple $\big(h(x_1),\ldots,h(x_k)\big)$ is uniformly distributed on $\{1,\ldots,n\}^k$.

The load on each index i is a binomial random variable with parameters $\left(|S|, \frac{1}{n}\right)$.

Fact. *When p is small and N is large enough that Np is moderately sized, the binomial distribution with parameters (N,p) is approximated by the* **Poisson distribution** *with parameter Np. That is, if X is a binomial random variable with parameters (N,p), then*

$$\Pr\left[X = k\right] \approx \frac{(Np)^k}{k!} e^{-Np} \quad (k \geq 0).$$

In our case, $N = |S|$ and $p = \frac{1}{n}$. Thus, if L_i is the load on index i, then

$$\Pr\left[L_i = k\right] \approx \frac{\left(\frac{|S|}{n}\right)^k}{k!} e^{-|S|/n}.$$

For example, if $|S| = n$, then

$$\begin{aligned} \Pr\left[L_i = 0\right] &\approx e^{-1} \approx 0.3679, \\ \Pr\left[L_i = 1\right] &\approx e^{-1} \approx 0.3679, \\ \Pr\left[L_i = 2\right] &\approx \tfrac{1}{2} e^{-1} \approx 0.1839, \\ &\vdots \end{aligned}$$

Further calculation shows that, when $|S| = n$, we have

$$\mathbb{E}\left[\max_{1 \leq i \leq n} L_i\right] = \Theta\left(\frac{\lg n}{\lg\lg n}\right).$$

Moreover, with high probability, $\max L_i$ does not exceed $O\left(\frac{\lg n}{\lg\lg n}\right)$. Thus, a typical hash table with $|S| = n$ and h chosen uniformly from the set of all functions looks like Figure 10.3: about 37% of the buckets empty, about 37% of the buckets having one element, and about 26% of the buckets having more than one element, incuding some buckets with $\Theta\left(\frac{\lg n}{\lg\lg n}\right)$ elements.

2. In Problem Set 4 we considered the hash family

$$\mathscr{H} = \left\{h_p : p \leq k \text{ and } p \text{ is prime}\right\},$$

where $h_p : \{0,\ldots,2^m - 1\} \to \{0,\ldots,k-1\}$ is the function

$$h_p(x) = x \bmod p.$$

In Problem 4(a) you proved that, for each $x \neq y$, we have

$$\Pr_p\left[h_p(x) = h_p(y)\right] \leq \frac{m \ln k}{k}.$$

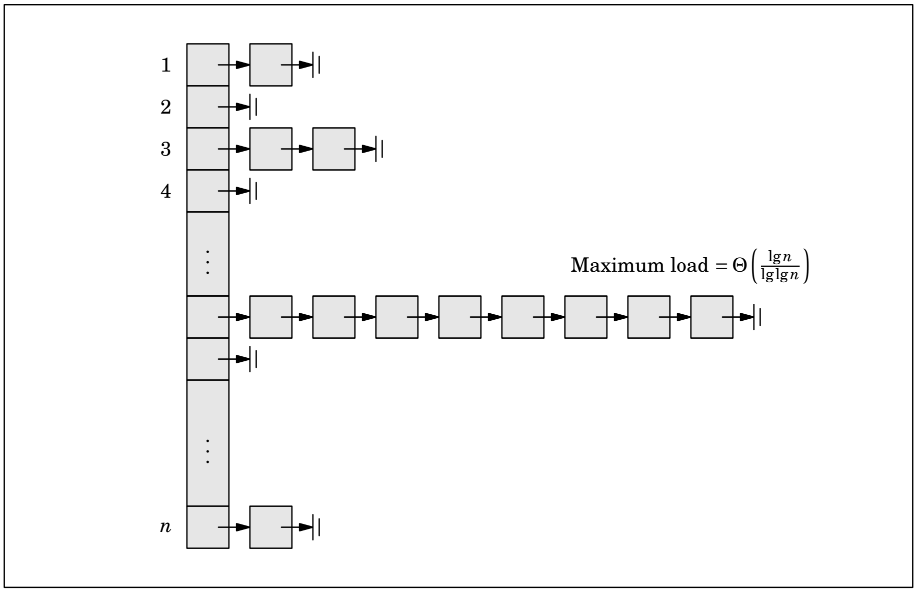


Figure 10.3. A typical hash table with $|S| = n$ and h chosen uniformly from the family of all functions $U \to \{1,\ldots,n\}$.

3. In Problem Set 5, we fixed a prime p and considered the hash family
$$\mathcal{H} = \left\{h_{\vec{a}} : \vec{a} \in \mathbb{Z}_p^m\right\},$$
where $h_{\vec{a}} : \mathbb{Z}_p^m \to \mathbb{Z}_p$ is the dot product
$$h_{\vec{a}}(\vec{x}) = \vec{x} \cdot \vec{a} = \sum x_i a_i \pmod{p}.$$

4. In Problem Set 6, we fixed a prime p and positive integers m and k and considered the hash family
$$\mathcal{H} = \left\{h_A : A \in \mathbb{Z}_p^{k\times m}\right\},$$
where $h_A : \mathbb{Z}_p^m \to \mathbb{Z}_p^k$ is the function
$$h_A(\vec{x}) = A\vec{x}.$$

5. If $\mathcal{H}_1$ is an ϵ_1-universal hash family of functions $\{0,1\}^m \to \{0,1\}^k$ and $\mathcal{H}_2$ is an ϵ_2-universal hash family of functions $\{0,1\}^k \to \{0,1\}^\ell$, then[2]
$$\mathcal{H} = \mathcal{H}_2 \circ \mathcal{H}_1 = \{h_2 \circ h_1 : h_1 \in \mathcal{H}_1, h_2 \in \mathcal{H}_2\}$$
is an $(\epsilon_1 + \epsilon_2)$-universal hash family of functions $\{0,1\}^m \to \{0,1\}^\ell$. To see this, note that for any $x \neq x'$, the union bound gives
$$\Pr_{\substack{h_1\in\mathcal{H}_1\\ h_2\in\mathcal{H}_2}}\left[h_2 \circ h_1(x) = h_2 \circ h_1(x')\right]$$

[2] To fully specify $\mathcal{H}$, we have to give not just a set but also a probability distribution. The hash families $\mathcal{H}_1$ and $\mathcal{H}_2$ come with probability distributions, so there is an induced distribution on $\mathcal{H}_1 \times \mathcal{H}_2$. We then equip $\mathcal{H}$ with the distribution induced by the map $\mathcal{H}_1 \times \mathcal{H}_2 \to \mathcal{H}$, $(h_1, h_2) \mapsto h_2 \circ h_1$. You could consider this a mathematical technicality if you wish: if $\mathcal{H}_1$ and $\mathcal{H}_2$ are given uniform distributions (as they typically are), then the distribution on $\mathcal{H}_1 \times \mathcal{H}_2$ is also uniform. The distribution on $\mathcal{H}$ need not be uniform, however: an element of $\mathcal{H}$ is more likely to be chosen if it can be expressed in multiple ways as the composition of an element of $\mathcal{H}_2$ with an element of $\mathcal{H}_1$.

$$
\begin{aligned}
&= \Pr\Big[h_1(x) = h_1(x') \text{ or } \Big(h_1(x) \neq h_1(x') \text{ and } h_2 \circ h_1(x) = h_2 \circ h_1(x')\Big)\Big] \\
&\leq \Pr\Big[h_1(x) = h_1(x')\Big] + \Pr\Big[h_1(x) \neq h_1(x') \text{ and } h_2 \circ h_1(x) = h_2 \circ h_1(x')\Big] \\
&\leq \epsilon_1 + \epsilon_2.
\end{aligned}
$$

In choosing the parameters to build a hash table, there is a tradeoff. Making n larger decreases the likelihood of collisions, and thus decreases the expected running time of operations on the table, but also requires the allocation of more memory, much of which is not even used to store data. In situations where avoiding collisions is worth the memory cost (or in applications other than hash tables, when the corresponding tradeoff is worth it), we can make n much larger than S.

Proposition 10.5. *Let $\mathcal{H}$ be a universal hash family $U \to \{1, \ldots, n\}$. Let $S \subseteq U$ be the the set of keys that occur. Then the expected number of collisions is at most $\binom{|S|}{2} \cdot \frac{1}{n}$. In symbols,*

$$\mathop{\mathbb{E}}_{h \in \mathcal{H}}\left[\sum_{x \neq x' \in U} I_{h(x)=h(x')}\right] \leq \binom{|S|}{2} \cdot \frac{1}{n}.$$

Proof. There are $\binom{|S|}{2}$ pairs of distinct elements in S, and each pair has probability at most $\frac{1}{n}$ of causing a collision. The result follows from linearity of expectation. □

Corollary 10.6. *If $n \geq 100\,|S|^2$, then the expected number of collisions is less than* 1/200*, and the probability that a collision exists is less than* 1/200*.*

Proof. Apply the Markov bound. □

Thus, if n is sufficiently large compared to S, a typical hash table consists mostly of empty buckets, and with high probability, there is at most one element in each bucket.

As we mentioned above, choosing a large n for a hash table is expensive in terms of space. While the competing goals of fast table operations and low storage cost are a fact of life if nothing is known about S in advance, we will see in recitation that, if S is known in advance, it is feasible to construct a **perfect** hash table, i.e., a hash table in which there are no collisions. Of course, the smallest value of n for which this is possible is $n = |S|$. As we will see in recitation, there are reasonably efficient algorithms to construct a perfect hash table with $n = O(|S|)$.

10.3 Amortization

What if the size of S is not known in advance? In order to allocate the array for a hash table, we must choose the size n at creation time, and may not change it later. If $|S|$ turns out to be significantly greater than n, then there will always be lots of collisions, no matter which hash function we choose.

Luckily, there is a simple and elegant solution to this problem: **table doubling**. The idea is to start with some particular table size $n = O(1)$. If the table gets filled, simply create a new table of size $2n$ and migrate all the old elements to it. While this migration operation is costly, it happens infrequently enough that, on the whole, the strategy of table doubling is efficient.

Let's take a closer look. To simplify matters, let's assume that only insertions and lookups occur, with no deletions. What is the worst-case cost of a single operation on the hash table?

- Lookup: $O(1)$, as usual.
- Insertion: $O(n)$, if we have to double the table.

Thus, the worst-case total running time of k operations ($k = |S|$) on the hash table is

$$O(1 + \cdots + k) = O\left(k^2\right).$$

The crucial observation is that this bound is *not* tight. Table doubling only happens after the second, fourth, eighth, etc., insertions. Thus, the total cost of k insertions is

$$k \cdot O(1) + O\left(\sum_{j=0}^{\lg k} 2^j\right) = O(k) + O(2k) = O(k).$$

Thus, in any sequence of insertion and lookup operations on a dynamically doubled hash table, the average, or **amortized**, cost per operation is $O(1)$. This sort of analysis, in which we consider the total cost of a sequence of operations rather than the cost of a single step, is called **amortized analysis**. In the next lecture we will introduce methods of analyzing amortized running time.

Lecture 11

Amortized Analysis

Supplemental reading in CLRS: Chapter 17

Data structures typically support several different types of operations, each with its own cost (e.g., time cost or space cost). The idea behind amortized analysis is that, even when expensive operations must be performed, it is often possible to get away with performing them rarely, so that the average cost per operation is not so high. It is important to realize that these "average costs" are not expected values—there needn't be any random events.[1] Instead, we are considering the worst-case average cost per operation in a sequence of many operations. In other words, we are interested in the asymptotic behavior of the function

$$C(n) = \frac{1}{n} \cdot \big(\text{worst-case total cost of a sequence of } n \text{ operations}\big)$$

(possibly with some condition on how many times each type of operation may occur). "Worst-case" means that no adversary could choose a sequence of n operations that gives a worse running time.

In this lecture we discuss three methods of amortized analysis: aggregate analysis, the accounting method, and the potential method.

11.1 Aggregate Analysis

In **aggregate analysis**, one assumes that there is no need to distinguish between the different operations on the data structure. One simply asks, what is the cost of performing a sequence of n operations, of any (possibly mixed) types?

Example. Imagine a stack[2] S with three operations:

- $\textsc{Push}(S,x)$ – $\Theta(1)$ – pushes object x onto the stack
- $\textsc{Pop}(S)$ – $\Theta(1)$ – pops and returns the top object of S

[1] There could be random events, though. It makes sense to talk about the worst-case amortized expected running time of a randomized algorithm, for example: one considers the average expected cost per operation in the string of operations which has the longest expected running time.

[2] A **stack** is a data structure S with two primitive operations: $\textsc{Push}(S,x)$, which stores x, and $\textsc{Pop}(S)$, which removes and returns the most recently pushed object. Thus, a stack is "last in, first out," whereas a queue is "first in, first out."

- MULTIPOP(S,k) – $\Theta(\min\{|S|,k\})$ – pops the top k items from the stack (or until empty) and returns the last item:

```
while $S$ is not empty and $k > 0$ do
    $x \leftarrow$ POP$(S)$
    $k \leftarrow k - 1$
return $x$
```

Suppose we wish to analyze the the running time for a sequence of n PUSH, POP, and MULTIPOP operations, starting with an empty stack. Considering individual operations without amortization, we would say that a MULTIPOP operation could take $\Theta(|S|)$ time, and $|S|$ could be as large as $n-1$. So in the hypothetical worst case, a single operation could take $\Theta(n)$ time, and n such operations strung together would take $\Theta\left(n^2\right)$ time.

However, a little more thought reveals that such a string of operations is not possible. While a single POP could take $\Theta(n)$ time, it would have to be preceded by $\Theta(n)$ PUSH operations, which are cheap. Taken together, the $\Theta(n)$ PUSH operations and the one MULTIPOP operation take $\Theta(n) \cdot \Theta(1) + 1 \cdot \Theta(n) = \Theta(n)$ time; thus, each operation in the sequence takes $\Theta(1)$ time on average. In general, if there occur r MULTIPOP operations with arguments $k_1, \ldots, k_r$, then there must also occur at least $k_1 + \cdots + k_r$ PUSH operations, so that there are enough items in the stack to pop. (To simplify notation, we assume that k is never chosen larger than $|S|$.) Thus, in a string of n operations, the total cost of all non-$O(1)$ operations is bounded above by $O(n)$, so the total cost of all operations is $O(n) \cdot O(1) + O(n) = O(n)$. Thus, the amortized running time per operation is $O(1)$.

11.2 Accounting Method

Unlike aggregated analysis, the **accounting method** assigns a different cost to each type of operation. The accounting method is much like managing your personal finances: you can estimate the costs of your operations however you like, as long as, at the end of the day, the amount of money you have set aside is enough to pay the bills. The estimated cost of an operation may be greater or less than its actual cost; correspondingly, the surplus of one operation can be used to pay the debt of other operations.

In symbols, given an operation whose actual cost is c, we assign an amortized (estimated) cost $\widehat{c}$. The amortized costs must satisfy the condition that, for *any* sequence of n operations with actual costs $c_1, \ldots, c_n$ and amortized costs $\widehat{c}_1, \ldots, \widehat{c}_n$, we have

$$\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i.$$

As long as this condition is met, we know that the amortized cost provides an upper bound on the actual cost of any sequence of operations. The difference between the above sums is the total surplus or "credit" stored in the data structure, and must at all times be nonnegative. In this way, the accounting model is like a debit account.

Example. Perhaps you have bought pre-stamped envelopes at the post office before. In doing so, you pay up-front for both the envelopes and the postage. Then, when it comes time to send a letter, no additional charge is required. This accounting can be seen as an amortization of the cost of sending a letter:

Operation	Actual cost c_i	Amortized cost $\widehat{c}_i$
Buy an envelope	5¢	49¢
Mail a letter	44¢	0¢

Obviously, for any valid sequence of operations, the amortized cost is at least as high as the actual cost. However, the amortized cost is easier to keep track of—it's one fewer item on your balance sheet.

Example. For the stack in Section 11.1, we could assign amortized costs as follows:

Operation	Actual cost c_i	Amortized cost $\widehat{c}_i$
Push	1	2
Pop	1	0
Multipop	$\min\{\lvert S\rvert, k\}$	0

When an object is pushed to the stack, it comes endowed with enough credit to pay not only for the operation of pushing it onto the stack, but also for whatever operation will eventually remove it from the stack, be that a Pop, a Multipop, or no operation at all.

11.3 Potential Method

The **potential method** is similar in spirit to the accounting method. Rather than assigning a credit to each element of the data structure, the potential method assigns a credit to the entire data structure as a whole. This makes sense when the total credit stored in the data structure is a function only of its state and not of the sequence of operations used to arrive at that state. We think of the credit as a "potential energy" (or just "potential") for the data structure.

The strategy is to define a potential function Φ which maps a state D to a scalar-valued potential $\Phi(D)$. Given a sequence of n operations with actual costs $c_1, \ldots, c_n$, which transform the data structure from its initial state D_0 through states $D_1, \ldots, D_n$, we define heuristic costs

$$\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

This rule can be seen as the conservation of energy: it says that the surplus (or deficit) cost $\widehat{c}_i - c_i$ caused by a given operation must be equal to the change in potential of the data structure caused by that operation. An operation which increases the potential of the system is assigned a positive heuristic cost, whereas an operation which decreases the potential of the system is assigned a negative heuristic cost.

Summing the heuristic costs of all n operations, we find

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c}_i &= \sum_{i=1}^{n} c_i + \Phi(D_i) - \Phi(D_{i-1}) \qquad \text{(a telescoping sum)} \\ &= \left(\sum_{i=1}^{n} c_i\right) + \Phi(D_n) - \Phi(D_0).\end{aligned}$$

Thus, the total credit stored in the data structure is $\Phi(D_n) - \Phi(D_0)$. This quantity must remain nonnegative at all times in order to ensure that the amortized cost provides an upper bound on the actual cost of any sequence of operations. (Any function Φ which satisfies this property can be used as the potential function.) One often chooses the potential function Φ so that $\Phi(D_0) = 0$; then one must check that Φ remains nonnegative at all times.

Example. Continuing the stack example from Sections 11.1 and 11.2, we define the potential of a stack S to be $\Phi(S) = |S|$, the number of elements in the stack. An empty stack has zero potential, and clearly Φ is always nonnegative, so Φ is an admissible potential function. Then the heuristic costs of the stack operations are as follows:

- A PUSH operation increases the size of S by 1, and has actual cost $c_{\text{PUSH}} = 1$. Thus, the amortized cost of a PUSH operation is

$$\widehat{c}_{\text{PUSH}} = c_{\text{PUSH}} + \Phi(D_{\text{new}}) - \Phi(D_{\text{old}}) = 1 + (|S_{\text{old}}| + 1) - |S_{\text{old}}| = 2.$$

- A POP operation decreases the size of S by 1, and has actual cost $c_{\text{POP}} = 1$. Thus, the amortized cost of a POP operation is

$$\widehat{c}_{\text{POP}} = c_{\text{POP}} + \Phi(D_{\text{new}}) - \Phi(D_{\text{old}}) = 1 + (|S_{\text{old}}| - 1) - |S_{\text{old}}| = 0.$$

- The operation MULTIPOP(S,k) decreases the size of S by $\min\{|S|,k\}$, and has actual cost $c_{\text{MULTIPOP}} = \min\{|S|,k\}$. Thus, the amortized cost of a MULTIPOP operation is

$$\widehat{c}_{\text{MULTIPOP}} = c_{\text{MULTIPOP}} + \Phi(D_{\text{new}}) - \Phi(D_{\text{old}}) = \min\{|S|,k\} + (|S_{\text{new}}| - |S_{\text{old}}|) = 0.$$

Thus, the amortized costs for this application of the potential method are the same as those we came up with using the accounting method in Section 11.2:

Operation	Actual cost c_i	Amortized cost $\widehat{c}_i$
PUSH	1	2
POP	1	0
MULTIPOP	$\min\{|S|,k\}$	0

11.4 Example: A Dynamically Resized Table

We now extend the example we started in Lecture 10: a dynamically doubled table T. This time we will forget about keys, inserting each new value into the next available block in the array $T.arr$. If $T.arr$ is full, then we create a new array of size $2|T.arr|$, copy over the old elements of T, perform the insertion, and reassign $T.arr$ to point to that array. Likewise, in §11.4.5 we ask you to devise an extension of our data structure which supports deletions (specifically, pops) as well as insertions, and halves the array $T.arr$ when appropriate in addition to doubling it when appropriate. In this way, T never takes up much more space than it has to.

For now, assume that T only supports insertions and lookups. We will store in T two things: the underlying array $T.arr$, and an integer $T.num$ which keeps track of the number of entries in $T.arr$ which have yet been populated. Thus, the array is full and needs to be doubled upon the next insertion if and only if $|T.arr| = T.num$. So the table lookup will be a simple array lookup operation, which takes $O(1)$ time, and the insertion operation will be defined as follows:

```
Algorithm: TABLE-INSERT($T, x$)
if $|T.arr| = 0$ then
    $T.arr \leftarrow$ new array of size 1
▷ If table is full, double it
if $T.num = |T.arr|$ then
    $arr' \leftarrow$ new array of size $2|T.arr|$
    Migrate all items from $T.arr$ to $arr'$
    $T.arr \leftarrow arr'$
▷ Insert $x$ into the array
$T.arr[T.num] \leftarrow x$
$T.num \leftarrow T.num + 1$
```

11.4.1 Actual cost per operation

A lookup operation on T is simply an array lookup, which takes $O(1)$ time. The running time of TABLE-INSERT, on the other hand, depends on whether the table needs to be doubled, which depends on the number of insertions which have happened prior to the current one. Specifically, starting from an initial table of size zero, the cost c_i of the ith insertion is

$$c_i = \begin{cases} 1 & \text{if the table has space} \\ \Theta(i) & \text{if the table has to be expanded: } \begin{pmatrix} i-1 \text{ to allocate the new table} \\ i-1 \text{ to migrate the old entries} \\ 1 \text{ to make the insertion} \\ 1 \text{ to reassign } T.arr \end{pmatrix} \end{cases}$$

The "Chicken Little" analysis would say that the worst possible string of n operations would consist entirely of operations which have the worst-case single-operation running time $\Theta(i)$, so that the sequence has running time

$$\Theta\left(1+2+\cdots+n\right) = \Theta\left(n^2\right).$$

This bound is not tight, however, because it is not possible for every operation to be worst-case.

11.4.2 Aggregate analysis

Aggregate analysis of our table's operations amounts to keeping track of precisely which operations will be worst-case. The table has to be expanded if and only if the table is full, so

$$c_i = \begin{cases} \Theta(i) & \text{if } i-1 \text{ is zero or an exact power of 2} \\ 1 & \text{otherwise.} \end{cases}$$

Thus, a string of n operations, m of which are insertion operations, would take time

$$\begin{aligned} & T\left(n-m \text{ lookups}\right) + T\left(m \text{ insertions}\right) \\ = & \left(\left(n-m\right)\cdot\Theta(1)\right) + \left(\sum_{\substack{k-1 \text{ is not an} \\ \text{exact power of 2}}} \Theta(1) \quad + \sum_{\substack{k-1 \text{ is an} \\ \text{exact power of 2}}} \Theta(k) \right) \end{aligned}$$

$$
\begin{aligned}
&= \Big((n-m)\cdot\Theta(1)\Big) + \left(\big(m - (2 + \lfloor \lg(m-1)\rfloor)\big)\cdot \Theta(1) \quad + \quad \sum_{j=0}^{\lfloor \lg(m-1)\rfloor} \Theta\left(2^j\right)\right) \\
&= \Theta(n-m) + \big(\Theta(m) - \Theta(\lg m) + \Theta(2m)\big) \\
&= \Theta(n) + \Theta(m) \\
&= \Theta(n) \quad \text{(since } m \le n\text{)}.
\end{aligned}
$$

Thus, a string of n operations takes $\Theta(n)$ time, regardless of how many of the operations are insertions and how many are lookups. Thus, aggregate analysis gives an amortized running time of $O(1)$ per operation.

11.4.3 Accounting analysis

We can set up a balance sheet to pay for the operations on our table as follows:

- Lookups cost \$1
- Insertions cost \$3
  - \$1 to insert our element into the array itself
  - \$2 to save up for the next time the array has to be expanded
    - One dollar will pay for migrating the element itself
    - One dollar will go to charity.

Because the current incarnation of $T.arr$ started with $\frac{1}{2}|T.arr|$ elements pre-loaded, it follows that, when $T.arr$ is full, only $\frac{1}{2}|T.arr|$ elements will have enough money in the bank to pay for their migration. However, there will also be $\$\frac{1}{2}|T.arr|$ of charity money saved up from the $\frac{1}{2}|T.arr|$ most recent insertions. This charity is exactly sufficient to pay for migrating the elements that don't have their own money. Thus, our balance sheet checks out; our assigned costs of \$1 and \$3 work. In particular, each operation's cost is $O(1)$.

11.4.4 Potential analysis

In our potential analysis, the potential energy will play the role of money in the bank: when the array needs to be doubled, the potential energy will exactly pay for migrating the old elements into the new array. Thus, we want to find a function which equals zero immediately after doubling the table and grows to size $|T.arr|$ as it is filled up. (To simplify notation, assume that copying an array of size k takes exactly k steps, rather than always using the more precise notation $\Theta(k)$.) The simplest function which satisfies these properties, and the one that will be most convenient to work with, is

$$\Phi(T) = 2\cdot T.num - |T.arr|.$$

Right after expansion, we have $T.num = \frac{1}{2}|T.arr|$, so $\Phi(T) = 0$; right before expasnion, we have $T.num = |T.arr|$, so $\Phi(T) = 2\cdot|T.arr| - |T.arr| = |T.arr|$.

The actual cost of a lookup is, as we said, $c_{\text{lookup}} = 1$. (Again, to simplify notation, let's just write 1 instead of $\Theta(1)$.) Using the potential Φ, the amortized cost of a lookup is

$$\widehat{c}_{\text{lookup}} = c_{\text{lookup}} + \Phi(T_{\text{new}}) - \Phi(T_{\text{old}}) = c_{\text{lookup}} = 1,$$

since $T_{\text{new}} = T_{\text{old}}$.

To find the amortized cost of an insertion, we must split into cases. In the first case, suppose the table does not need to be doubled as a result of the insertion. Then the amortized cost is

$$\begin{aligned}
\widehat{c} &= c + \Phi(T_{\text{new}}) - \Phi(T_{\text{old}}) \\
&= 1 + \left(2 \cdot T_{\text{new}}.num - |T_{\text{new}}.arr|\right) - \left(2 \cdot T_{\text{old}}.num - |T_{\text{old}}.arr|\right) \\
&= 1 + 2\left(T_{\text{new}}.num - T_{\text{new}}.num\right) - \left(|T_{\text{new}}.arr| - |T_{\text{old}}.arr|\right) \\
&= 1 + 2(1) + 0 \\
&= 3.
\end{aligned}$$

In the second case, suppose the table does need to be doubled as a result of the insertion. Then the actual cost of insertion is $|T_{\text{new}}.num|$, so the amortized cost of insertion is

$$\begin{aligned}
\widehat{c} &= c + \Phi(T_{\text{new}}) - \Phi(T_{\text{old}}) \\
&= T_{\text{new}}.num + \left(2 \cdot T_{\text{new}}.num - |T_{\text{new}}.arr|\right) - \left(2 \cdot T_{\text{old}}.num - |T_{\text{old}}.arr|\right) \\
&= \left(|T_{\text{old}}.arr| + 1\right) + 2\left(T_{\text{new}}.num - T_{\text{old}}.num\right) - \left(|T_{\text{new}}.arr| - |T_{\text{old}}.arr|\right) \\
&= \left(|T_{\text{old}}.arr| + 1\right) + 2(1) - \left(2\,|T_{\text{old}}.arr| - |T_{\text{old}}.arr|\right) \\
&= 3.
\end{aligned}$$

In both cases, the amortized running time of insertion is 3. (If we had picked a less convenient potential function Φ, the amortized running time would probably be different in the two cases. The potential analysis would still be valid if done correctly, but the resulting amortized running times probably wouldn't be as easy to work with.)

11.4.5 Exercise: Allowing deletions

Suppose we wanted our dynamically resized table to support pops (deleting the most recently inserted element) as well as insertions. It would be reasonable to want the table to halve itself when fewer than half the slots in the array are occupied, so that the table only ever takes up at most twice the space it needs to. Thus, a first attempt at a pop functionality might look like this (with the TABLE-INSERT procedure unchanged):

```
Algorithm: NAÏVE-POP($T$)
if $T.num = 0$ then
    error "Tried to pop from an empty table"
$r \leftarrow T.arr[T.num - 1]$
Unset $T.arr[T.num - 1]$
$T.num \leftarrow T.num - 1$
if $T.num \le \frac{1}{2}|T.arr|$ then
    $arr' \leftarrow$ new array of size $\lfloor \frac{1}{2}|T.arr| \rfloor$
    Migrate all items from $T.arr$ to $arr'$
    $T.arr \leftarrow arr'$
return $r$
```

Unfortunately, the worst-case running time of NAÏVE-POP is not so good.

Exercise 11.1. *Given n sufficiently large, produce a sequence of n TABLE-INSERT, lookup, and/or NAÏVE-POP operations which have total running time $\Theta(n^2)$. Thus, in any amortized analysis, at least one of the operations must have amortized running time at least $\Omega(n)$.*

Exercise 11.2. *Devise a way of supporting pops which still retains $O(1)$ worst-case amortized running time for each operation, and do an amortized analysis to prove that this is the case. You may redefine TABLE-INSERT or add more data fields to T if you wish (and of course you will have to define TABLE-POP), but it must still remain the case that, for every possible sequence of operations, T only takes up $O(T.num)$ space in memory at any given point in time.*

[Hint: An adversary can choose a series of operations for which TABLE-INSERT and NAÏVE-POP run slowly by making sure $T.num$ repeatedly crosses the critical thresholds for table doubling and table halving. (You must have used this fact somehow in your solution to Exercise 11.1.) So your first step will be to figure out how to place decisions about doubling and halving outside the control of your adversary.]

Lecture 12

Competitive Analysis

Supplemental reading in CLRS: None

12.1 Online and Offline Algorithms

For some algorithmic problems, the optimal response to a sequence of calls to a procedure depends not just on each input separately, but on the sequence of inputs as a whole. This is analogous to amortized analysis, in which the best running time analysis required us to consider the running time of an entire sequence of operations. When an algorithm expects to be given the entire sequence of inputs ahead of time, it is called an **offline algorithm**; when the algorithm is expected to give an answer after each individual input without knowledge of future inputs, it is called an **online algorithm**.

Example. Traders are faced with the following algorithmic problem: given today's stock prices (and the history of stock prices in the past), decide which stocks to buy and which stocks to sell. Obviously, this is an online problem for everybody. An attempt to treat the trading problem as an offline algorithmic problem is called *insider trading*.

Example. *Tetris* is an online game. The offline version of Tetris would be akin to a jigsaw puzzle, and would not require quick thinking. Of course, we would expect players of the offline game to produce much better packings than players of the online game.

Let's lay out a general setup for analyzing online and offline algorithms. Assume we are given an algorithm A, along with a notion of "cost" C_A. That is, if S is a sequence of inputs, then $C_A(S)$ is the cost of trusting A's response to the inputs in S. For example, A might be a stock broker and $C_A(S)$ might be the net profit after a sequence of days on the stock market. In that case, clients will probably try to learn as much as the can about the function C_A (for example, by observing its past outputs) when deciding whether to hire the stock broker.

In general, no online algorithm will perform optimally on all sequences of inputs—in other words, the so-called "**God's algorithm**" (which performs optimally on every sequence of inputs) will usually be impossible to describe in an online way. This is obvious in the stock market example: if you chose to buy today, what if prices drop tomorrow? If you chose to sell today, what if prices go up tomorrow?

Although an online algorithm may not ever be able to match the performance of God's algorithm, it may be possible for an online algorithm to perform *almost* as well as God's algorithm on every input.

Definition. An online algorithm A is said to be **α-competitive**[1] (where α is a positive constant) if there exists a constant k such that, for every sequence S of inputs, we have

$$C_A(S) \le \alpha \cdot C_{\text{OPT}}(S) + k,$$

where OPT is the optimal offline algorithm, A.K.A. God's algorithm.

12.2 Example: A Self-Organizing List

Problem 12.1. Design a data structure L representing a list of n key–value pairs (with distinct keys), satisfying the following constraints:

- The key–value pairs are stored in a linked list
- There is only one supported operation: ACCESS(x), which returns the element with key x. (It is assumed that the input x is always one of the keys which occurs in L.) The implementation of ACCESS(x) has two phases:
    1. Walk through the list until you find the element with key x. The cost of this phase is $\text{rank}_L(x)$, the rank of the element with key x.
    2. Reorder the list by making a sequence of adjacent transpositions[2] (in this lecture, the word "transposition" will always refer to adjacent transpositions). The cost of each transposition is 1; the sequence of transpositions performed is up to you, and may be empty.

Try to choose a sequence of transpositions which optimizes the performance of ACCESS, i.e., for which ACCESS is α-competitive where α is minimal among all possible online algorithms.

12.2.1 Worst-case inputs

No matter how we define our algorithm, an adversary (who knows what algorithm we are using) can always ask for the last element in our list, so that phase 1 always costs n, and the cost of a sequence S of such costly operations is at least

$$C_A(S) = \Omega\left(|S| \cdot n\right),$$

even disregarding the cost of any reordering we do.

12.2.2 Heuristic: Random inputs

As a heuristic (which will seem particularly well-chosen in retrospect), let's suppose that our inputs are random rather than worst-case. Namely, suppose the inputs are independent random variables, where the key x has probability $p(x)$ of being chosen. It should be believable (and you can prove if

[1] This is not the only notion of "competitive" that people might care about. We could also consider additive competitiveness, which is a refinement of the notion considered here: We say an algorithm A is *k-additively competitive* if there exists a constant k such that $C_A(S) \le C_{\text{OPT}}(S) + k$ for every sequence S of inputs. Thus, every k-additively competitive algorithm is 1-competitive in the sense defined above.

[2] An *adjacent transposition* is the act of switching two adjacent entries in the list. For example, transposing the elements B and C in the list $\langle A, B, C, D\rangle$ results in the list $\langle A, C, B, D\rangle$.

you wish) that, when $|S|$ is much larger than n, the optimal expected cost is achieved by immediately sorting L in decreasing order of $p(x)$, and keeping that order for the remainder of the operations.[3] In that case, the expected cost of a sequence S of ACCESS operations is

$$\mathop{\mathbb{E}}_{S}[C_A(S)] = \sum_{x \in L} p(x) \cdot \operatorname{rank}_P\left(p(x)\right) \cdot |S|,$$

where $P = \{p(y) : y \in L\}$.

In practice, we won't be given the distribution p ahead of time, but we can still estimate what p would be if it existed. We can keep track of the number of times each key appears as an input, and maintain the list in decreasing order of frequency. This *counting heuristic* has its merits, but we will be interested in another heuristic: the **move-to-front** (MTF) heuristic, which works as follows:

- After accessing an item x, move x to the head of the list. The total cost of this operation is $2 \cdot \operatorname{rank}_L(x) - 1$:
  - $\operatorname{rank}_L(x)$ to find x in the list
  - $\operatorname{rank}_L(x) - 1$ to perform the transpositions.[4]

To show that the move-to-front heuristic is effective, we will perform a **competitive analysis**.

Proposition 12.2. *The MTF heuristic is 4-competitive for self-organizing lists.*

As to the original problem, we are not making any assertion that 4 is minimal; for all we know, there could exist α-competitive heuristics for $\alpha < 4$.[5]

Proof.

- Let L_i be MTF's list after the ith access.
- Let L_i^* be OPT's list after the ith access.
- Let c_i be MTF's cost for the ith operation. Thus $c_i = 2\operatorname{rank}_{L_{i-1}}(x) - 1$.
- Let c_i^* be OPT's cost for the ith operation. Thus $c_i^* = \operatorname{rank}_{L_{i-1}^*}(x) + t_i$, where t_i is the number of transpositions performed by OPT during the ith operation.

We will do an amortized analysis. Define the potential function[6,7]

$$\Phi(L_i) = 2 \cdot \left(\#\text{ of inversions between } L_i \text{ and } L_i^*\right)$$

[3] The intuition is that the most likely inputs should be made as easy as possible to handle, perhaps at the cost of making some less likely inputs more expensive.

[4] The smallest number of transpositions needed to move x to the head of the list is $\operatorname{rank}_L(x) - 1$. These $\operatorname{rank}_L(x) - 1$ transpositions can be done in exactly one way. Once they are finished, the relative ordering of the elements other than x is unchanged. (If you are skeptical of these claims, try proving them.) For example, to move D to the front of the list $\langle A, B, C, D, E\rangle$, we can perform the transpositions $D \leftrightarrow C$, $D \leftrightarrow B$, and $D \leftrightarrow A$, ultimately resulting in the list $\langle D, A, B, C, E\rangle$.

[5] The question of whether $\alpha = 4$ is minimal in this example is not of much practical importance, since the costs used in this example are not very realistic. For example, in an actual linked list, it would be easy to move the nth entry to the head in $O(1)$ time (assuming you already know the location of both entries), whereas the present analysis gives it cost $\Theta(n)$. Nevertheless, it is instructive to study this example as a first example of competitive analysis.

[6] The notation $x <_{L_i} y$ means $\operatorname{rank}_{L_i}(x) < \operatorname{rank}_{L_i}(y)$. In other words, we are using L_i to define an order $<_{L_i}$ on the keys, and likewise for L_i^*.

[7] Technically, this is an abuse of notation. The truth is that $\Phi(L_i)$ depends not just on the list L_i, but also on L_0 and i (since L_i^* is gotten by allowing OPT to perform i operations starting with L_0). We will stick with the notation $\Phi(L_i)$, but if you like, you can think of Φ as a function on "list histories" rather than just lists.

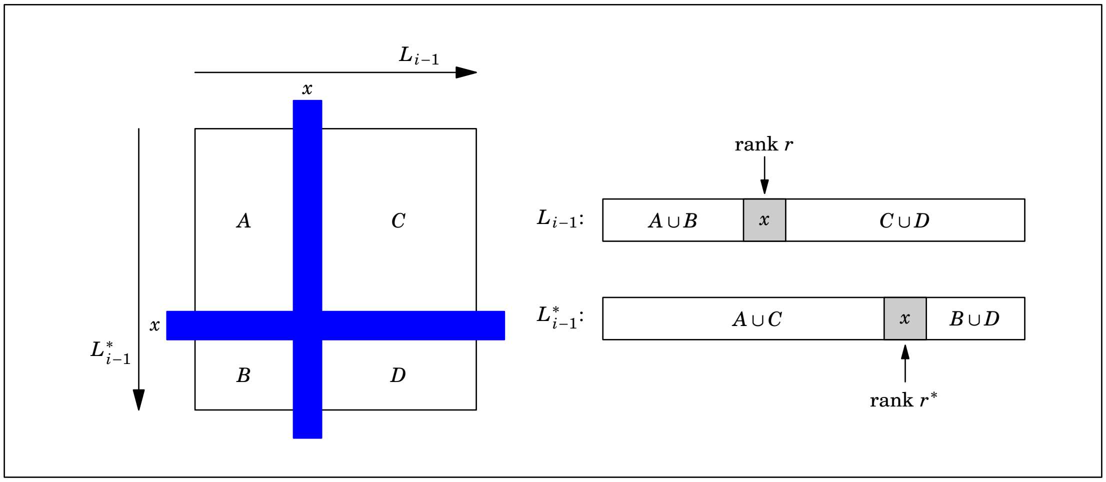


Figure 12.1. The keys in L_{i-1} other than x fall into four categories, as described in the proof of Proposition 12.2.

$$= 2 \cdot \left| \left\{ \text{unordered pairs } \{x,y\} : \, x <_{L_i} y \text{ and } y <_{L_i^*} x \right\} \right|.$$

For example, if $L_i = \langle E,C,A,D,B \rangle$ and $L_i^* = \langle C,A,B,D,E \rangle$, then $\Phi(L_i) = 10$, seeing as there are 5 inversions: $\{E,C\}$, $\{E,A\}$, $\{E,D\}$, $\{E,B\}$, and $\{D,B\}$. Note the following properties of Φ:

- $\Phi(L_i) \geq 0$ always, since the smallest possible number of inversions is zero.
- $\Phi(L_0) = 0$ if MTF and OPT start with the same list.
- A transposition either creates exactly 1 inversion or destroys exactly 1 inversion (namely, transposing $x \leftrightarrow y$ toggles whether $\{x,y\}$ is an inversion), so each transposition changes Φ by $\Delta\Phi = \pm 2$.

Let us focus our attention on a single operation, say the ith operation. The keys of L fall into four categories (see Figure 12.1):

- A: elements before x in L_{i-1} and L^*_{i-1}
- B: elements before x in L_{i-1} but after x in L^*_{i-1}
- C: elements after x in L_{i-1} but before x in L^*_{i-1}
- D: elements after x in L_{i-1} and L^*_{i-1}.

Let $r = \operatorname{rank}_{L_{i-1}}(x)$ and $r^* = \operatorname{rank}_{L^*_{i-1}}(x)$. Thus

$$r = |A| + |B| + 1 \quad \text{and} \quad r^* = |A| + |C| + 1.$$

Now, if we hold L^*_{i-1} fixed and pass from L_{i-1} to L_i by moving x to the head, we create exactly $|A|$ inversions (namely, the inversions $\{a,x\}$ for each $a \in A$) and destroy exactly $|B|$ inversions (namely, the inversions $\{b,x\}$ for each $b \in B$). Next, if we hold L_i fixed and pass from L^*_{i-1} to L^*_i by making whatever set of transpositions OPT chooses to make, we create at most t_i inversions (where t_i is the number of transpositions performed), since each transposition creates at most one inversion. Thus

$$\Phi(L_i) - \Phi(L_{i-1}) \leq 2\left(|A| - |B| + t_i\right).$$

The amortized cost of the ith insertion of MTF with respect to the potential function is therefore

$$\widehat{c}_i = c_i + \Phi(L_i) - \Phi(L_{i-1})$$

$$
\begin{aligned}
&\leq 2r + 2\left(|A| - |B| + t_i\right)\\
&= 2r + 2\left(|A| - (r - 1 - |A|) + t_i\right)\\
&= 4|A| + 2 + 2t_i\\
&\leq 4\left|r^* + t_i\right| \qquad \text{(since } r^* = |A| + |C| + 1 \geq |A| + 1\text{)}\\
&= 4c_i^*.
\end{aligned}
$$

Thus the total cost of a sequence S of operations using the MTF heuristic is

$$
\begin{aligned}
C_{\text{MTF}}(S) &= \sum_{i=1}^{|S|} c_i\\
&= \sum_{i=1}^{|S|} \left(\widehat{c}_i + \Phi(L_{i-1}) - \Phi(L_i)\right)\\
&\leq \left(\sum_{i=1}^{|S|} 4c_i^*\right) + \underbrace{\Phi(L_0)}_{0} - \underbrace{\Phi\left(L_{|S|}\right)}_{\geq 0}\\
&\leq 4 \cdot C_{\text{OPT}}(S).
\end{aligned}
$$

□

Note:

- We never found the optimal algorithm, yet we still successfully argued about how MTF competed with it.
- If we decrease the cost of a transposition to 0, then MTF becomes 2-competitive.
- If we start MTF and OPT with different lists (i.e., $L_0 \neq L_0^*$), then $\Phi(L_0)$ might be as large as $\Theta\left(n^2\right)$, since it could take $\Theta\left(n^2\right)$ transpositions to perform an arbitrary permutation on L. However, this does not affect α for purposes of our competitive analysis, which treats n as a fixed constant and analyzes the asymptotic cost as $|S| \to \infty$; the MTF heuristic is still 4-competitive under this analysis.

Lecture 13

Network Flow

Supplemental reading in CLRS: Sections 26.1 and 26.2

When we concerned ourselves with shortest paths and minimum spanning trees, we interpreted the edge weights of an undirected graph as distances. In this lecture, we will ask a question of a different sort. We start with a *directed* weighted graph G with two distinguished vertices s (the source) and t (the sink). We interpret the edges as unidirectional water pipes, with an edge's capacity indicated by its weight. The *maximum flow problem* then asks, how can one route as much water as possible from s to t?

To formulate the problem precisely, let's make some definitions.

Definition. A **flow network** is a directed graph $G = (V, E)$ with distinguished vertices s (the source) and t (the sink), in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v)$. We require that E never contain both (u, v) and (v, u) for any pair of vertices u, v (so in particular, there are no loops). Also, if $u, v \in V$ with $(u, v) \notin E$, then we define $c(u, v)$ to be zero. (See Figure 13.1).

In these notes, we will always assume that our flow networks are finite. Otherwise, it would be quite difficult to run computer algorithms on them.

Definition. Given a flow network $G = (V, E)$, a **flow** in G is a function $f : V \times V \to \mathbb{R}$ satisfying

1. Capacity constraint: $0 \le f(u, v) \le c(u, v)$ for each $u, v \in V$

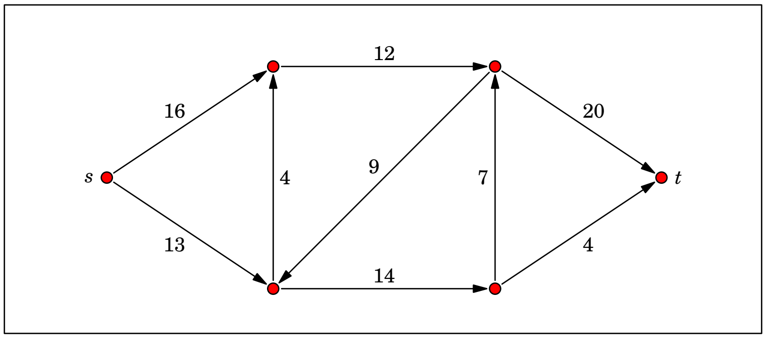


Figure 13.1. A flow network.

2. Flow conservation: for each $u \in V \setminus \{s,t\}$, we have[1]

$$\underbrace{\sum_{v \in V} f(v,u)}_{\text{flow into } u} = \underbrace{\sum_{v \in V} f(u,v)}_{\text{flow out of } u}.$$

In the case that flow conservation is satisfied, one can prove (and it's easy to believe) that the net flow out of s equals the net flow into t. This quantity is called the **flow value**, or simply the magnitude, of f. We write

$$\underbrace{|f|}_{\text{flow value}} = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) = \sum_{v \in V} f(v,t) - \sum_{v \in V} f(t,v).$$

Note that the definition of a flow makes sense even when G is allowed to contain both an edge and its reversal (and therefore is not truly a flow network). This will be important in §13.1.1 when we discuss augmenting paths.

13.1 The Ford–Fulkerson Algorithm

The **Ford–Fulkerson algorithm** is an elegant solution to the maximum flow problem. Fundamentally, it works like this:

```
1 while there is a path from $s$ to $t$ that can hold more water do
2     Push more water through that path
```

Two notes about this algorithm:

- The notion of "a path from s to t that can hold more water" is made precise by the notion of an *augmenting path*, which we define in §13.1.1.
- The Ford–Fulkerson algorithm is essentially a greedy algorithm. If there are multiple possible augmenting paths, the decision of which path to use in line 2 is completely arbitrary.[2] Thus, like any terminating greedy algorithm, the Ford–Fulkerson algorithm will find a locally optimal solution; it remains to show that the local optimum is also a global optimum. This is done in §13.2.

13.1.1 Residual Networks and Augmenting Paths

The Ford–Fulkerson algorithm begins with a flow f (initially the zero flow) and successively improves f by pushing more water along some path p from s to t. Thus, given the current flow f, we need

[1] In order for a flow of water to be sustainable for long periods of time, there cannot exist an accumulation of excess water anywhere in the pipe network. Likewise, the amount of water flowing into each node must at least be sufficient to supply all the outgoing connections promised by that node. Thus, the amount of water entering each node must equal the amount of water flowing out. In other words, the net flow into each vertex (other than the source and the sink) must be zero.

[2] There are countless different versions of the Ford–Fulkerson algorithm, which differ from each other in the heuristic for choosing which augmenting path to use. Different situations (in which we have some prior information about the nature of G) may call for different heuristics.

a way to tell how much more water a given path p can carry. To start, note that a chain is only as strong as its weakest link: if $p = \langle v_0, \ldots, v_n \rangle$, then

$$\begin{pmatrix}\text{amount of additional water} \\ \text{that can flow through } p\end{pmatrix} \quad = \quad \min_{1 \le i \le n} \begin{pmatrix}\text{amount of additional water that} \\ \text{can flow directly from } v_{i-1} \text{ to } v_i\end{pmatrix}.$$

All we have to know now is how much additional water can flow directly between a given pair of vertices u, v. If $(u,v) \in E$, then clearly the flow from u to v can be increased by up to $c(u,v) - f(u,v)$. Next, if $(v,u) \in E$ (and therefore $(u,v) \notin E$, since G is a flow network), then we can simulate an increased flow from u to v by decreasing the throughput of the edge (v,u) by as much as $f(v,u)$. Finally, if neither (u,v) nor (v,u) is in E, then no water can flow directly from u to v. Thus, we define the **residual capacity** between u and v (with respect to f) to be

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise.} \end{cases} \tag{13.1}$$

When drawing flows in flow networks, it is customary to label an edge (u,v) with both the capacity $c(u,v)$ and the throughput $f(u,v)$, as in Figure 13.2.

Next, we construct a directed graph G_f, called the **residual network** of f, which has the same vertices as G, and has an edge from u to v if and only if $c_f(u,v)$ is positive. (See Figure 13.2.) The weight of such an edge (u,v) is $c_f(u,v)$. Keep in mind that $c_f(u,v)$ and $c_f(v,u)$ may both be positive for some pairs of vertices u,v. Thus, the residual network of f is in general not a flow network.

Equipped with the notion of a residual network, we define an **augmenting path** to be a path from s to t in G_f. If p is such a path, then by virtue of our above discussion, we can perturb the flow f at the edges of p so as to increase the flow value by $c_f(p)$, where

$$c_f(p) = \min_{(u,v) \in p} c_f(u,v). \tag{13.2}$$

The way to do this is as follows. Given a path p, we might as well assume that p is a simple path.[3] In particular, p will never contain a given edge more than once, and will never contain both an edge and its reversal. We can then define a new flow f' in the residual network (even though the residual network is not a flow network) by setting

$$f'(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \in p \\ 0 & \text{otherwise.} \end{cases}$$

Exercise 13.1. *Show that f' is a flow in G_f, and show that its magnitude is $c_f(p)$.*

Finally, we can "**augment**" f by f', obtaining a new flow $f \uparrow f'$ whose magnitude is $|f| + |f'| =$

[3] Recall that a *simple path* is a path which does not contain any cycles. If p is not simple, we can always pare p down to a simple path by deleting some of its edges (see Exercise B.4-2 of CLRS, although the claim I just made is a bit stronger). Doing so will never decrease the residual capacity of p (just look at (13.2)).

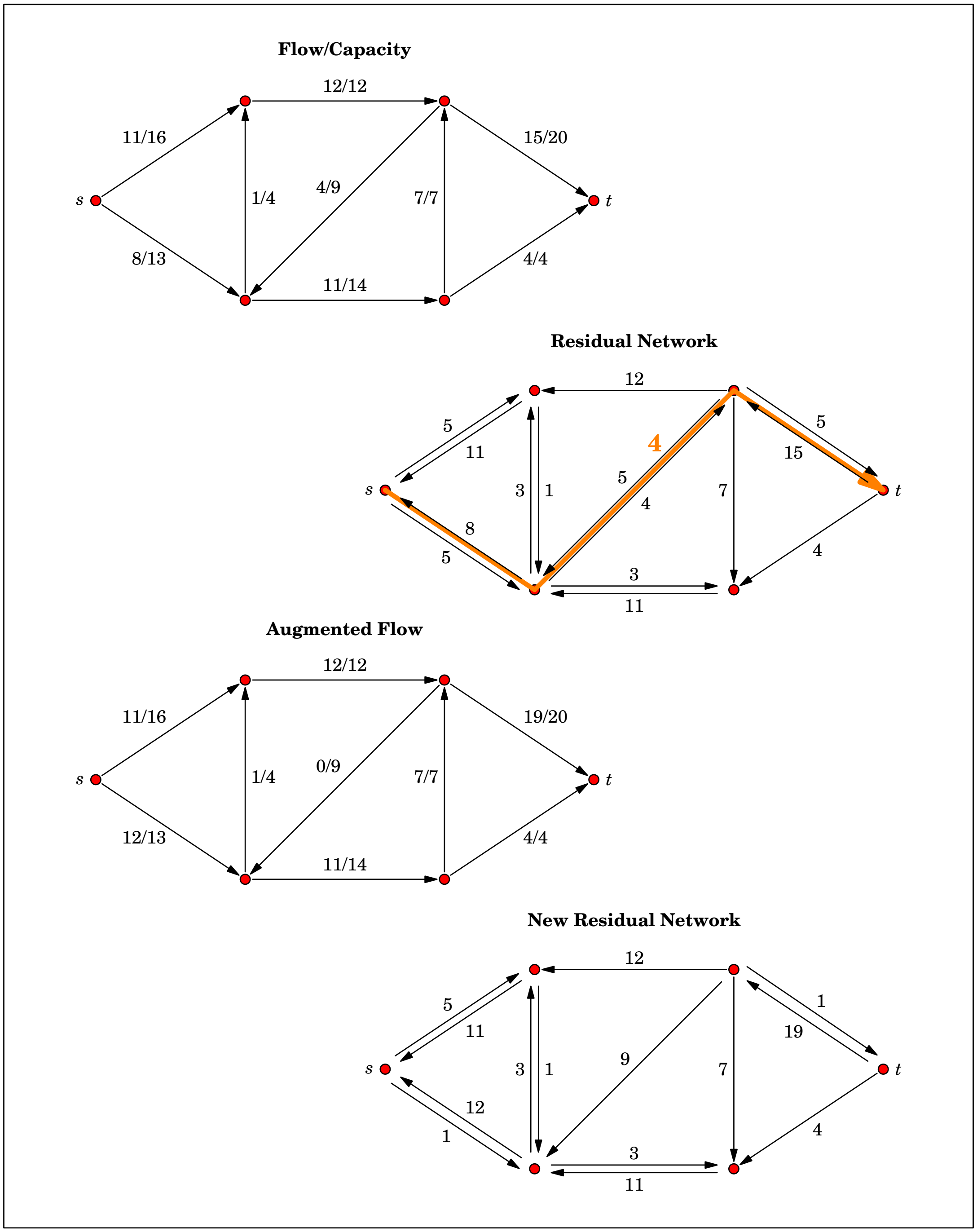


Figure 13.2. We begin with a flow network G and a flow f: the label of an edge (u,v) is "a/b," where $a = f(u,v)$ is the flow through the edge and $b = c(u,v)$ is the capacity of the edge. Next, we highlight an augmenting path p of capacity 4 in the residual network G_f. Next, we augment f by the augmenting path p. Finally, we obtain a new residual network in which there happen to be no more augmenting paths. Thus, our new flow is a maximum flow.

$|f| + c_f(p)$. It is defined by[4]

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + c_f(p) & \text{if } (u,v) \in p \text{ and } (u,v) \in E \\ f(u,v) - c_f(p) & \text{if } (v,u) \in p \text{ and } (u,v) \in E \\ f(u,v) & \text{otherwise.} \end{cases}$$

Lemma 13.1 (CLRS Lemma 26.1)**.** *Let f be a flow in the flow network $G = (V, E)$ and let f' be a flow in the residual network G_f. Let $f \uparrow f'$ be the augmentation of f by f', as described in* (13.3)*. Then*

$$|f \uparrow f'| = |f| + |f'|.$$

Proof sketch. First, we show that $f \uparrow f'$ obeys the capacity constraint for each edge in E and obeys flow conservation for each vertex in $V \setminus \{s,t\}$. Thus, $f \uparrow f'$ is truly a flow in G. Next, we obtain the identity $|f \uparrow f'| = |f| + |f'|$ by simply expanding the left-hand side and rearranging terms in the summation. □

13.1.2 Pseudocode Implementation of the Ford–Fulkerson Algorithm

Now that we have laid out the necessary conceptual machinery, let's give more detailed pseudocode for the Ford–Fulkerson algorithm.

```
Algorithm: FORD–FULKERSON(G)
▷ Initialize flow $f$ to zero
for each edge $(u,v) \in E$ do
    $(u,v).f \leftarrow 0$
▷ The following line runs a graph search algorithm (such as BFS or DFS)* to find a
  path from $s$ to $t$ in $G_f$
while there exists a path $p : s \rightsquigarrow t$ in $G_f$ do
    $c_f(p) \leftarrow \min\{c_f(u,v) : (u,v) \in p\}$
    for each edge $(u,v) \in p$ do
        ▷ Because $(u,v) \in G_f$, it must be the case that either $(u,v) \in E$ or $(v,u) \in E$.
        ▷ And since $G$ is a flow network, the "or" is exclusive: $(u,v) \in E$ xor $(v,u) \in E$.
        if $(u,v) \in E$ then
            $(u,v).f \leftarrow (u,v).f + c_f(p)$
        else
            $(v,u).f \leftarrow (v,u).f - c_f(p)$
```

* For more information about breath-first and depth-first searches, see Sections 22.2 and 22.3 of CLRS.

Here, we use the notation $(u,v).f$ synonymously with $f(u,v)$; though, the notation $(u,v).f$ suggests a convenient implementation decision in which we attach the value of $f(u,v)$ as satellite data to the

[4] In a more general version of augmentation, we don't require p to be a simple path; we just require that f' be some flow in the residual network G_f. Then we define

$$f'(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E \\ 0 & \text{otherwise.} \end{cases} \tag{13.3}$$

edge (u,v) itself rather than storing all of f in one place. Also note that, because we often need to consider both $f(u,v)$ and $f(v,u)$ at the same time, it is important that we equip each edge $(u,v) \in E$ with a pointer to its reversal (v,u). This way, we may pass from an edge (u,v) to its reversal (v,u) without performing a costly search to find (v,u) in memory.

We defer the proof of correctness to §13.2. We do show, though, that the Ford–Fulkerson algorithm halts if the edge capacities are integers.

Proposition 13.2. *If the edge capacities of G are integers, then the Ford–Fulkerson algorithm terminates in time $O\left(E \cdot |f^*|\right)$, where $|f^*|$ is the magnitude of any maximum flow for G.*

Proof. Each time we choose an augmenting path p, the right-hand side of (13.2) is a positive integer. Therefore, each time we augment f, the value of $|f|$ increases by at least 1. Since $|f|$ cannot ever exceed $|f^*|$, it follows that lines 5–13 are repeated at most $|f^*|$ times. Each iteration of lines 5–13 takes $O(E)$ time if we use a breadth-first or depth-first search in line 5, so the total running time of FORD–FULKERSON is $O\left(E \cdot |f^*|\right)$. □

Exercise 13.2. *Show that, if the edge capacities of G are rational numbers, then the Ford–Fulkerson algorithm eventually terminates. What sort of bound can you give on its running time?*

Proposition 13.3. *Let G be a flow network. If all edges in G have integer capacities, then there exists a maximum flow in G in which the throughput of each edge is an integer. One such flow is given by running the Ford–Fulkerson algorithm on G.*

Proof. Run the Ford–Fulkerson algorithm on G. The residual capacity of each augmenting path p in line 5 is an integer (technically, induction is required to prove this), so the throughput of each edge is only ever incremented by an integer. The conclusion follows if we assume that the Ford–Fulkerson algorithm is correct. The algorithm is in fact correct, by Corollary 13.8 below. □

Flows in which the throughput of each edge is an integer occur frequently enough to deserve a name. We'll call them **integer flows**.

Perhaps surprisingly, Exercise 13.2 is not true when the edge capacities of G are allowed to be arbitrary real numbers. This is not such bad news, however: it simply says that there *exists* a sufficiently foolish way of choosing augmenting paths so that FORD–FULKERSON never terminates. If we use a reasonably good heuristic (such as the shortest-path heuristic used in the Edmonds–Karp algorithm of §13.1.3), termination is guaranteed, and the running time needn't depend on $|f^*|$.

13.1.3 The Edmonds–Karp Algorithm

The **Edmonds–Karp algorithm** is an implementation of the Ford–Fulkerson algorithm in which the the augmenting path p is chosen to have *minimal length* among all possible augmenting paths (where each edge is assigned length 1, regardless of its capacity). Thus the Edmonds–Karp algorithm can be implemented by using a breadth-first search in line 5 of the pseudocode for FORD–FULKERSON.

Proposition 13.4 (CLRS Theorem 26.8)**.** *In the Edmonds–Karp algorithm, the total number of augmentations is $O(VE)$. Thus total running time is $O\left(VE^2\right)$.*

Proof sketch.

- First one can show that the lengths of the paths p found by breadth-first search in line 5 of FORD–FULKERSON are monotonically nondecreasing (this is Lemma 26.7 of CLRS).

- Next, one can show that each edge $e \in E$ can only be the bottleneck for p at most $O(V)$ times. (By "bottleneck," we mean that e is the (or, an) edge of smallest capacity in p, so that $c_f(p) = c_f(e)$.)
- Finally, because only $O(E)$ pairs of vertices can ever be edges in G_f and because each edge can only be the bottleneck $O(V)$ times, it follows that the number of augmenting paths p used in the Edmonds–Karp algorithm is at most $O(VE)$.
- Again, since each iteration of lines 5–13 of FORD–FULKERSON (including the breadth-first search) takes time $O(E)$, the total running time for the Edmonds–Karp algorithm is $O\left(VE^2\right)$.

□

The shortest-path heuristic of the Edmonds–Karp algorithm is just one possibility. Another interesting heuristic is *relabel-to-front*, which gives a running time of $O\left(V^3\right)$. We won't expect you to know the details of relabel-to-front for 6.046, but you might find it interesting to research other heuristics on your own.

13.2 The Max Flow–Min Cut Equivalence

Definition. A **cut** $(S,\ T = V \setminus S)$ of a flow network G is just like a cut (S,T) of the graph G in the sense of §3.3, except that we require $s \in S$ and $t \in T$. Thus, any path from s to t must cross the cut (S,T). Given a flow f in G, the **net flow** $f(S,T)$ across the cut (S,T) is defined as

$$f(S,T) = \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u). \tag{13.4}$$

One way to picture this is to think of the cut (S,T) as an oriented dam in which we count water flowing from S to T as positive and water flowing from T to S as negative. The **capacity** of the cut (S,T) is defined as

$$c(S,T) = \sum_{u\in S}\sum_{v\in T} c(u,v). \tag{13.5}$$

The motivation for this definition is that $c(S,T)$ should represent the maximum amount of water that could ever possibly flow across the cut (S,T). This is explained further in Proposition 13.6.

Lemma 13.5 (CLRS Lemma 26.4)**.** *Given a flow f and a cut (S,T), we have*

$$f(S,T) = |f|.$$

We omit the proof, which can be found in CLRS. Intuitively, this lemma is an easy consequence of flow conservation. The water leaving s cannot build up at any of the vertices in S, so it must cross over the cut (S,T) and eventually pour out into t.

Proposition 13.6. *Given a flow f and a cut (S,T), we have*

$$f(S,T) \le c(S,T).$$

Thus, applying Lemma 13.5, we find that for any flow f and any cut (S,T), we have

$$|f| \le c(S,T).$$

Proof. In (13.4), $f(v,u)$ is always nonnegative. Moreover, we always have $f(u,v) \le c(u,v)$ by the capacity constraint. The conclusion follows. □

Proposition 13.6 tells us that the magnitude of a maximum flow is at most equal to the capacity of a minimum cut (i.e., a cut with minimum capacity). In fact, this bound is tight:

Theorem 13.7 (Max Flow–Min Cut Equivalence)**.** *Given a flow network G and a flow f, the following are equivalent:*

(i) f is a maximum flow in G.

(ii) The residual network G_f contains no augmenting paths.

(iii) $|f| = c(S,T)$ for some cut (S,T) of G.

If one (and therefore all) of the above conditions hold, then (S,T) is a minimum cut.

Proof. Obviously (i) implies (ii), since an augmenting path in G_f would give us a way to increase the magnitude of f. Also, (iii) implies (i) because no flow can have magnitude greater than $c(S,T)$ by Proposition 13.6.

Finally, suppose (ii). Let S be the set of vertices u such that there exists a path $s \rightsquigarrow u$ in G_f. Since there are no augmenting paths, S does not contain t. Thus (S,T) is a cut of G, where $T = V \setminus S$. Moreover, for any $u \in S$ and any $v \in T$, the residual capacity $c_f(u,v)$ must be zero (otherwise the path $s \rightsquigarrow u$ in G_f could be extended to a path $s \rightsquigarrow u \to v$ in G_f). Thus, glancing back at (13.1), we find that whenever $u \in S$ and $v \in T$, we have

$$f(u,v) = \begin{cases} c(u,v) & \text{if } (u,v) \in E \\ 0 & \text{if } (v,u) \in E \\ 0 \text{ (but who cares)} & \text{otherwise.} \end{cases}$$

Thus we have

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v) - \sum_{u \in S} \sum_{v \in T} 0 = c(S,T) - 0 = c(S,T);$$

so (iii) holds. Because the magnitude of any flow is at most the capacity of any cut, f must be a maximum flow and (S,T) must be a minimum cut. □

Corollary 13.8. *The Ford–Fulkerson algorithm is correct.*

Proof. When FORD–FULKERSON terminates, there are no augmenting paths in the residual network G_f. □

13.3 Generalizations

The definition of a flow network that we laid out may seem insufficient for handling the types of flow problems that come up in practice. For example, we may want to find the maximum flow in a directed graph which sometimes contains both an edge (u,v) and its reversal (v,u). Or, we might want to find the maximum flow in a directed graph with multiple sources and sinks. It turns out that both of these generalizations can easily be reduced to the original problem by performing clever graph transformations.

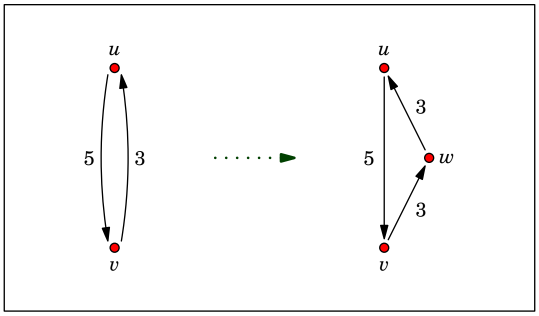


Figure 13.3. We can resolve the issue of E containing both an edge and its reversal by creating a new vertex and rerouting one of the old edges through that vertex.

13.3.1 Allowing both an edge and its reversal

Suppose we have a directed weighted graph $G = (V, E)$ with distinguished vertices s and t. We would like to use the Ford–Fulkerson algorithm to solve the flow problem on G, but G might not be a flow network, as E might contain both (u, v) and (v, u) for some pair of vertices u, v. The trick is to construct a new graph $G' = (V', E')$ from G in the following way: Start with $(V', E') = (V, E)$. For every unordered pair of vertices $\{u, v\} \subseteq V$ such that both (u, v) and (v, u) are in E, add a dummy vertex w to V'. In E', replace the edge (u, v) with edges (u, w) and (w, v), each with capacity $c(u, v)$ (see Figure 13.3). It is easy to see that solving the flow problem on G' is equivalent to solving the flow problem on G. But G' is a flow network, so we can use FORD–FULKERSON to solve the flow problem on G'.

13.3.2 Allowing multiple sources and sinks

Next suppose we have a directed graph $G = (V, E)$ in which there are multiple sources $s_1, \ldots, s_k$ and multiple sinks $t_1, \ldots, t_\ell$. Again, it makes sense to talk about the flow problem in G, but the Ford–Fulkerson algorithm does not immediately give us a way to solve the flow problem in G. The trick this time is to add new vertices s and t to V. Then, join s to each of $s_1, \ldots, s_k$ with a directed edge of capacity ∞,[5] and join each of $t_1, \ldots, t_\ell$ to t with a directed edge of capacity ∞ (see Figure 13.4). Again, it is easy to see that solving the flow problem in this new graph is equivalent to solving the flow problem in G.

13.3.3 Multi-commodity flow

Even more generally, we might want to transport multiple types of commodities through our network simultaneously. For example, perhaps G is a road map of New Orleans and the commodities are emergency relief supplies (food, clothing, flashlights, gasoline...) in the wake of Hurricane Katrina. In the **multi-commodity flow** problem, there are commodities $1, \ldots, k$, sources $s_1, \ldots, s_k$, sinks $t_1, \ldots, t_k$, and quotas (i.e., positive numbers) $d_1, \ldots, d_k$. Each source s_i needs to send d_i units of commodity i to sink t_i. (See Figure 13.5.) The problem is to determine whether there is a way to do so while still obeying flow conservation and the capacity constraint, and if so, what that way is.

[5] The symbol ∞ plays the role of a *sentinel value* representing infinity (such that $\infty > x$ for every real number x). Depending on your programming language (and on whether you cast the edge capacities as integers or floating-point numbers), the symbol ∞ may or may not be supported natively. If it is not supported natively, you will have to either implement it or add extra code to the implementation of FORD–FULKERSON so that operations on edge capacities support

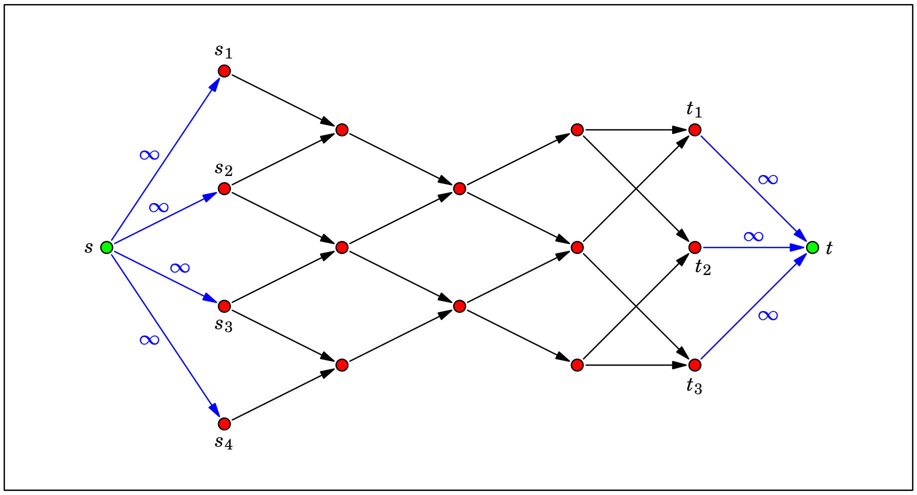


Figure 13.4. We can allow multiple sources and sinks by adding a "master" source that connects to each other source via an edge of capacity ∞, and similarly for the sinks.

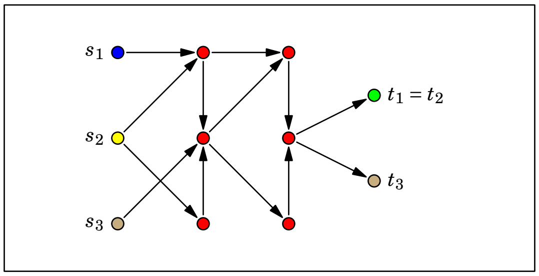


Figure 13.5. An instance of the multi-commodity flow problem with three commodities.

It turns out that the multi-commodity flow problem is NP-complete, even when $k = 2$ and all the edge capacities are 1. Thus, most computer scientists currently believe that there is no way to solve the multi-commodity flow problem in polynomial time, though it has not been definitively proved yet.

both numbers and the symbol ∞.

Lecture 14

Interlude: Problem Solving

Supplemental reading in CLRS: None

This lecture was originally given as a pep talk before the take-home exam. In note form, this chapter will be light reading, a break in which we look back at the course material as veterans.

14.1 What to Bring to the Table

In most technical undergraduate courses, questions are posed in such a way that all solutions are equally valid; if a student is able to come up with any solution to a given problem, she is considered to be in good shape. In 6.046, this is not the case. For most algorithmic problems, there is an obvious but inefficient solution; your task is to find a *better* solution. It is worth appreciating that this way of thinking may be new to you, and taking a moment to reflect metacognitively on strategies to help you solve algorithmic problems.

There is no recipe for solving problems, but there are many things you can do which often, eventually, lead to good solutions.

- *Knowledge.* In academic research, one will usually pick up the relevant background material over the course of working on a given problem. In an exam setting, you should come to the table with as complete an understanding of the background material as possible; you will not have time to learn many new concepts. In either case, you should strive to understand the relevant background as deeply as possible. You should be able to implement a given algorithm correctly in your computer language of choice. You should be able to answer CLRS-style exercises about the pseudocode: What would be the effect on performance if we changed this line? Would the algorithm still be correct? What would be the effect on performance if we used a different data structure?

- *Belief.* Be optimistic. Remind yourself that you might solve this problem, even if it doesn't look tractable at first.

- *Motivation.* There is no substitute for a strong motivation to solve the problem at hand. Given that you're trying to learn algorithms from a set of lecture notes, I probably don't even need to tell you this.

14.2 How to Attack a Problem

We'll now discuss several tips that might help you to crack a problem open. We'll keep the following concrete example in the back of our mind:

Problem 14.1 (Bipartite Matching)**.** In a group of n heterosexual people, each woman has a list of the men she is interested in marrying, and each man has a list of the women he is interested in marrying. Find an algorithm that creates the largest possible number of weddings. (Each person can marry at most one other person. In a couple, each member must be on the other member's list.)

14.2.1 Understand the Problem

- Note everything that is given.
- Find some upper bounds and lower bounds. For example:
    - If the algorithm has to read the entire input, then the running time is at least $\Omega(n)$. (This is not always the case, however! For example, binary search runs in $O(\lg n)$ and does not read the entire input.)
    - If the output has to output k things, then the running time is at least $\Omega(k)$.
    - If the algorithm can be used to perform a comparison-based sort, then the running time is at least $\Omega(n \lg n)$.[1]
    - There are usually a finite number of possibilities for the output. In the case of our current problem, arrange the men in a line (by alphabetical order of last name, say). For each man who has a wife, place his wife behind him. Place the unmarried women at the front of the line (by alphabetical order of last name, say). In this way, each possible matching gives a different line. Thus, the total number of possible matchings is at most $n!$, the number of ways to arrange all n people in a line.
    - Usually there is an obvious algorithm that runs in exponential time. Sometimes it runs faster.
    - Sometimes it is easy to reduce the problem to another problem whose solution is well-known. (Be careful, though—your special case may be amenable to a more efficient solution than the general problem!)
    - Whatever your solution is, its running time must of course fall within the lower and upper bounds you find.
- Is there a useful diagram?

[1] A *comparison-based sorting algorithm* is a sorting algorithm which makes no assumptions about the items being sorted except that there is a well-defined notion of "less than" and that, for objects a and b, we can check whether $a < b$ in constant time. The fact we are using here is that any comparison-based sorting algorithm takes at least $\Omega(n \lg n)$ time. The proof is as follows. Suppose we have a comparison-based sorting algorithm A which takes as input a list $L = \langle a_1, \dots, a_n \rangle$ and outputs a list of indices $I = \langle i_1, \dots, i_n \rangle$ such that $a_{i_1} \le \cdots \le a_{i_n}$. Let k be the number of comparison operations performed by A on a worst-case input of length n (where k depends on n). Because each comparison is a yes-or-no question, there are at most 2^k possible sets of answers to the at most k comparison operations. Since the output of a comparison-based sorting algorithm must depend only on these answers, there are at most 2^k possible outputs I. Of course, if A is correct, then all $n!$ permutations of the indices $\langle 1, \dots, n \rangle$ must be possible, so $2^k \ge n!$. Thus $k \ge \lg(n!)$. Finally, Stirling's approximation tells us that $\lg(n!) = \Theta(n \lg n)$. Thus the running time of A is $\Omega(k) = \Omega(n \lg n)$.

- Is there a graph?
- Try examples!

14.2.2 Try to Solve the Problem

- Can we solve the problem under a set of simplifying assumptions? (What if every vertex has degree 2? What if the edge weights are all 1?)
- Are there useful subproblems?
- Will a greedy algorithm work? Divide-and-conquer? Dynamic programming?
- Can this problem be reduced to a known problem?
- Is it similar to a known problem? Can we modify the known algorithm to suit it to this version?
- Can the problem be solved with convolution? If so, the fast Fourier transform will improve efficiency.
- Is there any further way to take advantage of the given information?
- Can randomization help? It is often better to find a fast algorithm with a small probability of error than a slower, correct algorithm.

Remember that *a hard problem usually cannot be solved in one sitting*. Taking breaks and changing perspective help. In the original setting of a pre–take-home exam pep talk, it was important to remind students that “incubation time” is completely necessary and should be part of the exam schedule. Thus, students should start the test early—the away-time when they are not actively trying to solve test questions is crucial.

14.2.3 Work Out the Details

- What is the algorithm? (Write the algorithm down in as much detail as possible—it is easy to end up with an incorrect algorithm or an incorrect running time analysis because you forgot about a few of what seemed like unimportant implementation details.)
- Why is it correct?
- What is its asymptotic running time?

14.2.4 Communicate Your Findings

- Don’t be shy about writing up partial solutions, solutions to simplified versions of the problem, or observations. This is the most common sort of writing that occurs in academia: often, the complete solution to a problem is just the finishing touch on a large body of prior work by other scientists. And even more often, no complete solution is found—all we have are observations and solutions to certain tractable special cases. (In an exam setting, this sort of writing could win you significant partial credit.)

- To be especially easy on your reader, you can format your algorithm description as an essay, complete with diagrams and examples. The essay should begin with a high-level "executive summary" of how the algorithm works.

Now that you have been thinking about the bipartite matching problem for a while, here is a solution:

Algorithm:
Let M be the set of men and let W be the set of women; let s and t be objects representing a source and a sink. Create a bipartite directed graph* G with vertex set $M \cup W \cup \{s,t\}$. Draw an edge from s to each man and an edge from each woman to t. Next, for each man m and each woman w, draw an edge from m to w if m is on w's list and w is on m's list. Give all edges capacity 1. The graph G is now a flow network with source s and sink t (see Figure 14.1).

Note that there is a bijection between valid matchings and integer flows in G. In one direction, given a matching, we can saturate each edge between a man and a woman who have been matched. The flows out of s and into t are then uniquely determined. In the other direction, given an integer flow f in G, the fact that each edge out of s or into t has capacity 1 means that there is at most one edge out of each man and at most one edge into each woman. Thus, we obtain a valid matching by wedding a man m to a woman w if and only if $f(m,w) = 1$. The magnitude of f is equal to the number of couples, so the number of couples in any matching is bounded above by the maximum flow in G.

This bound is tight if and only if there exists a maximum flow f^* in G which is an integer flow. Such a flow does exist, and is found by the Ford–Fulkerson algorithm, by Proposition 13.3. Since the number of couples is at most $O(n)$, we have $|f^*| = O(n)$, and the running time of the Ford–Fulkerson algorithm is $O(E \cdot |f^*|) = O\left(n^2 \cdot n\right) = O\left(n^3\right)$. Actually, assuming each person has a relatively small list of acceptable spouses, we ought to be more granular about this bound: we have

$$E = O\left(n + \sum_{i=1}^{n} \begin{pmatrix}\text{size of the } i\text{th} \\ \text{person's list}\end{pmatrix}\right),$$

and the running time is at most

$$T = O\left(n \cdot \left(n + \sum_{i=1}^{n} \begin{pmatrix}\text{size of the } i\text{th} \\ \text{person's list}\end{pmatrix}\right)\right).$$

* A graph $G = (V,E)$ is called **bipartite** if the vertex set V can be written as the union of two disjoint sets $V = V_1 \sqcup V_2$ such that every edge in E has one endpoint in V_1 and one endpoint in V_2.

14.2.5 Reflect and Improve

- Can we achieve a better running time?[2]

[2] For the bipartite matching problem, there exist better algorithms than the one given above. Still, I love this solution. It shows how useful flow networks are, even in contexts that seem to have nothing to do with flow. There are many more examples in which this phenomenon occurs.

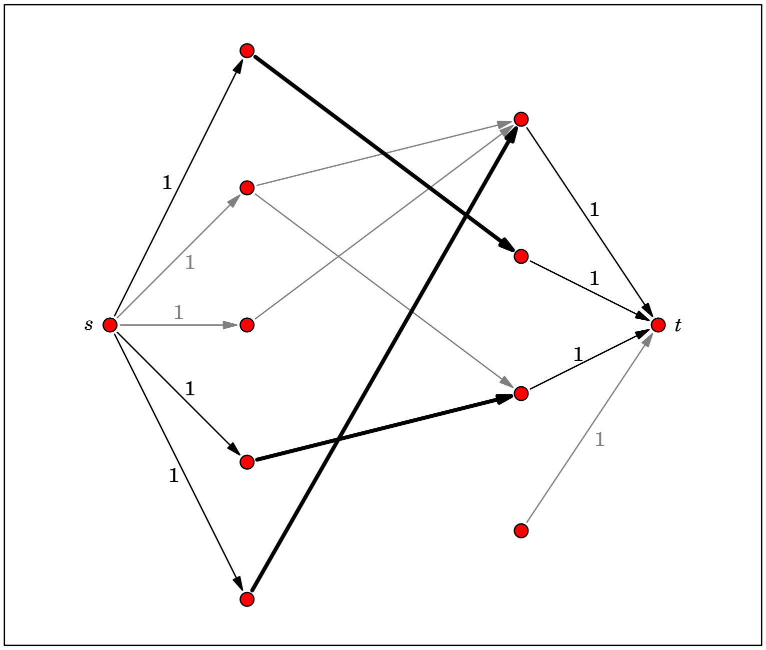


Figure 14.1. In this flow network, the nodes on the left (other than s) represent women and the nodes on the right (other than t) represent men. All edges have capacity 1. A maximum flow is highlighted in black; it shows that the maximal number of couples is 3.

- Can randomization help?
- Can amortization help?
- Can we use less space?

14.3 Recommended Reading

There are many great references and exercise books for algorithmic problem solving and mathematical problem solving in general. Two particular selections are:

- George Pólya, *How to Solve It*
- *The Princeton Companion to Mathematics*, VIII.1: "The Art of Problem Solving."

Another good source of algorithms practice is web sites designed to help people prepare for technical interviews. It is common for tech companies to ask algorithm questions to job applications during interviews; students who plan to apply for software jobs may especially want to practice. Even for those who aren't looking for jobs, web sites like TechInterview.org can be a useful source of challenging algorthmic exercises.

Lecture 15

van Emde Boas Data Structure

Supplemental reading in CLRS: Chapter 20

Given a fixed integer n, what is the best way to represent a subset S of $\{0,\dots,n-1\}$ in memory, assuming that space is not a concern? The simplest way is to use an n-bit array A, setting $A[i]=1$ if and only if $i \in S$, as we saw in Lecture 10. This solution gives $O(1)$ running time for insertions, deletions, and lookups (i.e., testing whether a given number x is in S).[1] What if we want our data structure to support more operations, though? Perhaps we want to be able not just to insert, delete and lookup, but also to find the minimum and maximum elements of S. Or, given some element $x \in S$, we may want to find the *successor* and *predecessor* of x, which are the smallest element of S that is greater than x and the largest element of S that is less than x, respectively. On a bit array, these operations would all take time $\Theta(|S|)$ in the worst case, as we might need to examine all the elements of S. The **van Emde Boas (vEB) data structure** is a clever alternative solution which outperforms a bit array for this purpose:

Operation	Bit array	van Emde Boas
INSERT	$O(1)$	$\Theta(\lg\lg n)$
DELETE	$O(1)$	$\Theta(\lg\lg n)$
LOOKUP	$O(1)$	$\Theta(\lg\lg n)$
MAXIMUM, MINIMUM	$\Theta(n)$	$O(1)$
SUCCESSOR, PREDECESSOR	$\Theta(n)$	$\Theta(\lg\lg n)$

We will not discuss the implementation of SUCCESSOR and PREDECESSOR; those will be left to recitation.

15.1 Analogy: The Two-Coconut Problem

The following riddle nicely illustrates the idea of the van Emde Boas data structure. I am somewhat embarrassed to give the riddle because it shows a complete misunderstanding of materials science and botany, but it is a standard example and I can't think of a better one.

Problem 15.1. There exists some unknown integer k between 1 and 100 (or in general, between 1 and some large integer n) such that, whenever a coconut is dropped from a height of k inches or

[1] This performance really is unbeatable, even for small n. Each operation on the bit array requires only a single memory access.

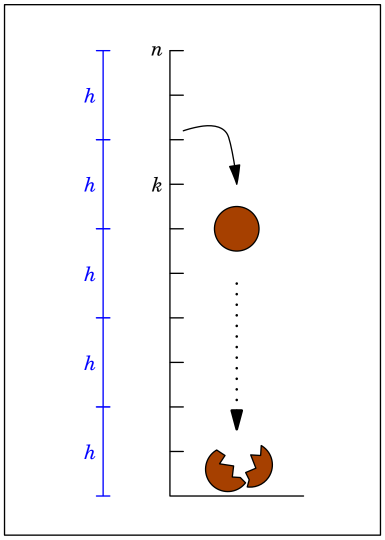


Figure 15.1. One strategy for the two-coconut problem is to divide the integers $\{1,\dots,n\}$ into blocks of size h, and then use the first coconut to figure out which block k is in. Once the first coconut breaks, it takes at most $h-1$ more drops to find the value of k.

more, the coconut will crack, and whenever a coconut is dropped from a height of less than k inches, the coconut will be completely undamaged and it will be as if we had not dropped the coconut at all. (Thus, we could drop the coconut from a height of $k-1$ inches a million times and nothing would happen.) Our goal is to find k with as few drops as possible, given a certain number of test coconuts which cannot be reused after they crack. If we have one coconut, then clearly we must first try 1 inch, then 2 inches, and so on. The riddle asks, what is the best way to proceed if we have two coconuts?

An approximate answer to the riddle is given by the following strategy: Divide the n-inch range into b blocks of height h each (we will choose b and h later; see Figure 15.1). Drop the first coconut from height h, then from height $2h$, and so on, until it cracks. Say the first coconut cracks at height $b_0 h$. Then drop the second coconut from heights $(b_0-1)h+1$, $(b_0-1)h+2$, and so on, until it cracks. This method requires at most $b+h-1=\frac{n}{h}+h-1$ drops, which is minimized when $b=h=\sqrt{n}$.

Notice that, once the first coconut cracks, our problem becomes identical to the one-coconut version (except that instead of looking for a number between 1 and n, we are looking for a number between $(b_0-1)h+1$ and b_0h-1). Similarly, if we started with three coconuts instead of two, then it would be a good idea to divide the range $\{1,\dots,n\}$ into equally-sized blocks and execute the solution to the two-cononut problem once the first coconut cracked.

Exercise 15.1. *If we use the above strategy to solve the three-coconut problem, what size should we choose for the blocks? (Hint: it is not $\sqrt{n}$.)*

15.2 Implementation: A Recursive Data Structure

Just as in the two-coconut problem, the van Emde Boas data structure divides the range $\{0,\dots,n-1\}$ into blocks of size $\sqrt{n}$, which we call *clusters*. Each cluster is itself a vEB structure of size $\sqrt{n}$. In addition, there is a "summary" structure that keeps track of which clusters are nonempty (see Figure

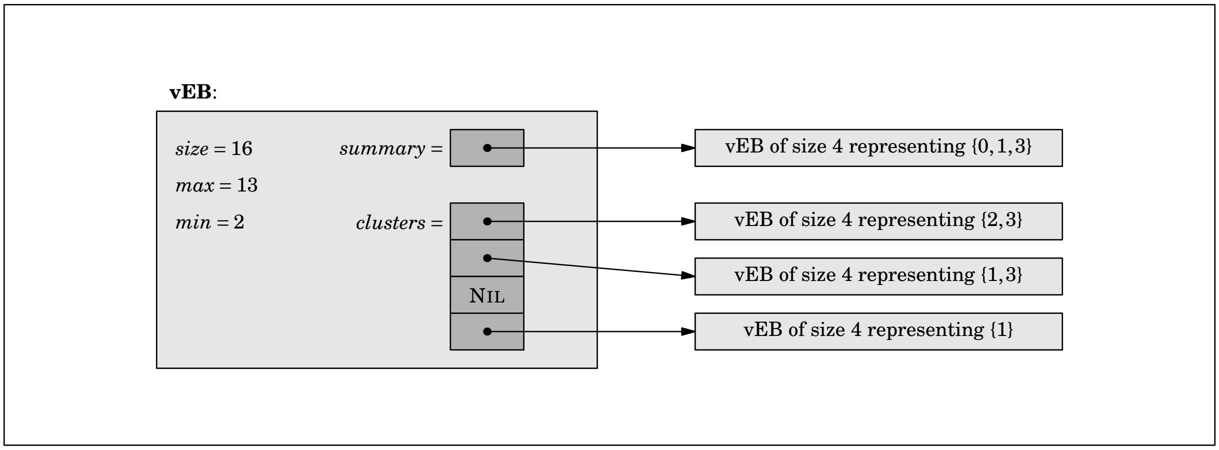

vEB:
size = 16
max = 13
min = 2
summary =
clusters =
NIL
vEB of size 4 representing {0,1,3}
vEB of size 4 representing {2,3}
vEB of size 4 representing {1,3}
vEB of size 4 representing {1}

Figure 15.2. A vEB structure of size 16 representing the set $\{2,3,5,7,13\} \subseteq \{0,\ldots,15\}$.

15.2). The summary structure is analogous to the first coconut, which told us what block k had to lie in.

15.2.1 Crucial implementation detail

The following implementation detail, which may seem unimportant at first, is actually crucial to the performance of the van Emde Boas data structure:

> Do not store the minimum and maximum elements in clusters. Instead, store them as separate data fields.

Thus, the data fields of a vEB structure V of size n are as follows:

- $V.size$ – the size of V, namely n
- $V.max$ – the maximum element of V, or NIL if V is empty
- $V.min$ – the minimum element of V, or NIL if V is empty
- $V.clusters$ – an array of size $\sqrt{n}$ which stores the clusters. For performance reasons, the value stored in each entry of $V.clusters$ will initially be NIL; we will wait to build each cluster until we have to insert something into it.
- $V.summary$ – a van Emde Boas structure of size $\sqrt{n}$ that keeps track of which clusters are nonempty. As with the entries of $V.clusters$, we initially set $V.summary \leftarrow$ NIL; we do not build the recursive van Emde Boas structure referenced by $V.summary$ until we have to insert something into it (i.e., until the first time we create a cluster for V).

15.2.2 Insertions

To simplify the exposition, in this lecture we use a model of computation in which it takes constant time to initialize any array (setting all entries equal to NIL), no matter how big the array is.[2] Thus,

[2] Of course, this is cheating; real computers need an initialization time that depends on the size of the array. Still, there are use cases for which this assumption is warranted. We can preload our vEB structure by creating all possible vEB

it takes constant time to create an empty van Emde Boas structure, and it also takes constant time to insert the first element into a van Emde Boas structure:

```
Algorithm: vEB-First-Insertion($V, x$)
$V.min \leftarrow x$
$V.max \leftarrow x$
```

Using this fact, we will show that the procedure V.Insert(x) has only one non–constant-time step. Say $V.clusters[i]$ is the cluster corresponding to x (for example, if $n = 100$ and $x = 64$, then $i = 6$). Then:

- If $V.clusters[i]$ is Nil, then it takes constant time to create a vEB structure containing only the element corresponding to x. (For example, if $n = 100$ and $x = 64$, then it takes constant time to create a vEB structure of size 10 containing only 4.) We update $V.clusters[i]$ to point to this new vEB structure. Thus, the only non–constant-time operation we have to perform is to update $V.summary$ to reflect the fact that $V.clusters[i]$ is now nonempty.

- If $V.clusters[i]$ is empty[3] (we can check this by checking whether $V.clusters[i].min =$ Nil), then it takes constant time to insert the appropriate entry into $V.clusters[i]$. So again, the only non–constant-time operation we have to perform is to update $V.summary$ to reflect the fact that $V.clusters[i]$ is now nonempty.

- If $V.clusters[i]$ is nonempty, then we have to make the recursive call $V.clusters[i]$.Insert(x). However, we do not need to make any changes to $V.summary$.

In each case, we find that the running time T of Insert satisfies the recurrence

$$T(n) = T\left(\sqrt{n}\right) + O(1). \tag{15.1}$$

As we will see in §15.3 below, the solution to this recurrence is $T_{\text{Insert}} = \Theta(\lg\lg n)$.

15.2.3 Deletions

Similarly, the procedure V.Delete(x) requires only one recursive call. To see this, let i be as above. Then:

- First suppose V has no nonempty clusters (we can check this by checking whether $V.summary$ is either Nil or empty; the latter happens when $V.summary.min =$ Nil).
  - If x is the only element of V, then we simply set $V.min \leftarrow V.max \leftarrow$ Nil. Thus, *deleting the only element of a single-element vEB structure takes constant time*; we will use this fact later.
  - Otherwise, V contains only two elements including x. Let y be the other element of V. If $x = V.min$, then $y = V.max$ and we set $V.min \leftarrow y$. If $x = V.max$, then $y = V.min$ and we set $V.max \leftarrow y$.

substructures up front rather than using Nil for the empty ones. This way, after an initial $O(n)$ preloading time, array initialization never becomes an issue. Another possibility is to use a dynamically resized hash table for $V.clusters$ rather than an array. Each of these possible improvements has its share of extra details that must be addressed in the running time analysis; we have chosen to ignore initialization time for the sake of brevity and readability.

[3] This would happen if $V.clusters[i]$ was once nonempty, but became empty due to deletions.

- Next suppose V has some nonempty clusters.
  - If $x = V.min$, then we will need to update $V.min$. The new minimum takes only constant time to calculate, though: it is y, where

    $$y \leftarrow V.clusters[V.summary.min].min;$$

    in other words, it's the smallest element in the lowest nonempty cluster of V. After making this update, we need to make a recursive call to

    $$V.clusters[V.summary.min].\text{DELETE}(y)$$

    to remove the new value of $V.min$ from its cluster. At this point, there are two possibilities:

    * y is not the only element in its cluster. Then $V.summary$ does not need to be updated.
    * y is the only element in its cluster. Then we must make a recursive call to

      $$V.summary.\text{DELETE}(V.summary.min)$$

      to reflect the fact that y's cluster is now empty. However, since y was the only element in its cluster, it took constant time to remove y from its cluster: as we said above, it takes only constant time to remove the last element from a one-element vEB structure.

    Either way, there is only one non–constant-time step in $V.\text{DELETE}$: a recursive call to DELETE on a vEB structure of size $\sqrt{n}$.
  - By entirely the same argument (perhaps in mirror-image), we find that the case $x = V.max$ has identical running time to the case $x = V.min$.
  - If x is neither $V.min$ nor $V.max$, then we must delete x from its cluster and, if this causes x's cluster to become empty, make a recursive call to $V.summary.\text{DELETE}$ to reflect this update. As above, the recursive call to $V.summary.\text{DELETE}$ will only happen when the deletion of x from its cluster took constant time.

In each case, the only non–constant-time step in DELETE is a single recursive call to DELETE on a vEB structure of size $\sqrt{n}$. Thus, the running time T for DELETE satisfies (15.1), which we repeat here for convenience:

$$T(n) = T\left(\sqrt{n}\right) + O(1). \qquad \text{(copy of 15.1)}$$

Again, the solution is $T_{\text{DELETE}} = \Theta(\lg\lg n)$.

15.2.4 Lookups

Finally, we consider the operation $V.\text{LOOKUP}(x)$, which returns TRUE or FALSE according as x is or is not in V. The implementation is easy: First we check whether $x = V.min$, then whether $x = V.max$. If neither of these is true, then we recursively call LOOKUP on the cluster corresponding to x. Thus, the running time T of LOOKUP satisfies (15.1), and we have $T_{\text{LOOKUP}} = \Theta(\lg\lg n)$.

Exercise 15.2. *Go through this section again, circling each step or claim that relies on the decision not to store $V.min$ and $V.max$ in clusters. Be careful—there may be more things to circle than you think!*

15.3 Solving the Recurrence

As promised, we now solve the recurrence (15.1), which we repeat here for convenience:

$$T(n) = T\left(\sqrt{n}\right) + O(1). \qquad \text{(copy of 15.1)}$$

15.3.1 Base case

Before we begin, it's important to note that we have to lay down a *base case* at which recursive structures stop occurring. Mathematically this is necessary because one often uses induction to prove solutions to recurrences. From an implementation standpoint, the need for a base case is obvious: how could a vEB structure of size 2 make good use of smaller vEB substructures? So we will lay down $n = 2$ as our base case, in which we simply take V to be an array of two bits.

15.3.2 Solving by descent

The equation (15.1) means that we start on a structure of size n, then pass to a structure of size $\sqrt{n} = n^{1/2}$, then to a structure of size $\sqrt{\sqrt{n}} = n^{1/4}$, and so on, spending a constant amount of time at each level of recursion. So the total running time should be proportional to the number of levels of recursion before arriving at our base case, which is the number ℓ such that $n^{1/2^\ell} = 2$. Solving for ℓ, we find

$$\ell = \lg\lg n.$$

Thus $T(n) = \Theta(\lg\lg n)$.

15.3.3 Solving by substitution

Another way to solve the recurrence is to make a substitution which reduces it to a recurrence that we already know how to solve. Let

$$T'(m) = T\left(2^m\right).$$

Taking $m = \lg n$, (15.1) can be rewritten as

$$T'(m) = T'(m/2) + O(1),$$

which we know to have solution $T'(m) = \Theta(\lg m)$. Substituting back $n = 2^m$, we get

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

Lecture 16

Disjoint-Set Data Structures

Supplemental reading in CLRS: Chapter 21 (§21.4 is optional)

When implementing Kruskal's algorithm in Lecture 4, we built up a minimum spanning tree T by adding in one edge at a time. Along the way, we needed to keep track of the connected components of T; this was achieved using a disjoint-set data structure. In this lecture we explore disjoint-set data structures in more detail.

Recall from Lecture 4 that a **disjoint-set data structure** is a data structure representing a dynamic collection of sets $\mathbf{S} = \{S_1, \ldots, S_r\}$. Given an element u, we denote by S_u the set containing u. We will equip each set S_i with a representative element $\text{rep}[S_i]$.[1] This way, checking whether two elements u and v are in the same set amounts to checking whether $\text{rep}[S_u] = \text{rep}[S_v]$. The disjoint-set data structure supports the following operations:

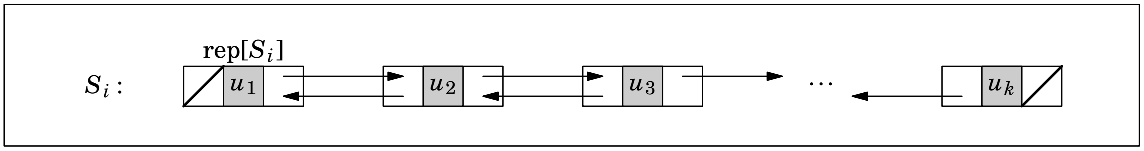
- MAKE-SET(u): Creates a new set containing the single element u.
  - u must not belong to any already existing set
  - of course, u will be the representative element initially
- FIND-SET(u): Returns the representative $\text{rep}[S_u]$.
- UNION(u,v): Replaces S_u and S_v with $S_u \cup S_v$ in $\mathbf{S}$. Updates representative elements as appropriate.

In Lecture 4, we looked at two different implementations of disjoint sets: doubly-linked lists and trees. In this lecture we'll improve each of these two implementations, ultimately obtaining a very efficient tree-based solution.

16.1 Linked-List Implementation

Recall the simple linked-list implementation of disjoint sets that we saw in Lecture 4:

[1] This is not the only way to do things, but it's just as good as any other way. In any case, we'll need to have some symbol representing each set S_i (to serve as the return value of FIND-SET); the choice of what kind of symbol to use is essentially a choice of notation. We have chosen to use a "representative element" of each set rather than create a new symbol.



MAKE-SET(u)	–	initialize as a lone node	$\Theta(1)$
FIND-SET(u)	–	walk left from u until you reach the head of S_u	$\Theta(n)$ worst-case
UNION(u,v)	–	walk right (towards the tail) from u and left (towards the head) from v. Reassign pointers so that the tail of S_u and the head of S_v become neighbors. The representative is updated automatically.	$\Theta(n)$ worst-case

Do we really need to do all that walking? We could save ourselves some time by augmenting each element u with a pointer to rep[S_u], and augmenting rep[S_u] itself with a pointer to the tail. That way, the running time of FIND-SET would be $O(1)$ and all the walking we formerly had to do in UNION would become unnecessary. However, to maintain the new data fields, UNION(u,v) would have to walk through S_v and update each element's "head" field to point to rep[S_u]. Thus, UNION would still take $O(n)$ time in the worst case.

Perhaps amortization can give us a tighter bound on the running time of UNION? At first glance, it doesn't seem to help much. As an example, start with the sets $\{1\},\{2\},\ldots,\{n\}$. Perform UNION($2,1$), followed by UNION($3,1$), and so on, until finally UNION($n,1$), so that $S_1 = \{1,\ldots,n\}$. In each call to UNION, we have to walk to the tail of S_1, which is continually growing. The ith call to UNION has to walk through $i-1$ elements, so that the total running time of the $n-1$ UNION operations is $\sum_{i=1}^{n-1}(i-1) = \Theta\left(n^2\right)$. Thus, the amortized cost per call to UNION is $\Theta(n)$.

However, you may have noticed that we could have performed essentially the same operations more efficiently by instead calling UNION($1,2$) followed by UNION($1,3$), and so on, until finally UNION($1,n$). This way, we never have to perform the costly operation of walking to the tail of S_1; we only ever have to walk to the tails of one-element sets. Thus the running time for this smarter sequence of $n-1$ UNION operations is $\Theta(n)$, and the amortized cost per operation is $O(1)$.

The lesson learned from this analysis is that, when performing a UNION operation, it is best to always merge the smaller set into the larger set, i.e., the representative element of the combined set should be chosen equal to the representative of the larger constituent—that way, the least possible amount of walking has to occur. To do this efficiently, we ought to augment each S_i with a "size" field, which we'll call $S_i.\mathit{weight}$ (see Figure 16.1).

It turns out that the "smaller into larger" strategy gives a significant improvement in the amortized worst-case running time of the UNION operation. We'll show that the total running time of any sequence of UNION operations on a disjoint-set data structure with n elements (i.e., in which MAKE-SET is called n times) is $O(n \lg n)$. Thus, the running time of m operations, n of which are MAKE-SET operations, is

$$O\left(m + n \lg n\right).$$

To start, focus on a single element u. We'll show that the total amount of time spent updating u's "head" pointer is $O(\lg n)$; thus, the total time spent on all UNION operations is $O(n \lg n)$. When u is added to the structure via a call to MAKE-SET, we have $S_u.\mathit{weight} = 1$. Then, every time S_u merges with another set S_v, one of the following happens:

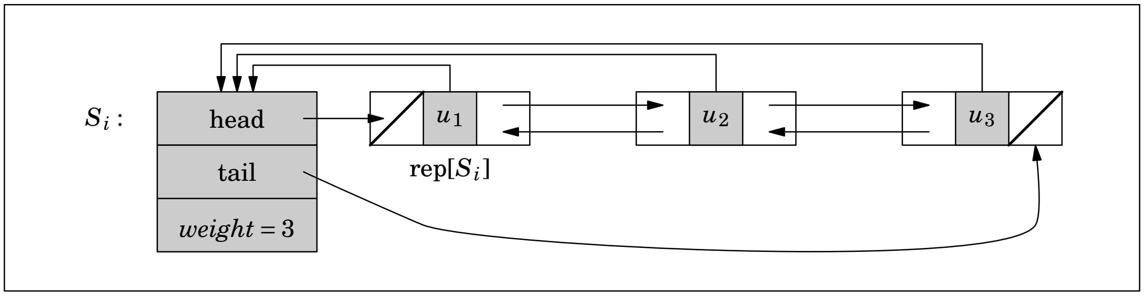


Figure 16.1. A linked list augmented with data fields for the head, the tail, and the size (weight).

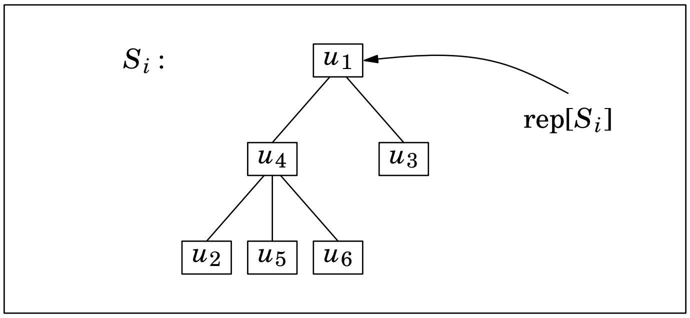
- $S_u.weight > S_v.weight$. Then no update to u's "head" pointer is needed.
- $S_v.weight \geq S_u.weight$. Then, we update u's "head" pointer. Also, in this case, the value of $S_u.weight$ at least doubles.

Because $S_u.weight$ at least doubles every time we update u's "head" pointer, and because $S_u.weight$ can only be at most n, it follows that the total number of times we update u's "head" pointer is at most $\lg n$. Thus, as above, the total cost of all UNION operations is $O(n \lg n)$ and the total cost of any sequence of m operations is $O(m + n \lg n)$.

Exercise 16.1. *With this new augmented structure, do we still need the list to be doubly linked? Which pointers can we safely discard?*

16.2 Forest-of-Trees Implementation

In addition to the linked-list implementation, we also saw in Lecture 4 an implementation of the disjoint-set data structure based on trees:



MAKE-SET(u)	–	initialize new tree with root node u	$\Theta(1)$
FIND-SET(u)	–	walk up tree from u to root	$\Theta(\text{height}) = \Theta(\lg n)$ best-case
UNION(u, v)	–	change rep[S_v]'s parent to rep[S_u]	$O(1) + 2T_{\text{FIND-SET}}$

The efficiency of the basic implementation hinges completely on the height of the tree: the shorter the tree, the more efficient the operations. As the implementation currently stands, the trees could

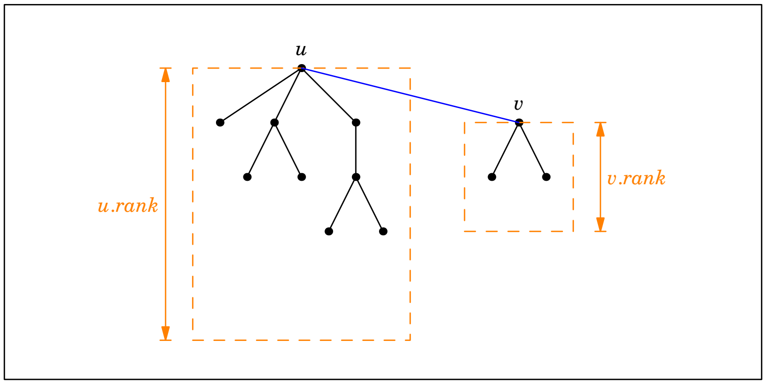


Figure 16.2. Union by rank attempts to always merge the shorter tree into the taller tree, using rank as an estimate (always an overestimate) of height.

be unbalanced and FIND-SET could take as long as $\Theta(n)$ in the worst case. However, this behavior can be dramatically improved, as we will see below.

16.2.1 Union by rank

When we call UNION(u,v), rep$[S_v]$ becomes a child of rep$[S_u]$. Merging S_v into S_u in this way results in a tree of height

$$\max\{height[S_u],\;\; height[S_v]+1\} \tag{16.1}$$

(why?). Thus, the way to keep our trees short is to always merge the shorter tree into the taller tree. (This is analogous to the "smaller into larger" strategy in the linked-list implementation.) To help us do this, we will introduce a new data field called *rank*. If u is the root of a tree (i.e., if $u = \text{rep}[S_u]$), then $u.rank$ will be an *upper bound* on the height of S_u.[2] In light of (16.1), the pseudocode for UNION will be as follows (see Figure 16.2):

```
Algorithm: UNION($\tilde{u}$,$\tilde{v}$)
$u \leftarrow$ FIND-SET($\tilde{u}$)
$v \leftarrow$ FIND-SET($\tilde{v}$)
if $u.rank = v.rank$ then
    $u.rank \leftarrow u.rank + 1$
    $v.parent \leftarrow u$
else if $u.rank > v.rank$ then
    $v.parent \leftarrow u$
else
    $u.parent \leftarrow v$
```

The following lemma shows that UNION preserves the fact that rank is an upper bound on height.

Lemma 16.1. *Suppose initially the following hold:*

- $S_u \neq S_v$

[2] If union by rank is the only improvement we use, then $u.rank$ will actually be the exact height of S_u. But in general, we wish to allow other improvements (such as path compression) to decrease the height of S_u without having to worry about updating ranks. In such cases, the upper bound provided by $u.rank$ may not be tight.

- *S_u has height h_1 and S_v has height h_2*
- $\text{rep}[S_u].\mathit{rank} = r_1$ *and* $\text{rep}[S_v].\mathit{rank} = r_2$
- *$h_1 \le r_1$ and $h_2 \le r_2$.*

Suppose we then call UNION(u,v)*, producing a new set $S = S_u \cup S_v$. Let h be the height of S and let $r = \text{rep}[S].\mathit{rank}$. Then $h \le r$.*

Proof. First, suppose $r_1 > r_2$. Then S_v has been merged into S_u and $r = r_1$. By (16.1), we have

$$\begin{aligned} h &= \max\{h_1,\ h_2+1\} \\ &\le \max\{r_1,\ r_2+1\} \\ &= r_1 \\ &= r. \end{aligned}$$

A similar argument shows that $h \le r$ in the case that $r_2 > r_1$. Finally, suppose $r_1 = r_2$. Then S_v has been merged into S_u and $r = r_1 + 1 = r_2 + 1$, so

$$\begin{aligned} h &= \max\{h_1,\ h_2+1\} \\ &\le \max\{r_1,\ r_2+1\} \\ &= r_2+1 \\ &= r. \end{aligned}$$

□

It turns out that the rank of a tree with k elements is always at most $\lg k$. Thus, the worst-case performance of a disjoint-set forest with union by rank having n elements is

MAKE-SET	$O(1)$
FIND-SET	$\Theta(\lg n)$
UNION	$\Theta(\lg n)$.

Exercise 16.2. *Amortization does not help this analysis. Given sufficiently large n and given m which is sufficiently large compared to n, produce a sequence of m operations, n of which are* MAKE-SET *operations (so the structure ultimately contains n elements), whose running time is* $\Theta(m \lg n)$*.*

Exercise 16.3. *Above we claimed that the rank of any tree with k elements is at most* $\lg k$*. Use induction to prove this claim. (You may assume that* UNION *is the only procedure that modifies ranks. However, you should not assume anything about the height of a tree except that it is less than the rank.) What is the base case?*

16.2.2 Path compression

The easiest kind of tree to walk up is a *flat* tree, where all non-root nodes are direct children of the root (see Figure 16.3). The idea of path compression is that, every time we invoke FIND-SET and walk up the tree, we should reassign parent pointers to make each node we pass a direct child of the root (see Figure 16.4). This locally flattens the tree. With path compression, the pseudocode for FIND-SET is as follows:

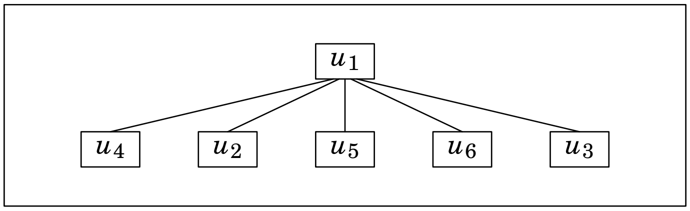


Figure 16.3. In a flat tree, each FIND-SET operation requires us to traverse only one edge.

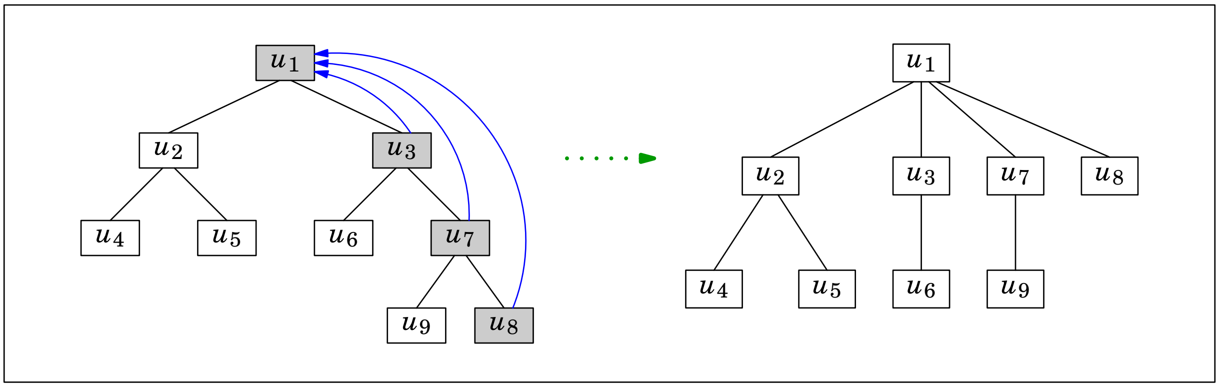


Figure 16.4. With path compression, calling FIND-SET(u_8) will have the side-effect of making u_8 and all of its ancestors direct children of the root.

Algorithm: FIND-SET(u)

```
$A \leftarrow \emptyset$
$n \leftarrow u$
while $n$ is not the root do
    $A \leftarrow A \cup \{n\}$
    $n \leftarrow n.parent$
for each $x \in A$ do
    $x.parent \leftarrow n$
return $n$
```

What data structure should we use for A? In an ideal world, where n can truly be arbitrarily large, we would probably want A to be a dynamically doubled array of the kind discussed in Lecture 10. In real life, however, some assumptions can be made. For example, if you have less than a petabyte (1024 TB, or 2^{53} bits) of available memory, then the rank (and therefore the height) of any tree is at most $\lg\left(2^{53}\right) = 53$, and it would be slightly more efficient to maintain A as a static array of size 53 (with an end-marking sentinel value, perhaps).

It can be shown that, with path compression (but not union by rank), the running time of any sequence of n MAKE-SET operations, f FIND-SET operations, and up to $n-1$ UNION operations is

$$\Theta\left(n + f\left(1 + \log_{2+f/n} n\right)\right).$$

16.2.3 Both improvements together

The punch-line of this lecture is that, taken together, union by rank and path compression produce a spectacularly efficient implementation of the disjoint-set data structure.

Theorem 16.2. *On a disjoint-set forest with union by rank and path compression, any sequence of m operations, n of which are* MAKE-SET *operations, has worst-case running time*

$$\Theta\big(m\alpha(n)\big),$$

where α is the inverse Ackermann function. Thus, the amortized worst-case running time of each operation is $\Theta(\alpha(n))$. If one makes the approximation $\alpha(n) = O(1)$, which is valid for literally all conceivable purposes, then the operations on a disjoint-set forest have $O(1)$ amortized running time.

The proof of this theorem is in §21.4 of CLRS. You can read it if you like; it is not essential. You might also be interested to know that, in a 1989 paper, Fredman and Saks proved that $\Theta(\alpha(n))$ is the fastest possible amortized running time per operation for any implementation of the disjoint-set data structure.

The *inverse Ackermann function* α is defined by

$$\alpha(n) = \min\left\{k : A_k(1) \geq n\right\},$$

where $(k, j) \mapsto A_k(j)$ is the **Ackermann function**. Because the Ackermann function is an extremely rapidly growing function, the inverse Ackermann function α is an extremely slowly growing function (though it is true that $\lim_{n\to\infty} \alpha(n) = \infty$).

The Ackermann function A (at least, one version of it) is defined by

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) \quad \text{(that is, } A_{k-1} \text{ iterated } j+1 \text{ times)} & \text{for } k \geq 1. \end{cases}$$

Some sample values of the Ackermann function are

$$\begin{aligned} A_1(1) &= 3 & \qquad A_1(j) &= 2j+1 \\ A_2(1) &= 7 & \qquad A_2(j) &= 2^{j+1}(j+1) \\ A_3(1) &= 2047 \\ A_4(1) &\gg 10^{80}. \end{aligned}$$

By current estimates, 10^{80} is roughly the same order of magnitude as the number of particles in the observable universe. Thus, even if you are a theoretical computer scientist or mathematician, you will still most likely never end up considering a number n so large that $\alpha(n) > 4$.

Exercise 16.4. *Write down a sequence of operations on the disjoint-set forest with union by rank and path compression (including* MAKE-SET *operations) which cause a taller tree to be merged into a shorter tree. Why do we allow this to happen?*

Lecture 17

Complexity and NP-completeness

Supplemental reading in CLRS: Chapter 34

As an engineer or computer scientist, it is important not only to be able to solve problems, but also to know which problems one can expect to solve efficiently. In this lecture we will explore the *complexity* of various problems, which is a measure of how efficiently they can be solved.

17.1 Examples

To begin, we'll review three problems that we already know how to solve efficiently. For each of the three problems, we will propose a variation which might not be so easy to solve.

- **Flow.** Given a flow network G with integer capacities, we can efficiently find an integer flow that is optimal using the Edmonds–Karp algorithm.
- *Multi-Commodity Flow.* In §13.3, we considered a varation of network flow in which there are k commodities which need to simultaneously flow through our network, where we need to send at least d_i units of commodity i from source s_i to sink t_i.
- **Minimum Cut.** Given an undirected weighted graph G, we can efficiently find a cut $(S, V \setminus S)$ of minimum weight.

  Exercise 17.1. *Design an efficient algorithm to solve the minimum cut problem. (Hint in this footnote.*[1]*)*
- *Maximum Cut.* What if we want to find a cut of maximum weight?
- **Minimum Spanning Tree.** Given an undirected weighted graph $G = (V, E)$, we know how to efficiently find a spanning tree of minimum weight. A spanning tree of G is a subgraph $G' = (V', E') \subseteq G$ such that
  - G' is connected and contains no cycles
  - $V' = V$.

[1] Hint: Turn G into a flow network and find a maximum flow. In Theorem 13.7, we saw how to turn a maximum flow into a minimum cut.

- *Steiner Tree.* What if, instead of requiring $V' = V$, we require $V' = S$ for some given subset $S \subseteq V$?

Each of these three variations is *NP-hard*. The typical attitude towards NP-hard problems is

> Don't expect an efficient algorithm for this problem.

However, I prefer the following more optimistic interpretation:

> If you find an efficient algorithm for this problem, you will get \$1 million.[2]

Unlike the pessimistic interpretation, the above statement is 100% factual. The so-called **P vs. NP** problem is one of seven important open research questions for which Clay Mathematics Institute is offering a \$1 million prize.[3]

17.2 Complexity

So far in the course, we have been ignoring the low-level details of the mathematical framework underlying our analyses. We have relied on the intuitive notion that our ideal computer is "like a real-world computer, but with infinite memory"; we have not worried about explicitly defining what a "step" is. The fact of the matter is that there are many reasonable *models of computation* which make these notions explicit. Historically the first one was the **Turing machine**, invented by Alan Turing, considered by many to be the founding father of computer science. The model we have been using throughout the course is similar to a Turing machine; the main difference is that a "step" on our ideal computer closely resembles a processor cycle on a modern computer, whereas the "steps" of a Turing machine involve sequentially reading and writing to the so-called *tape* which represents its memory. For this lecture, you won't have to go to the trouble of working with a particular model of computation in full detail; but it is worth noting that such details are important in theoretical computer science and should not be regarded as a triviality.

In what follows, we will define the complexity classes P and NP. Before doing so, we will need a couple more definitions:

Definition. A **decision problem** is a computation problem to which the answer is either "yes" or "no." In mathematical language, we can think of a decision problem as a function whose domain is the set of possible input strings[4] and whose range is $\{0,1\}$ (with 0 meaning "no" and 1 meaning "yes").

Definition. A **complexity class** is simply a set of decision problems.

Most of the problems we have considered so far in the course are not decision problems but rather **search problems**—they ask not just whether a solution exists, but also what the solution is. Given a search problem, we can derive decision problems which ask yes-or-no questions about the solution; for example, we might ask:

[2] I should warn you though, most computer scientists believe that it is not possible to find one. (In other words, most computer scientists believe that $P \neq NP$.)

[3] One of the questions has already been solved, so currently there are six prizes remaining.

[4] Strings over what alphabet? A typical choice is $\{0,1\}$ (i.e., binary); another possible choice is the ASCII alphabet. The main reason the choice of alphabet matters is that it determines what "an input of size n" is. The number 255 has size 8 in binary, size 1 in ASCII, and size 255 in unary. An algorithm whose running time is linear with respect to a unary input would be exponential with respect to a binary input.

Problem 17.1. Given a graph G and an integer k, is there a spanning tree of size less than k?

For most real-world applications, search problems are much more important than decision problems. So why do we restrict our attention to decision problems when defining complexity classes? Here are a few reasons:

- The answer to a decision problem is simple.
- The answer to a decision problem is unique. (A search problem might have multiple correct answers, e.g., a given graph might have multiple minimum spanning trees.)
- A decision problem which asks about the answer to a search problem is at most as difficult as the search problem itself. For example, if we can find a minimum spanning tree efficiently, then we can certainly also solve Problem 17.1 efficiently.

17.2.1 P and NP

The existence of many different models of computation is part of the reason for the following definition:

Definition. "**Efficient**" means "polynomial-time." An algorithm is **polynomial-time** if there exists a constant r such that the running time on an input of size n is $O(n^r)$. The set of all decision problems which have polynomial-time solutions is called **P**.

Polynomial time is the shortest class of running times that is invariant across the vast majority of reasonable, mainstream models of computation. To see that shorter running times need not be invariant, consider the following program:

```
Read the first bit of memory
Read the $n$th bit of memory
```

In our model of computation, which has random access to memory, this would take constant time. However, in a model of computation with only serial access to memory (such as a Turing machine), this would take linear time. It is true, though, that any polynomial-time program in our model is also polynomial-time on a Turing machine, and vice versa.

Search problems have the property that, once a solution is found, it can be verified quickly. This verifiability is the motivation for the complexity class NP.

Definition. A decision problem P is in **NP** if there exists a polynomial-time algorithm $A(x, y)$ such that, for every input x to the problem P,

$$P(x) = 1 \quad \Longleftrightarrow \quad \text{there exists some } y \text{ such that } A(x, y) = 1.$$

The string y is called a **witness** or **certificate**; the algorithm A is called a **verifier** or a **nondeterministic algorithm**.[5]

For example, in Problem 17.1, the witness y could be the spanning tree itself—we can certainly verify in polynomial time that a given object y is a spanning tree of size less than k.

[5] The abbreviation NP stands for "nondeterministic polynomial-time." The reason for this name is as follows. Imagine receiving x (the input to the problem P) but leaving the choice of y unspecified. The result is a set of possible running times of A, one for each choice of y. The problem P is in NP if and only if at least one of these possible running times is bounded by a polynomial $p(|x|)$ in the size of x. (The choice of y can depend on x, but p cannot depend on x.)

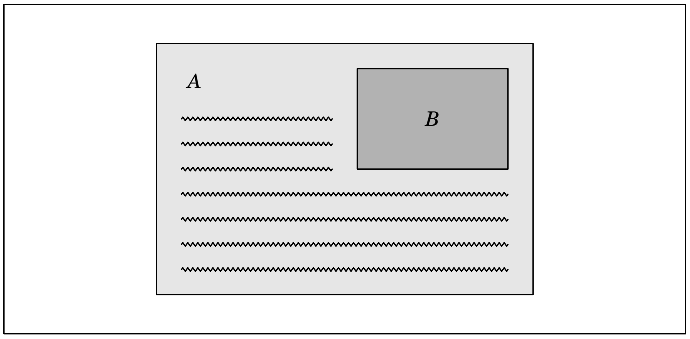


Figure 17.1. A Cook reduction of A to B is a program that would run in polynomial time on an *oracle machine*—that is, a Turing machine equipped with an oracle for B. If some day we find an efficient algorithm for B, then we can create an efficient algorithm for A by replacing the oracle with a subroutine.

Proposition 17.2. P ⊆ NP.

Proof. Given a decision problem P, view P as a function whose domain is the set of strings and whose range is $\{0,1\}$. If P can be computed in polynomial time, then we can just take $A(x,y) = P(x)$. In this case, the verifier just re-solves the entire problem. □

The converse to the above proposition is a famous open problem:

Problem 17.3 (P vs. NP)**.** Is it true that P = NP?

The vast majority of computer scientists believe that P ≠ NP, and so the P vs. NP problem is sometimes called the P ≠ NP problem. If it were true that P = NP, then lots of problems that seem hard would actually be easy: one such example is the algorithm search problem described in §17.3.

17.2.2 Polynomial-time reductions

It is possible to know that one problem is "at least as hard" as another without knowing exactly how hard each problem is. If problem A can be **polynomial-time reduced** to problem B, then it stands to reason B is at least as hard as A. There are two different notions of polynomial-time reduction, which we now lay out.

Definition. We say that the decision problem A is **Karp-reducible** to the decision problem B if there exists a polynomial-time computable function f such that, for every input x,

$$A(x) = 1 \quad \Longleftrightarrow \quad B(f(x)) = 1.$$

Definition. Problem A is **Cook-reducible** to problem B if there exists an algorithm which, given an oracle[6] for B, solves A in polynomial time. (See Figure 17.1.)

Note that a Karp reduction is also a Cook reduction, but not vice versa. Historically, Karp reductions and Cook reductions correspond to different traditions.

Definition. A problem is **NP-hard** if every problem in NP can be Cook-reduced to it.

[6] An **oracle** for B is a magical black-box which solves any instance of problem B in one step. A Cook reduction is a program which is allowed to do any of the normal things a computer program does, and is also allowed to query the oracle.

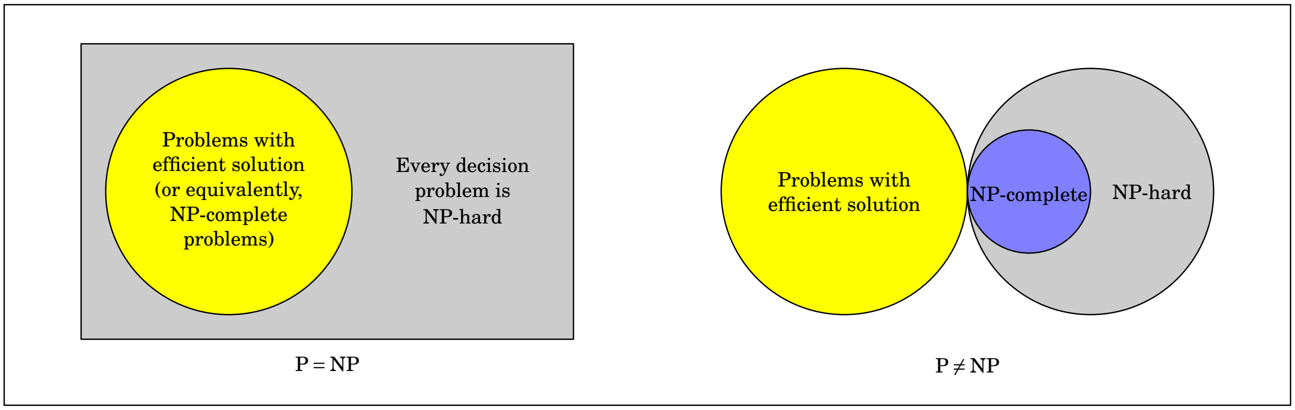

Problems with efficient solution (or equivalently, NP-complete problems)
Every decision problem is NP-hard
P = NP
Problems with efficient solution
NP-complete
NP-hard
P ≠ NP

Figure 17.2. Left: If P = NP, then every decision problem is NP-hard (why?). Right: If P ≠ NP.

Definition. A problem is **NP-complete** if it is both NP-hard and in NP.

Using the notion of NP-completeness, we can make an analogy between NP-hardness and big-O notation:

$O(f(n))$ *on the order of at most* $f(n)$	$\rightsquigarrow$	in NP *at most as hard as an* NP*-complete problem*
$\Theta(f(n))$ *tightly on the order of* $f(n)$	$\rightsquigarrow$	NP-complete *exactly as hard as any other* NP*-complete problem*
$\Omega(f(n))$ *on the order of at least* $f(n)$	$\rightsquigarrow$	NP-hard *at least as hard as an* NP*-complete problem*

Showing that a given problem is in NP is relatively straightforward (or at least, it is clear what the proof should look like): one must give a polynomial-time verifier. By contrast, it is much less clear how one might show that a given problem is NP-hard. One strategy is to reduce another NP-hard problem to it. But this strategy only works if one already knows certain problems to be NP-hard; it could not have been used as the first ever proof that a problem was NP-hard. That first proof was accomplished by Cook in 1971:

Theorem 17.4 (Cook's Theorem)**.** 3SAT *is* NP*-complete.*

Problem 17.5 (3SAT)**.** Given a set of atomic statements $x_1, \ldots, x_n$, a *literal* is either an atom x_i or its negation $\neg x_i$. A *clause* is the disjunction ("or") of a finite set of literals. The 3SAT problem asks, given a propositional formula $\varphi(x_1, \ldots, x_n)$ which is the "and" of finitely many clauses of length 3, does there exist an assignment of either TRUE or FALSE to each x_i which makes $\varphi(x_1, \ldots, x_n)$ evaluate to TRUE?

For example, one instance of 3SAT asks whether there exists an assignment of $x_1, \ldots, x_4$ which makes the proposition

$$\underbrace{(x_1 \vee x_2 \vee \neg x_3)}_{\text{clause}} \wedge \underbrace{(\neg x_1 \vee x_3 \vee x_4)}_{\text{clause}} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee \neg x_3)}_{\text{clause}} \wedge \underbrace{(x_1 \vee x_3 \vee \neg x_4)}_{\text{clause}}.$$

evaluate to TRUE. The answer to this instance happens to be "yes," as shown by the assignment

$$x_2 \mapsto \text{TRUE}, \quad x_1, x_3, x_4 \mapsto \text{FALSE}.$$

17.3 Example: Algorithm Search

In this section, we solve an algorithmic problem whose output is itself an algorithm:

Problem 17.6. Given an algorithmic problem P and a function $T(n)$, find an algorithm which runs in time at most $T(n)$, if such an algorithm exists. Output not just a description of the algorithm, but also a proof of correctness and running time analysis.

Proposition 17.7. *There exists a "meta-algorithm" which solves Problem 17.6 (but runs forever if no algorithm exists). If* $\mathrm{P} = \mathrm{NP}$*, then the running time of this meta-algorithm is polynomial in the size of the shortest possible output.*

The formats of both the input and the output of this algorithm deserve some explanation. By "an algorithmic problem," we mean a mathematical description of the relationship between the input and the output. For example, we could express the notion of "a flow with magnitude at least k" in symbols as

$$\left\{\begin{array}{cc} \forall u,v \in V & 0 \le f(u,v) \le c(u,v) \\ \forall u \in V \setminus \{s,t\} & \sum_{v\in V} f(u,v) = \sum_{v\in V} f(v,u) \\ & \sum_{v\in V} f(s,v) - \sum_{v\in V} f(v,s) \ge k \end{array}\right\}.$$

In the case of a decision problem, this would be more along the lines of

$$\begin{cases} 1 & \text{if there exists } f \text{ satisfying the above} \\ 0 & \text{otherwise.} \end{cases}$$

Regarding the output of our meta-algorithm, we need a format for "a description of the algorithm." One possibility is a text file containing the source code of an implementation in a particular programming language. Next, we need a format for "a proof of correctness and running time analysis." For this we appeal to a *machine-checkable proof language* of the sort used by theorem-checking software. One such software suite is Coq.[7] If you've never seen Coq before, I suggest you check it out!

The key to the proof of Proposition 17.7 is to consider the following problem:

Problem 17.8. Given an algorithmic problem P, a function $T(n)$ and an integer k, does there exist a solution to Problem 17.6 in which the output has length at most k?

Problem 17.8 is in NP, so if $\mathrm{P} = \mathrm{NP}$ then there exists a polynomial-time algorithm for it.

Proof of Proposition 17.7. Consider the following search problem:

Problem 17.9. Given an algorithmic problem P, a function $T(n)$, an integer k, and a prefix string s, does there exist a solution to Problem 17.8 in which the output starts with s?

[7] http://coq.inria.fr/

Problem 17.9 is in NP, so if P = NP then there exists a polynomial-time algorithm for it. Thus, if we let $|P|$ denote the length of the description of P, $|T|$ the length of the definition of the function T, and $|s|$ the length of s, then there exist constants a,b,c,d such that Problem 17.9 has a solution $A(P,T,k,s)$ which runs in time

$$O\left(|P|^a\,|T|^b\,k^c\,|s|^d\right).$$

We can use this algorithm to solve Problem 17.8 by probing longer and longer prefixes s. For example, supposing we use the 26-letter alphabet `A`, …, `Z` for inputs, we would proceed as follows:

- Run $A(P,T,k,\texttt{A})$. If the answer is 0, then run $A(P,T,k,\texttt{B})$. Proceed in this way until you find a prefix which returns 1. If all 26 letters return 0, then the answer to Problem 17.8 is "no."
- Otherwise, let's say for the sake of a concrete example that $A(P,T,k,\texttt{F}) = 1$. Then, run

  $$A(P,T,k,\texttt{FA}), \quad A(P,T,k,\texttt{FB}), \quad \text{etc.},$$

  until you find a two-letter prefix that returns 1.
- Proceeding in this way, the prefix s will eventually become the answer to Problem 17.8.

The above procedure solves Problem 17.8. Because the length of s ranges from 1 to at most k, the running time is

$$\begin{aligned}
&O\left(26\,|P|^a\,|T|^b\,k^c 1^d + 26\,|P|^a\,|T|^b\,k^c 2^d + \cdots + 26\,|P|^a\,|T|^b\,k^c k^d\right)\\
=&O\left(|P|^a\,|T|^b\,k^c\left(1^d + 2^d + \cdots + k^d\right)\right)\\
=&O\left(|P|^a\,|T|^b\,k^c k^{d+1}\right)\\
=&O\left(|P|^a\,|T|^b\,k^{c+d+1}\right).
\end{aligned}$$

Thus, we have done more than just show that Problem 17.8 is in P. We have shown that, for some constants α,β,γ, there exists a solution to Problem 17.8 which runs in time $O\left(|P|^\alpha\,|T|^\beta\,k^\gamma\right)$ and also *returns the full algorithm-and-proofs*, not just 0 or 1.

To conclude the proof, our meta-algorithm is to run the above procedure for $k = 1,2,3,\ldots$. If the shortest possible algorithm-and-proofs has length ℓ, then the running time of the meta-algorithm is

$$O\left(|P|^\alpha\,|T|^\beta\,1^\gamma + |P|^\alpha\,|T|^\beta\,2^\gamma + \cdots + |P|^\alpha\,|T|^\beta\,\ell^\gamma\right) = O\left(|P|^\alpha\,|T|^\beta\,\ell^{\gamma+1}\right). \qquad \square$$

Lecture 18

Polynomial-Time Approximations

Supplemental reading in CLRS: Chapter 35 except §35.4

If you try to design algorithms in the real world, it is inevitable that you will come across NP-hard problems. So what should you do?

1. Maybe the problem you want to solve is actually less general than the NP-hard problem.
   - Perhaps the input satisfies certain properties (bounded degree, planarity or other geometric structure, density/sparsity...)
   - Remember, if you are able to reduce your problem to an NP-hard problem, that doesn't mean your problem is NP-hard—it's the other way around!
2. Maybe you can settle for an approximation.
3. Maybe an exponential algorithm is not so bad.
   - There might be an $O(c^n)$ algorithm with $c \approx 1$.

In this lecture we will focus on item 2: approximation algorithms. As the name suggests, an **approximation algorithm** is supposed to return an answer that is close to the correct answer. There are several types of approximations one might consider, each based on a different notion of closeness.

Definition. Suppose A is an algorithm for the optimization problem Π. (So the answer to Π is a number which we are trying to maximize or minimize, e.g., the weight of a spanning tree.) Given input x, we denote the output of A by $A(x)$, and the optimal answer by $\Pi(x)$. The following are senses in which A can be an "approximation algorithm" to the problem Π:

- The most common type of approximation is multiplicative. For a positive number $\alpha \in \mathbb{R}$ (which may depend on the input), we say that A is a **multiplicative α-approximation** (or simply an **α-approximation**) to Π if, for every input x,

$$\left\{ \begin{array}{ccccccl} \alpha\Pi(x) & \leq & A(x) & \leq & \Pi(x) & & \text{if } \Pi \text{ is a maximization problem} \\ \Pi(x) & \leq & A(x) & \leq & \alpha\Pi(x) & & \text{if } \Pi \text{ is a minimization problem} \end{array} \right\}.$$

  Of course, we must have $0 < \alpha \leq 1$ for a maximization problem and $\alpha \geq 1$ for a minimization problem. If someone talks about a "2-approximation" to a maximization problem, they are probably referring to what here would be called a $\frac{1}{2}$-approximation.

- For a positive number $\beta \in \mathbb{R}$, we say that A is a **additive β-approximation** to Π if, for every input x,
$$\Pi(x) - \beta \le A(x) \le \Pi(x) + \beta.$$
- There are lots of other possibilities.[1] There is no need to worry about memorizing different definitions of approximation. When someone talks about an "approximation," it is their job to define precisely in what way it approximates the answer.

In this lecture we will see approximation algorithms for three NP-hard problems:

Problem	**Approximation factor**
Vertex Cover	2
Set Cover	$\ln n + 1$ (where n is the total number of elements)
Partition	$1 + \epsilon$ for any given parameter $\epsilon > 0$

Note that the $(1+\epsilon)$-approximation algorithm to Partition is a so-called **polynomial-time approximation scheme (PTAS)**. Given a parameter $\epsilon > 0$, the PTAS generates a polynomial-time algorithm which approximates the Partition problem to a factor of $1 + \epsilon$. Naturally, the polynomials will get larger as we decrease ϵ, since we are asking for a better approximation.

The approximations to Vertex Cover and Set Cover in this lecture are conjectured to be optimal, in the sense that giving a better approximation would be NP-hard.[2,3] There exist better approximations to Partition, though.[4]

18.1 Vertex Cover

NP-hard problem (Vertex Cover)**.** Given an undirected graph $G = (V, E)$, a *vertex cover* of G is a subset $V' \subseteq V$ such that, for every edge $(u, v) \in E$, we have either $u \in V'$ or $v \in V'$. The Vertex Cover problem asks, given a graph G, what is the smallest possible vertex cover?

Approximation Algorithm:

```
$V' \leftarrow \emptyset$
while $E$ is nonempty do
    Pick any $(u,v) \in E$
    $V' \leftarrow V' \cup \{u,v\}$
    Remove from $E$ all edges touching $u$ or $v$
    Remove $u$ and $v$ from $V$
```

[1] As a final example, we might combine multiplicative and additive approximations to define an "α,β–affine-linear approximation," in which
$$\alpha^{-1}\Pi(x) - \beta \le A(x) \le \alpha\Pi(x) + \beta$$
for all inputs x, where $\alpha \ge 1$.

[2] If the so-called "Unique Games Conjecture" holds, then it is NP-hard to α-approximate Vertex Cover for any constant $\alpha < 2$. It was recently proved that it is NP-hard to $(c \ln n)$-approximate Set Cover for any constant $c < 1$.

[3] Proofs of such optimality typically use the *probabilistically checkable proofs (PCP) theorem*. The PCP theorem states that a problem is in NP if and only if solutions to the problem can be verified (with high probability) by a certain kind of randomized algorithm.

[4] There exists a PTAS for Partition in which the running time depends only polynomially on $1/\epsilon$; this sort of PTAS is known as a **fully polynomial-time approximation scheme (FPTAS)**. As a consequence, for any given integer r there exists a polynomial-time $\left(1 + 1/n^r\right)$-approximation to Partition. Moreover, there exists a pseudo-polynomial–time algorithm which solves the Partition problem exactly. A **pseudo-polynomial–time** algorithm is one whose running time is polynomial in the size of the input and the numeric value of the output (in this case, the "cost" of the optimal partition).

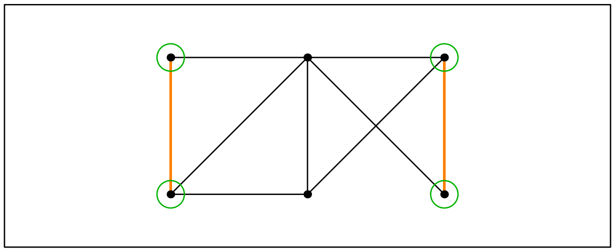


Figure 18.1. Example run of the approximation algorithm for Vertex Cover. The edges chosen by line 3 are highlighted in orange and the chosen vertices are circled. The algorithm returns a cover of size 4, whereas the optimal covering has size 3.

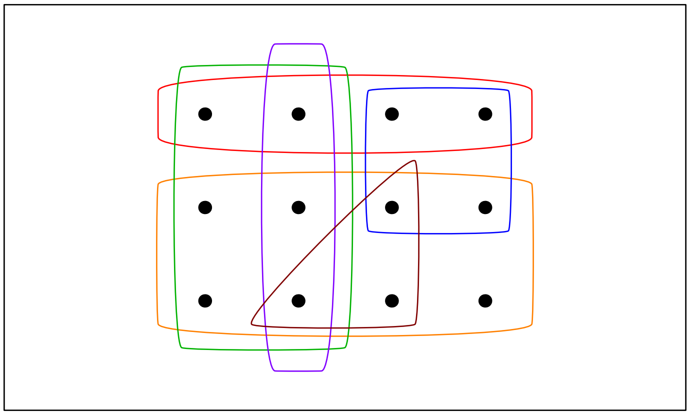


Figure 18.2. The Set Cover problem gives us a covering of a universe set U and asks us to find a small subcovering. How many of the sets in this picture are required to cover the array of dots?

It is clear that this algorithm always returns a vertex cover. Moreover, if G is given in the adjacency-list format, then the algorithm runs in linear time.

Claim. The cover returned by this algorithm is at most twice the size of an optimal cover.

Proof. Let E' denote the set of edges that get chosen by line 3. Notice that no two edges in E' share an endpoint. Thus, every vertex cover must contain at least one endpoint of each edge in E'. Meanwhile, the output of our algorithm consists precisely of the *two* endpoints of each edge in E'. □

18.2 Set Cover

NP-hard problem (Set Cover)**.** Given a set U and subsets $S_1, \ldots, S_m \subseteq U$ such that $\bigcup_{i=1}^m S_i = U$, find indices $I \subseteq \{1, \ldots, m\}$, with $|I|$ minimal, such that

$$\bigcup_{i \in I} S_i = U.$$

Approximation Algorithm:

```
while $U$ is not empty do
    Pick the largest subset $S_i$
    Remove all elements of $S_i$ from $U$ and from the other subsets
return a list of the sets we chose
```

The running time for a good implementation of this algorithm is

$$O\left(\sum_{i=1}^{m} |S_i|\right)$$

(see Exercise 35.3-3 of CLRS).

Claim. The above algorithm gives a $(\ln|U|+1)$-approximation.

Proof. Assume the optimal cover has size k. Let U_i denote the value of U (our universe set) after i iterations of the loop. Clearly, for all i, the set U_i can be covered by k sets. So one of these k sets must contain at least $\frac{|U_i|}{k}$ elements, and therefore the set chosen on line 2 has size at least $\frac{|U_i|}{k}$. Thus, we have

$$|U_{i+1}| \le \left(1-\tfrac{1}{k}\right)|U_i|$$

for all i. This implies that

$$|U_i| \le \left(1-\tfrac{1}{k}\right)^i |U_0|$$

for all i. Since $1-\frac{1}{k} \le e^{-1/k}$, it follows that

$$|U_i| \le e^{-i/k}|U_0|.$$

In particular, letting $n = |U_0|$, we have

$$|U_{k(\ln n+1)}| < 1 \quad \Longrightarrow \quad |U_{k(\ln n+1)}| = 0.$$

Thus, the loop exits after at most $k(\ln n+1)$ iterations. □

18.3 Partition

NP-hard problem (Partition)**.** Given a sorted list of numbers $s_1 \ge s_2 \ge \cdots \ge s_n$, partition the indices $\{1,\ldots,n\}$ into two sets $\{1,\ldots,n\} = A \sqcup B$ such that the "cost"

$$\max\left\{\sum_{i\in A} s_i,\ \sum_{i\in B} s_i\right\}$$

is minimized.

Example. For the partition

$$\boxed{12} \quad \boxed{10} \quad 9 \quad 7 \quad 4 \quad 3 \quad \boxed{2}$$

it is optimal to take $A = \{1,2,7\}$ and $B = \{3,4,5,6\}$, so that

$$\sum_{i\in A} s_i = 24 \quad \text{and} \quad \sum_{i\in B} s_i = 23.$$

Approximation Algorithm:

```
1  $\triangleright$ $\epsilon$ is a parameter: for a given $\epsilon$, the running time is polynomial in $n$ when we view $\epsilon$
   as a constant
2  $m \leftarrow \lfloor 1/\epsilon \rfloor$
3  By brute force, find an optimal partition $\{1,\dots,m\} = A' \sqcup B'$ for $s_1,\dots,s_m$
4  $A \leftarrow A'$
5  $B \leftarrow B'$
6  for $i \leftarrow m+1$ to $n$ do
7      if $\sum_{j\in A} s_j \le \sum_{j\in B} s_j$ then
8          $A \leftarrow A \cup \{i\}$
9      else
10         $B \leftarrow B \cup \{i\}$
11 return $\langle A, B\rangle$
```

Note that this algorithm always returns a partition. The running time of a reasonable implementation is[5]

$$\Theta\left(2^m + n\right) = \Theta(n).$$

Claim. The above algorithm gives a $(1+\epsilon)$-approximation.

Proof. Without loss of generality,[6] assume

$$\sum_{i\in A'} s_i \ge \sum_{i\in B'} s_i.$$

Let

$$H = \frac{1}{2}\sum_{i=1}^{n} s_i.$$

Notice that solving the partition problem amounts to finding a set $A \subseteq \{1,\dots,n\}$ such that $\sum_{i\in A} s_i$ is as close to H as possible. Moreover, since $\sum_{i\in A} s_i + \sum_{i\in B} s_i = 2H$, we have

$$\max\left\{\sum_{i\in A} s_i,\ \sum_{i\in B} s_i\right\} \ = \ H + \frac{D}{2},$$

where

$$D = \left|\sum_{i\in A} s_i - \sum_{i\in B} s_i\right|.$$

- *Case 1:*

  $$\sum_{i\in A'} s_i > H.$$

  In that case, the condition on line 7 is always false, seeing as $\sum_{i\in A'} s_i > \sum_{i\notin A'} s_i$. So $A = A'$ and $B = \{1,\dots,n\} \setminus A'$. In fact, approximation aside, I claim that this must be the *optimal* partition. To see this, first note that $\sum_{i\in B} s_i < H < \sum_{i\in A'} s_i$. If there were a better partition

[5] Of course, the dependence on ϵ is horrible. In a practical context, we would have to choose ϵ small enough to get a useful approximation but big enough that our algorithm finishes in a reasonable amount of time. This would be helped if we replaced line 3 with a more efficient subroutine.

[6] We can have our algorithm check whether this equation holds, and switch the roles of A' and B' if it doesn't.

$\{1,\ldots,n\} = A^* \sqcup B^*$, then taking $A'' = A^* \cap \{1,\ldots,m\}$ and $B'' = B \cap \{1,\ldots,m\}$ would give a partition of $\{1,\ldots,m\}$ in which

$$\max\left\{ \sum_{i\in A''} s_i,\ \sum_{i\in B''} s_i \right\} < \sum_{i\in A'} s_i,$$

contradicting the brute-force solution on line 3.

- *Case 2:*

  $$\sum_{i\in A'} s_i \le H.$$

  Note that, if A ever gets enlarged (i.e., if the condition on line 7 is ever true), then we have $D \le s_i$ (where i is as in line 6). And if A is never enlarged, then it must be the case that $\sum_{i\in B} s_i$ never exceeded $\sum_{i\in A} s_i$ until the very last iteration of lines 6–10, in which s_n is added to B. In that case we have $D \le s_n$ (why?). Either way, we must have

  $$D \le s_{m+1}$$

  and consequently

  $$\max\left\{ \sum_{i\in A} s_i,\ \sum_{i\in B} s_i \right\} \quad = \quad H + \frac{D}{2} \quad \le \quad H + \tfrac{1}{2} s_{m+1}.$$

  Thus,

  $$\begin{aligned} \frac{\text{cost of the algorithm's output}}{\text{optimal cost}} &\le \frac{\text{cost of the algorithm's output}}{H} \\ &\le \frac{H + \frac{1}{2}s_{m+1}}{H} \\ &= 1 + \frac{\frac{1}{2}s_{m+1}}{H} \\ &= 1 + \frac{\frac{1}{2}s_{m+1}}{\frac{1}{2}\sum_{i=1}^{m+1} s_i} \\ &= 1 + \frac{s_{m+1}}{\sum_{i=1}^{m+1} s_i} \\ &\le 1 + \frac{s_{m+1}}{(m+1)s_{m+1}} \\ &= 1 + \frac{1}{m+1} \\ &< 1 + \epsilon. \end{aligned}$$

  □

Lecture 19

Compression and Huffman Coding

Supplemental reading in CLRS: Section 16.3

19.1 Compression

As you probably know at this point in your career, **compression** is a tool used to facilitate storing large data sets. There are two different sorts of goals one might hope to achieve with compression:

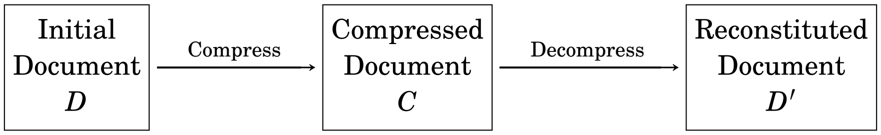
- *Maximize* ease of access, manipulation and processing
- *Minimize* size—especially important when storage or transmission is expensive.

Naturally, these two objectives are often at odds with each other. In this lecture we will focus on the second objective.

In general, data cannot be compressed. For example, we cannot losslessly represent all m-bit strings using $(m-1)$-bit strings, since there are 2^m possible m-bit strings and only 2^{m-1} possible $(m-1)$-bit strings. So when is compression possible?

- If only a relatively small number of the possible m-bit strings appear, compression is possible.
- If the same "long" substring appears repeatedly, we could represent it by a "short" string.
- If we relax the requirement that every string have a unique representation, then compression might work but make "similar" strings identical.

19.1.1 Lossless and lossy



In **lossless** compression, we require that $D = D'$. This means that the original document can always be recovered exactly from the compressed document. Examples include:

- Huffman coding
- Lempel–Ziv (used in `gif` images)

In **lossy** compression, D' is close enough but not necessarily identical to D. Examples include:

- `mp3` (audio)
- `jpg` (images)
- `mpg` (videos)

19.1.2 Adaptive and non-adaptive

Compression algorithms can be either adaptive or non-adaptive.

- *Non-adaptive* – assumes prior knowledge of the data (e.g., character frequncies).
- *Adaptive* – assumes no knowledge of the data, but builds such knowledge.

19.1.3 Framework

For the remainder of this lecture, we consider the following problem:

Input: Known alphabet *(e.g., English letters a,b,c,...)*
Sequence of characters from the known alphabet *(e.g., "helloworld")*

We are looking for a **binary code**—a way to represent each character as a binary string (each such binary string is called a **codeword**).

Output: Concatenated string of codewords representing the given string of characters.

19.1.4 Fixed-length code

In a *fixed-length code*, all codewords have the same length. For example, if our alphabet is

$$\{\text{a, b, c, d, e, f, g, h}\},$$

then we can represent each of the 8 characters as a 3-bit string, such as

$$\left\{\begin{matrix} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} & \text{h} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \end{matrix}\right\}.$$

Such a code is easy to encode and decode:

$$\text{baba} = 001|000|001|000$$

Just as in DNA encoding and decoding, it is important to keep track of register: the deletion or insertion of a single bit into the binary sequence will cause a frame shift, corrupting all later characters in the reconstituted document.

19.1.5 Run-length encoding

Imagine you are given the string of bits

$$\underbrace{000000}_{6}\underbrace{111}_{3}\underbrace{000}_{3}\underbrace{11}_{2}\underbrace{00000}_{5}.$$

Rather than create codewords, we could simply store this string of bits as the sequence $\langle 6,3,3,2,5\rangle$. This strategy is used in fax-machine transmission, and also in `jpeg`.

19.1.6 Variable-length code

If we allow our codewords to have different lengths, then there is an obvious strategy:

> Use shorter codewords for more frequent characters; use longer codewords for rarer characters.

For example, consider a six-letter alphabet with the following character frequencies:

Character	a	b	c	d	e	f
Frequency	45%	13%	12%	16%	9%	5%
Codeword	0	101	100	111	1101	1100
	(1 bit)	(3 bits)			(4 bits)	

Using this code, the average number of bits used to encode 100 characters is

$$(45)1+(13)3+(12)3+(16)3+(9)4+(5)4=224.$$

Compare this to a 3-bit fixed-length code, in which it would take 300 bits to encode 100 characters.

Notice that I didn't use 1, 01, or 10 as codewords, even though they would have made the encoding shorter. The reason for this is that we want our code to be **uniquely readable**: it should never be ambiguous as to how to decode a compressed document.[1] One standard way to make uniquely readable codes is to use **prefix coding**: no codeword occurs as a prefix (initial substring) of another codeword. This allows unambiguous, linear-time decoding:

$$\underbrace{101}_{\text{b}}\,\underbrace{111}_{\text{d}}\,\underbrace{1100}_{\text{f}}\,\underbrace{0}_{\text{a}}\,\underbrace{100}_{\text{c}}\,\underbrace{1101}_{\text{e}}$$

Prefix coding means that we can draw our code as a binary tree, with the leaves representing codewords (see Figure 19.1).

19.2 The Huffman Algorithm

The problem of §19.1.3 amounts to the following. We are given an alphabet $\{a_i\}$ with frequencies $\{f(a_i)\}$. We wish to find a set of binary codewords $C=\{c(a_1),\ldots,c(a_n)\}$ such that the average number of bits used to represent the data is minimized:

$$B(C)=\sum_{i=1}^{n} f(a_i)\left|c(a_i)\right|.$$

Equivalently, if we represent our code as a tree T with leaf nodes $a_1,\ldots,a_n$, then we want to minimize

$$B(T)=\sum_{i=1}^{n} f(a_i)\,d(a_i),$$

where $d(a_i)$ is the depth of a_i, which is also equal to the number of bits in the codeword for a_i.

The following algorithm, due to Huffman, creates an optimal prefix tree for a given set of characters $C=\{a_i\}$. Actually, the Huffman code is optimal among *all* uniquely readable codes, though we don't show it here.

[1] For an example of non-unique readibility, suppose we had assigned to "d" the codeword 01 rather than 111. Then the string 0101 could be decoded as either "ab" or "dd."

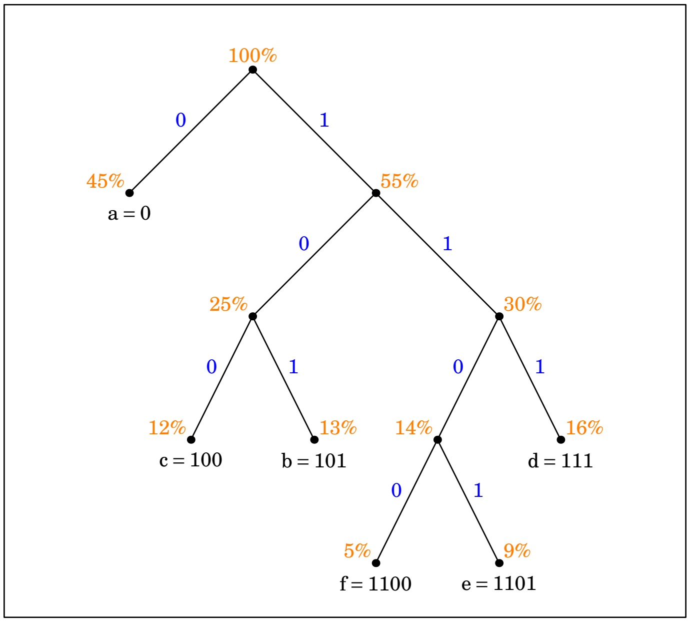


Figure 19.1. Representation of a binary code as a binary tree.

```
Algorithm: HUFFMAN-TREE($C$)
$n \leftarrow |C|$
$Q \leftarrow C$ ▷ a min–priority queue keyed by frequency
for $i \leftarrow 1$ to $n-1$ do
    Allocate new node $z$
    $z.left \leftarrow x \leftarrow$ EXTRACT-MIN($Q$)
    $z.right \leftarrow y \leftarrow$ EXTRACT-MIN($Q$)
    $z.freq \leftarrow x.freq + y.freq$
    $Q$.INSERT($z$)
▷ Return the root of the tree
return EXTRACT-MIN($Q$)
```

19.2.1 Running time

Initially, we build a minqueue Q with n elements. Next, the main loop runs $n-1$ times, with each iteration consisting of two EXTRACT-MIN operations and one INSERT operation. Finally, we call EXTRACT-MIN one last time; at this point Q has only one element left. Thus, the running time for Q a binary minheap would be

$$\underbrace{\Theta(n)}_{\text{BUILD-QUEUE}} + \underbrace{\Theta(n \lg n)}_{\text{Loop}} + \underbrace{O(1)}_{\text{EXTRACT-MIN}} = \Theta(n \lg n).$$

If instead we use a van Emde Boas structure for Q, we achieve the running time

$$\Theta(n \lg \lg n).$$

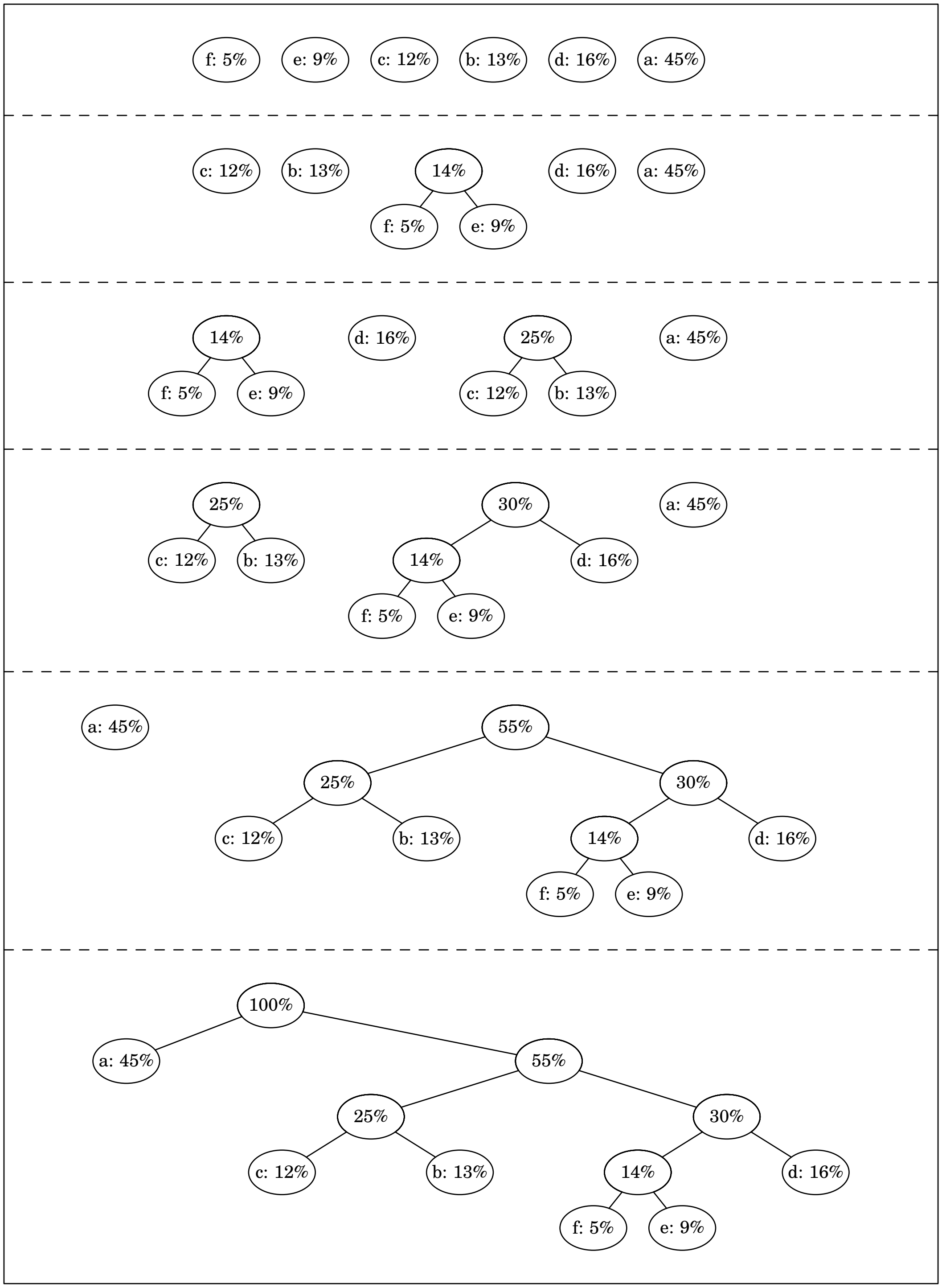


Figure 19.2. Example run of the Huffman algorithm. The six rows represent the state of the graph each of the six times line 3 is executed.

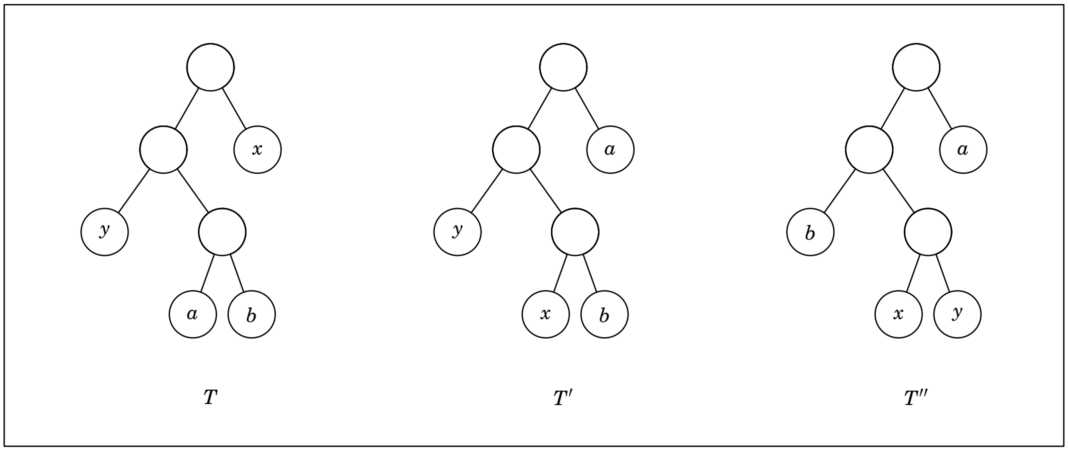


Figure 19.3. Illustration of the proof of Lemma 19.1.

19.2.2 Correctness

The Huffman algorithm is a greedy algorithm: at each stage, we merge together the two nodes of lowest frequency.

Lemma 19.1 (CLRS Lemma 16.2)**.** *Suppose x, y are the two most infrequent characters of C (with ties broken arbitrarily). Then there exists an optimal prefix code for C with codewords for x and y of the same length and differing only in the last bit. (In other words, there exists an optimal tree in which x and y are siblings.)*

Proof. Let T be an optimal tree. Let a be a leaf of maximal depth. If a has no sibling, then deleting a from the tree (and using a's parent to represent the character formerly represented by a) produces a better code, contradicting optimality. So a has a sibling b, and since a has maximum depth, b is a leaf. So a and b are nodes of maximal depth, and without loss of generality we can say $a.\mathit{freq} \le b.\mathit{freq}$.

Also without loss of generality, say $x.\mathit{freq} \le y.\mathit{freq}$. Switch the leaves a and x; call the resulting tree T' (see Figure 19.3). Then $B(T') \le B(T)$, seeing as $x.\mathit{freq} \le a.\mathit{freq}$. So it must be the case that $x.\mathit{freq} = a.\mathit{freq}$ and T' is optimal as well. Similarly, switch the leaves b and y in T'; call the resulting tree T''. It must be the case that $y.\mathit{freq} = b.\mathit{freq}$ and T'' is optimal. In T'', the leaves x and y are siblings. □

Lemma 19.2 (CLRS Lemma 16.3)**.** *Given an alphabet C, let:*

- *x, y be the two most infrequent characters in C (with ties broken arbitrarily)*
- *z be a new symbol, with $z.\mathit{freq} \leftarrow x.\mathit{freq} + y.\mathit{freq}$*
- *$C' = \big(C \setminus \{x, y\}\big) \cup \{z\}$*
- *T' be an optimal tree for C'.*

Then T is an optimal tree for C, where T is a copy of T' in which the leaf for z has been replaced by an internal node having x and y as children.

Proof. Suppose T is not optimal. By Lemma 19.1, let $\mathscr{T}$ be an optimal tree for C in which x and y are siblings. Delete x and y from $\mathscr{T}$, and label their parent (now a leaf) with the symbol z. Call the resulting tree $\mathscr{T}'$. Notice that $\mathscr{T}'$ is a tree for C', and furthermore,

$$\begin{aligned} B(\mathscr{T}') &= B(\mathscr{T}) - x.\mathit{freq} - y.\mathit{freq} \\ &< B(T) - x.\mathit{freq} - y.\mathit{freq} \\ &= B(T'). \end{aligned}$$

This contradicts the optimality of T'. We conclude that T must have been optimal in the first place. □

Corollary 19.3. *The Huffman algorithm is correct.*

Proof sketch. Let T be the tree produced by the Huffman algorithm. Construct a new tree U as follows. Initially, let U consist of just one vertex. Then, perform a series of transformations on U. In each transformation, append two children to what was previously a leaf node. In this way, we can eventually transform U into T. Moreover, Lemma 19.2 guarantees that, at each stage, U is an optimal tree for the alphabet formed by its leaves. □

Lecture 20

Sublinear-Time Algorithms

Supplemental reading in CLRS: None

If we settle for approximations, we can sometimes get much more efficient algorithms. In Lecture 18, we saw polynomial-time approximations to a few NP-hard problems. In what follows, we will concern ourselves with *sublinear*-time (that is, $o(n)$-time) approximation algorithms to problems whose exact solutions require at least linear time. We will see two examples:

1. Estimating the number of connected components of a graph G
2. Estimating the size of a minimum spanning tree of a graph G, given that all of G's edge weights are integers between 1 and $Ϝ$ for some given constant $Ϝ$.[1]

20.1 Estimating the Number of Connected Components

In this section we exhibit a polynomial-time approximation scheme for counting the connected components of a graph. Given a graph G (in adjacency-list format) with n vertices and parameters $\epsilon, \delta > 0$, we exhibit a randomized approximation algorithm which, with probability $1-\delta$, provides an additive ϵn-approximation. That is,

$$\Pr\left[\ \left|\begin{pmatrix}\text{output of}\\ \text{algorithm}\end{pmatrix} - \begin{pmatrix}\text{correct number of}\\ \text{connected components}\end{pmatrix}\right| > \epsilon n\ \right] \quad \le \quad \delta. \tag{20.1}$$

The running time of the algorithm is $\operatorname{Poly}(1/\epsilon,\ \lg(1/\delta))$.[2]

Given a vertex v, let m_v denote the number of vertices in v's connected component. Take a moment to convince yourself of the following lemma:

Lemma 20.1. *The number of connected components in G is*

$$\sum_{v \in V} \frac{1}{m_v}.$$

[1] The archaic Greek letter $Ϝ$ (digamma) stood for the sound /w/. Although digamma fell out of use by the time of Classical Greek, it is still occasionally used in Greek numerals (which are used in Greece in approximately the same way that we use Roman numerals) to represent the number 6.

[2] This notation means that the running time is polynomial in $1/\epsilon$ and $\lg(1/\delta)$; i.e., there exist positive constants r_1, r_2 such that the running time of the algorithm is $O\left(\left(\frac{1}{\epsilon}\right)^{r_1}\left(\lg\frac{1}{\delta}\right)^{r_2}\right)$.

The idea of the algorithm is to approximate m_v by a quantity $\widetilde{m}_v$ which can be computed in constant time. Then, for a set K of k randomly chosen vertices, we have

$$\begin{pmatrix}\text{number of connected}\\ \text{components}\end{pmatrix} = \sum_{v\in V}\frac{1}{m_v} \approx \sum_{v\in V}\frac{1}{\widetilde{m}_v} \approx \frac{n}{k}\sum_{v\in K}\frac{1}{\widetilde{m}_v}.$$

```
Algorithm: APPROX-#CC$(G, \epsilon, \delta)$
1  For some $k = \Theta\left(\frac{1}{\epsilon^2}\lg\frac{1}{\delta}\right)$, pick $k$ vertices $v_1, \ldots, v_k$ at random
2  for $i \leftarrow 1$ to $k$ do
3      Set
           $\widetilde{m}_{v_i} \leftarrow \min\left\{m_{v_i}, \frac{2}{\epsilon}\right\}$
       ▷ computed using breadth-first search
4  return the value of
           $\frac{n}{k}\sum_{i=1}^{k}\frac{1}{\widetilde{m}_{v_i}}$
```

The key line to examine is line 3.

Exercise 20.1.

(i) *The running time of a breadth-first search depends on the size of the graph. So how can there be a bound on the running time of line 3 that doesn't depend on n?*

(ii) *What is the running time of line 3? Show that the total running time of the algorithm is*

$$O\left(\frac{1}{\epsilon^2}k\right) = O\left(\frac{1}{\epsilon^4}\lg\frac{1}{\delta}\right).$$

20.1.1 Correctness

We will prove (20.1) in two steps. First we will prove

$$\left|\sum_{v\in V}\frac{1}{\widetilde{m}_v} - \sum_{v\in V}\frac{1}{m_v}\right| \le \frac{\epsilon n}{2}; \tag{20.2}$$

then we will prove

$$\Pr\left[\ \left|\frac{n}{k}\sum_{i=1}^{k}\frac{1}{\widetilde{m}_{v_i}} - \sum_{v\in V}\frac{1}{\widetilde{m}_v}\right| \ge \frac{\epsilon n}{2}\ \right] \le \delta. \tag{20.3}$$

Combining these two, we obtain

$$\begin{aligned}&\left|\frac{n}{k}\sum_{i=1}^{k}\frac{1}{\widetilde{m}_{v_i}} - \sum_{v\in V}\frac{1}{m_v}\right|\\ \le\ &\left|\frac{n}{k}\sum_{i=1}^{k}\frac{1}{\widetilde{m}_{v_i}} - \sum_{v\in V}\frac{1}{\widetilde{m}_v}\right| + \left|\sum_{v\in V}\frac{1}{\widetilde{m}_v} - \sum_{v\in V}\frac{1}{m_v}\right|,\end{aligned}$$

and with probability $1-\delta$,

$$\cdots \le \frac{\epsilon n}{2} + \frac{\epsilon n}{2} = \epsilon n,$$

which is (20.1).

Proof of (20.2). This follows from the fact that

$$0 \le \frac{1}{\widetilde{m}_v} - \frac{1}{m_v} < \frac{1}{\widetilde{m}_v} \le \frac{1}{\left(\frac{2}{\epsilon}\right)} = \frac{\epsilon}{2}$$

for each $v \in V$. □

Proof of (20.3). We use *Hoeffding's inequality*, a relative of the Chernoff bound which is stated as follows. Given independent real-valued random variables $X_1, \ldots, X_k$, let $Y = \frac{1}{k}\sum_{i=1}^{k} X_i$. Suppose $a, b \in \mathbb{R}$ are constants such that always[3] $a \le X_i \le b$ for each i. Then Hoeffding's inequality states that for any $\eta > 0$, we have

$$\Pr\left[\left|Y - \mathbb{E}[Y]\right| \ge \eta\right] \le 2\exp\left(\frac{-2k\eta^2}{(b-a)^2}\right).$$

We take $X_i = \frac{1}{\widetilde{m}_{v_i}}$, which gives $\mathbb{E}[Y] = \frac{1}{n}\sum_{v \in V} \frac{1}{\widetilde{m}_v}$; we take $a = 0$ and $b = 1$ and $\eta = \epsilon/2$. Then Hoeffding's equality becomes

$$\Pr\left[\ \left|\frac{1}{k}\sum_{i=1}^{k}\frac{1}{\widetilde{m}_{v_i}} - \frac{1}{n}\sum_{v \in V}\frac{1}{\widetilde{m}_v}\right| \ge \frac{\epsilon}{2}\ \right] \le 2\exp\left(-2k\left(\frac{\epsilon^2}{4}\right)\right).$$

Thus, for a suitable $k = \Theta\left(\frac{\lg(1/\delta)}{\epsilon^2}\right)$ (namely, $k = \frac{2\ln(2/\delta)}{\epsilon^2}$), we have

$$\cdots \le 2\exp\left(-\ln\tfrac{2}{\delta}\right) = \delta.$$

This is equivalent to (20.3). □

20.2 Estimating the Size of a Minimum Spanning Tree

In this section, we exhibit an algorithm which solves the following problem:

Input: An undirected weighted graph $G = (V, E, w)$ with n vertices

all edge weights are integers in the set $\{1, \ldots, \Gamma\}$, where $\Gamma \ge 2$ is a given parameter
all vertices have degree at most d
given in adjacency-list format

Parameters $\epsilon, \delta > 0$

Output: A number t such that, with probability at least $1 - \delta$,

$$(1-\epsilon)w^* \le t \le (1+\epsilon)w^*,$$

where w^* is the weight of a minimum spanning tree.

The running time of the algorithm is $\operatorname{Poly}\left(\frac{1}{\epsilon},\ \lg\frac{1}{\delta},\ \Gamma,\ d\right)$.

[3] Or at least, with probability 1.

20.2.1 Motivation

Imagine running Kruskal's MST algorithm on a graph $G = (V, E, w)$ whose edge weights are all integers from the set $\{1, \dots, \digamma\}$. The procedure would look like this:

```
1 $T \leftarrow \emptyset$
2 for $i \leftarrow 1$ to $\digamma$ do
3     while there exists an edge of weight $i$ which has its endpoints in different
      connected components do
4         add the edge to $T$
5         if $|T| = n - 1$ then
6             return $T$
```

The number of connected components in T starts at n and decreases by 1 every time we add an edge. This leads to the following insight, which will be crucial for us:

Observation 20.2. Let $G^{(i)} = (V, E^{(i)})$, where $E^{(i)} = \{e \in E : w(e) \le i\}$, and let $T^{(i)}$ be the restriction of T to $G^{(i)}$. Let $c^{(i)}$ be the number of connected components in $G^{(i)}$. Lines 3–6 (plus induction) guarantee that the number of connected components in $T^{(i)}$ is also $c^{(i)}$. Moreover, the number of connected components in $T^{(i)}$ is equal to

$$n - \left(\text{\# edges in } T^{(i)}\right)$$

(this is true for any forest with n vertices). Thus, we have

$$\begin{pmatrix}\text{\# edges in } T \text{ with} \\ \text{weight at most } i\end{pmatrix} = \left(\text{\# edges in } T^{(i)}\right) = n - c^{(i)}.$$

Observation 20.2 plus some clever algebra lead to the following lemma:

Lemma 20.3. *With $G^{(i)}$, $T^{(i)}$ and $c^{(i)}$ as above, we have*

$$w(T) = n - \digamma + \sum_{i=1}^{\digamma - 1} c^{(i)}.$$

Proof. Let

$$A_i = \begin{pmatrix}\text{\# edges it } T \text{ with} \\ \text{weight exactly } i\end{pmatrix} \qquad \text{and} \qquad B_i = \begin{pmatrix}\text{\# edges in } T \text{ with} \\ \text{weight at least } i\end{pmatrix}.$$

Then

$$\begin{aligned} B_i &= |T| - \begin{pmatrix}\text{\# edges in } T \text{ with} \\ \text{weight at most } i - 1\end{pmatrix} \\ &= (n-1) - \left(n - c^{(i-1)}\right) \\ &= c^{(i-1)} - 1. \end{aligned}$$

Now, the clever algebra trick is to notice that

$$w(T) = \sum_{i=1}^{\digamma} i \cdot A_i = \sum_{i=1}^{\digamma} B_i.$$

(Make sure you see how this works.) Completing the computation,

$$\begin{aligned} w(T) &= \sum_{i=1}^{F} B_i \\ &= \sum_{i=1}^{F} \left(c^{(i-1)} - 1\right) \\ &= -F + \sum_{i=0}^{F-1} c^{(i)} \\ &= n - F + \sum_{i=1}^{F-1} c^{(i)}. \end{aligned}$$ □

```
Algorithm: MST-APPROX($G,\epsilon,\delta,F,d$)
1  for $i \leftarrow 1$ to $F-1$ do
2        ▷ Let $G^{(i)}$ denote the subgraph of $G$ consisting of those edges whose weight is at
         most $i$
3        $\hat{c}^{(i)} \leftarrow$ APPROX-#CC$\left(G^{(i)}, \frac{\epsilon}{2F}, \frac{\delta}{F}\right)$
4  return the value of
                      $|G.V| - F + \sum_{i=1}^{F-1} \hat{c}^{(i)}$
```

The tricky part here is that we cannot compute $G^{(i)}$ and store it in memory, as that would take $\Omega(n + G.E)$ time. So how does line 3 work? We must modify APPROX-#CC to so as to ignore any edges of weight greater than i. However, this modification forces us to reconsider the running time of APPROX-#CC, and is the reason for the dependence on d.

Exercise 20.2.

(i) *Suppose we modify* APPROX-#CC *so as to ignore any edges of weight greater than i. Use an example to show that, if we treat ϵ, δ and F as constants but do not allow any dependence on d, then the breadth-first search on line 3 of* APPROX-#CC *has worst-case running time $\Omega(n)$. (Hint: Every time we ignore an edge, it takes one step.)*

(ii) *Despite part (i), we can put a good bound on the running time of the modified version of* APPROX-#CC *if we allow the bound to depend on d. Show that the modified breadth-first search in line 3 of* APPROX-#CC *(with $\frac{\epsilon}{2F}$ and $\frac{\delta}{F}$ standing in for what in* APPROX-#CC *are denoted ϵ and δ) takes $O\left(\frac{dF}{\epsilon}\right)$ time, and thus that the total running time of* APPROX-MST *is*

$$O\left(\frac{dF^4}{\epsilon^3} \lg \frac{F}{\delta}\right).$$

20.2.2 Correctness

The return value of APPROX-MST is

$$n - F + \sum_{i=1}^{F-1} \hat{c}^{(i)},$$

while in §20.2.1 we showed that

$$w^* = n - \digamma + \sum_{i=1}^{\digamma-1} c^{(i)}.$$

Thus, to show correctness, we need to show that

$$\Pr\left[\left| \sum_{i=1}^{\digamma-1} c^{(i)} - \sum_{i=1}^{\digamma-1} \hat{c}^{(i)} \right| > \epsilon w^* \right] \le \delta.$$

By the union bound, it suffices to show that

$$\Pr\left[\left| c^{(i)} - \hat{c}^{(i)} \right| > \frac{\epsilon}{\digamma - 1} w^* \right] \le \frac{\delta}{\digamma - 1}$$

for each i. Now, because each edge has weight at least 1, we know that $w^* \ge n-1$. So it suffices to show that

$$\Pr\left[\left| c^{(i)} - \hat{c}^{(i)} \right| > \frac{\epsilon}{\digamma - 1}(n-1) \right] \le \frac{\delta}{\digamma - 1}.$$

The correctness of APPROX-#CC guarantees that

$$\Pr\left[\left| c^{(i)} - \hat{c}^{(i)} \right| > \frac{\epsilon}{2\digamma} n \right] \le \frac{\delta}{\digamma},$$

so we are fine as long as

$$\frac{\epsilon}{\digamma - 1}(n-1) \ge \frac{\epsilon}{2\digamma} n,$$

which holds whenever $n > 1$. That is good enough.

Lecture 21

Clustering

Supplemental reading in CLRS: None

Clustering is the process of grouping objects based on similarity as quantified by a metric. Each object should be similar to the other objects in its cluster, and somewhat different from the objects in other clusters.

Clustering is extremely useful; it is of fundamental importance in data analysis. Some applications include

- Scientific data from a wide range of fields
- Medical data, e.g., for patient diagnosis
- Identifying patterns of consumer behavior
- Categorizing music, movies, images, genes, etc.

Clustering is conceptually related to

- *Unsupervised learning* – the notion that objects which produce similar measurements may share some intrinsic property.
- *Dimensionality reduction.*

Depending on the application, we may have to carefully choose which metric to use out of many possible metrics. Or, we may have to apply a transformation to our data before we can get good clustering. But ultimately, the (admittedly vague) problem we are trying to solve is this:

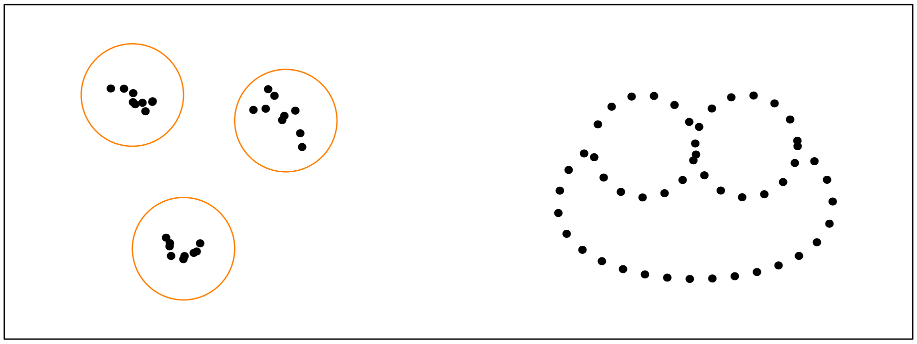


Figure 21.1. Left: Some data sets are relatively straightforward to cluster; most people would cluster these objects in basically the same way. Right: It's not always so easy, though. How would you make two clusters from this data set?

Input: Instance data $D = \{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n\}$
Desired number of clusters, k
Distance metric[1] $d(\vec{x}_i, \vec{x}_j)$

Output: Assignment of instance data D to clusters $\mathscr{C} = \{C_1, \ldots, C_k\}$.

21.1 Hierarchical Agglomerative Clustering

The main idea of hierarchical agglomerative clustering is to build up a graph representing the cluster set as follows:

- Initially, each object (represented as a vertex) is in its own cluster.
- Each time an edge is added, two clusters are merged together.
- Stops when we have k clusters.

The following is an implementation of hierarchical agglomerative clustering known as *single-linkage clustering*.

```
Algorithm: SINGLE-LINKAGE-CLUSTER($D, d, k$)
  1. Let $H$ be an undirected graph with one vertex for each object and no edges.
  2. Sort the set of unordered pairs $\{\{u,v\} : u, v \in D,\ u \neq v\}$ by distance:
         $d(\text{pair } 1) \le d(\text{pair } 2) \le \cdots \le d\left(\text{pair } \binom{n}{2}\right)$.
  3. Loop from $i = 1$ to $\binom{n}{2}$:
       • If the two members of pair $i$ are in different connected components:
           – Merge their clusters.
           – Join pair $i$ with an edge in $H$.
       • Exit the loop and halt when there are only $k$ components left.
```

Although we have couched the description of this algorithm in the language of graph theory, the use of a graph data structure is not essential—all we need is a disjoint-set data structure to store the connected components. The advantage of computing the graph is that the graph gives us more information about the influence each object had on cluster formation.

21.1.1 Running time

In step 1, we initialize the graph and make n calls to MAKE-SET. Next, there are $\binom{n}{2} = \Theta\left(n^2\right)$ unordered pairs of objects; sorting them in step 2 takes $\Theta\left(n^2 \lg n^2\right) = \Theta\left(n^2 \lg n\right)$ time. Finally, each iteration of the loop in step 3 makes two calls to FIND-SET and at most one call to UNION. The loop is iterated at most $O\left(n^2\right)$ times.

Thus, the total number of operations performed on the disjoint-set data structure is $n + O\left(n^2\right) = O\left(n^2\right)$. If we use a good implementation of the disjoint-set data structure (such as a disjoint-set forest

[1] What we call here a "distance metric" corresponds to the notion of a *metric space* in mathematics. That is, our distance metric is a symmetric, positive-definite scalar-valued function of two arguments which satisfies the triangle inequality.

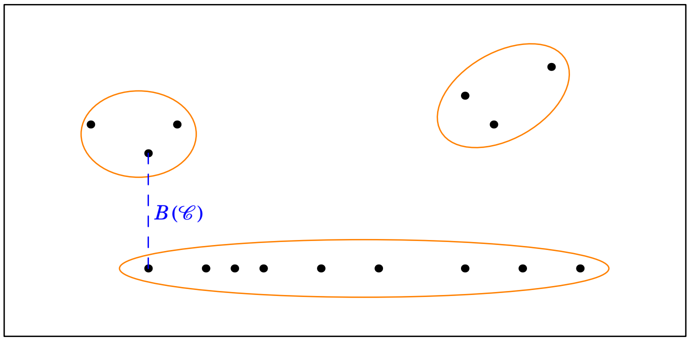


Figure 21.2. Illustration of the quantity $\beta(\mathscr{C})$, which is a measure of the amount of space between clusters.

with union by rank and path compression), these operations will take less time than the $\Theta(n^2 \lg n)$ sorting. Thus, the total running time of SINGLE-LINKAGE-CLUSTER is

$$\Theta(n^2 \lg n).$$

21.1.2 ~~Correctness~~ Discussion

This is where we would usually argue for the correctness of our algorithm. In the current case, there is no precise notion of correctness because we didn't state exactly what properties our clusters should have—just that there are k of them and they represent some sort of nearness-based grouping. So instead, we'll discuss the properties of this clustering and consider other possibilities.

- This algorithm is essentially a special case of Kruskal's MST algorithm. For $k = 1$, running SINGLE-LINKAGE-CLUSTER is exactly the same as running Kruskal's algorithm on the complete graph whose edges are weighted by distance.
- For $k > 1$, the graph produced is an MST with the $k - 1$ heaviest edges removed.
- This algorithm maximizes the "spacing" between clusters. More precisely, the minimum distance between a pair of clusters is maximized, in the sense that the quantity

  $$\beta(\mathscr{C}) = \min\left\{d(\vec{x}, \vec{y}) : \vec{x}, \vec{y} \text{ not in the same cluster}\right\} \tag{21.1}$$

  is maximized (see Figure 21.2), as we show below.

Proposition 21.1. *Let $\mathscr{C} = \{C_1, \ldots, C_k\}$ be the output of* SINGLE-LINKAGE-CLUSTER. *Then the quantity $\beta(\mathscr{C})$, as defined in (21.1), is maximized.*

Proof. Let $d^* = \beta(\mathscr{C})$. Then d^* is the distance between the first pair of vertices *not considered* by the algorithm. All edges in the graph H *were* considered, so their weights are all $\le d^*$.

Consider a different clustering $\mathscr{C}' = \left\{C'_1, \ldots, C'_k\right\}$. There must exist some $C_r \in \mathscr{C}$ which is not a subset of any $C'_s \in \mathscr{C}'$. (Make sure you can see why.[2]) So we can choose some $\vec{x}, \vec{y} \in C_r$ and some s such that $\vec{x} \in C'_s$ and $\vec{y} \notin C'_s$. Because C_r is a connected component of H, there exists a path $\vec{x} \rightsquigarrow \vec{y}$ in

[2] What would it mean if every C_r were a subset of some C'_s? This should not sit comfortably with the fact that $\mathscr{C}' \ne \mathscr{C}$.

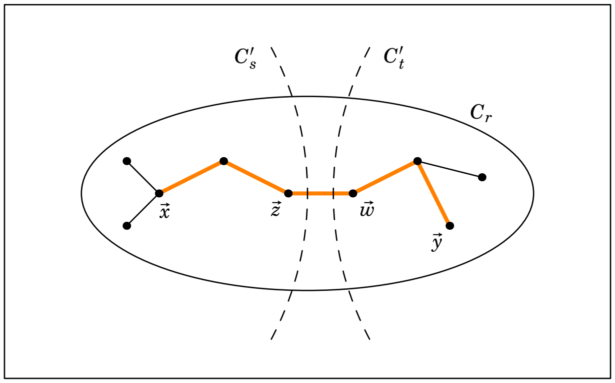


Figure 21.3. Illustration of the proof of Proposition 21.1.

H. Somewhere in that path there must be an edge $(\vec{z},\vec{w})$ which connects a vertex in C'_s to a vertex in some other cluster C'_t (see Figure 21.3). Thus

$$\beta(\mathscr{C}') \le \left(\text{distance between } C'_s \text{ and } C'_t\right) \le d(\vec{z},\vec{w}) \le d^*,$$

since $(\vec{z},\vec{w})$ is an edge in H. □

21.1.3 Other distance criteria

The procedure SINGLE-LINKAGE-CLUSTER is designed to always merge together the two “closest” clusters, as determined by the distance criterion

$$d_{\text{min}}\left(C_i, C_j\right) = \min_{\vec{x}\in C_i, \vec{y}\in C_j} d\left(\vec{x},\vec{y}\right).$$

Other forms of hierarchical agglomerative clustering use different distance criteria, such as[3]

$$\begin{aligned} d_{\text{max}}\left(C_i, C_j\right) &= \max_{\vec{x}\in C_i, \vec{y}\in C_j} d\left(\vec{x},\vec{y}\right) \\ d_{\text{mean}}\left(C_i, C_j\right) &= \frac{1}{|C_i|\cdot|C_j|}\sum_{\vec{x}\in C_i, \vec{y}\in C_j} d\left(\vec{x},\vec{y}\right) \\ d_{\text{centroid}}\left(C_i, C_j\right) &= d\left(\frac{1}{|C_i|}\sum_{\vec{x}\in C_i}\vec{x},\ \ \frac{1}{|C_j|}\sum_{\vec{y}\in C_j}\vec{y}\right). \end{aligned}$$

Note that new algorithms are needed to implement these new criteria. Also, different criteria tend to produce quite different clusters.

In general, hierarchical agglomerative clustering tends to perform worse on higher-dimensional data sets.

[3] The last of these, d_{centroid}, assumes that the points $\vec{x}$ belong to a vector space (hence, can be added together and scaled). If you care about this sort of thing, you probably already noticed my choice of notation $\vec{x}$ which suggests that the ambient space is $\mathbb{R}^m$. This is the most important case, but d_{centroid} makes sense in any normed vector space. (And the rest of this lecture makes sense in an arbitrary metric space.)

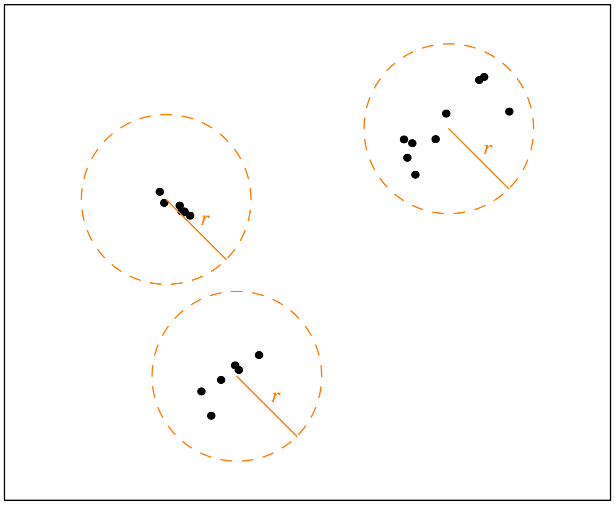


Figure 21.4. Given a positive integer k (in this case $k = 3$) and a desired radius r, we attempt to create k clusters, each having radius at most r.

21.2 Minimum-Radius Clustering

Hierarchical agglomerative clustering focuses on making sure that objects in different clusters are far apart. By contrast, minimum-radius clustering focuses on making sure that objects in the same cluster are close together. Thus, rather than maximizing the minimum distance $\beta(\mathscr{C})$, we will now try to minimize a maximum (actually, maximin) distance. Namely, we will try to assign a "center" $C.center$ to each cluster $C \in \mathscr{C}$ so as to minimize the quantity

$$F(\mathscr{C}) = \max_{\vec{x}\in D} \min_{C\in\mathscr{C}} d(\vec{x}, C.center).$$

In principle, given k, we need only find the best possible set of k centers—then, each vertex will belong to the cluster centered at whichever of these k points is closest to it. Thus, we can view the quantity $F(\mathscr{C})$ which we are trying to minimize as depending only on the choice of centers.

The problem of minimizing $F(\mathscr{C})$ in this way is quite difficult. We will make things easier on ourselves by assuming that we are given a goal radius r to strive for (rather than trying to figure out what the minimum possible radius is) (see Figure 21.4).

Input: Instance data $D = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_n\}$ with a distance metric $d\left(\vec{x}_i, \vec{x}_j\right)$
Desired number of clusters, k
Desired radius, r

Output: Ideally, a set of k points $\Gamma = \{\vec{c}_1, \dots, \vec{c}_k\} \subseteq D$ (the cluster centers) such that each $\vec{x} \in D$ is within distance r of some element of Γ.

21.2.1 First strategy: Approximate k

The following algorithm gives the correct r but approximates k: assuming a clustering with the desired k and r is possible, the number of clusters is at most $k(\ln n + 1)$.

Algorithm:
Imagine each $\vec{x} \in D$ as the center of a cluster of radius r. The set

$$S_i = \{\text{points in } D \text{ within distance } r \text{ of } \vec{x}_i\}$$

consists of all points which should be allowed to belong to this cluster. We now face the following problem: we have a collection of sets $S_1, \ldots, S_n \subseteq D$, and we wish to find a size-k subcollection $S_{i_1}, \ldots, S_{i_k}$ such that

$$\bigcup_{j=1}^{k} S_{i_j} = D.$$

This is the set-cover problem, which is NP-hard. However, we can obtain an approximate solution using the approximation algorithm of §18.2:

- Greedily choose the set with the largest coverage until all points are covered.
- Assuming a set cover exists, this algorithm gives a multiplicative α-approximation, where $\alpha = \ln|D| + 1$.

The running time of this algorithm is

$$O\left(\sum_{i=1}^{n} |S_i|\right),$$

which, depending on the geometric configuration of D, could be anywhere from $O(n)$ to $O\left(n^2\right)$.

21.2.2 Second strategy: Approximate r

Alternatively, we could strictly meet the k requirement but approximate r. Assuming that a clustering with the desired k and r is possible, the following algorithm produces a set of k clusters with radius at most $2r$.

Algorithm:

```
1 $i \leftarrow 0$
2 while $D$ is not empty do
3     $i \leftarrow i + 1$
4     Pick an arbitrary $\vec{x} \in D$
5     Define cluster $C_i$ to be all points within distance $2r$ of $\vec{x}$
6     $C_i.center \leftarrow \vec{x}$
7     $D \leftarrow D \setminus C_i$
8 return $\{C_1, \ldots, C_i\}$
```

The costliest line is line 5, which takes $O(n)$ time. As we show below in Lemma 21.2, line 5 is executed at most k times (assuming that a clustering with the desired k and r is possible). Thus, the running time of this algorithm is $O(kn)$.

Lemma 21.2. *Suppose that a clustering with the desired k and r is possible. Then, after k passes through the loop, D will be empty.*

Proof. Let $\mathscr{C}^* = \{C_1^*, \ldots, C_k^*\}$ be a clustering with the desired k and r. Let $\vec{x}$ be as in line 4, and let $C_j^* \in \mathscr{C}^*$ be such that $\vec{x} \in C_j^*$. Let $\vec{c} = C_j^*.center$. We claim that C_i (as defined on line 5) is a superset of $C_j^* \cap D$. (The intersection with D is necessary because D may have changed during prior iterations of line 7.) To prove this, note that for any $\vec{y} \in C_j^* \cap D$, we have by the triangle inequality

$$d(\vec{x}, \vec{y}) \leq d(\vec{x}, \vec{c}) + d(\vec{c}, \vec{y}) \leq r + r = 2r.$$

Thus $\vec{y} \in C_i$, which means that $C_j^* \cap D \subseteq C_i$.

Thus, each time line 7 is executed, a new element of $\mathscr{C}^*$ is removed from consideration. By the time line 7 has been executed k times, all k of the clusters in $\mathscr{C}^*$ will have disappeared, so that D is empty. □

Lecture 22

Derandomization

Supplemental reading in CLRS: None

Here are three facts about randomness:

- Randomness can speed up algorithms (e.g., sublinear-time approximations)
- Under reasonable assumptions, we know that the speed-up can't be too large. Advances in complexity theory in recent decades have provided evidence for the following conjecture, which is generally believed to be true by complexity theorists:[1]

  Conecture. *Suppose there exists a randomized algorithm A which, on an input of size n, runs in time T and outputs the correct answer with probability at least $2/3$. Then there exists a deterministic algorithm A' which runs in time $\operatorname{Poly}(n,T)$ and always outputs the correct answer.*

- In practice, randomization often doesn't buy any (or much) speed-up. Moreover, the randomized algorithm is often based on novel ideas and techniques which can equally well be used in a deterministic algorithm.

As we said in Lecture 9, a randomized algorithm can be viewed as a sequence of random choices along with a deterministic algorithm to handle the choices. *Derandomization* amounts to deterministically finding a possible sequence of choices which would cause the randomized algorithm to output the correct answer. In this lecture we will explore two basic methods of derandomization:

1. *Conditional expectations.* As we walk down the decision tree (see Figure 22.1), we may be able to use probabilistic calculations to guide our step. In this way, we can make choices that steer us toward a preponderance of correct answers at the bottom level.
2. *Pairwise independence.* Another way to find the correct answer is to simply run the randomized algorithm on *all* possible sequences of random choices and see which answer is reported most often (see Figure 22.2). In general this is not practical, as the number of sequences is exponential in the number of random choices, but sometimes it is sufficient to check only a relatively small collection of sequences (e.g., those given by a universal hash family).

In what follows, we will use the *big-cut problem* to explore each of these two methods of derandomization:

[1] See, for example, Impagliazzo and Wigderson, "P = BPP unless E has sub-exponential circuits: Derandomizing the XOR Lemma" ⟨http://www.math.ias.edu/~avi/PUBLICATIONS/MYPAPERS/IW97/proc.pdf⟩.

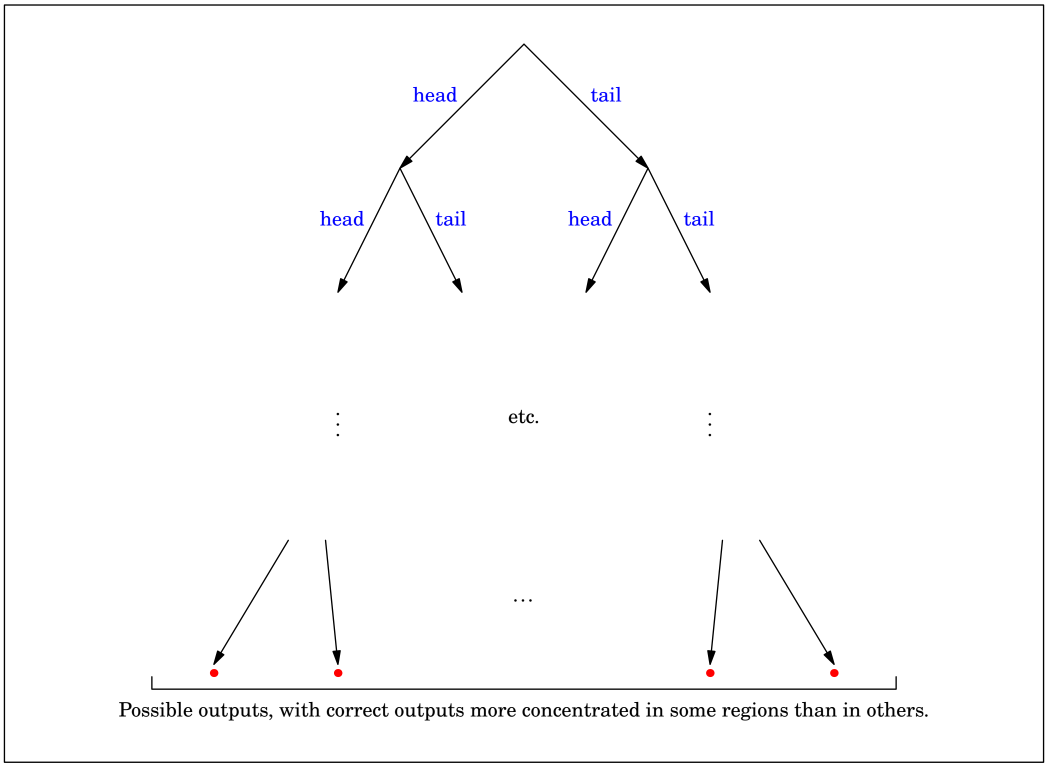

head
tail
head
tail
head
tail
etc.
…
Possible outputs, with correct outputs more concentrated in some regions than in others.

Figure 22.1. Probabilistic computations can often be used as a heuristic to find the regions in which correct answers are most concentrated.

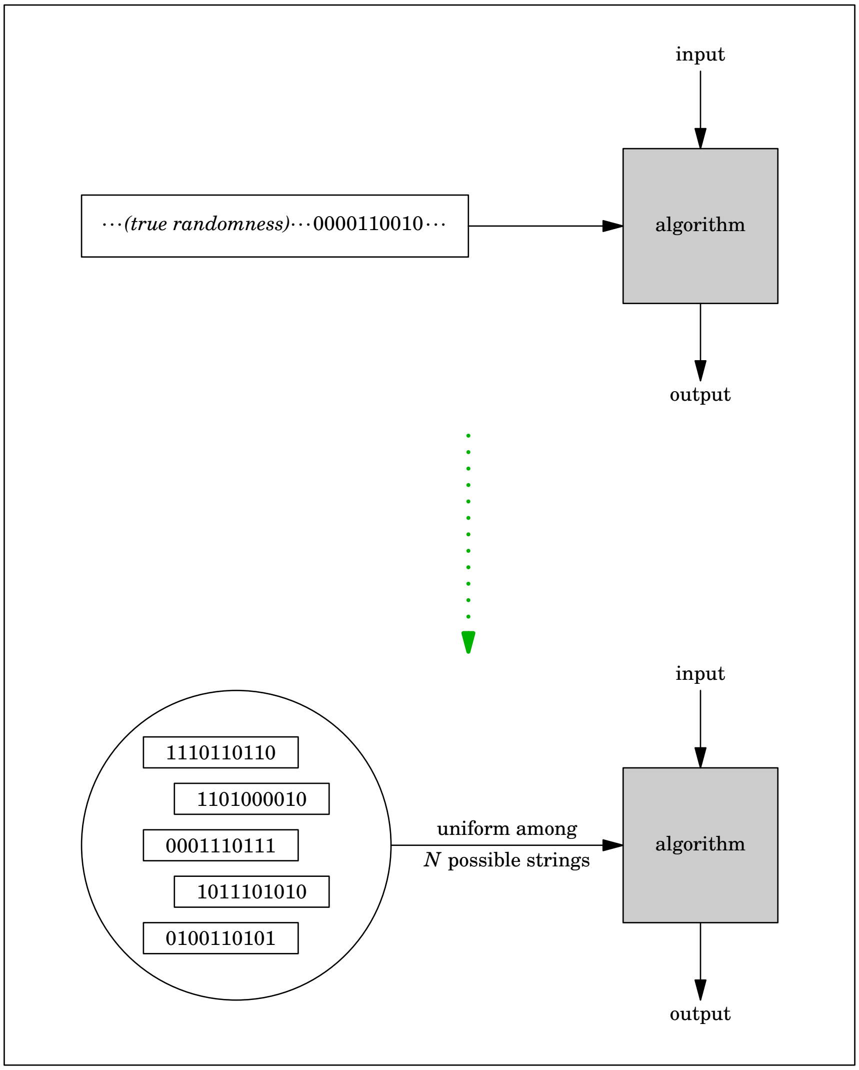


Figure 22.2. Sometimes the deterministic component of a randomized algorithm can't distinguish between a truly random sequence of choices and a well-chosen *pseudorandom* sequence of choices. Or, sometimes we can simulate a random choice from a large sample set (e.g., the set of all binary strings) using a random choice from a small sample set (e.g., a fixed set of N binary strings). If we can do this with N sufficiently small, it may be feasible to find the answer deterministically by running the randomized algorithm on all N of the possible random inputs.

Problem 22.1 (Big Cut)**.** Given an undirected graph $G = (V,E)$ with no loops, find a cut $(S, V \setminus S)$ which crosses at least half of the edges in E.

The solution to this problem is approximated by a simple randomized algorithm:

Algorithm:

```
S ← ∅
for each v ∈ V do
      Flip a coin
      If heads, then S ← S ∪ {v}
return (S, V \ S)
```

The running time is $\Theta(V)$.

Proof of approximation. For each edge $(u,v) \in E$, we have

$$\Pr\left[(u,v) \text{ crosses the cut}\right] = \Pr\begin{bmatrix}(u \in S \text{ and } v \notin S) \text{ or} \\ (u \notin S \text{ and } v \in S)\end{bmatrix} = \frac{1}{2}.$$

Thus, if $I_{(u,v)}$ denotes the indicator random variable which equals 1 if $(u,v) \in S$, then we have

$$\mathbb{E}\left[\# \text{ edges crossed by } S\right] = \mathbb{E}\left[\sum_{(u,v)\in E} I_{(u,v)}\right] = \sum_{(u,v)\in E} \mathbb{E}\left[I_{(u,v)}\right] = \tfrac{1}{2}\left|E\right|.$$

Chernoff's bound then tells us that, when E is large, it is extremely likely that the number of edges crossed is at least $0.499\left|E\right|$. □

22.1 Using Conditional Expectation

We can deterministically simulate the randomized approximation to Problem 22.1 by choosing to include a vertex v if and only if doing so (and making all future decisions randomly) would result in a higher expected number of crossed edges than choosing to exclude v. In more detail, say $V = \{v_1, \ldots v_n\}$, and suppose we have already decided whether S should contain each of the vertices $v_1, \ldots, v_k$. We pretend that all future decisions will be random (even though it's not true), so that we can use probabilistic reasoning to guide our choice regarding v_{k+1}. Ideally, our choice should maximize the value of

$$\underset{\substack{\text{future random}\\ \text{choices}}}{\mathbb{E}}\left[\# \text{ edges crossed} \;\middle|\; \text{our existing decisions about } v_1, \ldots, v_{k+1}\right]. \tag{22.1}$$

Consider the partition

$$E = E_1 \sqcup E_2 \sqcup E_3,$$

where

$$\begin{aligned} E_1 &= \left\{(v_i, v_j) \in E : \; i,j \leq k\right\} \\ E_2 &= \left\{(v_i, v_{k+1}) \in E : \; i \leq k\right\} \\ E_3 &= \left\{(v_i, v_j) \in E : \; j > k+1\right\}. \end{aligned}$$

Whether we choose to include or exclude v_{k+1} from S, the number of crossed edges in E_1 is unaffected. Moreover, since each edge in E_3 has at least one of its vertices yet to be randomly assigned to S or

$V \setminus S$, the expected number of crossed edges in E_3 is $\frac{1}{2}|E_3|$ regardless of where we decide to put v_{k+1}. So in order to maximize (22.1), we should choose to put v_{k+1} in whichever set (either S or $V \setminus S$) produces more crossings with edges in E_2. (It will always be possible to achieve at least $\frac{1}{2}|E_2|$ crossings.) We can figure out what the right choice is by simply checking each edge in E_2.

Proceeding in this way, the value of $\mathbb{E}\left[\text{\# edges crossed}\right]$ starts at $\frac{1}{2}|E|$ and is nondecreasing with each choice. Once we make the nth choice, there are no more (fake) random choices left, so we have

$$\left(\text{\# edges crossed}\right) = \mathbb{E}\left[\text{\# edges crossed}\right] \geq \tfrac{1}{2}\left|E\right|.$$

22.2 Using Pairwise Independence

The sequence of random choices in the randomized approximation to Problem 22.1 is equivalent to picking the set S uniformly at random from the collection $\mathscr{S} = \mathscr{P}(V)$ of all subsets of V. In hindsight, the only reason we needed randomness was to ensure that

$$\Pr_{S \in \mathscr{S}}\left[S \text{ doesn't cross } (u,v)\right] \leq \frac{1}{2} \qquad \text{for each } (u,v) \in E. \tag{22.2}$$

Recall from §10.1 that a subset of V can be represented as a function $V \to \{0,1\}$. In this way, $\mathscr{S}$ becomes a hash family $\mathscr{H} : V \to \{0,1\}$, and (22.2) becomes

$$\Pr_{h \in \mathscr{H}}\left[h(u) = h(v)\right] \leq \frac{1}{2} \qquad \text{for each } (u,v) \in E.$$

This will be satisfied as long as $\mathscr{H}$ is universal. So rather than taking $\mathscr{H}$ to be the collection $\{0,1\}^V$ of all functions $V \to \{0,1\}$, we can take $\mathscr{H}$ to be a much smaller universal hash family; there exist universal hash families of size $O(V)$.[2] In this way, we have

$$\mathop{\mathbb{E}}_{h \in \mathscr{H}}\begin{bmatrix}\text{\# edges crossed by the}\\ \text{cut corresponding to } h\end{bmatrix} \geq \tfrac{1}{2}\left|E\right|.$$

In particular, this guarantees that there exists some $h \in \mathscr{H}$ such that

$$\begin{pmatrix}\text{\# edges crossed by the}\\ \text{cut corresponding to } h\end{pmatrix} \geq \tfrac{1}{2}\left|E\right|.$$

We can simply check every $h \in \mathscr{H}$ until we find it.

Exercise 22.1. *What are the running times of the two deterministic algorithms in this lecture?*

[2] One such universal hash family is described as follows. To simplify notation, assume $V = \{1, \ldots, n\}$. Choose some prime $p \geq n$ with $p = O(n)$ and let

$$\mathscr{H} = \left\{h_{a,b} : a, b \in \mathbb{Z}_p,\ a \neq 0\right\},$$

where

$$h_{a,b}(x) = (ax + b) \bmod p.$$

Lecture 23

Computational geometry

Supplemental reading in CLRS: Chapter 33 except §33.3

There are many important problems in which the relationships we wish to analyze have geometric structure. For example, computational geometry plays an important role in

- Computer-aided design
- Computer vision
- Computer animation
- Molecular modeling
- Geographic information systems,

to name just a few.

23.1 Intersection Problem

Given a set of line segments $S_1, \dots, S_n$ in the plane, does there exist an intersection? Humans, with their sophisticated visual systems, are particularly well-equipped to solve this problem. The problem is much less natural (but still important, given the above applications) for a computer.

Input: Line segments $\mathscr{S} = \{S_1, \dots, S_n\}$ in the plane, represented by the coordinates of their endpoints

Output: *Goal I: Detect.* Determine whether there exists an intersection.
Goal II: Report. Report all pairs of intersecting segments.

The obvious algorithm is to check each pair of line segments for intersection. This would take $\Theta\left(n^2\right)$ time. For Goal II, this can't be improved upon, as it could take $\Theta\left(n^2\right)$ time just to report all the intersections (see Figure 23.1). We will focus on Goal I and show that there exists a $\Theta(n \lg n)$ algorithm.

For ease of exposition, we will assume that our line segments are in *general position*, i.e.,

- No two endpoints have the same x-coordinate. (In particular, there are no perfectly vertical segments.)
- There are no instances of three segments intersecting at a single point.

Both of these assumptions are unnecessary and can be dropped after a couple of minor (but careful) modifications to the algorithm.

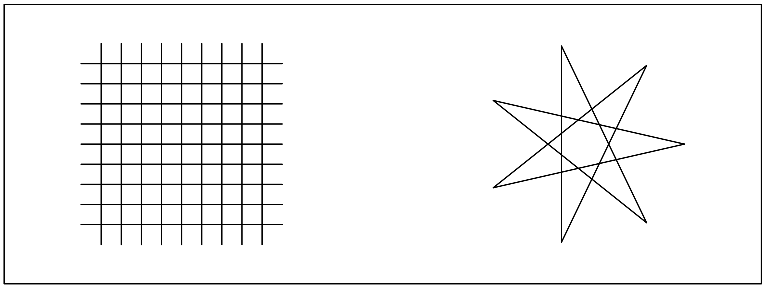


Figure 23.1. If the segments $S_1,\ldots,S_n$ form a square grid, there could be as many as $\left(\frac{n}{2}\right)\left(\frac{n}{2}\right)=\frac{n^2}{4}=\Theta\left(n^2\right)$ intersections. In other situations, it could happen that every pair of segments intersects.

23.1.1 Sweep line and preorder

Suppose you have a sheet of paper with the line segments $S_1,\ldots,S_n$ drawn on it. Imagine dragging a vertical ruler across the page from left to right (see Figure 23.2). This is exactly what we have in mind when we say,

> For each $x \in \mathbb{R}$, let L_x be the vertical line at distance x from the y-axis.[1]

We imagine x increasing over time, so that L_x represents a **sweep line** which moves across the plane from left to right.

For each $x \in \mathbb{R}$, there is a natural *preorder*[2] on the set of non-vertical line segments in the plane: for two non-vertical segments a, b, we say $a \geq_x b$ if the intersection of a (or its extension, if need be) with L_x lies *above* the intersection of b with L_x. This relation needn't be antisymmetric because it could happen that a and b intersect L_x at the same point. In that case, we say that both $a \geq_x b$ and $b \geq_x a$; but of course, this does not imply that $a = b$. Of crucial importance is the following obvious fact:

> Given $x \in \mathbb{R}$ and line segments a and b, there exists an $O(1)$-time algorithm to check whether $a \geq_x b$.[3]

Also important is the following observation:

Observation 23.1. Suppose line segments a and b intersect at the point $P = (x^*, y^*)$. Then one of the following two statements holds:

(i) For $x > x^*$, we have $a \geq_x b$. For $x < x^*$, we have $b \geq_x a$.
(ii) For $x > x^*$, we have $b \geq_x a$. For $x < x^*$, we have $a \geq_x b$.

Conversely, if a and b do *not* intersect, then one of the following two statements holds:

(i) For all x such that L_x intersects a and b, we have $a \geq_x b$.
(ii) For all x such that L_x intersects a and b, we have $b \geq_x a$.

Another way to view the second part of Observation 23.1 is as follows:

[1] It took me a while to figure out how to write down a definition of L_x that didn't include the equation "$x = x$."

[2] A **preorder** is a binary relation that is reflexive and transitive, but not necessarily symmetric or antisymmetric.

[3] It is not so obvious, however, what the best such algorithm is. Most naïve attempts involve calculating a slope, which is numerically unstable. To remedy this, the best algorithms use cross products to compute orientations without using division.

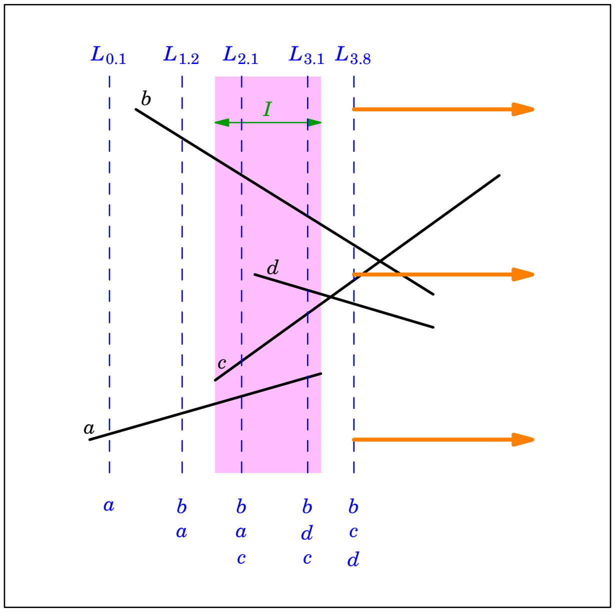


Figure 23.2. A sweep line traversing from left to right. Notice that $b \geq_x d \geq_x c \geq_x a$ for all $x \in I$, but $b \geq_{3.8} c \geq_{3.8} d$.

Thought Experiment 23.2. Consider a finite set of line segments; for concreteness let's say we are looking at the four segments $\{a, b, c, d\}$. There is some interval I (perhaps empty) such that, for each $x \in I$, L_x intersects all four segments (see Figure 23.2). For each $x \in I$, the relation $\geq_x$ induces a preorder structure on the set $\{a, b, c, d\}$. Observation 23.1 implies that, if there are no intersections among $\{a, b, c, d\}$, then this preorder structure does not depend on the choice of x.

Thought Experiment 23.2 leads to the following strategy, which is quite ingenious. To introduce the strategy, we first define the preorder data structure. The **preorder data structure** stores a dynamic collection T of objects along with a preorder on those objects, which we denote by $\geq$. The supported operations are[4]

- INSERT(a) – inserts a into the collection
- DELETE(a) – deletes a from the collection
- ABOVE(a) – returns the element immediately above a. If no such element exists, returns NIL.
- BELOW(a) – returns the element immediately below a. If no such element exists, returns NIL.

The obvious issue with using this data structure in conjunction with our plane sweep is that it only stores a single preorder relation, which we have denoted above by $\geq$ (and the usual counterparts $\leq$, $>$, $<$), whereas the relation $\geq_x$ has the potential to change as x varies. However, this needn't be a problem, as long as:

[4] For ease of exposition, we have obscured one detail from view. It is possible for T to contain two elements d, e such that $d \geq e$ and $e \geq d$. In order for the operations ABOVE(d) and BELOW(d) to work correctly, there must be some rule for breaking ties. To this end, T will also internally store an *order* $\succeq$ on its elements, such that $b \geq a$ always implies $b \succeq a$ (but not necessarily conversely). Then, "the element immediately above a" is the element b such that $b \succ a$ and, whenever $c \succ a$, we always have $c \succeq b$. (Similarly for "the element immediately below a.") The choice of order $\succeq$ does not matter, as long as it is persistent through insertions and deletions.

Condition 23.3. During the lifetime[5] of any particular element a, the ranking of a with respect to the other elements of T does not ever need to be changed.

We are now prepared to give the algorithm:

```
Algorithm: DETECT-INTERSECTION($\mathscr{S}$)
1  Sort the endpoints of segments in $\mathscr{S}$ by $x$-coordinate
2  $T \leftarrow$ new preorder structure
3  ▷ Run the sweepline from left to right
4  for each endpoint $(x, y)$, by order of $x$-coordinate do
5      if $(x, y)$ is the left endpoint of a segment $a$ then
6          Insert $a$ into $T$, using the relation $\geq_x$ to decide where in the preorder $a$ should
           belong*
7          Check $a$ for intersection with $T$.ABOVE($a$)
8          Check $a$ for intersection with $T$.BELOW($a$)
9          if there was an intersection then
10             return TRUE
11     else if $(x, y)$ is the right endpoint of a segment $a$ then
12         Check $T$.ABOVE($a$) for intersection with $T$.BELOW($a$)
13         if there was an intersection then
14             return TRUE
15         $T$.DELETE($a$)
16 return FALSE
```

* This step needs justification. See the proof of correctness.

Proof of correctness. First, note the need to justify line 6. In implementation form, T will probably store its entries in an ordered table or tree; then, when T.INSERT(a) is called, it will take advantage of T's internal order by using a binary search to insert a in $\Theta\left(\lg|T|\right)$ time. In line 6, we are asking T to use the relation $\geq_x$ to decide where to place a. This is fine, *provided that* $\geq_x$ agrees with T's internal preorder when considered as a relation on the elements that currently belong to T. (Otherwise, a binary search would not return the correct result.[6]) So, over the course of this proof, we will argue that

> Whenever line 6 is executed, the relation $\geq_x$ agrees with T's internal preorder when considered as a relation on the elements that currently belong to T.

From Thought Experiment 23.2 we see that the above statement holds as long as the sweep line has not yet passed an intersection point. Thus, it suffices to prove the following claim:

Claim. DETECT-INTERSECTION terminates before the sweep line passes an intersection point.

The claim obviously holds if there are no intersections, so assume there is an intersection. Let's say the leftmost intersection point is P, where segments a and b intersect. Assume without loss of generality that a's left endpoint lies to the right of b's left endpoint (so that b gets inserted into T

[5] By "the lifetime of a," I mean the period of time in which a is an element of T.

[6] Actually, in that situation, there would be no correct answer; the very notion of "binary search" on an unsorted list does not make sense.

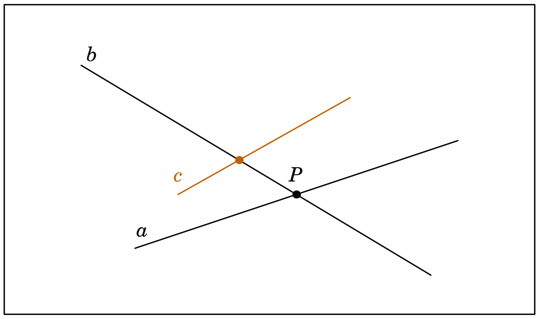


Figure 23.3. If the left endpoint of c lies between a and b to the left of P and the right endpoint of c lies to the right of P, then c must intersect either a or b.

before a does). If a ever becomes adjacent to b in T, then either lines 7–8 or line 12 will detect that a and b intersect, and the algorithm will halt.

So we are free to assume that a and b are never adjacent in T until after the sweep line has passed P. This means that there exists a line segment c such that the left endpoint of c lies between a and b[7] and to the left of P, and the right endpoint of c lies to the right of P. Geometrically, this implies that c must intersect either a or b, and that intersection point must lie to the *left* of P (see Figure 23.3). But this is impossibe, since we assumed that P was the leftmost intersection point. We conclude that the claim holds.

Actually, we have done more than prove the claim. We have shown that, if there exists an intersection, then an intersection will be reported. The converse is obvious: if an intersection is reported, then an intersection exists, since the only time we report an intersection is after directly checking for one between a specific pair of segments. □

23.1.2 Running time

The running time for this algorithm depends on the implementation of the preorder data structure. CLRS chooses to use a red–black tree[8], which has running time $O(\lg n)$ per operation. Thus, DETECT-INTERSECTION has total running time

$$\Theta(n \lg n).$$

23.2 Finding the Closest Pair of Points

Our next problem is simple:

Input: A set Q of points in the plane

Output: The two points of Q whose (Euclidean) distance from each other is shortest.

The naïve solution is to proceed by brute force, probing all $\binom{n}{2}$ pairs of points and taking $\Theta\left(n^2\right)$ time. In what follows, we will exhibit a subtle divide-and-conquer algorithm which runs in $\Theta(n \lg n)$ time.

[7] That is, if the left endpoint of c is (x, y), then either $a \le_x c \le_x b$ or $a \ge_x c \ge_x b$.

[8] For more information about red–black trees, see Chapter 13 of CLRS.

In §23.1, the pseudocode would not have made sense without a few paragraphs of motivation beforehand. By contrast, in this section we will give the pseudocode first; the proof of correctness will elucidate some of the strange-seeming choices that we make in the algorithm. This is more or less how the algorithm is presented in §33.4 of CLRS.

The algorithm begins by pre-sorting the points in Q according to their x- and y-coordinates:

```
Algorithm: CLOSEST-PAIR($Q$)
$X \leftarrow$ the points of $Q$, sorted by $x$-coordinate
$Y \leftarrow$ the points of $Q$, sorted by $y$-coordinate
return CLOSEST-PAIR-HELPER($X, Y$)
```

Most of the work is done by the helper function CLOSEST-PAIR-HELPER, which makes recursive calls to itself:

```
Algorithm: CLOSEST-PAIR-HELPER($X, Y$)
if $|X| \leq 3$ then
    Solve the problem by brute force and return
$x^* \leftarrow$ the median $x$-coordinate of $X$
Let $X_L \subseteq X$ consist of those points with $x$-coordinate $\leq x^*$
Let $X_L \subseteq X$ consist of those points with $x$-coordinate $> x^*$
Let $Y_L \subseteq Y$ consist of those points which are in $X_L$
Let $Y_R \subseteq Y$ consist of those points which are in $X_R$
▷ Find the closest two points in the left half
$\langle p_L, q_L \rangle \leftarrow$ CLOSEST-PAIR-HELPER($X_L, Y_L$)
▷ Find the closest two points in the right half
$\langle p_R, q_R \rangle \leftarrow$ CLOSEST-PAIR-HELPER($X_R, Y_R$)
$\delta_R \leftarrow$ distance from $p_L$ to $q_L$
$\delta_R \leftarrow$ distance from $p_R$ to $q_R$
$\delta \leftarrow \min\{\delta_L, \delta_R\}$
$Y' \leftarrow$ those points in $Y$ whose $x$-coordinate is within $\delta$ of $x^*$
▷ Recall that $Y'$ is already sorted by $y$-coordinate
$\epsilon \leftarrow \infty$
for $i \leftarrow 1$ to $|Y'| - 1$ do
    $p \leftarrow Y'[i]$
    for $q$ in $Y'[i+1, \ldots, \min\{i+7, |Y'|\}]$ do
        if $\epsilon >$ distance from $p$ to $q$ then
            $\epsilon \leftarrow$ distance from $p$ to $q$
            $p^* \leftarrow p$
            $q^* \leftarrow q$
if $\epsilon < \delta$ then
    return $\langle p^*, q^* \rangle$
else if $\delta_R < \delta_L$ then
    return $\langle p_R, q_R \rangle$
else
    return $\langle p_L, q_L \rangle$
```

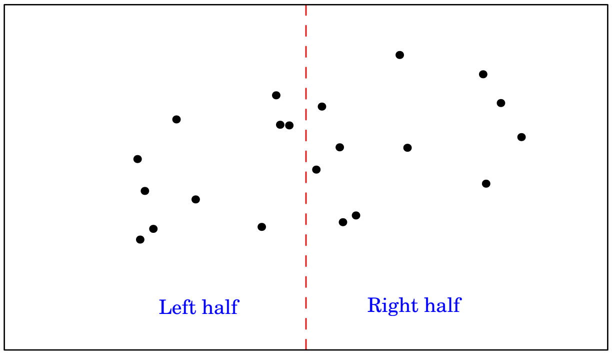


Figure 23.4. CLOSEST-PAIR-HELPER divides the set X into a left half and a right half, and recurses on each half.

23.2.1 Running time

Within the procedure CLOSEST-PAIR-HELPER, everything except the recursive calls runs in linear time. Thus the running time of CLOSEST-PAIR-HELPER satisfies the recurrence

$$T(n) = 2 \cdot T\left(\tfrac{n}{2}\right) + O(n), \tag{23.1}$$

which (by the Master Theorem in §4.5 of CLRS) has solution

$$T(n) = O(n \lg n). \tag{23.2}$$

Note that, if we had decided to sort within each recursive call to CLOSEST-PAIR-HELPER, the $O(n)$ term in (23.1) would have instead been an $O(n \lg n)$ term and the solution would have been $T(n) = O\left(n (\lg n)^2\right)$. This is the reason for creating a helper procedure to handle the recursive calls: it is important that the lists X and Y be pre-sorted so that recursive calls need only linear-time operations.

Note also that, if instead of lines 20–24 we had simply checked the distance between each pair of points in Y', the $O(n)$ term in (23.1) would have instead been an $O\left(n^2\right)$ term, and the solution would have been $T(n) = O\left(n^2\right)$.

23.2.2 Correctness

CLOSEST-PAIR-HELPER begins by recursively calling itself to find the closest pairs of points on the left and right halves. Thus, lines 15–24 are ostensibly an attempt to check whether there exists a pair of points $\langle p^*, q^* \rangle$, with one point on the left half and one point on the right half, whose distance is less than that of any two points lying on the same half. What remains to be proved is that lines 15–24 do actually achieve this objective.

By the time we reach line 15, the variable δ stores the shortest distance between any two points that lie on the same half. It is easy to see that there will be no problems if δ truly is the shortest possible distance. The case we need to worry about is that in which the closest pair of points—call it $\langle p, q \rangle$—has distance less than δ. In such a case, the x-coordinates of p and q would both have to be within δ of x^*; so it suffices to consider only points within a vertical strip V of width 2δ centered at $x = x^*$ (see Figure 23.5). These are precisely the points stored in the array Y' on line 15.

Say $p = Y'[i]$ and $q = Y'[j]$, and assume without loss of generality that $i < j$. In light of lines 20–24 we see that, in order to complete the proof, we need only show that $j - i \le 7$.

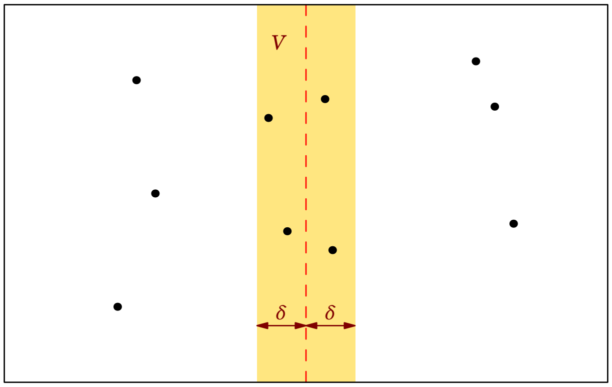


Figure 23.5. It suffices to consider a vertical strip V of width 2δ centered at $x = x^*$.

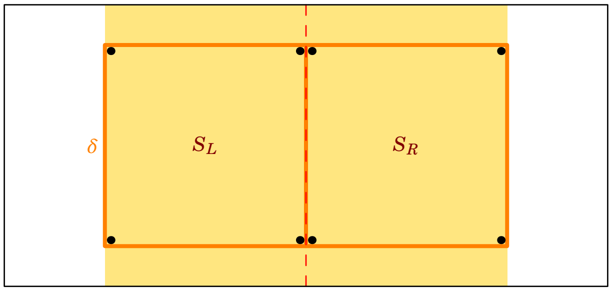


Figure 23.6. Each of S_L and S_R can hold at most 4 points. (Actually, in completely ideal geometry, S_R cannot contain 4 points because its left boundary is excluded. But since the coordinates in a computational geometry problem are typically given as floating point numbers, we are not always guaranteed correct handling of edge cases.)

Say $p = (p_x, p_y)$. Let S_L be the square (including boundaries) of side length δ whose right side lies along the vertical line $x = x^*$ and whose bottom side lies along the horizontal line $y = p_y$. Let S_R be the square (excluding the left boundary) of side length δ whose left side lies along the vertical line $x = x^*$ and whose bottom side lies along the horizontal line $y = p_y$. It is evident that q must lie within either S_L or S_R (see Figure 23.6). Moreover, any two points in the region S_L are separated by a distance of at least δ; the same is true for any two points in the region S_R. Thus, by a geometric argument[9], S_L and S_R each contain at most four points of Y'. In total, then, $S_L \cup S_R$ contains at most eight points of Y'. Two of these at-most-eight points are p and q. Moreover, since Y' consists of all points in V sorted by y-coordinate, it follows that the at-most-eight points of $S_L \cup S_R$ occur *consecutively* in Y'. Thus, $j - i \leq 7$.

[9] One such argument is as follows. Divide S_L into northeast, northwest, southeast and southwest quadrants. Each quadrant contains at most one point of Y' (why?), so S_L contains at most four points of Y'.

Index

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012