Selected Topics in Algorithms

Jeff Achter and Roberto Tamassia

Department of Computer Science
Brown University
Providence, RI 02912–1910
{jda,rt}@cs.brown.edu

September 1993

Contents

Chapter 1

Recurrence Relations

1.1 Introduction

Many algorithms are *recursive* — a large problem is broken up into sub-problems, each a smaller instance of the original problem. The time-complexity analysis of a recursive algorithm is usually determined by solving a *recurrence relation*, which is a recursive definition of a function.

We introduce recurrence relations by means of an example: the Towers of Hanoi problem. In this problem, we start off with three rods or towers. On one of the rods is a pile of some number of disks, say, N. Each disk in the stack is a little smaller than the one underneath it. The challenge is to transfer the tower of disks to another rod, subject to a couple of constraints:

1. only one disk may be moved at a time;
2. a larger disk may never be placed on top of a smaller disk.

As it turns out, there is an easy recursive solution to this problem. Clearly, if $N = 1$ we just move the unique disk and we are done. Now, let $N \geq 2$, and assume that we know how to solve the problem when there are $N-1$ disks, i.e., we know how to transfer $N-1$ disks subject to the above constraints. Our strategy for $N \geq 2$ disks consists of three phases:

1. Transfer the top $N-1$ disks to a free rod.
2. Move the bottom (largest) disk on to the other free rod.
3. Move the stack of $N-1$ disks on to the rod with the largest disk.

Hence, to solve the problem for N disks, we solve it for $N-1$ disks, move the bottom disk, then solve the problem for $N-1$ disks again. How can we analyze the efficiency of this recursive strategy? Let $T(N)$ be the number of moves performed to transfer N disks. Clearly, $T(1) = 1$. For $N \geq 2$, we have that Phases 1 and 3 make $T(N-1)$ moves each, while Phase 2 makes exactly one move, so that $T(N) = 2T(N-1) + 1$. We can write this more formally as a recurrence relation:

$$\begin{aligned} T(1) &= 1 \\ T(N) &= 2 \cdot T(N-1) + 1 \qquad \text{for } N \geq 2 \end{aligned}$$

The above recurrence relation allows us to compute $T(N)$ by repeated evaluations of the above formula. For example, we can compute $T(5)$ as follows:

$$\begin{aligned} T(1) &= 1 \\ T(2) &= 2T(1)+1 &&= 3 \\ T(3) &= 2T(2)+1 &&= 7 \\ T(4) &= 2T(3)+1 &&= 15 \\ T(5) &= 2T(4)+1 &&= 31 \end{aligned}$$

This method of evaluating the function $T(N)$ is rather inefficient; think, for example, how long it would take to compute $T(100)$. We would like to "solve" the recurrence relation, i.e., find a *closed form* for $T(N)$ that allows us to compute $T(N)$ with a constant number of arithmetic operations, for any value of N.

To solve the above recurrence relation, we will use a technique called *unfolding* (also known as "expansion," or "telescoping", or "repeated substitution"). We start by observing that the recurrence relation applied to $N-1$ gives:

$$T(N-1) = 2T(N-2) + 1$$

Hence, we have

$$T(N) = 2(2T(N-2)+1) + 1$$

By repeated substitutions we obtain:

$$\begin{array}{rcll} T(N) & = & 2T(N-1)+1 & \text{Step } 0 \\ & = & 2(2T(N-2)+1)+1 & \text{Step } 1 \\ & = & 4T(N-2)+2+1 & \\ & = & 8T(N-3)+4+2+1 & \text{Step } 2 \\ & \vdots & & \\ T(N) & = & 2^i T(N-i) + 2^{i-1} + 2^{i-2} + \cdots + 1 & \text{Step } i \\ & = & 2^i T(N-i) + 2^i - 1 & \\ & \vdots & & \end{array}$$

The substitution process terminates at Step i such that $N - i = 1$, i.e., at Step $N-1$. We conclude that

$$T(N) = 2^{N-1}T(1) + 2^{N-1} - 1 = 2^{N-1} + 2^{N-1} - 1 = 2^N - 1$$

.

Hence, the solution of the recurrence relation for $T(N)$ is

$$T(N) = 2^N - 1$$

Using the asymptotic notation we can also say

$$T(N) = \Theta(2^N)$$

1.2 Fundamental Recurrence Relations

In this section we examine several important recurrence relations and present techniques to solve them. There are three general strategies for solving recurrence relations. One is intuition, the second, unfolding, and the third, a combination of guessing and induction.

1.2.1 $T(N) = T(N/2) + 1; \quad T(1) = 1$

This recurrence relation describes the worst-case time complexity of binary search in a sorted array. Binary search compares the search value x to the middle element of the array. If they are equal, we are done. If x is less than the middle element, then binary search is recursively called on the first half of the array. If x is greater, then the second half of the array is recursively searched.

Let $T(N)$ be the time complexity of binary search on an array with N elements. In the worst case (when the x is not found or is found at the very last step), binary search spends some constant amount of time to compare x with the middle element, and then takes time $T(N/2)$ to search a subarray of size $N/2$. Hence, we have

$$\begin{aligned} T(1) &= 1 \\ T(N) &= T(N/2) + 1 \qquad \text{for } N \geq 2 \end{aligned}$$

By unfolding the recurrence, we obtain:

$$\begin{aligned} T(N) &= T(N/2) + 1 && \text{Step 1} \\ &= T(N/4) + 1 + 1 && \text{Step 2} \\ &= T(N/8) + 1 + 1 + 1 && \text{Step 3} \\ &\vdots \\ T(N) &= T(N/2^i) + i && \text{Step } i \\ &\vdots \end{aligned}$$

The substitution process terminates at step i such that $N/2^i = 1$, or $i = \log_2 N$. Thus, $T(N) = 1 + \log_2 N$, and the worst-case time complexity of binary search is $\Theta(\log N)$.

Notice the abstraction we used. The inner workings of the algorithm were not used in the evaluation of the recurrence relation. The only piece of information we used was T(N) = T(N/2) + 1.

A reasonable question is: why were we allowed to pick an integer i such that $N = 2^i$? What happens if N isn't a power of 2? If our ultimate goal is to get an asymptotic expression for $T(N)$, it turns out that we can fudge a little. Suppose that N is *not* a power of 2. We can take the integer n such that that $2^n \leq N < 2^{n+1}$, i.e., $n = \lfloor \log_2 N \rfloor$, where $\lfloor x \rfloor$ denotes the largest integer less than or equal to x. (This is equivalent to "padding" the array with dummy values so that the size of the array is a power of 2.) We have

$$T(2^n) \;\leq\; T(N) \;<\; T(2^{n+1})$$

Hence

$$\begin{aligned} n + 1 \;&\leq\; T(N) \;<\; n + 2 \\ \lfloor \log_2 N \rfloor + 1 \;&\leq\; T(N) \;<\; \lfloor \log_2 N \rfloor + 2 \end{aligned}$$

from which we derive $T(N) = \Theta(\log N)$

Note that we assumed $T(N)$ is nondecreasing. In other words, if $M \leq N$, then $T(M) \leq T(N)$. This is a reasonable assumption for time complexity functions since it should take more time to process more input data.

Finally, although $T(N) = \log_2 N + 1$, we write that it is $\Theta(\log N)$ instead of $\Theta(\log_2 N)$. Why can we ignore the subscript, i.e., the base of the logarithm? Recall that in general

$$\log_b x = \frac{\log_a x}{\log_a b}$$

which means that changing the base of the logarithm from a to b corresponds to multiplying by the constant $\frac{1}{\log_a b}$. But remember that *in asymptotic notation we ignore constant factors*, so that the base of the logarithm is irrelevant and we write simply $T(N) = \Theta(\log N)$.

1.2.2 Induction

The mathematical technique of *induction* can be used to prove the correctness of a solution for a recurrence relation. Here is how induction works. Suppose we're trying to prove that some property P is true for all positive integers n. We denote the assertion that "P is true for n" by writing $P(n)$. How can we show $P(n)$ for *all* integers $n \geq 1$? The inductive argument has two steps. First, we must prove the *basis case*, showing that $P(1)$ is true. (Usually this step is pretty straightforward.) Second, we need to prove the *inductive step*: that if $P(n)$ is true for all integers n such that $1 \leq n < k$, then $P(n)$ is also true for $n = k$.

It may not be obvious why proving $P(1)$, and that if $P(n)$ for $n < k$ then $P(k)$, should prove anything at all. Proof by induction is fairly subtle.[1] Well, suppose we were trying to verify a particular case, say, $P(5)$. From the basis step we know $P(1)$. By the inductive step, if $P(1)$ is true, then so is $P(2)$. Invoking the inductive step again, since $P(1)$ and $P(2)$ are true, so is $P(3)$, and so on up to $P(5)$.

At any rate, we shall use induction to prove that $T(N) = \log_2 N + 1$.

Basis step: In this case, it's easy, because $\log_2 1 + 1 = 1$ and we are given that $T(1) = 1$.

Inductive step: By the inductive hypothesis, we assume that $T(n) = \log_2 n + 1$ for all $n < N$. So what can we say about $T(N)$? By definition, $T(N) = T(N/2) + 1$. But from the inductive hypothesis, we know that $T(N/2) = \log_2(N/2) + 1 = \log_2 N$. (This is because $N/2 < N$.) So $T(N) = \log_2 N + 1$, and the inductive step is complete.

Since we were able to prove the basis step and the inductive step, we have formally proved by induction that $T(N) = \log_2 N + 1$ for all $N \geq 1$.

One slightly tricky thing about such proofs is knowing where to go. Unless you have a good idea of what the answer is, induction may not help you prove much. Typically, one makes an "educated guess" of the answer, using techniques such as telescoping. Then, this guess is (hopefully) proved correct using induction.

1.2.3 $T(N) = T(N/2) + N; \quad T(1) = 1$

Initial Attack

This recurrence relation describes a hypothetical algorithm which looks at every element (N), then recurses on half of them ($T(N/2)$). We'll use unfolding again:

$$\begin{aligned} T(N) &= T(N/2) + N && \text{Step 1} \\ &= T(N/4) + N/2 + N && \text{Step 2} \\ &= T(N/8) + N/4 + N/2 + N && \text{Step 3} \\ &\vdots \\ &= T(N/2^i) + N/2^{i-1} + N/2^{i-2} + \cdots + N/2 + N && \text{Step } i \\ &\vdots \end{aligned}$$

We stop substituting at step n such that $N/2^n = 1$, i.e., at step $n = \log_2 N$, and we obtain

$$T(N) = N\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^n}\right)$$

The above expression contains a sum which occurs fairly often.

[1] In fact, this important technique was completely unknown to Euclid, so most of his theorems were only proven up to $n = 3$ and merely vigorously asserted for the general case.

A Brief Detour

Consider the sum

$$G_{\frac{1}{2}}(n) = \sum_{i=0}^{n} \left(\frac{1}{2}\right)^i = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n}$$

This sum is a special case of a geometric sum, which is of the type

$$G_\alpha(n) = \sum_{i=0}^{n} \alpha^i \qquad (\alpha \neq 1)$$

From the theory of geometric sums we have that

$$G_\alpha(n) = \frac{\alpha^{n+1} - 1}{\alpha - 1}$$

Hence

$$1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^n} = 2 - \frac{1}{2^n} < 2 \qquad \text{for all } n \geq 1$$

Back to $T(N) = T(N/2) + N$

Anyways, we have found that

$$N \ < \ T(N) = N\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^n}\right) \ < \ 2N$$

We happily ignore the constants, and conclude that

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

.

Notice that we have not found an *exact* solution for the recurrence, but only an *asymptotic* solution. This is enough for the purpose of time complexity analysis that motivated our study of recurrences. A more accurate analysis shows that

$$T(N) = 2N - 1.$$

Even if we don't know how this answer was arrived at, we can *prove* this time bound using induction.

Basis step: This is simple; $T(1) = 1 = 2 \cdot 1 - 1$.

Inductive step: Assume that $T(n) = 2n - 1$ for $n < N$. Then $T(N) = T(N/2) + N$, by definition. Now, $N/2 < N$, so by the inductive hypothesis, assume $T(N/2) = 2 \cdot \frac{N}{2} - 1 = N - 1$. Thus, $T(N) = N - 1 + N = 2N - 1$, and the proof is complete.

1.2.4 $T(N) = T(\alpha N) + N; \quad T(1) = 1 \quad (0 < \alpha < 1)$

We now solve this slightly more general recurrence relation. It corresponds to an algorithm which looks at every element, then recursively operates on some fraction α of the original elements. Note that $0 < \alpha < 1$. If α weren't positive, the expression would be meaningless. If α were greater than one, the size of the array would increase with every recursion — it would never terminate. So, we'll make the assumption that $0 < \alpha < 1$.

As usual, we unfold the recurrence.

$$\begin{array}{rcll} T(N) & = & T(\alpha N) + N & \text{Step 1} \\ & = & T(\alpha^2 N) + \alpha N + N & \text{Step 2} \\ & = & T(\alpha^3 N) + \alpha^2 N + \alpha N + N & \text{Step 3} \\ & \vdots & & \\ T(N) & = & T(\alpha^i N) + \alpha^{i-1} N + \alpha^{i-2} N + \cdots + \alpha^2 N + \alpha N + N & \text{Step } i \\ & \vdots & & \end{array}$$

The substitution process terminates at Step m such that $\alpha^m N = 1$, or $m = \log_{\frac{1}{\alpha}}(N)$, and we have:

$$T(N) = N(1 + \alpha + \alpha^2 + \cdots + \alpha^{m-1} + \alpha^m)$$

We encounter again a geometric sum so that

$$T(N) = NG_\alpha(m) = N\frac{\alpha^{m+1} - 1}{\alpha - 1}$$

Since $\alpha < 1$, then $\alpha^m < 1$ and we have

$$N \ < \ T(N) \ < \ \frac{1}{1-\alpha} N$$

Again, we ignore constants and simply write

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

.

1.2.5 $T(N) = 2T(N/2) + N; \quad T(1) = 1$

This recurrence relation describes a typical "divide and conquer" algorithm (such as merge-sort) that divides the input into two halves, processes recursively each half, and then combines the solutions of the two subproblems in linear time.

As usual, we unfold the recurrence

$$\begin{array}{rcll} T(N) & = & 2T(N/2) + N & \text{Step 1} \\ & = & 2(2T(N/4) + N/2) + N & \text{Step 2} \\ & = & 4T(N/4) + 2N & \\ & = & 4(2T(N/8) + N/4) + 2N & \text{Step 3} \\ & = & 8T(N/8) + 3N & \\ & \vdots & & \\ & = & 2^i T(N/2^i) + iN & \text{Step } i \\ & \vdots & & \end{array}$$

The substitution process terminates at step n such that $N/2^n = 1$, i.e., $n = \log_2 N$, and we obtain:

$$T(N) = 2^n T(1) + nN = N + N \log_2 N$$

.

Remembering that the asymptotic notation allows us to discard asymptotically smaller terms and constant factors, we conclude that

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

Using the technique of induction, we can prove the claim that $T(N) = N + N \log_2 N$.

Basis step: Since $\log_b 1 = 0$, $T(1) = 1 = 1 + 1 \log_2 1$.

Inductive step: Assume that $T(n) = 2T(n/2) + n$ for $n < N$. Then

$$\begin{aligned} T(N) &= 2T(N/2) + N \\ &= 2[\tfrac{N}{2} + \tfrac{N}{2} \log_2 \tfrac{N}{2}] + N \\ &= N + N \log_2 \tfrac{N}{2} + N \\ &= N + N \log_2 N - N \log_2 + N \\ &= N + N \log_2 N. \end{aligned}$$

This completes the proof.

1.2.6 $T(N) = 2T(N/2) + N^\alpha; \quad T(1) = 1; \quad \alpha \neq 1$

Without much commentary, we proceed as always:

$$\begin{aligned} T(N) &= 2T(N/2) + N^\alpha && \text{Step 1} \\ &= 2(2T(N/4) + (\tfrac{N}{2})^\alpha) + N^\alpha && \text{Step 2} \\ &= 4T(N/4) + N^\alpha(1 + \tfrac{1}{2^\alpha}) \\ &= 4(2T(N/8) + (\tfrac{N}{4})^\alpha) + N^\alpha(1 + \tfrac{1}{2^\alpha}) && \text{Step 3} \\ &= 8T(N/8) + N^\alpha(1 + \tfrac{1}{2^\alpha} + \tfrac{1}{4^\alpha}) \\ &\vdots \\ &= 2^i T(N/2^i) + N^\alpha(1 + \tfrac{1}{2^\alpha} + \tfrac{1}{4^\alpha} + \cdots + \tfrac{1}{(2^{i-1})^\alpha}) && \text{Step } i \\ &\vdots \end{aligned}$$

At this point, we (hopefully!) have enough experience to recognize that the substitution process terminates at step $n = \log_2 N$. It now remains to evaluate the sum

$$1 + \frac{1}{2^\alpha} + \frac{1}{4^\alpha} + \cdots + \frac{1}{(2^{n-1})^\alpha} = \sum_{i=0}^{n-1} \frac{1}{(2^i)^\alpha}$$

We replace $\frac{1}{(2^i)^\alpha}$ with β^i, where $\beta = \left(\frac{1}{2}\right)^\alpha$. Note that $\beta < 1$. This gives us

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

Hence, we have

$$2^n + N^\alpha \;<\; T(N) \;<\; 2^n + \frac{1}{1-\beta} N^\alpha$$

$$N + N^\alpha \;<\; T(N) \;<\; N + \frac{1}{1-\beta} N^\alpha$$

To find an asymptotic expression for $T(N)$ we need to distinguish two cases. If $\alpha < 1$ then N^α is asymptotically smaller than N and we have $T(N) = \Theta(N)$. If instead $\alpha > 1$ then N^α is asymptotically larger than N so that $T(N) = \Theta(N^\alpha)$. Therefore

$$T(N) = \begin{cases} \Theta(N) & \text{if } \alpha < 1 \\ \Theta(N^\alpha) & \text{if } \alpha > 1 \end{cases}$$

1.3 Conclusion

We have shown how to solve several recurrence relations. For some recurrences we have found exact solutions, while for some others we have determined asymptotic solutions. It probably wouldn't hurt to summarize our results:

Recurrence Relation	**Solution**
$T(N) = 2T(N-1) + 1$	$\Theta(2^N)$
$T(N) = T(N/2) + 1$	$\Theta(\log N)$
$T(N) = T(N/2) + N$	$\Theta(N)$
$T(N) = T(\alpha N) + N$	$\Theta(N)$ if $\alpha < 1$
$T(N) = 2T(N/2) + N$	$\Theta(N \log N)$
$T(N) = 2T(N/2) + N^\alpha$	$\Theta(N)$ if $\alpha < 1$ $\Theta(N^\alpha)$ if $\alpha > 1$

Chapter 2

Priority Queues

2.1 Introduction

Certain abstract data structures appear in computer science in a wide variety of contexts. You are probably already familiar with one of these, the stack. While these structures are useful for specific problems, the study of their abstract properties also proves rewarding. If we study a structure and understand its behavior, we gain a new tool for solving other problems. We now consider one such abstract data structure, the *priority queue*. We'll use the priority queue in a number of algorithms for handling graphs. More immediately, we'll see that an easy solution to the sorting problem falls out of a study of priority queues.

The priority queue is perhaps best introduced with an example. Imagine a professor trying to figure out what to do. Several tasks demand her attention: she needs to submit her abstract to the International Conference on Pointless Computation; get the car washed; answer the telephone; prepare notes for the afternoon's lecture; pay her undergraduate research assistant; and extinguish the fire which recently ignited in her shoes. Clearly, some of these tasks are more pressing than others.

A priority queue lets us model this situation. A set of records is maintained, one for each of the tasks. Associated with each task is a priority, a score denoting its urgency. Furthermore, there are several basic procedures which act on this set of records:

create – Initialize an empty priority queue.

insert(task,priority) – Add a new task with the given priority to the priority queue.

remove_max – Find the task with the highest priority. Return this task, and delete it from the priority queue.

is_empty – Return true if the priority queue is empty, and false otherwise.

With these procedures, we can easily use a priority queue to model the professor:

```
Professor
create
insert(abstract, 5)
insert(car, 2)
insert(phone, 8)
insert(notes, 7)
insert(pay, 1)
insert(fire, 37)
```

```
while not is_empty
  next_task ← remove_max
  deal_with(next_task)
endwhile
```

When the professor executes the program, the first task handled will be fire. Its priority is 37, greater than that of any other issue. Once this task is removed from the priority queue, the task with the highest priority will be phone, then notes, etc. In this way, she addresses her problems in a reasonably intelligent order.

In this chapter we'll be assigning *high* values to important tasks. This may conflict with your preconceptions of a priority-one alert, etc. Nonetheless, we'll be using the convention that a low number means a low priority. On one hand, this is fairly arbitrary. On the other hand, it lets us model the fact that, no matter how urgent an issue is, something can always come up which overshadows it.

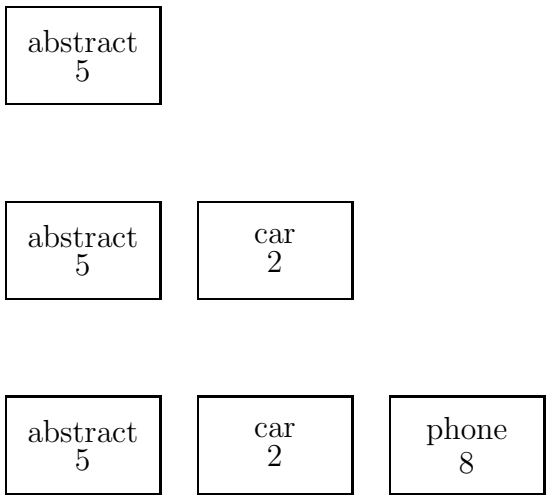
2.2 Implementation

So far, we have described what a priority queue does without mentioning implementation details. Note that, for our professor program, the details of the priority queue – is it an array, or linked list, or tree? – are completely irrelevant. Even knowing only the abstract properties of the data structure, we can solve problems with it.

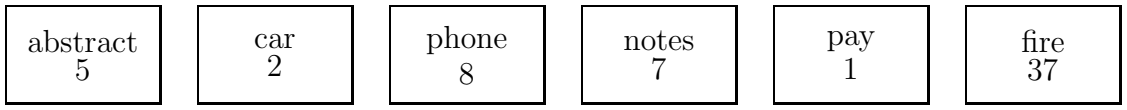
However, in order to analyze the complexity of a program using priority queues, we must know the complexity of insert and remove_max. (It's not hard to see that create and is_empty should be simple routines taking constant time. We'll not address them again.) This, in turn, requires that we examine the inner workings of the priority queue. We now consider three possible implementations of a priority queue.

2.2.1 Unsorted List

One of the easiest methods of implementation is the unsorted list. We maintain an unsorted list as an array or linked list of records. New records are added to the end of the list. Thus, while the priority queue is being constructed, it looks like this:



and so on until we get to:

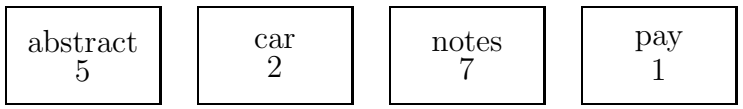


Little processing occurs during insert. If we're reasonably intelligent about keeping track of the last record added, we need merely add the new task to the end of the list. The time it takes to insert is independent of N, the number of records already in the priority queue. Thus, insert takes time $O(1)$.

The function remove_max is slightly more complicated. We must scan each record to find the one with the highest priority. Once the largest record has been identified, we must somehow delete it from the queue. If the record is at the end, as for fire, deletion poses no particular problem.

What happens on the next remove_max, when the record phone with top priority is in the middle of the list?

There are two obvious ways of filling the gap left by the removal. We can join the right half to the left half.

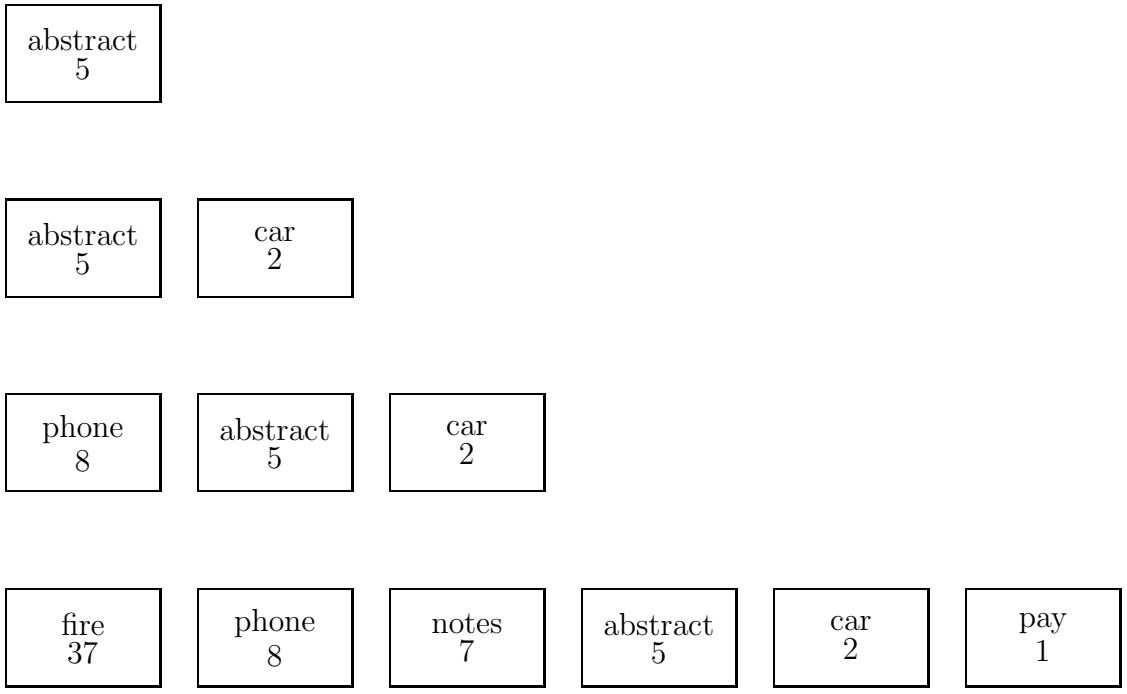


Alternatively, since we don't care about the ordering, we can move the last record into the hole.

The former is probably easier in a linked-list representation; the latter, in an array. Regardless, the deletion takes some constant amount of time. Recall, though, that we initially had to process each of the N records to find the one with highest priority. Therefore, remove_max takes time $O(N+1) = O(N)$.

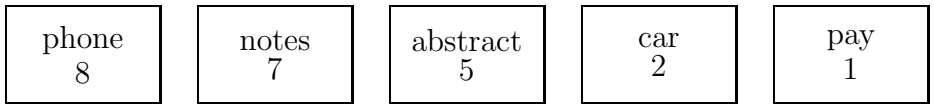
2.2.2 Sorted List

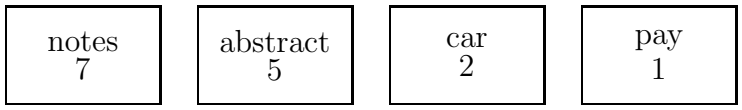
remove_max always retrieves the task with the highest priority. We might try to speed up this process by ensuring the the first record always has top priority. This means that the records must always be *sorted* by decreasing priority. Each time we add an event to the priority queue, we want the list of tasks to remain sorted, as in the following example:



To find where to insert a task, we check the priority of each task until we find one with a lower priority then the new task. The new task is then inserted just before this task. In the worst case, the task will be inserted at the end, and each of the N tasks must be examined. So we see that insert runs in time $O(N)$. (Note that on average, a task will be inserted halfway through the list, yielding $N/2$ comparisons.)

Although we have made insert more costly, we have considerably simplified remove_max. The largest task is guaranteed to be the first task. We need merely delete and return it.





This deletion scheme takes a constant amount of time, so the time complexity of remove_max is $O(1)$.

2.2.3 Heap

A New Representation

We now look at a third representation of a priority queue. Unlike those considered so far, this representation will require *another* new abstraction. Thus far we have examined priority queues based on either sorted or unsorted lists. These, in turn, can be represented as linked lists or arrays. We now introduce the heap, a representation based on a binary tree.

Before defining the heap, we recall that a *complete* binary tree is one in which all levels are filled, with the possible exception of the last one. Additionally, internal nodes on the last level are on the left of the external (or unused) nodes on that level.

A heap is a complete binary tree with keys stored at the internal nodes, with a special condition maintained:

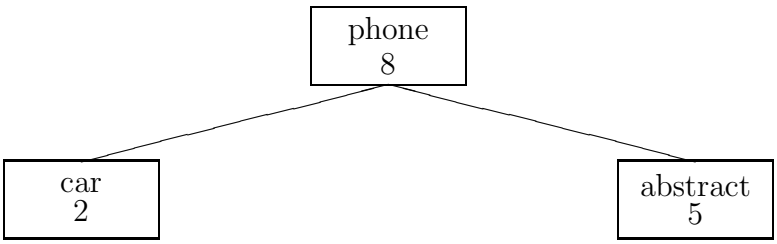
Heap Condition: The key of a node is always greater than or equal to the key of its children.

Each node corresponds to one task, with the key given by the priority.

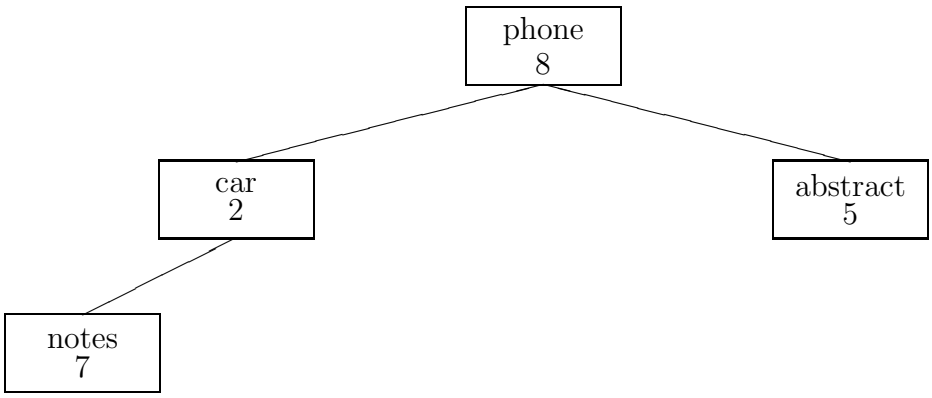
Operation insert

Inserting a task into a heap is not too difficult. Initially, the new task is placed in the next free space in the tree, so that the tree remains complete. (Below we'll see an efficient method for finding the next free space in a binary tree.) The resulting tree may violate the heap condition; the new task may have a higher priority than it parent in the tree. In this case, we switch the new task with its parent. We continue this comparison between the new task and its parent until the heap condition is once again satisfied.

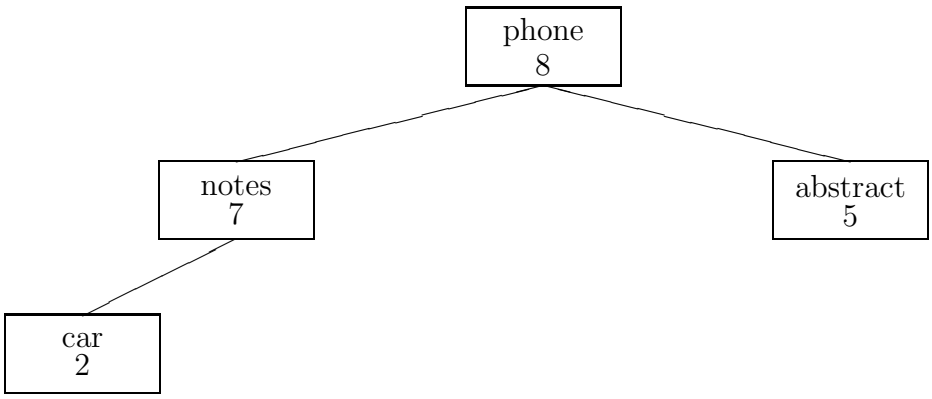
In the following example, we look at how the notes task is inserted, and assume that the first three tasks have already been inserted. Suppose we have the following heap:



We insert notes with priority 7.



The priority of notes is higher than that of its parent, car. Therefore, we exchange these two tasks.



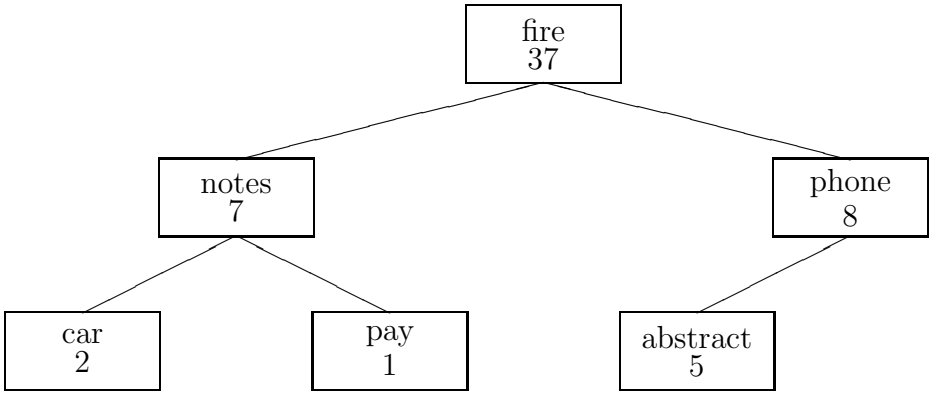
We now compare notes to its new parent, phone. The phone task has higher priority than notes, so the insertion is complete.

Observe that the number of exchanges is bounded by the height of the tree. Since the heap is a complete binary tree, its height is $\lfloor \log_2 N \rfloor$. (Remember, $\lfloor x \rfloor$ denotes the greatest integer which is smaller than or equal to x.) Therefore, insert takes $O(\log N)$ time.

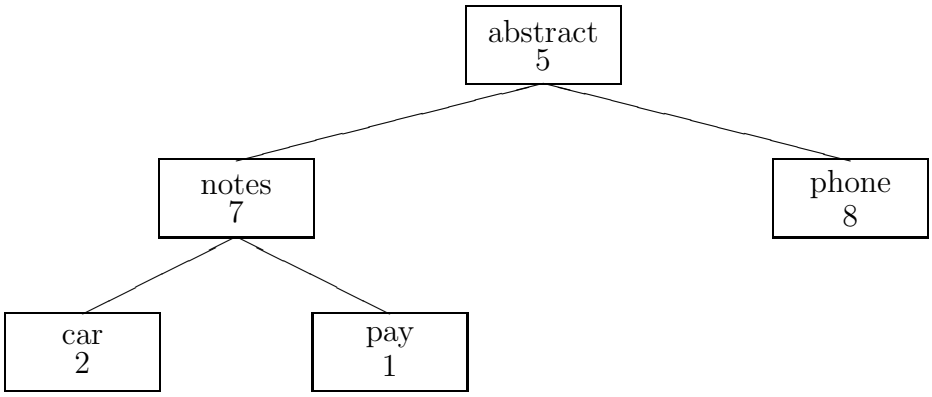
Operation remove_max

It should be obvious that the task with the highest priority is at the top of the heap. What is less than obvious is how to fill the gap left by its removal. One idea is to "promote," say, the left subtree to the top position. Unfortunately, this leaves the right subtree hanging. Instead, we'll move the last task up to the root position of the tree. (We'll call this the new task, even though it hasn't been actually added to the tree.) This, of course, probably violates the heap condition. So we compare the priority of the new task to those of its children. If a child's priority is greater than that of the new task, we exchange the two. (If both children have greater priority, we exchange with the higher-priority child). We continue comparing the new task to its children, exchanging if necessary, until the heap condition is restored.

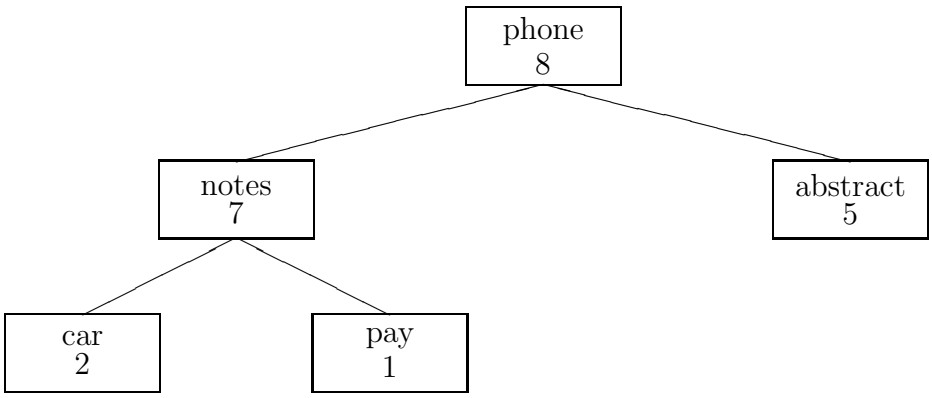
Hopefully, an example will clarify. Suppose we call remove_max when all tasks have been added. The initial heap looks like



We return fire as the largest task, and put abstract in its place.



The priority of abstract is smaller than those of either of its children. Since phone has higher priority than notes, we exchange phone and abstract.



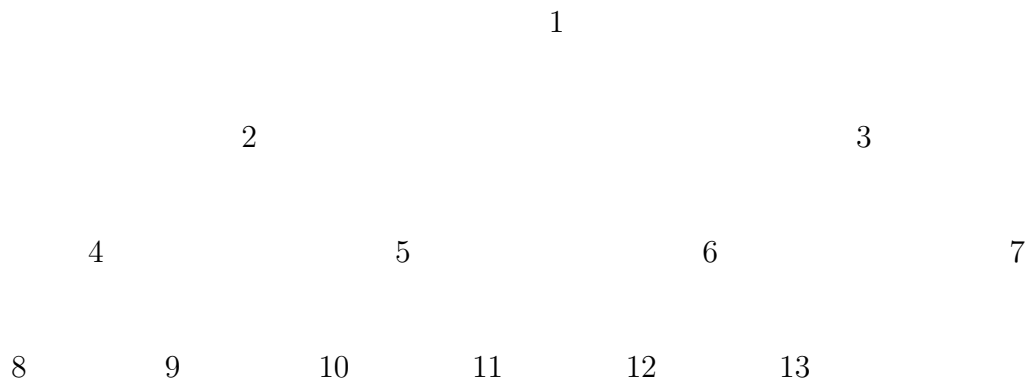
abstract is now at a leaf, so we're finished.

Again, the number of comparisons in remove_max is bounded by the height of the tree. The new node can propagate no further than the heap's height. Therefore, like insert, the time complexity of remove_max is $O(\log N)$.

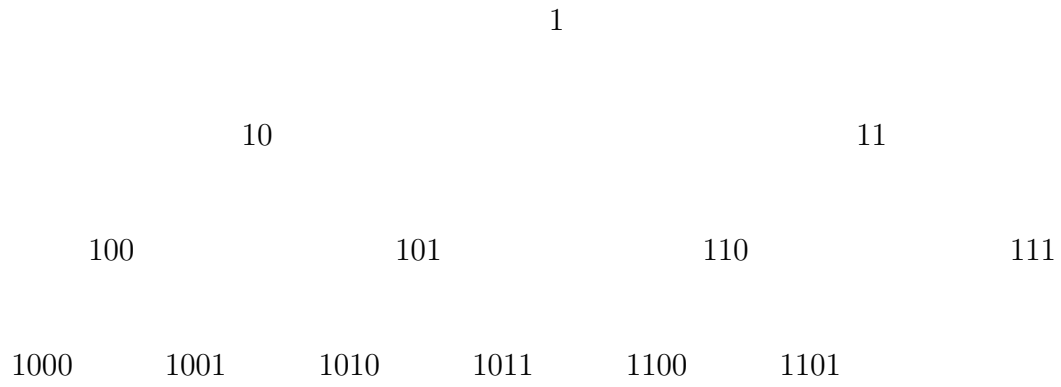
It might seem that we could also restore the heap condition by propagating the "hole" down the tree. When we initially remove the highest-priority task, we leave a hole at the root. We can move the root's largest child into the hole, and thus move the hole down one level. We could repeat this operation, moving the hole down to the bottom level. It might seem that this would re-assert the heap condition. However, a heap must be a *complete* tree. The lowest nodes are no longer guaranteed to be arranged so that the internal nodes are to the left of the external nodes.

A Neat Trick

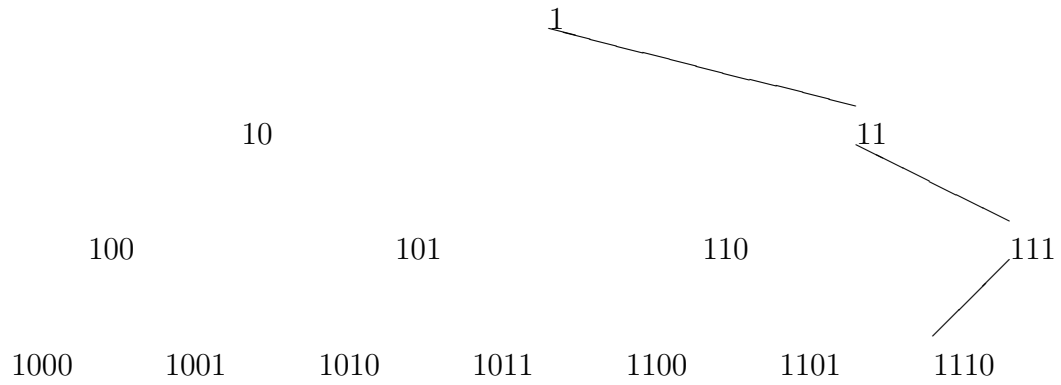
There is an elegant method for finding the next available position in a complete binary tree. Consider the order in which nodes are added:



Let's rewrite these as binary numbers:



Observe that, as we traverse the tree, if we go left we add a 0 to the end of the binary representation of the current node. Similarly, going right corresponds to adding a 1. Using this observation, we can easily find the N^{th} node by examining the bits of N from left to right, or most significant to least significant. Hence, the next position, corresponding to node $N+1$, is found by considering the binary representation of $N+1$. For example, to find the next location, we look for node 14, or 1110 in binary notation:



2.3 Sorting with a Priority Queue

Using a priority queue, we can easily sort an array $a[1 \ldots N]$ of N records with integer keys:

```
PQ_SORT
create
for i ← 1 to N
  insert(a[i], a[i].key)
for i ← N downto 1
  a[i] ← remove_max
```

This sorting algorithm calls insert and remove_max N times. Assume that create takes constant time. If $I(N)$ is the time of insert, and $R(N)$ is the time of remove_max, then the complexity of PQ_sort is $O(N \cdot I(N) + N \cdot R(N))$.

For the unsorted list, the time complexity is $O(N + N^2) = O(N^2)$. Similarly, for the sorted list, it is $O(N^2+N) = O(N^2)$. Using the heap implementation, we get time complexity $O(N \log N + N \log N) = O(N \log N)$, which is faster than the other two. We can summarize this information in a short table:

Data Structure	insert	remove_max	PQ_sort
unsorted list	$O(1)$	$O(N)$	$O(N^2)$
sorted list	$O(N)$	$O(1)$	$O(N^2)$
heap	$O(\log N)$	$O(\log N)$	$O(N \log N)$

We have taken one high-level data structure, the priority queue, and used it to define a sorting algorithm. The details of the sort vary according to the implementation details of the priority queue. In fact, there are correspondences between some implementations of PQ_sort and sorts we've already seen.

Consider how the unsorted list version works. The initial insertion process isn't very interesting. After all of the insert calls have been made, the priority queue is an unsorted list of tasks. Each time we call remove_max, the entire list is scanned for the largest remaining priority. By reading off the tasks in this order, we get a sorted list. But this is nothing but a selection sort! Remember, a selection sort consists merely of searching for the largest, then next largest, etc., element of a list. So in the unsorted list implementation, PQ_sort is basically a selection sort.

The algorithm works a little differently with a sorted list. In this case, each priority is inserted into its proper place in a list of already-sorted priorities. The successive calls to remove_max simply read off this sorted list. The process of inserting elements into a list which is always sorted is exactly what insertion sort does.

Finally, there is the heap-based implementation of PQ_sort. This sort doesn't really correspond well to any other sort we've encountered. In fact, such a sort is usually called a *heapsort*.

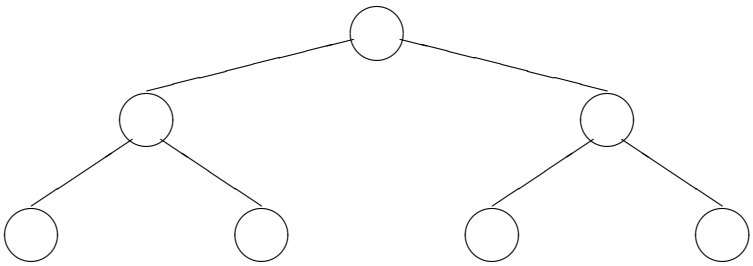
It may be instructive to review our general development of priority queues, heaps, and heapsort. Initially, we described the priority queue as an abstract object and a set of operations on it. Then, we examined some specific implementations of the priority queue, and analyzed each of them. Finally, we described a new algorithm – PQ_sort – based on our definition of a priority queue. Since we had already studied the behavior of priority queues, the analysis of PQ_sort was relatively easy.

Chapter 3

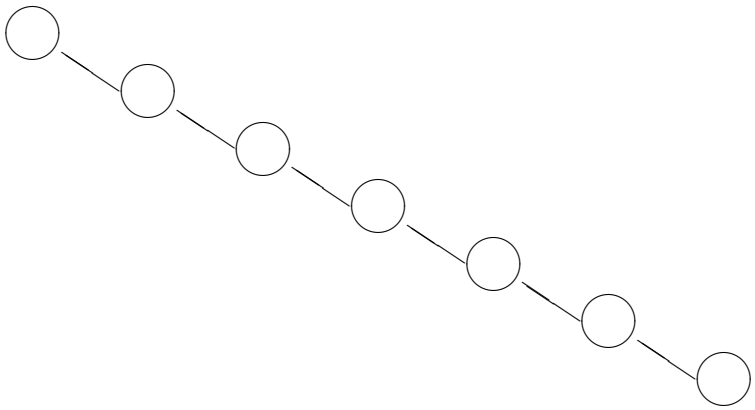
Red-Black Trees

3.1 Introduction

In this chapter we'll be taking a look at red-black trees. Before we start, it probably wouldn't hurt to recall what they are and why they might be useful. A common data structure for the dictionary problem is the binary search tree. Typically, we store the keys with an inorder ordering; the key of a node v is greater than the keys in the left subtree of v, and less than the keys in the right subtree of v. It's tempting to assert that all operations (search, insert, delete) on a binary search tree with N keys take time $O(\log N)$. Certainly, this appears to be the case in a "nice" tree such as

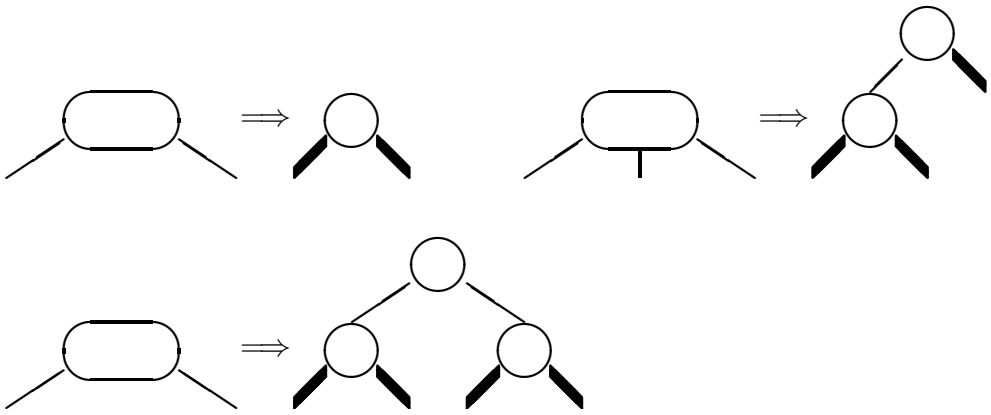


Unfortunately, if we aren't careful we can get trees like



Balanced tree schemes in general, and red-black trees in particular, are methods for guarding against such unbalanced trees. In class we took a brief look at the 2-3-4 tree.

While conceptually simple, it's rather difficult to implement. It turns out that by labeling some edges red and others black, we can model a 2-3-4 tree using a regular binary tree. Briefly, we have the following correspondences between 2-3-4 trees and red-black trees:



(Please note that the heavy lines represent black edges, while the slightly lighter lines represent red edges.)

Don't worry if you don't remember the details of 2-3-4 trees. We're not going to mention 2-3-4 trees again; the focus will be on red-black trees. So, a *red-black tree* is a binary search tree with edges colored red or black, subject to the following constraints, the *red-black invariants*:

1. No path from the root to any node contains two consecutive red edges.
2. Let $B(u)$ be the black depth of a leaf u, i.e., the number of black edges along the path from the root to u. Then $B(u) = B(v)$ for any leaves u and v.
3. Assume that edges are directed downwards, from parent to child. Then the incoming edge of each leaf is colored black.

Incidentally, some authors prefer to color the nodes instead of the edges.[1] The definitions are equivalent; let the color of a node be the color of its incoming edge, or vice-versa.

From the definition, we can immediately deduce a few properties of red-black trees. For a red-black tree T let $B = B(u)$ for some leaf u. We call B the *black height* of T. What, then, is H, the height of T? Since leaves have B black edges between them and the root, H is at least B. We can also find an upper bound for the height. Consider the longest possible path from the root to some leaf. This path has alternating black and red edges. If there were two consecutive black edges, we could make the path longer by inserting a red edge between them (without affecting the black height). If there were two consecutive red edges, this would violate red-black invariant . So the longest possible path has just as many red edges as black edges. Thus, we have $B \leq H \leq 2B$. Notice that we have been able to figure this out using only the definition of red-black trees.

Additionally, there are at least $2^B - 1$ and at most $2^{2B} - 1$ internal nodes in T. So if N is the number of internal nodes, then $B = \Theta(\log N)$ and $H = \Theta(\log N)$. For any search operation, in the worst case we will examine H nodes. Hence, a search takes $O(\log N)$ time, which is

[1] Some authors also prefer colouring to coloring.

optimal for a binary tree. So if we can (relatively) easily construct and maintain red-black trees, we have a good method for building balanced trees with optimal characteristics.

3.2 Insertion

Insertion into a red-black tree is easily described in high-level terms. Assume we're trying to insert some key k.

1. Find the leaf in the tree where k would be, if it were present.
2. Replace the leaf with a new internal node, u, with key k.
3. Let u's incoming edge be red; that way, the black height doesn't change.
4. Restore the red-black invariants.

Typically, the student requests a slightly less abstract discussion of insertion, especially step 4.

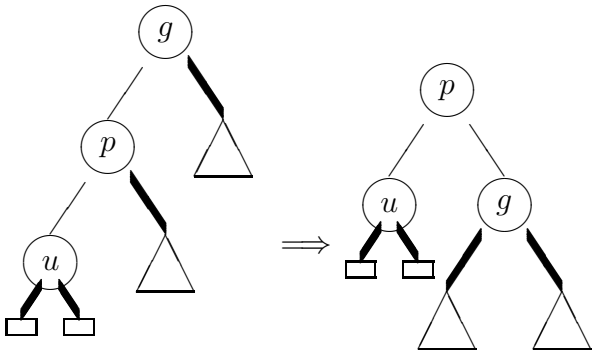
The classic introduction to red-black trees obliges; this paper will be no different. The classic introduction "helpfully" enumerates a half-dozen cases in which the red- black invariants have been violated, and how to restore them in each case. Here is where this paper differs. Instead of listing lots of special cases, we'll simply give two general principles. These two rules are pretty easy to remember. Furthermore, all of the special cases are easily analyzed with the two rules. Let u be the new node, p its parent, and g the parent of the parent. If the edge from g to p is black, then the invariants haven't been violated, and we're done. Otherwise, we're going to have to somehow modify the tree. Without further ado, we present *The Rules*:

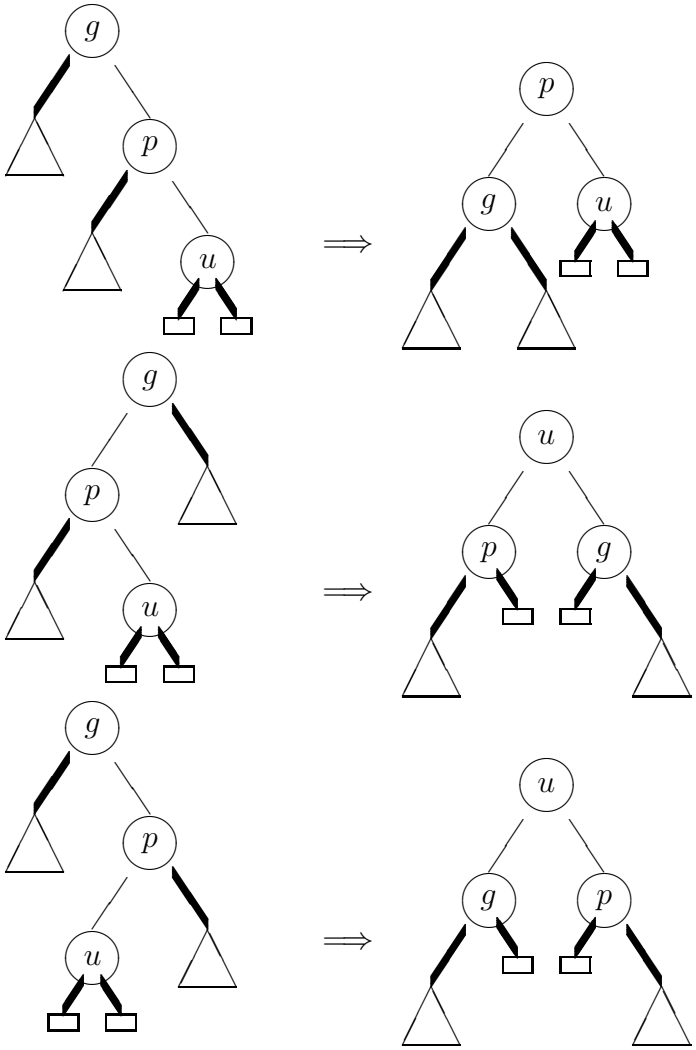
1. *Faced with two adjacent reds, subtrees have to lose their heads*
2. *If after doing this they're stuck, they've no choice but to pass the buck.*

Alternatively, we have

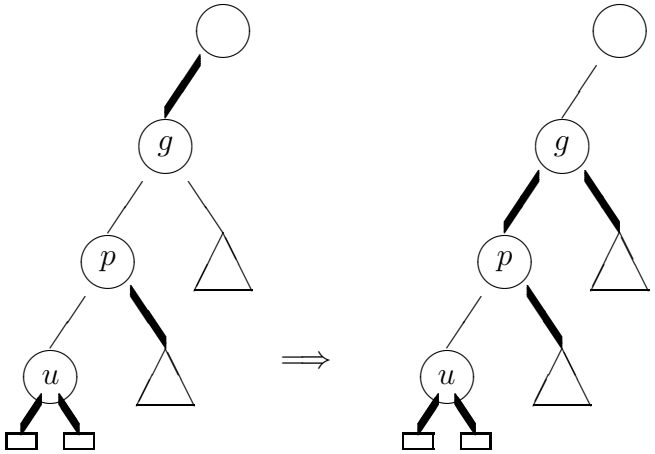
1. *When reds derange, root change.*
2. *When 1 can't cope, promote.*

While pithy, these two rules may not be (initially) transparent. Embodied in Rule 1 is our general approach to restructuring. Consider the subtree T' rooted at g. Notice that T' is unbalanced. To rebalance T', we change the root of T' to be the median of g, p and u. The other two nodes (and the subtrees rooted by them) will fall neatly to the left and right of the median. This simple heuristic accounts for many of the cases encountered in standard texts:





The incoming edge of the new root of T' remains black. That way, the black height of the subtree, and thus the entire tree, is maintained. In all of the examples given above, this restructuring was sufficient to remove the problem. Using *rotations* or *double rotations*, we can restore the red-black invariants. There is, however, a case in which the restructuring would not correct the violation. This is the case where p's sibling — the other child of g — has a red incoming edge. In this instance, we make both p's and p's sibling's incoming edges black. Then, g's incoming edge is changed to red:



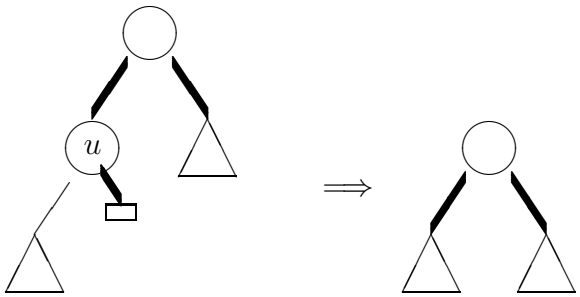
This process is called *promotion*. Of course, by changing g's incoming edge to red, we may violate red-black invariant 1 again (g's parent may have an incoming red edge). This time, the violation (if any) will be further up on the tree. If necessary, we can keep promoting all the way up to the root. To avoid special cases, think of the root has having a virtual black incoming edge.

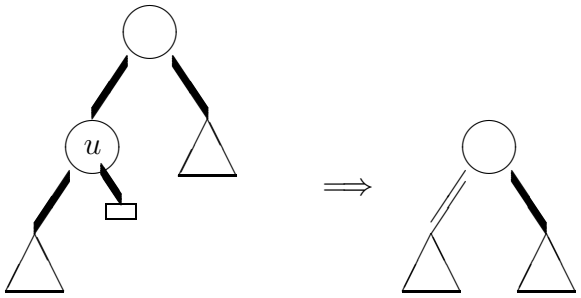
For one insertion, one restructuring may be necessary, plus some promotions. Each time we promote, we move two levels up the tree. Since the height of the tree is $O(\log N)$, there are $O(\log N)$ promotions. Since restructuring and promotion each take constant time, insertion takes time $O(\log N)$.

3.3 Deletion

Deletion on red-black trees isn't much harder than insertion. While it doesn't fit quite as nicely into a pair of couplets, the general approach is the same. We identify the node to be deleted, make some structural changes, and then restore the red-black invariants.

Suppose we are trying to delete a key k. As for standard binary search trees, we can move k to a node u that has at least one leaf child, and then remove u. After removing u (and its leaf child) we have to join the two edges formerly incident on u. To preserve the black height, if one of the two edges is red and the other is black, we color the resulting edge black, otherwise (both edges are black) we color the resulting edge *double black*.

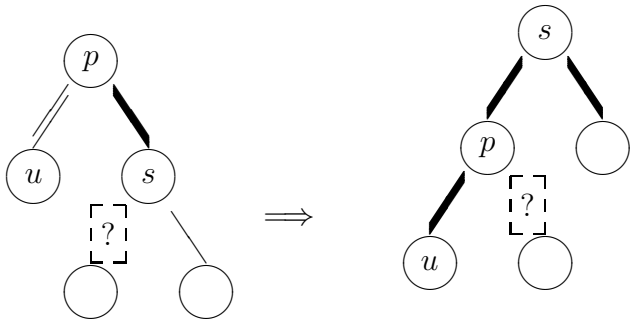


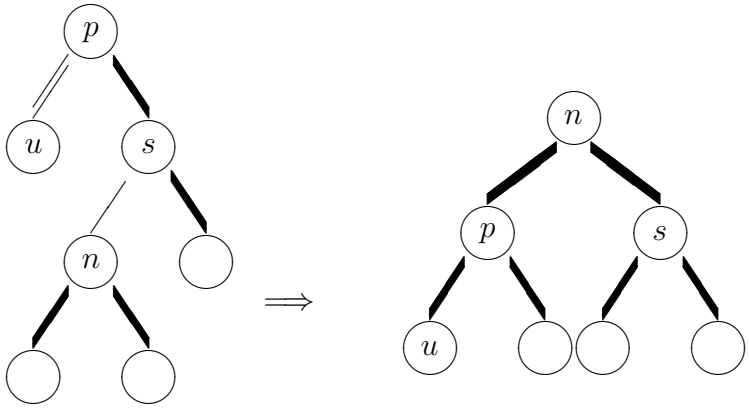


By convention, the double black edge counts as a sequence of two black edges. It has been temporarily introduced to preserve red-black invariant 2. However, it is not allowed in a red-black tree, and we have to get rid of it somehow. The idea is to "compensate" for the double black edge. We want to find a nearby red edge to color black, thereby removing the "extra black" from the double black edge.

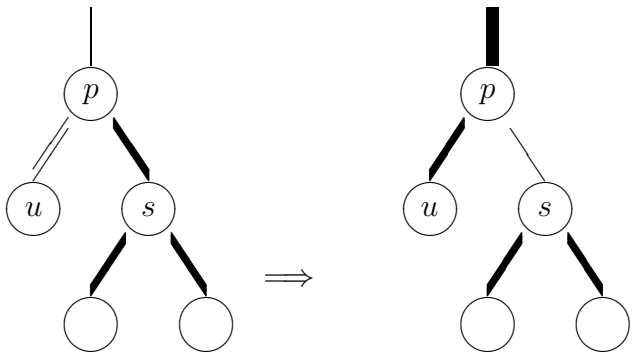
Let p denote the parent of u and s the sibling of u. We consider first the case where s has a black incoming edge.

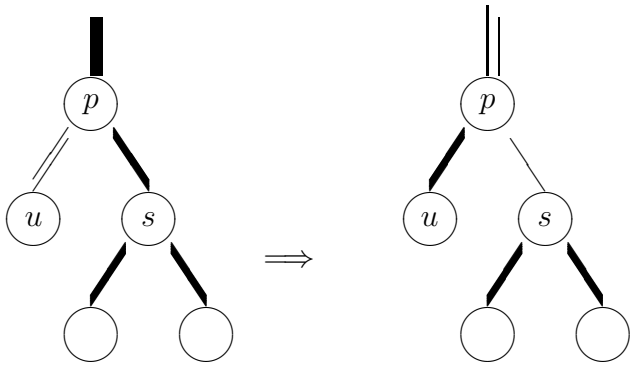
Assume there is a red edge nearby, i.e., one of the children of s has a red incoming edge. If we can find such a red edge, all is well. We eliminate the double black edge with a restructuring as follows:





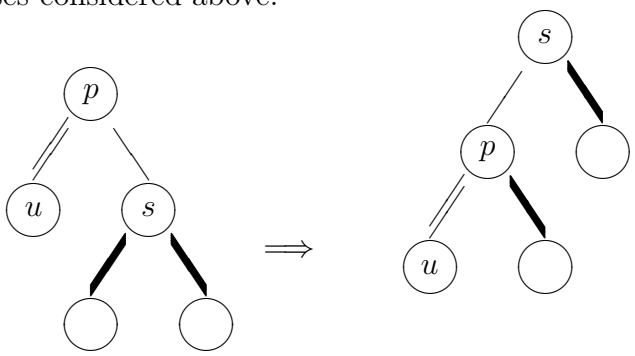
(A question mark denotes an edge which can be either red or black.) If there is no red edge available to absorb the extra black, i.e., the incoming edges of s and its children are black, we perform a *demotion*, which consists of coloring u's edge black, s's edge red, and p's edge black or double black depending on whether it was formerly red or black.





Note that demotion may not solve the double-black problem. However, it pushes the problem higher up in the tree.

Finally, if s has a red incoming edge, with one restructuring (a rotation) we can reduce the tree to one of the cases considered above.



Note that now at most one demotion will be performed since p's incoming edge is red.

The time complexity analysis of deletion is quite similar to that of insertion. The search for the key to be deleted takes time $O(\log N)$. The time to move u to the appropriate node is also $O(\log N)$. A restructuring or demotion takes constant time. The number of restructurings is at most two, and the number of demotions is bounded by the height of the tree. Thus, the time complexity of the entire deletion algorithm is $O(\log N)$.

Chapter 4

Geometric Algorithms

4.1 Introduction

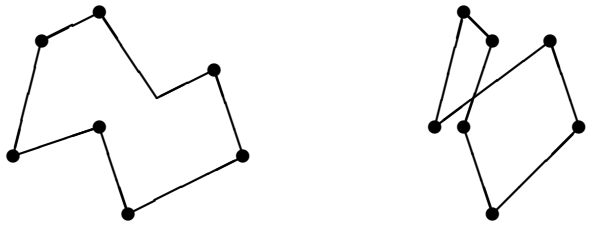
Computational geometry is a branch of computer science which takes geometry as its domain. The abstract objects considered include points, lines and polygons. Examples of problems which are addressed are testing whether two line segments intersect and whether a point is inside or outside a polygon. We'll take a look at algorithms to solve these and other geometric problems.

First, some definitions are in order. Unless stated otherwise, assume that we are working in two dimensions. All objects lie in the Euclidean plane. Thus, a point is defined by a pair of coordinates, (x, y). (We can be a bit vague about whether x and y are integers or real numbers; the same algorithms apply in either case.) A line segment s is given by its two endpoints, $s = (p_0, p_1) = ((x_0, y_0), (x_1, y_1))$. Of course, a segment is actually an infinite set of points (x, y) satisfying

$$\begin{aligned} x &= x_0 + \alpha \cdot (x_1 - x_0) \\ y &= y_0 + \alpha \cdot (y_1 - y_0) \end{aligned}$$

for some $0 \leq \alpha \leq 1$. However, the segment is completely determined by its endpoints, and so it is described by merely listing its two ends.

A polygon is given by a circular sequence of points $p_0, p_1, p_2, \ldots, p_{n-1}$, which gives the vertices in counterclockwise order. The edges of the polygon are (p_0, p_1), (p_1, p_2), ..., (p_{n-2}, p_{n-1}) and (p_{n-1}, p_0). In a simple polygon, none of those edges intersect, except for the trivial intersection of adjacent edges at the vertices they share. In an intersecting polygon, at least one pair of edges intersects in a nontrivial way.



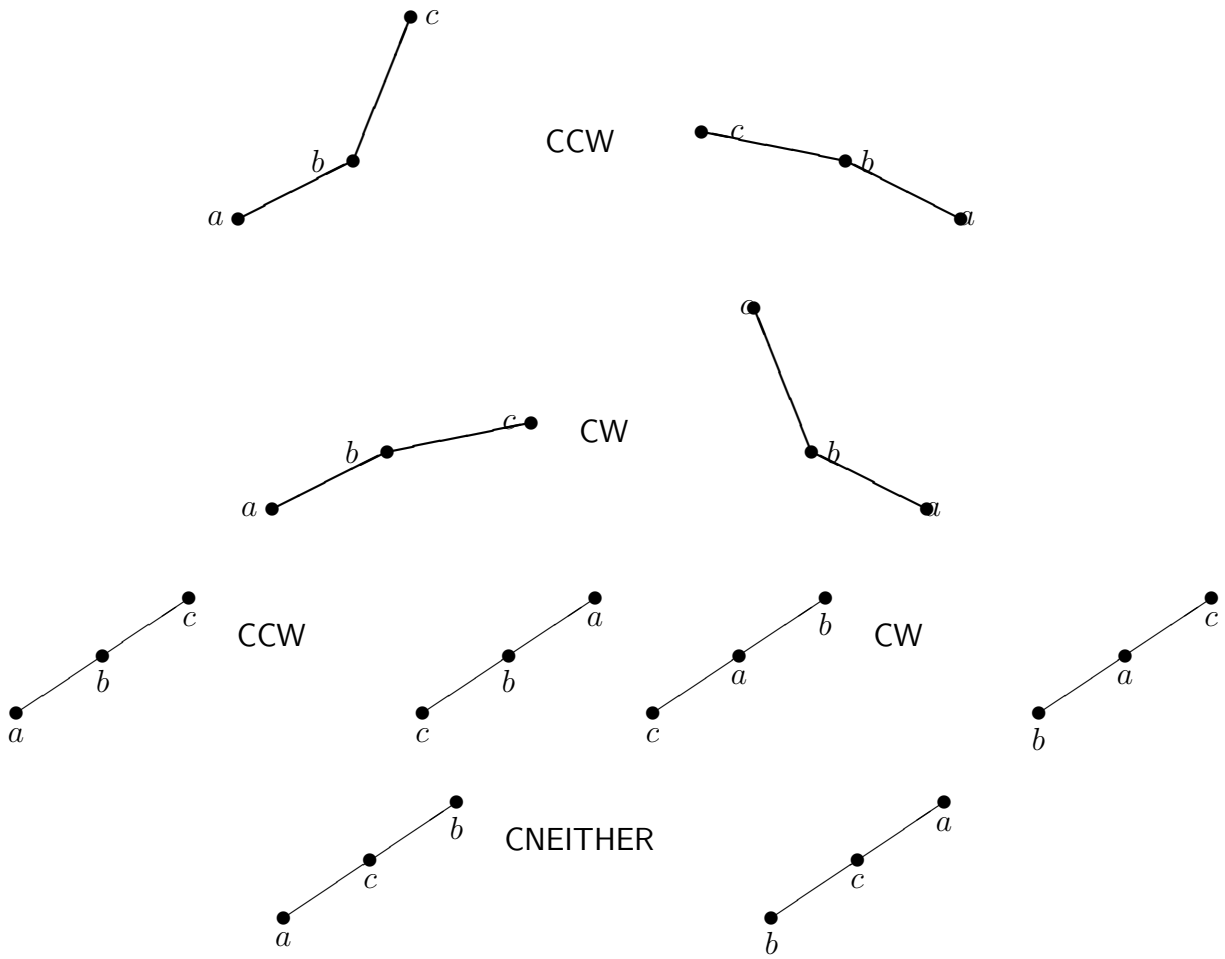
4.2 CCW

In this section, we'll craft a routine ccw. Given three points a, b and c, ccw determines whether one goes clockwise or counterclockwise in traveling from a to b to c. The routine

ccw will return the value CW or CCW as appropriate, and CNEITHER in certain degenerate cases. It would be nice if the algorithm could just return true or false; however, we will need to distinguish a third case.

This basic algorithm finds applications in almost all computational geometry. It easily leads to a test for segment-intersection. Additionally, it will give a way to order points by angle without actually computing an angle, and to determine if a point lies inside a polygon. Suffice it to say that this is an important building block.

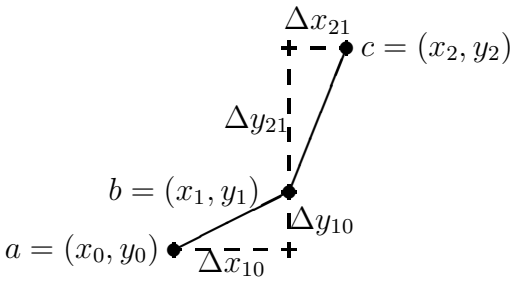
A caveat is in order before we start. The field of computational geometry is full of degenerate and special cases. The construction of ccw is no different. For what should the function return if the three points are collinear? Somewhat arbitrarily, we define what the behavior of ccw is:



This completely specifies what ccw should do. However, it leaves open the question of how it should do it. There are probably many different ways of explaining it; two will be explored here.

4.2.1 Slopes

We are examining the points $a = (x_0, y_0)$, $b = (x_1, y_1)$, $c = (x_2, y_2)$. Let $\Delta x_{10} = x_1 - x_0$ and $\Delta x_{21} = x_2 - x_1$. Define Δy_{10} and Δy_{21} similarly.



For now, assume that all the Δ's are nonnegative. The line abc goes in a counterclockwise direction when the slope of bc is greater than the slope of ab. Recall that the slope of a line segment is the rise over the run; $slope(ab) = \Delta y_{10}/\Delta x_{10}$, and $slope(bc) = \Delta y_{21}/\Delta x_{21}$. Thus, in order to check if abc is counterclockwise, we can test whether

$$\frac{\Delta y_{21}}{\Delta x_{21}} > \frac{\Delta y_{10}}{\Delta x_{10}}.$$

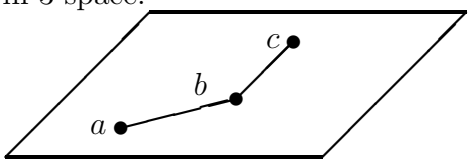
This will work, but it isn't the best test. If either of the Δx's is zero, things will go awry. Furthermore, division is often a more expensive operation than multiplication. So it makes more sense to test whether

$$\Delta y_{21} \cdot \Delta x_{10} > \Delta y_{10} \cdot \Delta x_{21}.$$

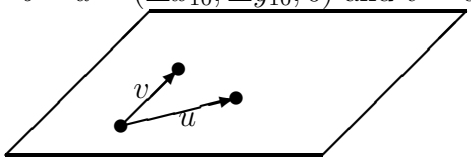
All of this has assumed that the Δ's are nonnegative. If some or all of the values are negative, the same formula holds. It isn't hard to convince one's self that multiplying by negative numbers changes the direction of the ">", so everything works out.

4.2.2 Cross Products

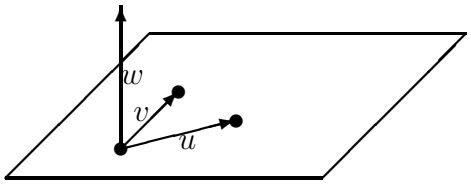
There is also a second way to analyze the question posed by ccw. As before, we work with three points $a = (x_0, y_0)$, $b = (x_1, y_1)$, $c = (x_2, y_2)$. This time, imagine that the points a, b and c lie on the plane $z = 0$ in 3-space.



Construct the vectors $u = b - a = (\Delta x_{10}, \Delta y_{10}, 0)$ and $v = c - b = (\Delta x_{21}, \Delta y_{21}, 0)$.



This isn't the place to review the entire derivation of cross-products and the right-hand rule. However, a smattering of physics or engineering reveals that $u \times v$ gives the area of the parallelogram defined by vectors u and v. Furthermore, the *sign* of $u \times v$ is positive if v is counterclockwise from v, and negative otherwise. (To see why this is called the right-hand rule, curl the fingers of your right hand in the direction from u to v, and see which way your thumb points.) So let $w = u \times v$.



The cross-product formula gives

$$w = (0, 0, \Delta x_{10} \cdot \Delta y_{21} - \Delta x_{21} \cdot \Delta y_{10})$$

What we are concerned with is the sign of the third coordinate of w. In other words, ccw should be true if $w_x = \Delta x_{10} \cdot \Delta y_{21} - \Delta x_{21} \cdot \Delta y_{10} > 0$, i.e.,

$$\Delta x_{10} \cdot \Delta y_{21} > \Delta x_{21} \cdot \Delta y_{10}.$$

This formula should seem familiar.

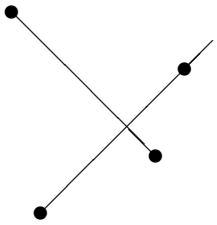
4.2.3 Pseudocode

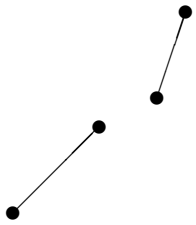
The code follows easily enough from these observations. The only tricky aspect is to correctly handle the degenerate cases. Without further comment, the pseudocode for ccw is presented here. It is instructive to check a few of the cases by hand to confirm that they behave properly.

```
CCW(a,b,c)
Δx1 ← bx–ax; Δx2 ← cx–bx
Δy1 ← by–ay; Δy2 ← cy–by
if ( Δy2 · Δx1 > Δy1 · Δx2 )                          /* Regular ccw */
  return(CCW)
else if ( Δy2 · Δx1 < Δy1 · Δx2 )                     /* Regular cw */
  return(CW)
else                                                  /* Collinear case */
  if ( Δx1 · Δx2 < 0 ) or ( Δy1 · Δy2 < 0 )           /* Did we change direction? */
                                                      /* If so, check if we went past a on second leg */
    if ( Δx1 · Δx1 + Δy1 · Δy1 ) ≥ ( Δx2 · Δx2 + Δy2 · Δy2 )
      return( CNEITHER )                              /* If not, then it's neither */
    else                                              /* If so, then it's clockwise */
      return(CW)
  else
                                                      /* If no change in direction */
    return(CCW)                                       /* It's counterclockwise */
```

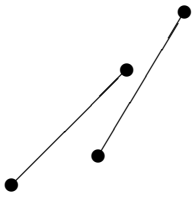
4.3 Segment Intersection

The whole counterclockwise analysis may have seemed trivial or useless. However, it is the cornerstone for the solution of many geometric problems. For example, given two line segments $l = (a, b)$ and $l' = (a', b')$, do they intersect? As a first approach, people often try to make sure that the x and y coordinates overlap. This is certainly a good first step:





Unfortunately, overlapping x and y coordinates aren't enough to insure that the two segments intersect:



As it turns out, if the segments' coordinates don't overlap, we can say with certainty that the segments don't intersect. The converse, however, is false; we need a better test.

Fortunately, ccw allows us to construct just such a test. Thinking about what it means for two segments intersect, it is clear that a' and b' must be on opposite sides of l. So traveling from a to b to a' must take us in a different direction than traveling from a to b to b'. In other words,

$$\mathsf{ccw}(a, b, a') \neq \mathsf{ccw}(a, b, b').$$

Similarly, a and b must be on opposite sides of (a', b'):

$$\mathsf{ccw}(a', b', a) \neq \mathsf{ccw}(a', b', b).$$

This is a sufficient test for segment intersection. We easily get the following algorithm:

```
INTERSECT(l = (a,b), l'=(a',b') )
if (ccw(a,b,a')≠ccw(a,b,b')) and (ccw(a',b',a)≠(a',b',b))
  return(TRUE)
else
  return(FALSE)
```

What happens if a segment has one or both endpoints on the line through the other segment? It is interesting to check the behavior of the algorithm under various degenerate cases.

4.4 Simple Closed Path

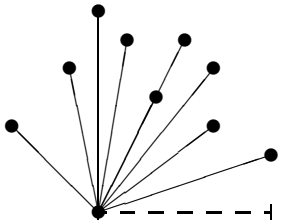
For the next problem, we consider a set S of points lying in the plane. A path is any sequence of points, and the line segments between successive points. A simple path is one which doesn't cross itself. A closed path is one in which the last point visited is also the first point. Thus, a simple closed path connecting all the points will produce a nice simple polygon with vertex set S. How can we find such a path?

There are two closely-related ways to accomplish this. Both start by picking an anchor or reference point, a. (In practice, it is usually convenient to pick the point with the lowest

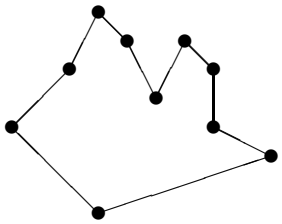
y-coordinate. If there are several points with minimal y-coordinate, pick the point among them with the smallest x-coordinate.) Then, we can sort the points in some manner, so that a traversal of the points goes counterclockwise around the polygon. There are a couple of choices about this.

4.4.1 Sort on θ

We could sort based on angle. Consider a line extending from a to another (imaginary) point with the same y-coordinate, but a little further over in the x direction. Call this point b. For any point p in the point-set, we can calculate the angle between the line segments ap and ab, denoted $\theta_a(p)$. (This calculation can be performed using a $\tan^{-1}$ computation. Another way will be presented below.)



Assign to each point its angle $\theta_a(p)$ with respect to a. Then, we can sort the points by saying that $p_1 \prec p_2$ when $\theta_a(p_1) < \theta_a(p_2)$. Suppose the points are sorted in this manner. It's not hard to see that visiting the points in this order will yield a simple path. If the path were to cross itself, it would necessarily go from the right, to the left, and again to the right of some line. This would mean that the angles of the points visited wouldn't be increasing.



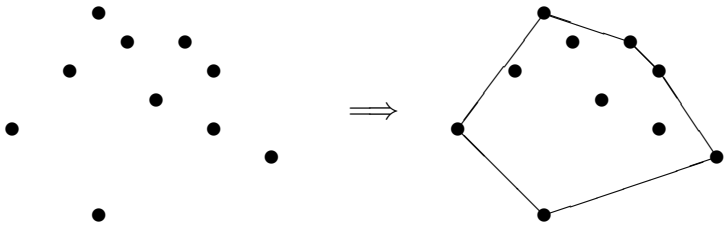
4.4.2 Sort on ccw

Actually, we don't really need to calculate any angles at all. We can define an ordering relation on the points by $p_1 \prec p_2$ if and only if $\mathsf{ccw}(a, p_1, p_2) = \mathsf{CCW}$. This captures precisely the information we need; it gives the vertices in clockwise order from a. For most of the examples of sorts given in class, the ordering relation has been a natural one, based on the key. Typically, keys were integers, and records were sorted by comparing their keys. However, many of the sorts we've looked at – in fact, all of them except for radix sort and bucket sort – only require that there exist a comparison function. Where a normal sort algorithm might compare the keys of two records, substitute in the appropriate call to ccw. (Of course, one could also write a completely general sorting routine which accepts a comparison function as one of its parameters. The standard qsort in the C library is an example of such a routine.)

4.5 Convex Hull

The next problem considered here is the computation of the convex hull of a set of points. A region is called convex if, for any two points lying anywhere in the region, the segment between those two points is contained within the region.

So, given a set of points $S = \{p_1, p_2, \cdots, p_N\}$, the convex hull of S is the minimal convex region containing those points. As it turns out, the convex hull of S will actually be a polygon whose vertices are a subset of S. In other words, there is some subset of the points, such that the polygon connecting them is the convex hull of S.



We'll take a look at two methods of solving the convex hull problem, package-wrapping and the Graham scan.

4.5.1 Package Wrapping

The first method is based on a fairly intuitive notion of how to wrap a package. Imagine we're trying to wrap a package in the plane, by wrapping some string or ribbon around it. Initially, we could tape one end of the string to some point which we know will be on the outside. (As in the algorithm for constructing a simple closed path, we could take the point with the lowest y-coordinate.) Then, we could pull the string taut (hold it off to the right), and wind it around until it hits another point in the point set. This point, too, will be on the convex hull. We can continue this way, winding the string around until it hits another point. This procedure will eventually produce the convex hull of the point set. It remains merely to formalize this algorithm.

We'll denote the points on the hull by h_1, h_1, ..., h_M, where M is the number of points which ultimately wind up on the hull. Start off by setting h_1 to be the point with the lowest y- coordinate. This point is guaranteed to be on the convex hull; for if this point were not on the hull, since it does have the lowest y- coordinate, the segment between it and any other point would not be contained in the hull. Again, if there are several points with the same lowest y-coordinate, we pick the one among them with the smallest x-coordinate. Subsequently, if we currently know the hull up to point h_i, we have to identify the next point as we continue winding the string around. This is equivalent to finding the point p_j with the minimum angle relative to h_i, provided that the angle between p_j and h_i is greater than the angle between h_i and h_{i-1}. This second condition is necessary, in order to not wind up "going backwards" in constructing the hull. Since we don't need the angle, merely the ordering relation it induces on the points, we can use ccw here.

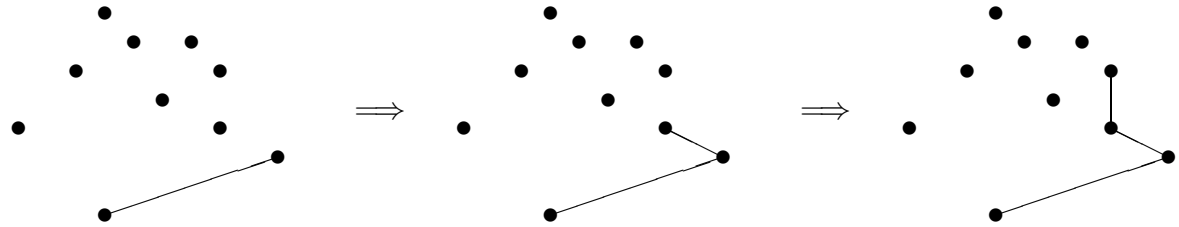
The time complexity of this algorithm isn't too hard to calculate. As before, let M be the number of points which wind up on the convex hull, and N be the initial size of the point-set. For each of the M points, we have to find the next point on the hull. This involves finding the point with some minimum property, out of a set of size N. So the algorithm takes time $\Theta(N)$ for each of the M points; package wrapping takes time $\Theta(MN)$. If M is just a small constant, then this is good asymptotic behavior. However, on average M grows with

N. For example, if the points are randomly distributed inside a square, then $M = \Theta(\log N)$ on average, and the algorithm takes average time $\Theta(N \log N)$. If the points are randomly distributed inside a circle, then $M = \Theta(N^{1/3})$ on average, and the algorithm takes average time $\Theta(N^{4/3})$. Of course, in the worst case all of the points are on the hull. When this happens, $M = N$, and the algorithm takes time $\Theta(N^2)$.

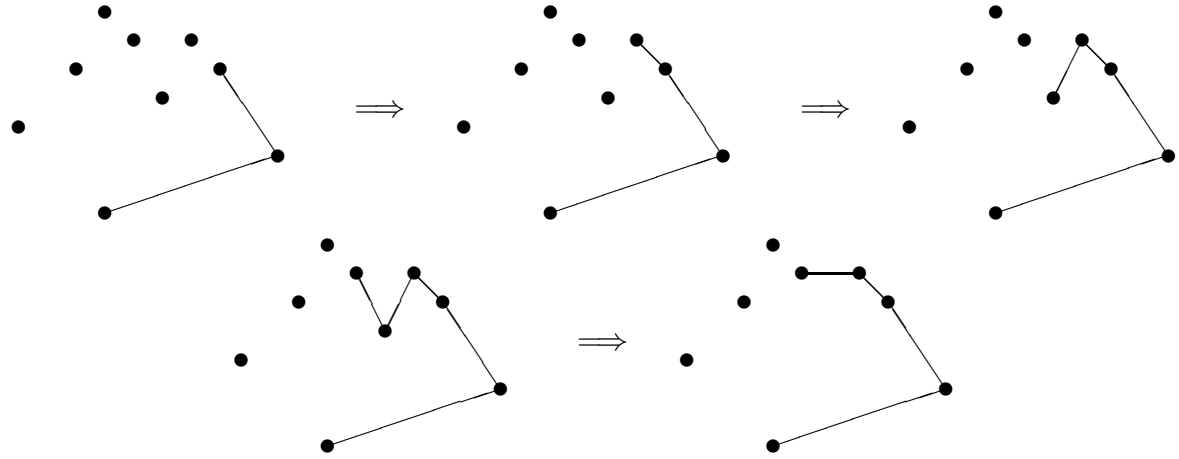
4.5.2 Graham Scan

The Graham scan takes a slightly different approach to the problem of computing the convex hull. We start out by computing a closed, simple polygon, using the method given above. Obviously, this polygon is contained in the convex hull. However, there are some interior points which shouldn't be on the hull. How can we identify these interior points?

Observe that, in following the convex hull of a point set, we always go in the same direction, say, counterclockwise. Now, suppose there is one point in the polygon which shouldn't be on the convex hull. In that case, we go left, then right, then left. In fact, the points which aren't on the convex hull are precisely the points which cause us to change direction. If, for example, we go left-right-left in traveling from p to q to r, then the line segment pr isn't contained in the polygon. In the point set above, we start off by scanning around the simple closed polygon which the points induce:



Shortly, however, we encounter a "bend" in the path around the polygon. Thus, we eliminate the point(s) which caused the bend, and continue.



This algorithm is guaranteed to produce the convex hull. There are a couple of options in implementing this algorithm. We could store the newly created hull in an array, adding elements as we go along. Alternatively, we can use a stack. Observe that, when we remove points from the hull, we only remove the most-recently-added point. This situation can be modeled with a modified stack. Assume that, in addition to the standard pop and push operations, we have functions peek_top and peek_next which return the top value on the stack and the one just below it, respectively.

```
GrahamScan(S)                                    /* Points are p[1]...p[N] */
find lowest point in S and exchange it with p[1]
sort S by angle with respect to p[1]
push( p[1] )                                     /* These two points are definitely on hull */
push( p[2] )
for i ← 3 to N
  while ccw( peek_next, peek_top, p[i] ) = CW
    pop
  push p[i]
```

The bulk of the processing time for this algorithm lies in the initial sorting stage. Since the sort is comparison-based, it takes time $O(N \log N)$. For the second stage, we merely scan through the points one at a time, performing a constant amount of work at each point. Thus, the second stage takes time $O(N)$, and the entire algorithm has time complexity $O(N \log N)$.

4.6 Range searching

For the next problem, we briefly retreat to one dimension for *range searching*. A range-search data structure supports several different operations.

create(key) Create a new range-search data structure which will store records, with comparisons done on the basis of the field key. It returns the structure rs.

insert(rs, element) Add a new record element to the data structure, rs, with key value element→key.

range_search(rs, min, max) Report all records element with min $\leq$ element $\rightarrow$ key $\leq$ max.

delete(rs, element) Remove the element element.

destroy(rs) Destroy the structure rs.

In other words, we need to have a dynamic structure which allows us to find all elements within a certain range. If you imagine the elements as lying on a line, according to their key values, a range_search query asks which elements lie within a given segment of the line. Now, we've already seen one dynamic data structure, the red-black tree, which efficiently supports insertion and deletion. As it turns out, we can use this same tree structure to support range_search operations.

Imagine the red-black tree as lying "above" the line on which the elements lie. Identifying which elements lie within some range min, max is easy. Using a binary search, we can find the largest element less than or equal to min. Similarly, we can also identify the smallest element greater than or equal to max. Then, we need merely perform an inorder-traversal of the tree, starting with the minimum element and ending with the greatest element.

The time complexity of insert and delete are simply $O(\log N)$. This is because these operations are simply the red-black operations we looked at earlier. It remains to figure out the time for a range-search query. It takes time $O(\log N)$ to find the minimum and maximum elements. If I values lie between min and max, then each of those nodes will be reported in time $O(1)$. Additionally, some number of nodes may be visited which aren't in the query range. The number of extra nodes is bounded by the height of the tree which is $O(\log N)$. Thus, a range_search takes time $O(\log N + I)$.

4.7 Segment intersection

We next turn to the problem of finding the intersections among a set of segments. In other words, given a set of segments $S = l_1, \ldots, l_N$, for which i and j do l_i and l_j intersect? This has applications in a number of areas in computational geometry. For example, in deciding how to display a 3-dimensional image, it is often necessary to remove the "hidden lines" – the lines which are obscured by polygons in front of them. This can be reduced to determining which lines in the scene intersect each other. Also, in the design of integrated circuits, it is important to know which wires intersect each other.

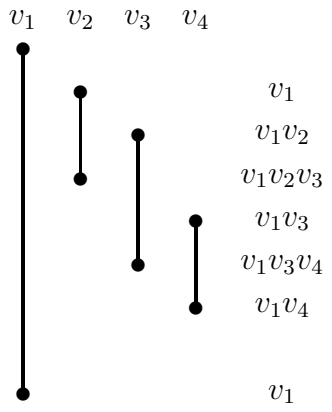
As an obvious solution, we could simply test all pairs of segments. However, there are $N(N+1)/2$ such pairs. Even though it takes time $O(1)$ to test a pair of segments for intersection, using the procedure above, it still would take time $\Theta(N^2)$ to find all pairs. The solution we'll be using instead takes time $O((N+I)\log N)$, where I is the actual number of intersections. This solution is preferable, as it is sensitive to the number of intersections.

The general technique is one which is fairly common in computational geometry. We'll introduce a scan line, or sweep line, which moves across the points. We perform computations on the line segments as the scan line passes them. This will become a little clearer below.

4.7.1 Status line

With range searching under control, we can attack the problem of segment intersection. Actually, we'll consider a slightly restricted problem. The line segments will be divided into two sets, H and V, consisting only of horizontal and vertical segments, respectively. For purposes of measuring the complexity of the algorithm, let $N = |H| + |V|$. Note that if the segments are divided equally among H and V, then $|H|$ and $|V|$ are both $\Theta(N)$. Also, for the moment we'll ignore intersections between two horizontal segments or two vertical segments. The algorithm presented here can be easily generalized to a broader context.

As mentioned above, we'll have a horizontal sweep-line L that moves from the bottom to the top of the plane considered. Of course, there is nothing sacred about sweeping from top to bottom. L could be in any orientation, and could sweep along the direction perpendicular to it. However, we lose nothing by restricting our attention to a horizontal sweep line moving from bottom to top. Along with the L, we will maintain a status data structure. The status variable will maintain all the vertical segments which L currently intersects.



A vertical segment v can be described by its bottom and top endpoints. As L sweeps upward, v is inserted into status when L sweeps through its bottom endpoint. As long as L

is between the bottom and top endpoints of v (considering only y-coordinates), L intersects v. Once L sweeps past the top endpoint of v, it will no longer intersect v. Since the status variable is supposed to maintain which vertical segments intersect L, v should be deleted from status when L goes past the top endpoint of v.

This describes how L changes as it encounters vertical line segments. What about horizontal line segments? As L encounters some horizontal segment h, h could intersect some of the vertical line segments. Certainly, if a vertical segment v doesn't intersect L at that time, then v can't intersect h. So h only has a chance of intersecting the segments stored in status. Additionally, considering only the x-coordinates of the points involved, the vertical segment must be between the left and right endpoints of the horizontal segment, h. This condition is both necessary and sufficient. Thus, suppose that the vertical segments in status are sorted by x-coordinate. Then when the sweep line L passes through some horizontal segment $h = h_1h_2$, we can find which segments intersect h with range_query($h_1 \rightarrow x, h_2 \rightarrow x$).

Thus, there are three kind of events which can happen as L sweeps up. We could encounter the bottom endpoint of a vertical segment v, in which case we should execute insert(rs, v). Or, we could encounter the top endpoint of v, and thus should perform delete(rs,v). Finally, we could encounter a horizontal segment. In that case, we should check for intersections, with range_search(rs, $h_1 \rightarrow$x, $h_2 \rightarrow$x).

It is unnecessary to perform any computation when nothing is happening, that is, when no vertical endpoints or horizontal segments are encountered. We can safely look for intersections only at the events listed above. Of course, these events have to happen at the right time. So in order to implement the algorithm, we should figure out the order of events, and set up the schedule ahead of time. We'll label the events EV_BOT, EV_TOP, EV_HOR.

First, we enter the vertical segments into an array that will be our event schedule. Each vertical segment is entered twice; once as an EV_BOT and once as an EV_TOP. The key of EV_BOT is the segment's minimum y-coordinate and the key of EV_TOP is the maximum y-coordinate. Next, all the horizontal segments are entered as EV_HORs. Their keys are their y-coordinates as well. The array representing the event schedule is then sorted.

We handle the events one at a time by scanning the event schedule. We initialize the status variable equal to the empy set, and then we process each event. We insert the segment into the status if it is an EV_BOT, delete the segment if it is an EV_TOP, and do a range search if the event is an EV_HOR. The range search takes the x-coordinates of the horizontal segment as parameter.

The above algorithm for detecting intersections among horizontal and vertical segments runs in time $O(N \log N + I)$, where I is the number of intersections. In the case of general segments, there is a more complex sweep line algorithm that runs in time $O(N \log N + I \log N)$.

4.8 Closest pair

For the last problem, we return to the domain of points. Given a set P of points, what is the closest pair of points in P? Recall that the distance between two points (x_1, y_1) and (x_2, y_2) is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. In this case, we wish to find the pair of points which minimizes this value. This problem creeps up in a variety of contexts. For example, we may be trying to find the pair of galaxies which is closest in a photograph of some sector of the sky, or trying to identify the two closest blots in a two-dimensional gel electrophoresis image.

As with most of these algorithms, there is an $\Theta(N^2)$-time brute-force algorithm, where $N = |P|$. The asymptotically faster solution we'll derive here has the general recursive form of many of the algorithms we've seen, namely, divide and conquer. Initially, sort the points on their x-coordinates. Divide the points into two sets, so that no point in one half is to the right of any point in the other half. In other words, draw a line down the middle, and label the two sets left and right. Points which lie on the dividing line can be put in either set. Now, either the closest pair of points lies in the left half, or it lies in the right half, or it consists of one point from each half. A recursive call to the procedure will find the closest pair in each half. It remains to check the last case.

Let Δ be the minimum of the closest-pair distances for the left and right halves. Suppose the closest pair in the whole set actually has one point in the left half and one in the right. Then each point in that pair must lie within Δ units of the midline. Thus, we can eliminate all points which do not lie in the strip within Δ of the midline.

This won't necessarily reduce the problem enough for a brute-force solution to make sense. It is possible that *all* the points will lie within Δ of the midline. In this case, a comparison of each left point with every right point in the strip will still take time $\Theta(N^2)$. Fortunately, there's a better way.

Now, suppose that the points outside the strip have been eliminated, and we are left with a set of points sorted by y-coordinate. As it turns out, we'll be able to compare each point to a constant number of points in order to find the closest pair. Consider any $\Delta \times \Delta$ square entirely contained in the left half. We know that the distance between any pair of points is at least Δ. Thus, we can only fit four points into a $\Delta \times \Delta$ square, by placing a point at each corner of the square.

Pick some point p in the right half. Are there any points in the left half closer than Δ to p? Examine the $\Delta \times \Delta$ square whose right edge is aligned with the dividing line, and whose lowest y-coordinate is p_y. Any point which is close to p and has a y-coordinate greater than that of p *must* be contained in that square. But there are only (at most) four such points! Similarly, there can only be four close points in the left half below p. Thus, if the points are sorted by y-coordinate, we can detect the closest cross-boundary pair by comparing each right point p to the eight left points with y-coordinate sufficiently close to p's. If the points are sorted by y-coordinate, then it will take time $\Theta(N)$ to perform this stage of the computation.

This is a moderately complex algorithm, and we should be careful with the analysis. What is $T(N)$, the time to find the closest pair of points among N points? There is an initial sort-on-x stage which takes time $O(N \log N)$. This doesn't affect the recurrence relation, as it need be done only once, at the beginning. We need to make recursive calls for the left and right halves, which contributes time $2T(N/2)$. Additionally, we need to sort the points by y-coordinate at each stage, which takes time $O(N \log N)$. Thus, a recurrence relation for this algorithm is:

$$\begin{aligned} T(N) &= 2T(N/2) + N \log N \\ T(1) &= 1 \end{aligned}$$

Using techniques we've developed earlier, we can discover that the closed form for this is $T(N) = O(N \log^2 N + N \log N) = O(N \log^2 N)$. This isn't bad, but we can do better. The algorithm takes time $O(N \log^2 N)$, rather than $O(N \log N)$, since we have to sort on the y-coordinates at each stage. As it turns out, we can work around this. Initially, we should

sort the points twice, once by x-coordinate, and once by y-coordinate. Before each recursive call, the y-sorted array is partitioned into sorted left and right subarrays. This takes time $\Theta(N)$. After the recursive calls, the sorted subarrays are merged back together. This also takes time $\Theta(N)$. Thus, the recurrence relation is now

$$\begin{aligned} T(N) &= 2T(N/2) + N \\ T(1) &= 1, \end{aligned}$$

and we have an algorithm for closest-pair which takes time $O(N \log N)$.

Chapter 5

Priority-First Searching for Weighted Graphs

5.1 Introduction

Recall that a graph G consists of a set V of vertices and of a set E of edges connecting pairs of vertices. A *weighted graph* is a graph with a numerical value, called *weight*, associated with each edge. For example, in a graph representing a road network, edge weights may represent distances or driving times. All of the algorithms for unweighted graphs work on weighted graphs too. Procedures for depth-first search, breadth-first search, and determining connectivity can be used on weighted graphs as well. However, they ignore the additional information of a weighted graph. We can ask additional questions about weighted graphs, e.g., what is the shortest path between two points? What is the minimum spanning tree of the graph? We now proceed to answer these questions. Note that we can safely restrict ourselves to a *connected* graph. The algorithms we'll study can easily be applied to each connected component of a disconnected graph.

We have already seen algorithms which use depth-first search and breadth-first search techniques. In this chapter, we'll be looking at a problems which admit *greedy* solutions. At each step, we can make the "best" move without looking ahead and considering all combinations. We will evaluate the "goodness" of each vertex by assigning it a priority, and simply picking the vertex with the best priority. The algorithms presented here are all *priority-first searches*.

5.2 Minimum Spanning Tree

5.2.1 An Introduction to Minimum Spanning Trees

A minimum spanning tree (MST) is essentially just what it sounds like. Let the weight of a subgraph be the sum of the weights of its edges. A *spanning subgraph* of a graph G is a connected subgraph S that contains all the vertices of G. Hence, any two vertices of G can be connected by a path in S. A *minimum* spanning subgraph of G has minimum weight among all spanning subgraphs of G. Note that it is not unique in general (consider for example a graph whose edges have all the same weight).

It is not hard to prove that a minimum spanning subgraph must be a tree. We can prove this through a simple proof by contradiction. By supposing that the minimum spanning

subgraph were not a tree, and showing that this contradicts the initial definition of a MST, we show that the MST *must* be a tree. There are two ways a collection of edges could fail to be a tree: there could be more than one connected component; and there could be a cycle. We consider the two cases separately. On one hand, suppose the MST were not connected. In that case, the set of edges would not connect all of the vertices to each other. On the other hand, suppose the MST contained a cycle. Then we could make a smaller spanning tree by deleting one of the edges of the cycle. The resulting set of edges would still connect all of the vertices, but it would have a smaller weight. We conclude that a MST of a graph is, in fact, a tree.

Before designing algorithms to find MSTs, it wouldn't hurt to figure out some simple properties of them. Consider any partition of the vertices of a graph into two subsets.

What, you may ask, is a partition? A reasonable question, indeed. A partition of a set divides that set into several subsets, such that each element belongs to *exactly* one subset. For example, one partition of the integers is {even integers} and {odd integers}, since every integer is even or odd, but not both. Dividing the integers into {positive integers} and {negative integers} is *not* a partition, since zero is neither positive nor negative.

So, consider any partition of the vertices of a graph into two subsets, G_1 and G_2. We claim that a MST contains a shortest (i.e., of minimum weight) edge e connecting a vertex in G_1 to a vertex in G_2. (If there are two shortest edges, we can pick either one of them since, in general, the MST of a graph need not be unique.) Again, we can best prove this claim by contradiction. Suppose the MST didn't contain e. This means that either there is no edge at all between G_1 and G_2, or there is a different edge f connecting G_1 and G_2 with $weight(f) > weight(e)$. (Perhaps this seems obvious or trivial, but in a proof by contradiction it is important to explicitly delineate each case.)

If there were no edge between G_1 and G_2, then the two components G_1 and G_2 would be unconnected. This violates the definition of a MST. So there must be some edge between the two subsets.

Suppose the MST *didn't* contain the shortest edge e. Consider the graph resulting from adding e to the spanning tree. Then e will complete a cycle involving some edges in G_1, some edges in G_2, and the edge f between them. So delete f from the spanning subgraph. The resulting graph is still a spanning tree, but it also has a smaller total weight than the original tree. Thus, the original tree *couldn't* have been a minimum spanning tree, since we were able to find another spanning tree with a smaller weight. Since we have ruled out the two cases where e isn't in the MST, we know it must be in it. We formalize this result in the following property:

Theorem 1 For any partition of the vertices of a graph into two subsets G_1 and G_2, a minimum spanning tree for that graph *must* contain the shortest edge connecting a vertex in G_1 to a vertex in G_2.

5.2.2 A First Algorithm

The above theorem is true for *any* partition of the vertices into two subsets. While it may be easier to visualize a partition into two subsets of roughly equal size, $G_1 = \{u\}$ and $G_2 = \{\text{all the other vertices}\}$ is a perfectly legitimate partition. Therefore, any MST T must contain the shortest edge between u and any other vertex.

Suppose that the shortest such edge connected u to a vertex v. At this point, there's a new natural partition of the graph. We have one set of vertices which contains only u and

v. The other set, of course, contains the rest of the vertices. So far, T is fairly small. It contains u, v, and the edge (u, v).

Well, we have again a partition of the vertices into two sets. We might as well try to apply our only theorem which deals with such a situation. The theorem tells us that the shortest edge between u or v and some other node must be in T. So, we dutifully add such an edge, say, (u, w), and now $G_1 = \{u,\ v,\ w\}$ and $G_2 = \{\text{everything else}\}$.

Hopefully, at this point, a pattern is emerging. We can formalize this in a working, if naive, algorithm to build a MST:

```
NAIVE_MST
let s be an arbitrary vertex
add s to T
mark s
while ∃ unmarked vertices in G
  find a minimum weight edge (u, v) such that u is marked and v is not
  add edge (u,v) to T
  mark vertex v
```

It's not hard to make a rough estimate of the worst-case time complexity of this algorithm. Let n be the number of edges of the graph, and m the number of edges. The outer loop is executed exactly $n - 1$ times. At each iteration $O(m)$ edges are examined to find a shortest edge. Therefore, the algorithm takes time $O(nm)$.

5.2.3 A Better Algorithm

How can we proceed to get a more efficient algorithm? By using a priority queue — the heap we earlier considered — we can design a much more efficient algorithm. In the naive algorithm presented above, on average half of the edges were checked with each iteration. However, there's really no need to do this.

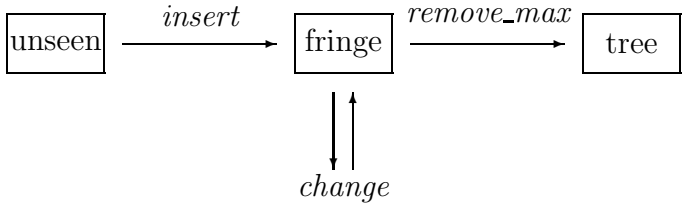
So far, we've been partitioning the vertices into those in the tree, and everything else. We will now further subdivide the vertices into those which are adjacent to vertices in the tree, and those which aren't. The *fringe* is the set of vertices which, although they don't yet lie in the tree, are adjacent to some vertex in the tree. The *unseen* vertices are the rest of the vertices.

Suppose that at some stage in the algorithm we have divided the vertices into *tree*, *fringe* and *unseen*. Which edge and vertex should be added to the tree next? It's not hard to see that the next vertex is the one in the fringe closest to some vertex in the tree. The shortest edge connecting it to the tree is added to the MST. Furthermore, the vertex is moved out of the fringe and into the tree. With the addition of this new vertex to the tree, it is quite possible that previously unseen vertices become now adjacent to the tree. Thus, some vertices from the unseen set may be moved to the fringe.

At each step, we are selecting the vertex from the fringe which is closest to the tree. Fortunately, we already have a dynamic data structure which allows us to select and remove the "best" out of a set of elements. Let the *priority* of a vertex be the weight of the shortest edge between that vertex and any vertex in the tree. By storing the priority of each vertex in the fringe in a priority queue, we gain efficient access to the closest vertex. The priority queue also allows easy insertion of vertices, in this case from the unseen set to the fringe.

The number of vertices in the fringe is $O(n)$. If the priority queue is based on a heap representation, then both the *insert* and *remove_max* operations take time $O(\log n)$.

Note that the priority of a vertex in the fringe can change. The priority of some vertex u will change if a new vertex v is added to the tree, and $weight(u, v) < priority(u)$. Our priority queue now needs an additional operation, *change*, which changes the priority of an entry. It's not hard to see that changing the priority of an element of a heap takes time $O(\log n)$. We see that vertices are moved from *unseen* to *fringe* to *tree* in the following way:



All the operations involved take time $O(\log n)$. Each of the vertices is inserted and deleted once from the fringe, giving a time complexity of $O(n \log n)$. Additionally, each time a vertex u is added to the tree, we have to check all its $degree(u)$ neighbors: if a neighbor is unseen, we insert it into the fringe; if a neighbor is in the fringe, we may have to update its priority. This takes time $O(\sum_u degree(u) \log n = O(m \log n)$. Hence, the entire algorithm takes time $O((n + m) \log n)$, which is certainly better than the naive algorithm given above.

Observe that we assigned each vertex a priority, and then visited the vertices in order of their priorities. The key step was defining the priority of a vertex. By assigning a priority which measures some other property of the vertices, we can solve different graph problems. We now consider another such problem.

5.3 Shortest Path

Given some vertex s of a weighted graph G, we wish to find the shortest paths between s and every other vertex in G. As it turns out, the algorithm for finding the shortest paths to s is practically identical to the one for finding a minimum spanning tree. We will wind up building a tree rooted at s. Indeed, we can show that there is a set of shortest paths whose edges are in a tree with an argument similar to the one proving that a minimum spanning subgraph must be a tree.

We shall restrict our attention to the case where all edge weights are nonnegative. Indeed, negative weight edges may cause problems with the definition of shortest path. Namely, if G had a cycle of total negative weight, we could traverse it infinitely many times to get an infinitely "small" path length. Hence, the restriction.

Obviously, the shortest edge between s and any other vertex must lie in the tree. So, start by adding that edge and vertex to the tree. We should now add the next closest vertex to s. Which vertex is closest? Suppose that the vertex just added is v. Then the distance from s of some vertex w adjacent to the the current tree is either $weight(s, w)$ or $weight(s, v) + weight(v, w)$, whichever is less (for convenience we consider nonexisting edges as having infinite weight). In general, the distance (and, thus, priority) of a vertex w is

$$dist(w) = \min_{v \in tree} (weight(v, w) + dist(v))$$

Using this notion of priority, we get a correct algorithm to determine the shortest paths to s which is quite similar to the algorithm for building a MST.

Again, we start off with some set of fringe vertices, the ones directly connected to s. The priority of a vertex is its distance to s. Once the shortest path between a vertex and s is found, that vertex is added to the tree. The addition of a vertex to the tree may necessitate changes in the distances (priorities) of vertices in the fringe, as well as the insertion of previously unseen vertices into the fringe. The analysis of this algorithm is identical to that for MSTs. The result is an algorithm which takes time $O((n + m) \log n)$.

Chapter 6

Relations

Relations are a fundamental mathematical concept. In this chapter we show that equivalence relations and order relations can be associated with graph concepts.

We start by recalling the concept of relation. Let S be a set. The *cartesian product* $S \times S$ is the set of all ordered pairs of elements of S. Hence, if S has n elements, $S \times S$ has n^2 elements. A *relation* $\sim$ over S is a subset of $S \times S$, i.e., it is a subset of ordered pairs of elements of S. We write $a \sim b$ if the pair (a, b) is in the relation.

Relations are classified according to their properties. Below are some of the most widely-studied properties of relations. Assume that a, b and c are in the set S over which relation $\sim$ is defined.

reflexive $a \sim a$.

transitive If $a \sim b$, and $b \sim c$, then $a \sim c$.

symmetric If $a \sim b$, then $b \sim a$.

antisymmetric If $a \sim b$, and $b \sim a$, then $a = b$.

(In order for a relation to be labeled reflexive, transitive, etc., the condition must be true for all a, or all pairs a and b, etc. So for $\sim$ to be reflexive, for example, it is not sufficient that there is a particular a_0 such that $a_0 \sim a_0$. We must have $a \sim a$ for *all* a.)

6.1 Equivalence Relations

A relation which is reflexive, transitive and symmetric is called an equivalence relation. One of the most intuitive equivalence relations is, in fact, equality. Consider the relation of equality among integers. Clearly, $x = x$ for all integers x. Also, if $x = y$ then $x = y$ for any pair of integers x and y. Finally, we know that if $x = y$, and $y = z$, then $x = z$. (This isn't an actual proof; in order to *prove* that equality is an equivalence relation, we would need a more formal notion of what equality means, and what integers are, for that matter.)

Not all equivalence relations are quite so obvious. For example, we can write $x \cong_b y$ if the rightmost digits of x and y (written in base b) are the same. It turns out that this defines an equivalence relation. Also, the domain need not be the integers; we can, for instance, talk about a relation among people. Say that a is related to b if a and b have the same maternal grandfather. This defines an equivalence relation on people.

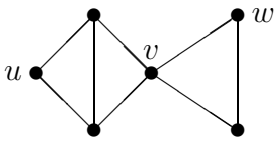
An equivalence relation partitions the set into equivalence classes. The equivalence class of a is the set of all b such that $b \sim a$. Since $\sim$ is symmetric, reflexive and transitive,

this is a true partition of the set. The equivalence relation $\cong_2$ divides the integers into two equivalence classes, the even integers and the odd integers.

Perhaps not surprisingly, equivalence relations find applications in graphs. One example is connectivity. We define the relation $\cong$ on the vertices of a graph by $u \cong v$ if and only if there is a path between u and v in the graph. (More formally, $u \cong v$ if there is a sequence of vertices w_0, w_1, ..., w_k such that $u = w_0$, $v = w_k$, and (w_i, w_{i+1}) is an edge for $i < k$.)

We can informally confirm that $\cong$ is actually an equivalence relation. First, $\cong$ is reflexive. There is a path from every vertex to itself; $v \cong v$. Second, $\cong$ is transitive: if there is a path from u to v, and a path from v to w, then we can get from u to w by simply following the paths from u to v and v to w. In other words, if $u \cong v$, and $v \cong w$, then $u \cong w$. Finally, $\cong$ is symmetric: if there's a path from u to v, then by following that path backwards, we can get from v to u. Thus, $u \cong v$ implies $v \cong u$. Since $\cong$ is reflexive, transitive and symmetric, it is an equivalence relation. What are the equivalence classes induced by this relation? A vertex v is related to u only if there is a path from u to v. So the equivalence class of u is the set of all vertices somehow connected to u. Each equivalence class corresponds precisely to a connected component of the graph.

We might try to generalize this, by looking at higher connectivities. Two vertices u and v are biconnected if there are two vertex-disjoint paths between u and v (not including the endpoints). In other words, the removal of any vertex or edge in the graph is not enough to break all paths between u and v. Is this an equivalence relation? This relation obviously is reflexive and symmetric. However, it isn't transitive. In the example below, u and v are biconnected, and v and w are biconnected, but u and w are *not* biconnected. The removal of vertex v will disconnect u from w. (In this case, we say that v is an articulation point.)



Since this relation on the vertices is not transitive, it is not an equivalence relation. As it turns out, there is a similar relation which *is* an equivalence relation. We have defined biconnectivity for vertices. We can analogously define biconnectivity for edges. Two edges e_1 and e_2 are biconnected if there is a simple cycle which contains them both. Biconnectivity is an equivalence relation for *edges*, but not vertices.

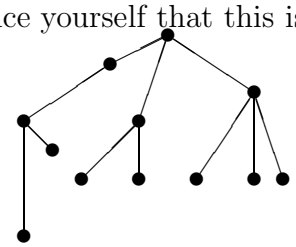
6.2 Partial-Order Relations

Many relations which aren't equivalence relations are, nonetheless, quite useful. Another class of relations are *partial orders* (some authors call them just *orders*). A relation $\preceq$ is a partial order if it satisfies the reflexive, transitive and *anti*symmetric properties.

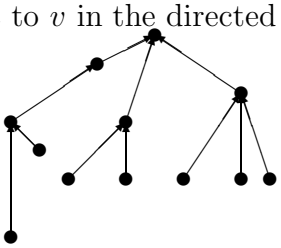
The most obvious example of a partial order is $\leq$, defined on the integers in the usual way. (In fact, this is the model on which the definition of partial order was based.) Naturally, the domain of the relation need not be the integers. One useful partial order relation can be defined on sets which are subsets of some larger set. For example, if $S = \{1, 2, 3\}$, then we define the set of subsets of S, call it P, to be $\{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. Then $a \subseteq b$ is a partial order. For example, $\{1\} \subseteq \{1, 3\}$, while $\{1, 3\} \not\subseteq \{1\}$. There are also elements for which $a \not\subseteq b$ and $b \not\subseteq a$. In this case, $\{1, 2\} \not\subseteq \{2, 3\}$, and $\{1, 2\} \not\subseteq \{2, 3\}$. Such

elements are called incomparable.

This, too, has applications to graphs. Consider the vertices in a rooted tree. We can say that, for two vertices u and v, $u \preceq v$ if and only if v is on the path from u to the root of the tree. It shouldn't be hard to convince yourself that this is a partial order.

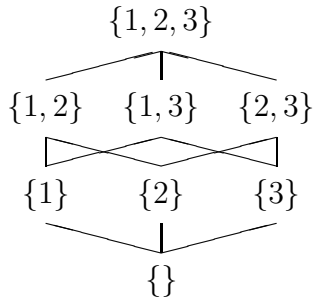


We can also view the situation slightly differently. Consider the same tree as a directed graph, with edges directed from a vertex to its parent. Then we can define the same relation, $\preceq$, as $u \preceq v$ if there is a path from u to v in the directed graph.



We can make a similar definition on any DAG, that is, on any directed, acyclic graph. For a pair of vertices in a DAG, say that $u \preceq v$ if there is a directed path from u to v. Under this definition, $\preceq$ is, in fact, a partial order. We can verify the properties. First, there is a trivial path from every vertex to itself. Thus, $u \preceq u$ for all vertices u. Second, if there is a directed path from u to v, and there is a directed path from v to w, then by simply concatenating the paths we get a path from u to w. Thus, $\preceq$ is transitive. Finally, we claim that $\preceq$ is antisymmetric. For suppose that $u \preceq v$, and $v \preceq$. Then there is a directed cycle in the graph, obtained by going from u to v and back again. But this contradicts the fact that the graph is *acyclic*. Thus, this can only happen when $u = v$. Thus, $\preceq$ is antisymmetric, and hence a partial order. Note that the graph had to be acyclic in order for $\preceq$ to be a partial order.

Thus, we have seen that any DAG induces a partial order on its vertices. We can also proceed the other way round. Given any partial order, we can construct a directed acyclic graph. There is a vertex in the DAG for every element in the domain of the partial order. Add edge (u, v) whenever $u \preceq v$ and the graph constructed so far has not already a directed path from u to v. Using logic similar to that used above, one can show that the resulting graph is acyclic. For example, given the relation defined earlier on P, we can construct a DAG.



A partial order can be extended, by insuring that either $a \preceq b$, $b \preceq a$, or $a = b$ for all a and b in the set. In other words, we can extend the relation so that there are no incomparable elements. Somewhat arbitrarily, we define the relation for pairs of incomparable elements so that *all* elements are comparable. This is sometimes called a linear ordering, as the resulting DAG is a straight line.

Such extensions have real-life applications. Consider the system of prerequisite courses at a university. If course c_i must be taken before c_j, write $c_i \preceq c_j$. Assuming that the courses are set up in a reasonable fashion – that no two courses are mutual prerequisites! – this will define a partial order. What is a proper order for taking the courses? The answer is that *any* linear ordering of the elements will satisfy the prerequisite system. There is an analogous problem for DAGs. Given a directed acyclic graph, we wish to enumerate the vertices in an order such that, if there is an edge from u to v, then u is output before v. Such an enumeration is called a topological sorting of the DAG, and can be accomplished with a modified depth first search.

Chapter 7

Parallel Algorithms

7.1 Introduction

Computer scientists – and computer users, for that matter – are always searching for faster solutions to problems. Often, these improvements are made in software. Most of this course is dedicated to figuring out how to accomplish tasks quickly, using a variety of algorithmic techniques. The other obvious place for improvement is in the hardware. To date, the power of computers has grown exponentially. Unfortunately, physics places an upper bound on the speed of any one computer. As our technologies improve, the speed of electrons becomes a limiting factor. No matter how fast the logic gates are, it still takes a finite (and significant) time for electrons to travel between these elements. Researchers have been looking for ways around this bound on computation speed.

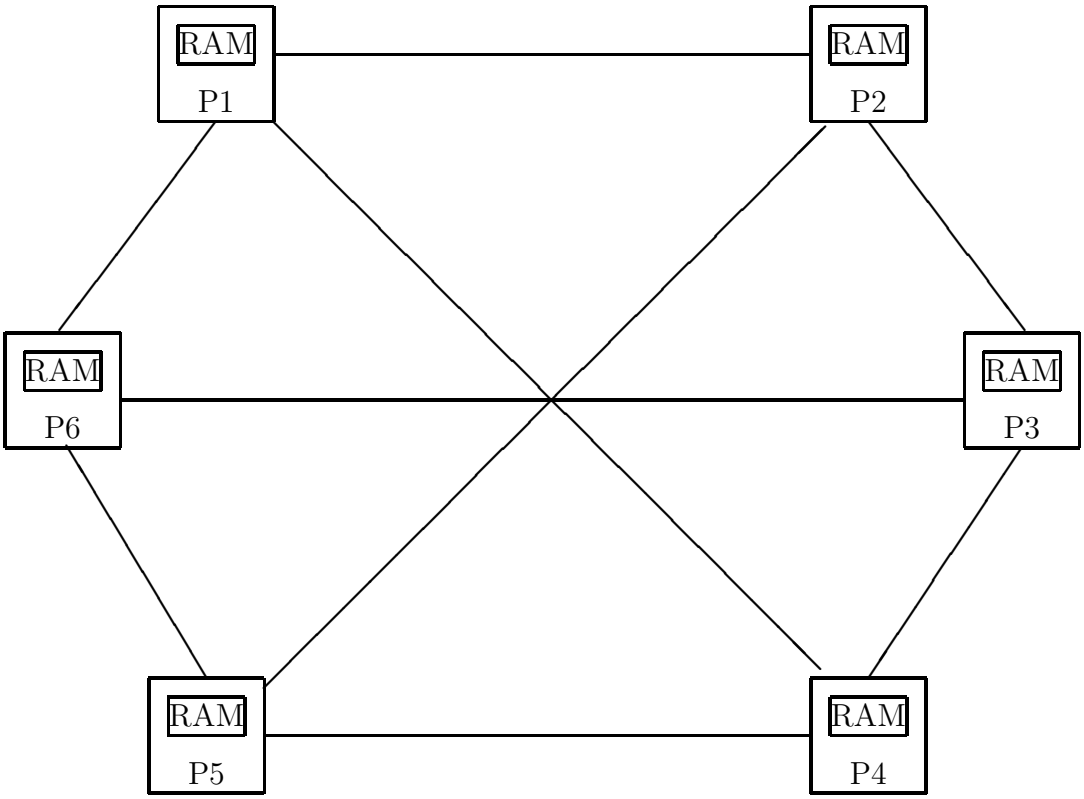
Thus, we see a growing interest in *parallel computation.* Instead of asking one processor to work on a particular problem, we have several processors work at the same time to solve it. Intuitively, it's not hard to see that this approach promises faster computation. For example, it seems that the computing power in the Sun lab is many times larger than the computing power of any single SPARC station. Making these computers work in concert, however, is not a trivial endeavor. We have to decide how to connect the machines, how they will communicate with each other, and how an algorithm can take advantage of the entire network.

Until now, it seems that we have been able to ignore hardware considerations in our analysis of algorithms. Actually, we have been making a set of assumptions; we just haven't explicitly stated them. We have been discussing algorithms based on the *sequential* model of computation. In this model, a computer (or *random access machine*) has a single processor which sequentially executes the instructions of a program. Additionally, this processor has a memory in which data are stored; this memory is accessed one element at a time.

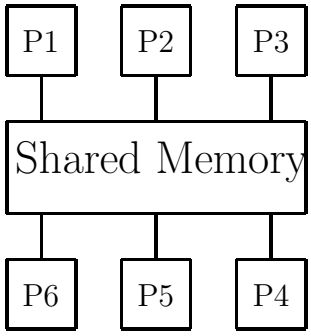
So much for the sequential model. There are several different models for parallel computation. Each corresponds to a different way in which parallel machines are actually built. In order to design algorithms for a machine, we have to know something about how it works. We now consider some of the more common models.

First, how do the processors communicate with each other? In the *network* model, the processors work reasonably independently and communicate only along specific communication lines. Typically, the time for one message exchange is much greater than the time to execute an instruction. An extreme example of this is the distributed factoring project. Computers all over the world have worked together to factor very large numbers. Commu-

nication among processors is done through electronic mail, which is vastly slower than a computation done by one computer. This network recently made the news by factoring the ninth Fermat number, $2^{512} - 1$.



Sometimes the processors are more tightly coupled. An example of this is a machine where many processors share the same memory bank. This configuration is *a parallel random access machine*, or *PRAM*. Communication between two processors is accomplished by having the "sending" processor post the message at a certain memory location, and having the "receiver" then read that location. In this way, the common memory acts as a kind of bulletin board for message passing. We will be using the PRAM model in most of our examples.



Our consideration of hardware doesn't end here. Once we have specified a PRAM machine, we have to describe how the memory is accessed. Depending on the implementation, it is possible that two processors can read the same memory location at the same time. Alternatively, no two processors can simultaneously read the same memory location. These two designs are called *concurrent read* and *exclusive read*, respectively, or simply *CR* and *ER*.

A similar choice between concurrent and exclusive operation exists for write operations. Concurrent write, however, is a bit problematic. What should happen if two processors try to write to the same memory location at the same time? There is no single, canonical answer. Different conflict-resolution schemes are useful for different applications. Examples include:

- *Arbitrary*: an arbitrary value is selected from among the conflicting processors.
- *Common*: if all values are the same, then that value is written; otherwise, there is no change.
- *Priority:* each processor is assigned a priority. The processor with the highest priority gets its value written in the memory location.

7.2 Analysis

When analyzing sequential algorithms, efficiency is usually pretty easy to define. Typically, we are primarily concerned with execution speed $T(n)$; the number of instructions executed is a good measure of performance. When we say that an algorithm runs in time $\Theta(n \log n)$, we are saying something about its speed.

A good analysis of parallel algorithms should incorporate a few pieces of information. Again, we will be interested in execution time. Thus, we will need an expression involving n, the size of the problem. Additionally, we want to relate the speed to p, the number of processors used. We will let $T_p(n)$ denote the time necessary to process n elements with p processors.

Perhaps a (trivial) example will help clarify matters. Let's design an algorithm to fill n memory locations with the value 0. Obviously, a sequential algorithm would run in linear time, so $T_1(n) = \Theta(n)$.

Suppose we directly implemented this algorithm on a parallel machine. Such a naive implementation would not take advantage of parallelism. All the processors (save one) would be ignored. Therefore, the procedure would still run in $\Theta(n)$: the time is independent of the number of processors used.

Next, consider a more sophisticated, truly parallel version of the algorithm. Each of the p processors is assigned a group of consecutive n/p array elements and sets them equal to zero. Now, the time complexity $T_p(n)$ of the algorithm is $\Theta(n/p)$. Observe that we are assuming $n > p$. If $n < p$, then the first n processors zero the array and the others have nothing to do.

The second algorithm is a superior solution, as it takes better advantage of the available resources. This concept is formalized in the definition of speedup:

$$speedup = \frac{\text{worst-case time for fastest sequential algorithm for problem}}{\text{worst-case time for parallel algorithm}}$$

In this example, the speedup is $\frac{\Theta(n)}{\Theta(n/p)} = \Theta(p)$. This corresponds to our intuition; adding p processors speeds up the performance by a factor of $\Theta(p)$.

Often, the number of processors required by an algorithm is determined by the input size. We express this by writing $p = p(n)$. The *cost* (or *work*) of the algorithm tells us how many instructions were actually executed by all the processors. Thus, we have

$$cost = p(n) \cdot T_p(n)$$

The *efficiency* measures the cost of the parallel algorithm as opposed to the best sequential algorithm for the problem:

$$efficiency = \frac{T_1(n)}{cost}$$

where $T_1(n)$ is the time complexity of the best known sequential algorithm. Using these concepts we can formalize notions of optimality. We say that an algorithm is *optimal* if $efficiency = \Theta(1)$ (i.e., $cost = \Theta(T_1(n))$), and that it is *efficient* if $efficiency = \Omega(1/\log n)$ (i.e., $cost = O(T_1(n) \cdot \log n)$).

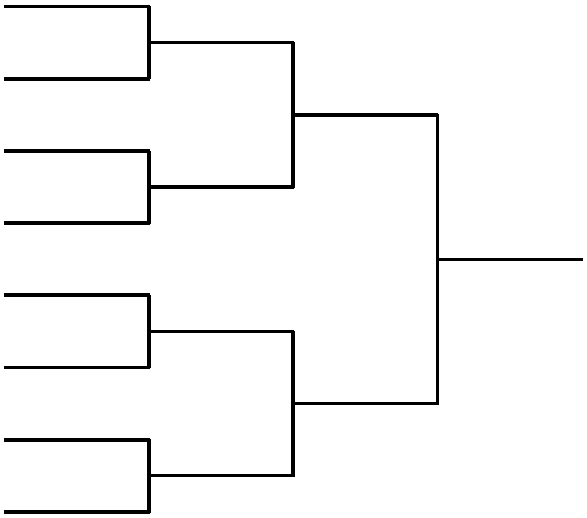
All of these definitions, based on the number of instructions executed, are appropriate for PRAMs. However, in different models of parallel computation the number of instructions is *not* the rate-determining part of an algorithm. In many distributed networks, the number of exchanges between processors far outweighs the effects of local computations. An analysis appropriate for such machines examines the number of inter-processor communications. Additionally, in some networks not all connections are equal; processors can be close (there is a direct connection between them) or far apart on the network. In such cases, it may be important to consider the net routing distance that messages traverse.

7.3 Maximum Value

The first problem we'll consider is finding the maximum element of some array a of n elements. The sequential algorithm isn't very surprising:

```
max ← 1
for i ← 2 to n
  if a[i] > a[max]
    max ← i
```

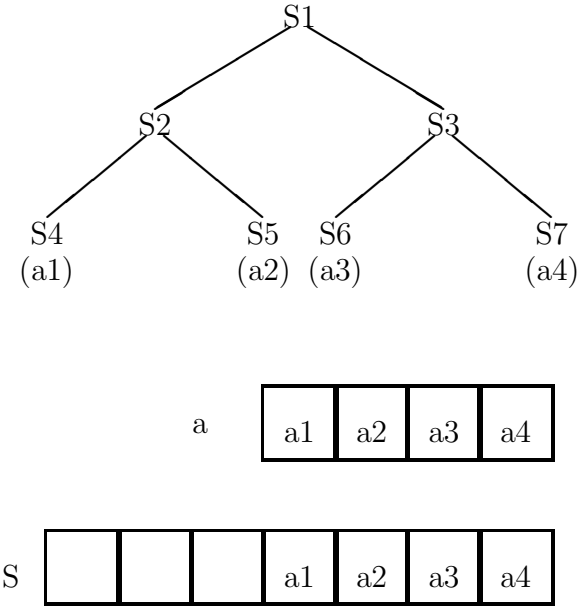
It runs in time $\Theta(N)$. How can we do this in parallel? One answer can be found in any bowling alley – the elimination tournament. Bowling alleys, tennis clubs, and many other groups have to solve just such a problem, finding the best player. Players are paired up, and the winner of each game advances to the next round of competition. The tournament eventually reaches the quarterfinals and semifinals, and in the final round the winner is determined. Typically, the schedule for such a tournament looks something like:



Notice that competitions on the same level can happen concurrently. This suggests that there is a certain parallelism inherent in the process.

In fact, there is. We'll substitute elements of the array for players in the tournament, and each processor arbitrates one "contest" at a time. It may not be obvious how to assign comparisons to processors. We turn to earlier techniques for representing heaps – and arbitrary binary trees, in fact – with arrays. Given an array, we say that the children of element i are the elements $2i$ and $2i + 1$. Similarly, the parent of element i is element i div 2.

Suppose we start with an array of elements a_1, ..., a_n. We load them into an array s in positions s_n, ..., s_{2n-1}. At this point, the array and the corresponding tree look like this:



We want the value stored at an internal node to be the greater of its two children. The height of the heap is $\log n$; it will take $\Theta(\log n)$ time before the final computation is performed. At the end, the value stored in s_1 is the maximum of all of the values. The program to implement this is pretty clean:

```
copy a[1..n] into s[n..2n-1]
for iterations ← 1 to log n
  for each processor P_i
    s[i] ← max(s[2i], s[2i+1])
maximum ← s[1]
```

A few comments are appropriate here.

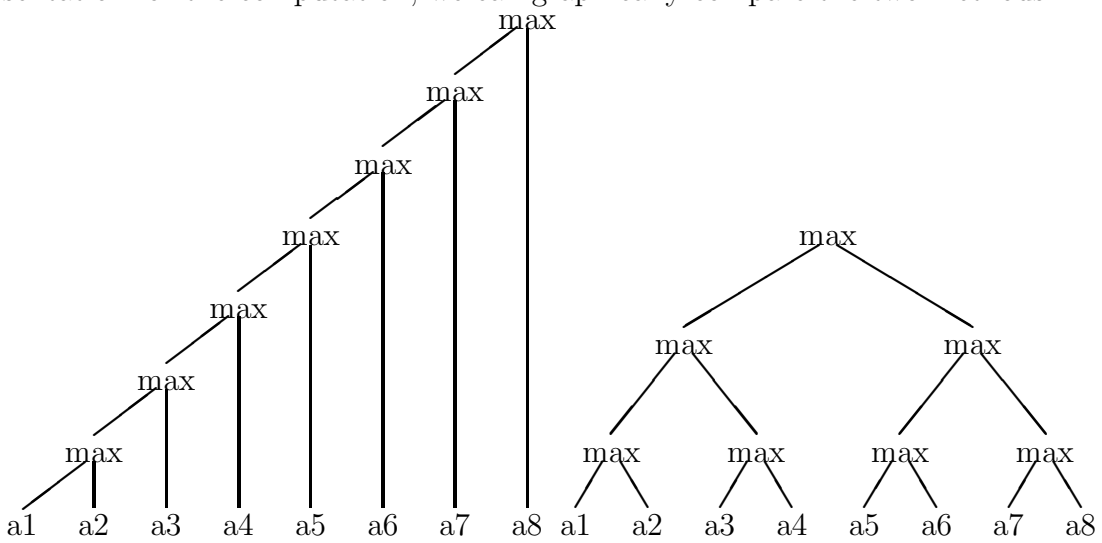
First, while the algorithm runs in $\Theta(\log n)$ time, it requires $p(n) = n$ processors. Thus, the total work (or cost) of the algorithm is $n \cdot \Theta(\log n) = \Theta(n \log n)$. Using a sequential algorithm, it takes $\Theta(n)$ time to determine the maximum element. By our earlier definitions, this algorithm is efficient but not optimal. Furthermore, the efficiency is $\Theta(n)/\Theta(n \log n) = \Theta(1/\log n)$. Note that, while the parallel algorithm runs faster than the serial one, it is considered less efficient.

Second, this routine runs on almost any PRAM. No two processors attempt to read the same memory location at the same time. Furthermore, only one processor writes to a given memory location. So we can get by with an EREW machine. (Certainly any algorithm which works on an EREW machine will also work on CREW and CRCW machines.)

Finally, it is instructive to compare the logic of this algorithm with the standard sequential method. The parallel method takes the maxima of pairs of elements, then the maxima of pairs of maxima, etc. This yields the maximum element, since *max* is an associative operator.

(In other words, $\max(a, b, c) = \max(\max(a, b), c) = \max(a, \max(b, c))$. An associative operator is a binary operation $\circ$ such that $a \circ (b \circ c) = (a \circ b) \circ c$. Alternatively, it can be regarded as a function f such that $f(f(a, b), c) = f(a, f(b, c))$.

Contrast this with the sequential algorithm, which determines the maximum of the first two elements, then the maximum of that value and the next element, etc. By using a tree representation for the computation, we can graphically compare the two methods:



In effect, we were able to achieve a speedup by balancing the computation tree. We can use this technique on *any* associative operator, not just max. Examples of associative operators include addition, multiplication, and function composition. We'll return to these in a bit. Before that, we can take a look at a new way to determine the maximum value in better than $\Theta(\log n)$ time, using the power of concurrent write.

This algorithm achieves constant running time on a CRCW machine, at the expense of using many processors. Suppose we have n^2 processors labeled $P_{i,j}$, where i and j are between 1 and n. We also use an auxiliary array boolean $b_1, \ldots, b_n$. Initially, each of the b_i is set to *true*. Subsequently, b_i = *true* means that "element i could be the maximum element." Clearly, b_i should be false if there is some j such that $a_j > a_i$. So, we use each processor to perform one of these comparisons:

```
for each processor P_{i,j}
  if a[j] > a[i]
    b[i] ← false
```

At the end, the only true elements of b are those corresponding to the maximum elements of a. It is now a simple matter to identify the maximum value:

```
for each processor P_{i,1}
  if b[i]
    maximum ← b[i]
```

This algorithm consists of only two steps, each of which takes constant time. Therefore, the algorithm takes time $\Theta(1)$. This is certainly a dramatic increase over the sequential running time of $\Theta(n)$. However, this speed comes with an increase in cost. Formally, the cost of the algorithm is $n^2 \cdot \Theta(1) = \Theta(n^2)$, which is greater than both the previous parallel algorithm and the standard sequential one.

This algorithm will work on all CRCW machines. Even if many processors try to write to some memory location at the same time, they will all be writing the same value. This makes conflict resolution easy. Additionally, we could have easily cut the number of processors in half. Suppose we know that all elements are distinct. Then the following algorithm will produce the correct result:

```
for each processor P_{i,j} 1 ≤ i < j ≤ n
  if a[j] > a[i]
    b[i] ← false
  else if a[j] < a[i]
    b[j] ← false
```

The number of processors necessary is now $(n^2 - n)/2$. (Drawing a picture of the matrix of processors shows that this is so.) However, $p(n)$ is still $\Theta(n^2)$, so the cost of the algorithm is still $\Theta(n^2)$.

7.4 Sum

Now, suppose we wish to find the sum of n numbers. The solution to this problem is identical to our first algorithm for finding the maximum element of an array. We simply replace the $\max(a, b)$ operation with $\text{sum}(a, b)$ or, more commonly, $a + b$. The algorithm looks like this:

```
copy a[1..n] into s[n..2n-1]
for iteration ← 1 to log n
  for each processor P_i
    s[i] ← s[2i] + s[2i+1]
sum ← s[1]
```

When the algorithm finishes, the sum of all the elements is stored in s_1. Note that, until s_{2i} and s_{2i+1} are calculated correctly, the calculation at s_i will quietly generate garbage. This doesn't affect the outcome; it merely means that the processors high up on the tree, i.e., the ones with low indices, will be doing more work than necessary. We could put in a check, if we wanted:

```
for each processor P_i
  if ( iteration ≤ log (n/i) < iteration + 1 )
    s[i] ← s[2i] + s[2i+1]
sum ← s[1]
```

This still leaves nodes with "dead" time. In fact, each processor executes the addition only once and does nothing but an unsuccessful if-test $\log n - 1$ times. In general, an efficient algorithm should pretty much constantly use its resources. Techniques known as *load balancing* try to insure that processors are always kept busy doing useful computations.

7.5 Merge Sort

A discussion of basic algorithms would hardly seem complete without some mention of sorting. Some uses of parallelism are obvious. For example, a quicksort recursively sorts its left and right halves, usually one after the other. There is no reason that these subarrays couldn't be passed to separate processors. This is an immediate parallelization. Instead of quicksort, however, we'll focus on mergesort here.

Recall that a mergesort recursively mergesorts the left and right halves of an array, and then merges them together. The two halves can clearly be processed simultaneously. Thus, for the time complexity of parallel mergesort we have the recurrence relation

$$\begin{aligned} T_1(1) &= 1 \\ T_n(n) &= T_{n/2}(n/2) + M_n(n) \end{aligned}$$

where $M_n(n)$ is the time to merge two arrays of size $n/2$ using n processors. Obviously, we need to figure out $M_n(n)$ before we can fully evaluate the recurrence relation.

For simplicity, assume that all elements are distinct. If this isn't true, we can always use the initial indices in the array as a secondary keys. Let us briefly retreat from the realm of parallel computation, and ask the following question:

> Suppose you are given sorted arrays a and b, with n elements each, and an element $e = a_i$. Give an algorithm to find the position of e in the sorted array that would result if a and b were merged.

Well, what is the final position of some element $e = a_i$? There are $i - 1$ elements less than it in a. How many elements in b are less than e? To determine this, perform a binary search for e on the array b. While the search will fail, it can return the location j where e *would* be if it were in b. The index of e in the final array is $i + j - 1$. The binary search takes time $\Theta(\log n)$, and so does the entire computation.

So suppose we are merging two arrays, a and b. It takes time $\Theta(\log n)$ to determine the final location of any element in a or b. Assign one processor to each element of a and b. Each processor can determine in time $\Theta(\log n)$ where its element should go, and then write it their in the final array. Thus, given n processors, we have $M_n(n) = \Theta(\log n)$. (Note that this assumes a sequential binary search. Later on we'll see parallel algorithms for searching, too.)

Going back to the original recurrence relation, we now have

$$\begin{aligned} T_n(n) &= T_p(n/2) + \log n \\ &= T_p(n/4) + \log n + \log \tfrac{n}{2} \\ &= T_p(n/4) + \log n + (\log n - 1) \\ &= T_p(n/8) + \log n + (\log n - 1) + (\log n - 2) \\ &\vdots \\ &= T_p(n/2^i) + \log n + (\log n - 1) + (\log n - 2) + \cdots + (\log n - i + 1) \end{aligned}$$

The expansion stops at step $i = \log n$. So we have

$$\begin{aligned} T_n(n) &= \log n + (\log n - 1) + (\log n - 2) + \cdots + 2 + 1 \\ &= \textstyle\sum_{i=1}^{\log n} i \\ &= \tfrac{(\log n)\cdot(\log n + 1)}{2}. \end{aligned}$$

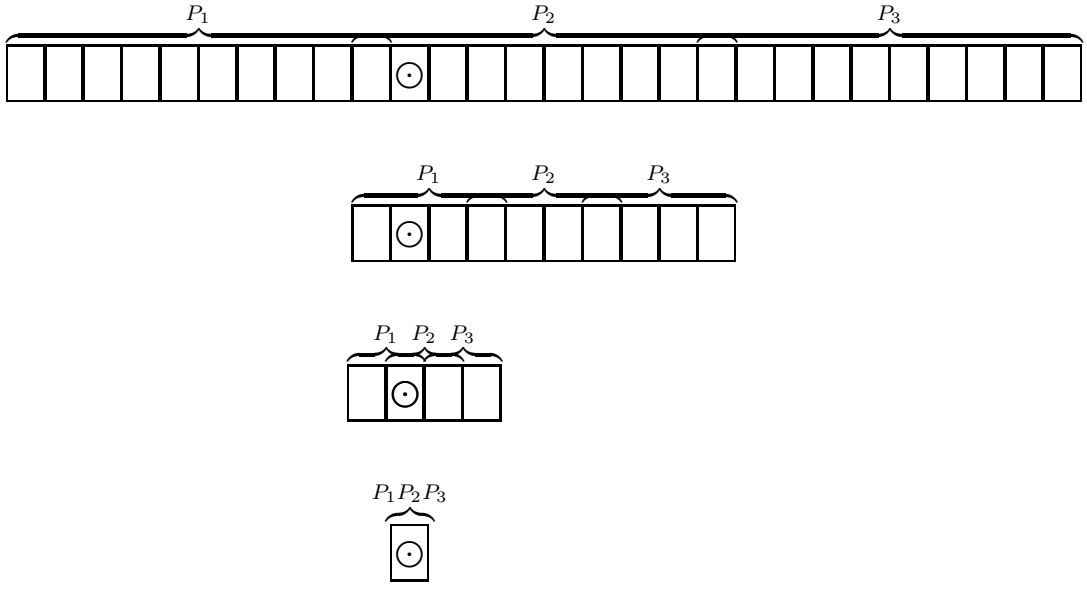
Thus, we conclude that the above parallel merge-sort runs in time $T_n(n) = \Theta(\log^2 n)$. It is an efficient, but not optimal, algorithm.

7.6 Searching

We continue our discussion of parallel algorithms. We want to write an algorithm to find some element in a sorted array. More formally, let a be an array of size n such that $a_i > a_j$ if $i > j$. (Notice that this means that all elements are distinct. Duplicate elements can be handled with some modifications.) Given a key x, we want to know if there is some index k such that $a_k = x$. As a starting point, we might as well take our standard sequential binary search. Suppose we have p processors. We can then divide the search range n into p sections of size n/p. Each processor can perform a binary search on its own range. This yields an algorithm which takes time $O(\log(n/p))$.

Intuitively, this isn't the most efficient parallel algorithm. At least $p-1$ of the p processors are guaranteed to fail to find the key x. For example, even using $\sqrt{n}$ processors, the search time is still $\Theta(\log n)$ in the worst case. We'll try a slightly different approach. Recall that, in a binary search, we pick the midpoint and split the array into two halves. With p processors, we can hopefully pick p different "midpoints" to narrow the search space. There are several variations on a p-ary search, all of which have the same asymptotic time and work complexity. For now, we pick one which is fairly easy to describe.

Given p processors, we can divide the array into p ranges of n/p elements. (Assume that n is a power of p. This won't really change the underlying algorithm; it will merely eliminate certain special cases. In actually implementing the search, it isn't necessary to make this assumption.) Since the array is sorted, each processor can determine if x lies in its range in time $O(1)$; it need merely compare x to the leftmost and rightmost elements in its range. Once the appropriate subrange has been determined, *all* the processors can then focus their efforts on the subrange. The range to search has been decreased from n to n/p.



Writing the pseudocode for this algorithm is not too hard. We assume that every processor has access to the global values left and right, which store the bounds of the array being searched, and p, the number of processors. Let a[0..n–1] be the array being searched. First, processor P_1 determines whether x < a[0] or x> a[n–1] and, if so, reports "not found" and terminates the algorithm. Otherwise, P_1 sets a[n] $\leftarrow +\infty$, left $\leftarrow$ 0, and right $\leftarrow$ n. Next, all the processors execute the following loop:

```
for each processor P_i
  while ( right > left + 1 )
    local_left  ← left + (i–1)·( right – left ) / p /* find left extent of local range */
    local_right ← left + i·( right – left ) / p                  /* and right extent */
    if (a[local_left] ≤ x) and (x < a[local_right])
      left  ← local_left
      right ← local_right
```

At the end, P_1 examines element a[left]. If a[left]=x then it reports "found at index left," and otherwise it reports "not found".

Note that we are assuming a concurrent read capability, since all processors must be able to read the global values left and right simultaneously. Note that there are no concurrent writes, i.e., only one processor updates the global variables left and right. The loop ends when right = left + 1, i.e., the search-range has been narrowed down to one element. With each iteration of the loop, the search-range [left..right–1] becomes smaller by a factor of p. This yields the recurrence relation:

$$T_p(n) = T_p(n/p) + 1$$

Thus, we see that the algorithm takes time $\Theta(\log_p n) = O(\log n / \log p)$. Now, if $p = \sqrt{n}$, the search time is constant (i.e., $O(1)$). The earlier algorithm ran in time $\Theta(\log(n/p))$. No matter what the constant factor is in each of these expressions, $\Theta(\log_p n)$ is asymptotically faster as n increases.

7.7 Pointer Jumping

Suppose we want to convert a linked list to an array format. This comes up fairly often; many applications require that we have direct access to all elements. For example, most sequential sorting and searching algorithms require arrays, rather than linked lists. Additionally, almost all of the parallel algorithms we've looked at need arrays for them to work; only when different processors can easily find specific data elements do we get good asymptotic time complexities. This requires calculating the index of each record in the array. Assume that the first record in the list is pointed to by ptr. We want to set the index field of each record to reflect its position in the new list. Assuming that the last record points to itself, we arrive easily at the sequential algorithm for accomplishing this:

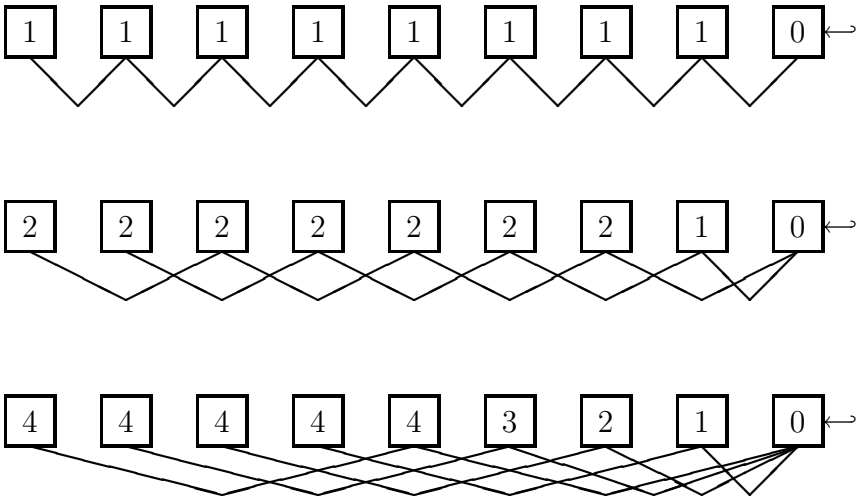
```
index ← 1
repeat
  index(ptr) ← index
  index ← index + 1
  ptr ← next(ptr)
until ( next(ptr) = ptr )
```

Obviously, this algorithm takes time $\Theta(n)$. Now assume that we have n processors, one to handle each record of the list. How shall we proceed?

Instead of actually calculating the index described above, we're going to compute a slightly different value. Specifically, we're going to design an algorithm to set the index of

each record to its distance from the last element. Thus, the last element will have index 0, and the first will have an index of $n-1$. It's simply a little easier to solve the problem this way. With a small, constant amount of processing at the end, we can convert from this index value to the one referred to above. (Specifically, we need merely set index to n – index – 1.)

The technique we'll be using is called *pointer jumping*, and it has applications beyond this simple problem. Initially, the elements are ordered in a standard linked list. Consider the next pointer of each record. We start off with the index of each record set to 1, except for the last record, which has index 0. The index represents the number of links (in the original linked list) between a record and the one pointed to by its next field. What we'll do is iteratively modify the next and index fields until every record points to the last one, and contains the correct distance between itself and that last record.



What we're doing is letting the pointers in the next fields "jump." The key observation is that, for any particular record pointed to by ptr, dist(ptr, next(next(ptr))) = dist(ptr, next(ptr)) + dist(next(ptr), next(next(ptr))) The entire algorithm looks like this:

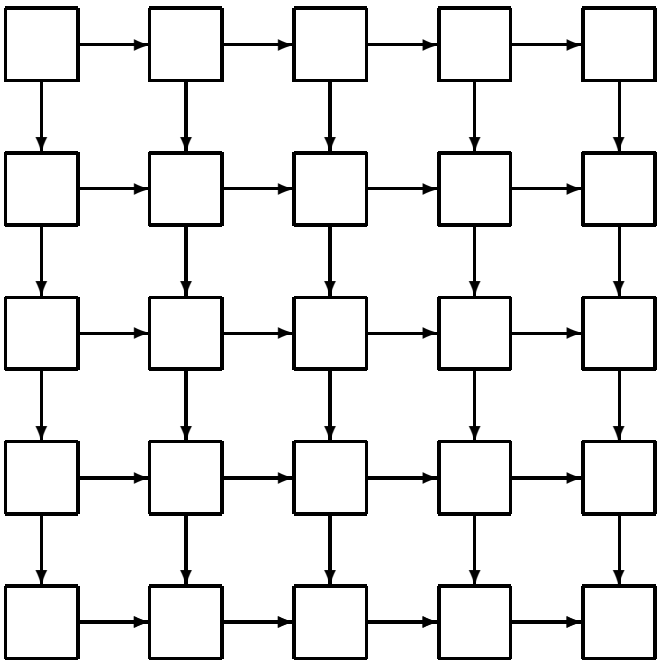
```
for each processor P_i
  if i is last element                              /*  Initialize index field  */
    index(ptr)  ← 0
  else
    index(ptr)  ← 1
  while ( index(next) ≠ 0)                          /*  while next isn't last record */
    index(ptr)  ← index(ptr) + index(next(ptr))
    next(ptr)  ← next(next(ptr))
```

At each iteration, for each node in the list, one of two things happens. It could be the case that next(next(ptr)) is the last element in the array, and the computation stops. Otherwise, we know that index(next(ptr)) = index(ptr), and so the index value doubles. Thus, after two iterations, the elements which are within two links of the last element are correctly indexed, and after three iterations, we know the final index of all elements which are within four links of the last element, and so on. In general, after i iterations, we know the final index of all elements within 2^{i-1} links of the last element. Thus, after $\log n$ iterations, we'll know the

index of all the elements. Since each iteration takes time $O(1)$ for each processor, the time complexity is $\Theta(\log n)$.

7.8 Matrix Multiplication

For our final problem we depart from the PRAM model of computation. Instead, we'll be considering matrix multiplication on *systolic* machines, a special class of network parallel computers. Computation is performed in three steps. In the first step processors receive input from their neighboring processors. In the second step, they perform some internal calculation. In the third step, they send output to their neighboring processors. While the architecture we'll be using is a two-dimensional array, such machines can have any architecture. Common ones include one-dimensional arrays of processors, and d-cubes, where the processors correspond to vertices on a d-dimensional cube, and connections between processors correspond to edges of the cube. At any rate, for now we will content ourselves with a two-dimensional array of processors looking something like this:



The problem we'll be attacking is matrix multiplication. Given two n by n matrices a and b, we wish to compute the matrix product c. Let a_{ij} denote the element of a in row i, column j. Then the product rule for matrices is given by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

The value at each of the n^2 entries c_{ij} is the sum of n products. Thus, there is an obvious $\Theta(n^3)$ sequential algorithm to calculate the matrix product. If we have $p(n) = n^2$ processors, how can we take advantage of them to make multiplication faster?

The key observation is that each a_{ij} and b_{jk} is used in the value of several different entries in c. For example, the product of two 3 by 3 matrices is:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} =$$

$$\begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{pmatrix}$$

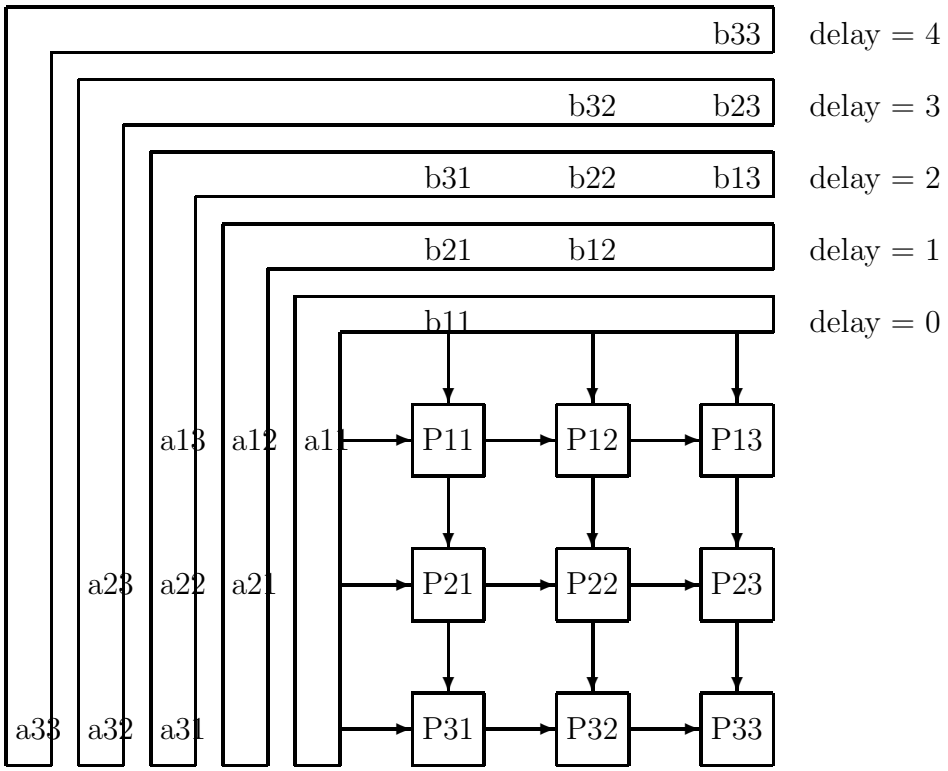
Observe that each a value seems to travel along one row, and the b values seem to move along columns. In our systolic array, we'll let P_{ij} be responsible for calculating c_{ij}. Suppose that, in our network, the upper-left processor P_{11} receives as input a_{11} and b_{11}. It can multiply them together, store the result, then pass b_{11} down the *column* and a_{11} down the *row*.

On the next iteration of computation, we have three processors performing calculations. P_{12} receives a_{11} from its neighboring processor. To calculate c_{12}, we have to multiply a_{11} by b_{12}. So assume that P_{12} is now also passed b_{12}. Then it can perform its proper calculation. Similarly, we set things up so that P_{21} simultaneously receives b_{11} (from P_{11}) and the value a_{21}.

Soon a pattern emerges. We feed in elements of a and b at the edges of the array. The elements are lagged, so that pairs to be multiplied arrive simultaneously at their proper processor. Each processor P_{ij} performs the following operations when two elements reach it:

1. input a_{ik} from the left and b_{kj} from above
2. compute $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
3. output a_{ik} to the right and b_{kj} below

A diagram is probably the best way to explain this:



Entries from the second row and column are delayed by one step, and entries from the third row and column are lagged for two steps. In the general n by n case, entries on the last row are delayed for $n-1$ steps before they are fed through the systolic array. Any particular entry takes time $\Theta(n)$ to travel across the network. The matrix multiplication is complete when the last entries a_{nn} and b_{nn} have traveled along the network and met. Hence, this algorithm for matrix multiplication on systolic arrays takes time $\Theta(n)$.

Chapter 8

Public-Key Cryptography

8.1 Introduction

The fundamental problem of cryptography is sending a message from Alice to Bob so that Eve cannot gain information from an intercepted copy of the message.[1] One of the main problems with traditional cryptosystems is *key transfer*, or how to distribute the key for encryption and decryption. Also, the encryption key is often closely tied to the decryption key; knowledge of one allows easy deduction of the other. This means that if, for example, agent Alice is compromised, then *all* messages to Bob are vulnerable.

In 1976, Diffie and Hellman described an abstract system which would avoid these problems, the *public-key cryptosystem*. While they didn't actually publish a particular public-key system, they discussed the features of such a system. Specifically, given a message M, encryption procedure E, and decryption procedure D, the following four properties must hold:

1. $D(E(M)) = M$
2. Both E and D are easy to compute.
3. It is computationally infeasible to derive D from E.
4. $E(D(M)) = M$

In retrospect, these properties seem fairly common-sense. The first property merely states that, once a message has been encrypted, applying the decryption procedure will restore it. Property two is perhaps more obvious. In order for a cryptosystem to be practical, encryption and decryption must be computationally fast.

The third property is the start of the innovation. It means that E only goes one way; it is computationally infeasible to invert E, unless you already know D. Thus, the encryption procedure E can be made public. Any party can send a message, while only one knows how to decrypt it.

If the fourth property holds, then the mapping is one-to-one. Thus, the cryptosystem is a solution to the *digital signature* problem. Given an electronic message from Bob to Alice, how can we prove that Bob actually sent it? Bob can apply his *decryption* procedure to some signature message M. Any other party can then verify that Bob actually sent the message by applying the public encyrption procedure E. Since only Bob knows the decryption function, only Bob can generate a message which can be correctly decoded by the function E.

[1] Traditionally, the protagonists are named Alice and Bob. However, some authors refer to B as Bill, and one source names his parties Aniuta and Björn.

Two years later, groups started publishing specific public-key cryptosystems. Before we can discuss them in detail, however, we'll need to review a little number theory.

8.2 Number Theory

The cryptosystems introduced in this paper require modular arithmetic. In this section we will develop some of the basic tools necessary for understanding them. Assume that all of the variables are integers.

8.2.1 Definitions

Initially, we require some notation and definitions. Write "$a|b$" if a divides b, i.e., b is a multiple of a. If $a|b$, then we know that there is some integer k such that $b = ak$. The *greatest common divisor* of a and b, or $\gcd(a, b)$, is exactly what it sounds like. It is the largest integer which evenly divides both a and b. If we felt like speaking a little more formally, we could say that $\gcd(a, b)$ is some number c such that if $d|a$ and $d|b$, then $d|c$. It's interesting to note that, for any integers a and b, we can find integers u and v so that $ua + vb = \gcd(a, b)$. If the gcd of two numbers is 1, we say that they are *relatively prime.*

A few words about the mod function are in order. Programming experience yields one understanding of the mod function; $a \bmod n$ is the remainder of a when divided by n. It is sometimes convenient to talk about "equivalence mod n." We write that $a \equiv b \bmod n$, and say "a is equivalent to b mod n." Observe that if $a \bmod n = r$, then there is some integer k so that $a = kn + r$. Therefore, if $a \equiv b \bmod n$ then $a - b = kn$ for some k.

We'll also need the Euler totient function, $\varphi(n)$. The Euler totient is the number of (positive) integers which are relatively prime to n. If p is a prime, then $\varphi(p) = p-1$. Since p is prime, no integer divides it; thus, each of the numbers 1, 2, ..., $p - 1$ are relatively prime to it, and $\varphi(p) = p - 1$.

What if n isn't a prime number? Suppose $n = pq$, where p and q are primes. How many numbers are relatively prime to n? Well, initially, we observe that there are $pq - 1$ positive integers between 1 and n. However, p of them are multiples of p, and so they have a gcd of p with n. Similarly, there are q multiples of q. Those multiples can't be counted in $\varphi(n)$. Thus, we see that $\varphi(n) = pq - 1 - p - q = (p - 1)(q - 1)$.

8.2.2 Two congruences

We now have enough machinery for our first major theorem, which is sometimes called Fermat's little theorem:

Theorem 2 *Let p be a prime, $1 \leq a \leq p - 1$. Then $a^{p-1} \equiv 1 \bmod p$.*

The proof of this may seem a little involved, but it isn't too bad. Consider the numbers 1, 2, ..., $p - 1$. Initially, we claim that multiplication by a yields distinct values (mod p) for each of the numbers between 1 and $p - 1$. Why? Well, suppose not. Then there are some i and j, $i > j$ such that $ai \equiv aj \bmod p$.[2] From our discussion above, this means that $ai - aj = kp$, or $a(i - j) = kp$. Now, p is prime; it can't be factored. This means that at least one of a and $i - j$ is a multiple of p, since their product is a multiple of p. But both a and $i - j$ are less than p, so this can't be true. Therefore, $ai \neq aj$ for each $i \neq j$.

[2] If $i < j$, then we can just switch the two, and continue with the proof.

But this means that multiplication of the nonzero numbers between 1 and $p-1$ effectively permutes them. We know that the sets $\{1, 2, \ldots, p-1\}$ and $\{a \cdot 1, a \cdot 2, \ldots, a \cdot (p-1)\}$ contain exactly the same elements. So when we multiply the elements of the sets together, we get the same value:

$$1 \cdot 2 \cdots (p-1) \equiv (a \cdot 1) \cdot (a \cdot 2) \cdots a \cdot (p-1) \bmod p$$

If we combine the a terms, we get

$$a^{p-1} 1 \cdot 2 \cdots (p-1) \equiv 1 \cdot 2 \cdots (p-1) \bmod p$$

We can cancel the constants from both sides (later, we'll see why), yielding $a^{p-1} \equiv 1 \bmod p$, the desired result.

Notice that, in this case, the exponent was $p-1 = \phi(p)$. As it turns out, a generalized form of Fermat's little theorem is true, too. The following theorem is usually attributed to Euler.

Theorem 3 *Let* $\gcd(a, n) = 1$. *Then* $a^{\varphi(n)} \equiv 1 \bmod m$.

Using more or less the same technique as in Fermat's little theorem, we can prove this proposition. We know that there are exactly $\varphi(n)$ distinct numbers relatively prime to n between 1 and n. Label them $u_1, u_2, \ldots, u_{\varphi(n)}$. Again, for each $i \neq j$, and some a relatively prime to n, we can easily show that $au_i \not\equiv au_j \bmod n$.[3]

Suppose that $au_i \equiv au_j \bmod n$. Then $a(u_i - u_j) \equiv 0 \bmod n$, and so $a(u_i - u_j)$ is a multiple of n. Now, we know that $\gcd(a, n) = 1$, which means that $u_i - u_j$ must be a multiple of n. But if that were true, then $u_i \equiv u_j \bmod n$, which contradicts our assumption. Therefore, $au_i \not\equiv au_j \bmod n$.

Once again, we know that multiplying by a merely permutes the set of elements relatively prime to n. Hence,

$$u_1 \cdot u_2 \cdots u_{\varphi(n)} \equiv (au_1) \cdot (au_2) \cdots (au_{\varphi(n)}) \bmod n$$

Again, we collect a term $a^{\varphi(n)}$ on one side, and divide by the product of the u_i. In the end, this gives us $a^{\varphi(n)} \equiv 1 \bmod n$.

8.2.3 Inverses

The two theorems above relied on our being able to "cancel" terms from the left and right. In other words, given some number a, we assumed we could find some a^{-1} such that $aa^{-1} \equiv 1 \bmod n$. Is this assumption warranted? As it turns out, it is, provided that $\gcd(a, n) = 1$. We can show this in two ways. First, by the definition of gcd given above, we know that there are numbers u and v such that $ua + vn = 1$. But if this is true, then $ua = (-v)n + 1$, which means that $ua \equiv 1 \bmod n$, and $u = a^{-1}$.

We can also prove this in a completely different fashion. Consider the powers of a: a, a^2, a^3, ..., all taken mod n. Now, there are only finitely many numbers between 1 and $n-1$, so some values must be repeated as we list all the powers. So find the smallest i and j, $i \neq j$, such that $a^i \equiv a^j \bmod n$. (Exponentiation is quite simple. The symbol a^i just means $a \cdot a \cdots a$ i times.) Because of the way we've defined exponentiation, it makes sense

[3] Since $\gcd(a, n) = 1$, we know that $a = u_k$ for some k. However, this doesn't affect the argument.

to talk of subtracting exponents from both sides. This yields $a^{i-j+1} \equiv a^{j-j} \equiv 1 \bmod n$. But if $a^{i-j+1} \equiv 1 \bmod n$, then $aa^{i-j} \equiv 1 \bmod n$, and a^{i-j} is the multiplicative inverse of a, considered mod$n$.

8.3 Some Public-Key Cryptosystems

The design of public-key cryptosystems can be desribed in general terms. The ideas is to find a very tough problem in computer science, and then somehow tie the cryptosystem to it. Ideally, one arrives at an actual proof that breaking the cryptosystem is computationally equivalent to solving the difficult problem. There's a large class of problems, *NP-complete*, which don't have known polynomial time algorithms for their solution.[4] Then, to generate the particular encryption and decryption keys, you create a particular set of parameters for this problem. Encrypting then means turning the message into an instance of the problem. Using a "trapdoor," the recipient can use secret information (the decryption key) to solve the puzzle effortlessly.

Some care must be taken in how the problem is tied to the cryptosystem. One of the earlier public-key cryptosystems, the Merkle-Hellman system, linked encryption to something called the knapsack problem, which is NP-complete. Unfortunately, the problems the system generates turn out to be a special subclass of the knapsack problem, which *is* easily soluble. So designing public-key cryptosystems has its share of subtleties.

8.3.1 RSA

Probably the most famous public-key algorithm is also one of the oldest. It is named RSA after its inventors, Rivest, Shamir and Adleman.

Two odd primes, p and q, are selected. Let N be their product, pq. Encryption and decryption keys e and d are selected so that $e \cdot d \equiv 1 \bmod \varphi(N)$. The values N and e form the public key; N and d, the private key.

Now, because of the way we've selected e and d, we know that $ed = k\varphi(N)+1$. So for any $a \neq 0$, by applying theorem 2, we can show that $a^{ed} = a^{k\varphi(N)+1} = (a^{\varphi(N)})^k a^1 \equiv a \bmod N$. This is the heart of the algorithm. If Alice wishes to send a message to Bob, she encrypts the message M using Bob's public key:

$$C = E(M) = M^{e_B} \bmod N_B$$

When Bob receives the message, he easily decrypts it:

$$D(C) = C^{d_B} \bmod N_B$$

Note that, even though Alice knows the value e_B, she can't figure d_B unless she knows $\varphi(N_B)$. This, in turn, requires factoring N_B. While there is no *proof* that factorization is computationally complex, a whole series of famous mathematicians[5] have worked on the problem over the past few hundred years. Especially if N is large (≈ 200 digits), it will take a very long time to factor it. To give you an idea of the state of the art, mathematicians were quite excited when a nationwide network of computers was able to factor the ninth

[4] In fact, it's largely believed that there are none. So if you tie a cryptosystem to such a problem, and later break the system, you have actually gained by solving an outstanding problem in computer science!

[5] Legendre, Fermat, and that crowd, in addition to contemporary mathematicians

Fermat number, $2^{512} - 1$. This number has "only" 155 decimal digits. Barring a major breakthrough, the RSA system will remain secure. For if technology somehow advances to a point where it is feasible to factor 200 digit numbers, Bob need only choose an N with three or four hundred digits.

8.3.2 El Gamal

We have seen that the security of the RSA cryptosystem is contingent on the difficulty of factoring large numbers. It is possible to construct cryptosystems based on other difficult number-theoretic problems. We now consider the El Gamal cryptosystem. It is probably more vulnerable than RSA, in that its core problem seems more tractable. Nonetheless, it is instructive to examine a different public-key cryptosystem.

When we're working with the real numbers, $\log_b x$ is the value y such that $b^y = x$. We can define an analogous *discrete logarithm*. Given a modulus q and some $b < q$, the discrete log of x to the base b is an integer y such that $b^y \equiv x \bmod q$. While it is quite easy to raise numbers to large powers (mod q), the inverse computation of the discrete logarithm is much harder. The El Gamal system relies on the difficulty of this computation.

In this system, everybody agrees on a prime modulus, q.[6] Additionally, a number g is selected. (The *order* of g is the smallest $e > 1$ such that $g^e \equiv 1 \bmod q$. Ideally, g will be a *generator*, that is, the order of g is $q - 1$. At any rate, we want the order of g to be large.) Then Bob (for example) picks his private key y_B, an integer between 1 and $q-2$. The public enciphering key is g^{y_B}. If taking discrete logarithms is as difficult as we hope it is, releasing g^{y_B} does not reveal y_B.

Suppose Alice wishes to send a message M to Bob. She picks some arbitrary integer k, and actually sends *two* values to Bob, namely, g^k and $Mg^{y_B k}$. Since g^{y_B} is public information, she can compute $Mg^{y_B k}$ without actually knowing y_B. This pair of numbers is enough for Bob to reconstruct M. Only Bob knows what y_B is, so given g^k, only Bob can calculate $g^{y_B k}$. Having done this, Bob can then calculate the *inverse* of $g^{y_B k}$. Multiplying this number by $Mg^{y_B k}$ recovers M fully.

8.4 Algorithms

Certain computations arise repeatedly in implementing these two public-key cryptosystems. These operations include gcd, exponentiation, and primality testing. There are several different ways of implementing each of these. This section introduces one algorithm for each of these problems.

8.4.1 Exponentiation

Exponentiation lies at the heart of these cryptosystems. Since we'll be doing a lot of it, to quite large powers, it behooves us to find a method other than the obvious brute-force. Most importantly, however, we must keep the result from getting too large. Multiplying, say, 30192 by itself 43791 times and *then* taking the result modulo 65301 will yield unpredictable results in most languages. Thus, we should take the modulus at each iteration:

NAIVE_EXP_MOD(base, pow, modulus)

[6] Actually, in some cases q can be a power of a prime. However, it makes the analysis a little trickier.

```
result ← 1
for p ← 1 to pow do
  result ← (result*base) mod modulus;
return(result)
```

This isn't very efficient, though; it still takes $O(pow)$ iterations. With large exponents, this is quite expensive. Fortunately, there's a better method. The key observation is that we can write pow as a binary number:

$$pow = k_n 2^n + k_{n-1} 2^{n-1} + \cdots + k_0 2^0$$

Of course, each of the k_i is 1 or 0. Once we know this, we can rewrite b^{pow} as

$$b^{pow} = b^{k_n 2^n + k_{n-1} 2^{n-1} + \cdots + k_0 2^0} = b^{k_n 2^n} b^{k_{n-1} 2^{n-1}} \cdots b^{k_0 2^0}$$

Experience in bit-flicking allows us to easily craft an algorithm to use this information.

8.4.2 Greatest Common Divisor

We've already discussed some properties of $\gcd(a, b)$. Additionally,

$$\gcd(a, b) = \gcd(b, a - qb)$$

for any integer r. Why? Let $g = \gcd(a, b)$. Clearly, g divides b. Furthermore, if two numbers are a multiple of g, then their difference must also be a multiple of g. Thus, g also divides $a - qb$, and $\gcd(a, b) = \gcd(b, a - qb)$. This leads us easily to the Euclidean algorithm:

```
GCD(a,b)
if (b > a)
  switch(a,b)
while ( b > 0 )
  temp ← a mod b;
  a ← b;
  b ← temp
end
return( a )
```

8.4.3 Primality

We'll need to generate primes p and q. All we have to do is randomly pick a number, and then test to make sure if it's prime. Thus, we need a good test for primality.

Fermat's little theorem seems to suggest a solution. Perhaps we can somehow use $a^{p-1} \equiv 1 \bmod p$ to form a test. Pick any number a, and raise it to the power $p - 1$. If the result is *not* 1, then the number p is definitely not prime. Otherwise, there's a chance it is. Would repeating this test somehow prove that p is prime?

Unfortunately, the answer is "no." There is a class of numbers, called Carmichael numbers, which have the property that $a^{n-1} \equiv 1 \bmod n$ for all $1 \leq a \leq n-1$, but n is composite. These numbers ruin such a simple test.

While this "probabilistic" test won't work, there are several which will. There are many tests for primality of the form $R(p)$, which return true or false. If $R(p)$, returns false, then p is definitely *not* prime. Otherwise, there is a probability $0 < \varepsilon < 1$ that p is not a prime.

The test we'll be describing here is the Solovay-Strassen test. Unfortunately, a detailed explanation of its workings is beyond the scope of this paper. However, we *can* take a look at how to implement it. For two numbers a and b, we can calculate something called the *Jacobi symbol* $(\frac{a}{b})$ as follows:

```
JACOBI(a, b)
if a = 1
  return( 1 )
if a = 0
  return( 0 )
if (a mod 2 = 0)
  if (b·b − 1)/8 mod 2 = 0
    return( jacobi(a/2, b) )
  else
    return( −jacobi(a/2, b) )
else if (a − 1)(b − 1)/4 mod 2 = 0
  return( jacobi(b mod a, a) )
else
  return( −jacobi(b mod a, a) )
```

Now, it can be shown that a number n is prime if and only if

$$a^{(n-1)/2} \equiv (\frac{a}{n}) \bmod n$$

for all choices of a. So, once we've written the routine for calculating the Jacobi symbol, we can write a probabilistic primality tester.

```
IS_PRIME( n )
for i ← 1 to 50
  a  ← random( 1, n-1 )
  if exp_mod( a, (n-1)/2, n ) ≠ jacobi(a, n) mod n
    return( false ) return( true )
```

If n successfully passes the test 50 times, then the chances are less than 1 in 2^{50} that n is composite.[7]

8.5 Conclusion

The introduction to public-key cryptosystems ends here. A lot of mathematics was introduced, and it may seem a bit intimidating. If the proofs are a little scary, read them over a few times, and they'll start to make sense. Also, it's important not to lose sight of the basic idea of public-key cryptography. The main innovation is that the encryption and decryption keys are kept separate. It provides an elegant solution to the problem of distributing keys. In closing, a couple of related ideas are probably worth mentioning.

[7]Note that, while $-1 \equiv n - 1 \bmod n$, many programming languages are unaware of this fact. Be careful!

The first is the *key exchange protocol.* Several techniques have been developed in which Alice and Bob can exchange private keys, even over an insecure communication line. Typically, these protocols involve several rounds. Alice picks a number, performs some operation on it, and then sends the number to Bob. Bob, in turn, takes that number, applies his own transformation, and then sends it back, etc.

The second is the *zero-knowledge proof.* While this isn't actually an instance of cryptography, it has similar applications. A zero-knowledge proof is an authentication procedure. Suppose Alice wants to prove to Bob that she is, in fact, Alice. However, she doesn't wish to give away her password, just in case Bob isn't really Bob. A zero-knowledge proof is a protocol whereby Alice can prove (to an arbitrary degree of confidence) that she is who she claims she is, without ever disclosing her knowledge to the interrogator.