CMSC 251: Algorithms[1]
Spring 1998
Dave Mount

Lecture 1: Course Introduction

(Tuesday, Jan 27, 1998)
Read: Course syllabus and Chapter 1 in CLR (Cormen, Leiserson, and Rivest).

What is algorithm design? Our text defines an *algorithm* to be any well-defined computational procedure that takes some values as *input* and produces some values as *output*. Like a cooking recipe, an algorithm provides a step-by-step method for solving a computational problem.

A good understanding of algorithms is essential for a good understanding of the most basic element of computer science: *programming*. Unlike a program, an algorithm is a mathematical entity, which is independent of a specific programming language, machine, or compiler. Thus, in some sense, algorithm design is all about the mathematical theory behind the design of good programs.

Why study algorithm design? There are many facets to good program design. Good algorithm design is one of them (and an important one). To be really complete algorithm designer, it is important to be aware of programming and machine issues as well. In any important programming project there are two major types of issues, *macro issues* and *micro issues*.

Macro issues involve elements such as how does one coordinate the efforts of many programmers working on a single piece of software, and how does one establish that a complex programming system satisfies its various requirements. These macro issues are the primary subject of courses on software engineering.

A great deal of the programming effort on most complex software systems consists of elements whose programming is fairly mundane (input and output, data conversion, error checking, report generation). However, there is often a small critical portion of the software, which may involve only tens to hundreds of lines of code, but where the great majority of computational time is spent. (Or as the old adage goes: 80% of the execution time takes place in 20% of the code.) The micro issues in programming involve how best to deal with these small critical sections.

It may be very important for the success of the overall project that these sections of code be written in the most efficient manner possible. An unfortunately common approach to this problem is to first design an inefficient algorithm and data structure to solve the problem, and then take this poor design and attempt to fine-tune its performance by applying clever coding tricks or by implementing it on the most expensive and fastest machines around to boost performance as much as possible. The problem is that if the underlying design is bad, then often no amount of fine-tuning is going to make a substantial difference.

As an example, I know of a programmer who was working at Boeing on their virtual reality system for the 777 project. The system was running unacceptably slowly in spite of the efforts of a large team of programmers and the biggest supercomputer available. A new programmer was hired to the team, and his first question was on the basic algorithms and data structures used by the system. It turns out that the system was based on rendering hundreds of millions of polygonal elements, most of which were invisible at any point in time. Recognizing this source of inefficiency, he redesigned the algorithms and data structures, recoded the inner loops, so that the algorithm concentrated its efforts on eliminating many invisible elements, and just drawing the few thousand visible elements. In a matter of two weeks he had a system that ran faster on his office workstation, than the one running on the supercomputer.

This may seem like a simple insight, but it is remarkable how many times the clever efforts of a single clear-sighted person can eclipse the efforts of larger groups, who are not paying attention to the basic principle that we will stress in this course:

Before you implement, first be sure you have a good design.

This course is all about how to design good algorithms. Because the lesson cannot be taught in just one course, there are a number of companion courses that are important as well. CMSC 420 deals with how to design good data structures. This is not really an independent issue, because most of the fastest algorithms are fast because they use fast data structures, and vice versa. CMSC 451 is the advanced version of this course, which teaches other advanced elements of algorithm design. In fact, many of the courses in the computer science department deal with efficient algorithms and data structures, but just as they apply to various applications: compilers, operating systems, databases, artificial intelligence, computer graphics and vision, etc. Thus, a good understanding of algorithm design is a central element to a good understanding of computer science and good programming.

Implementation Issues: One of the elements that we will focus on in this course is to try to study algorithms as pure mathematical objects, and so ignore issues such as programming language, machine, and operating system. This has the advantage of clearing away the messy details that affect implementation. But these details may be very important.

For example, an important fact of current processor technology is that of *locality of reference*. Frequently accessed data can be stored in registers or cache memory. Our mathematical analyses will usually ignore these issues. But a good algorithm designer can work within the realm of mathematics, but still keep an open eye to implementation issues down the line that will be important for final implementation. For example, we will study three fast sorting algorithms this semester, heap-sort, merge-sort, and quick-sort. From our mathematical analysis, all have equal running times. However, among the three (barring any extra considerations) quicksort is the fastest on virtually all modern machines. Why? It is the best from the perspective of locality of reference. However, the difference is typically small (perhaps 10–20% difference in running time).

Thus this course is not the last word in good program design, and in fact it is perhaps more accurately just the first word in good program design. The overall strategy that I would suggest to any programming would be to first come up with a few good designs from a mathematical and algorithmic perspective. Next prune this selection by consideration of practical matters (like locality of reference). Finally prototype (that is, do test implementations) a few of the best designs and run them on data sets that will arise in your application for the final fine-tuning. Also, be sure to use whatever development tools that you have, such as profilers (programs which pin-point the sections of the code that are responsible for most of the running time).

Course in Review: This course will consist of three major sections. The first is on the mathematical tools necessary for the analysis of algorithms. This will focus on asymptotics, summations, recurrences. The second element will deal with one particularly important algorithmic problem: sorting a list of numbers. We will show a number of different strategies for sorting, and use this problem as a case-study in different techniques for designing and analyzing algorithms.

The final third of the course will deal with a collection of various algorithmic problems and solution techniques. Finally we will close this last third with a very brief introduction to the theory of NP-completeness. NP-complete problem are those for which no efficient algorithms are known, but no one knows for sure whether efficient solutions might exist.

Lecture 2: Analyzing Algorithms: The 2-d Maxima Problem

(Thursday, Jan 29, 1998)
Read: Chapter 1 in CLR.

Analyzing Algorithms: In order to design good algorithms, we must first agree the criteria for measuring algorithms. The emphasis in this course will be on the design of efficient algorithm, and hence we will measure algorithms in terms of the amount of *computational resources* that the algorithm requires. These resources include mostly running time and memory. Depending on the application, there may be other elements that are taken into account, such as the number disk accesses in a database program or the communication bandwidth in a networking application.

In practice there are many issues that need to be considered in the design algorithms. These include issues such as the ease of debugging and maintaining the final software through its life-cycle. Also, one of the luxuries we will have in this course is to be able to assume that we are given a clean, fully-specified mathematical description of the computational problem. In practice, this is often not the case, and the algorithm must be designed subject to only partial knowledge of the final specifications. Thus, in practice it is often necessary to design algorithms that are simple, and easily modified if problem parameters and specifications are slightly modified. Fortunately, most of the algorithms that we will discuss in this class are quite simple, and are easy to modify subject to small problem variations.

Model of Computation: Another goal that we will have in this course is that our analyses be as independent as possible of the variations in machine, operating system, compiler, or programming language. Unlike programs, algorithms to be understood primarily by people (i.e. programmers) and not machines. Thus gives us quite a bit of flexibility in how we present our algorithms, and many low-level details may be omitted (since it will be the job of the programmer who implements the algorithm to fill them in).

But, in order to say anything meaningful about our algorithms, it will be important for us to settle on a mathematical model of computation. Ideally this model should be a reasonable abstraction of a standard generic single-processor machine. We call this model a *random access machine* or *RAM*.

A RAM is an idealized machine with an infinitely large random-access memory. Instructions are executed one-by-one (there is no parallelism). Each instruction involves performing some *basic operation* on two values in the machines memory (which might be characters or integers; let's avoid floating point for now). Basic operations include things like assigning a value to a variable, computing any basic arithmetic operation ($+$, $-$, $*$, integer division) on integer values of any size, performing any comparison (e.g. $x \leq 5$) or boolean operations, accessing an element of an array (e.g. $A[10]$). We assume that each basic operation takes the same constant time to execute.

This model seems to go a good job of describing the computational power of most modern (nonparallel) machines. It does not model some elements, such as efficiency due to locality of reference, as described in the previous lecture. There are some "loop-holes" (or hidden ways of subverting the rules) to beware of. For example, the model would allow you to add two numbers that contain a billion digits in constant time. Thus, it is theoretically possible to derive nonsensical results in the form of efficient RAM programs that cannot be implemented efficiently on any machine. Nonetheless, the RAM model seems to be fairly sound, and has done a good job of modeling typical machine technology since the early 60's.

Example: 2-dimension Maxima: Rather than jumping in with all the definitions, let us begin the discussion of how to analyze algorithms with a simple problem, called *2-dimension maxima*. To motivate the problem, suppose that you want to buy a car. Since you're a real swinger you want the fastest car around, so among all cars you pick the fastest. But cars are expensive, and since you're a swinger on a budget, you want the cheapest. You cannot decide which is more important, speed or price, but you know that you definitely do NOT want to consider a car if there is another car that is both faster and cheaper. We say that the fast, cheap car *dominates* the slow, expensive car relative to your selection criteria. So, given a collection of cars, we want to list those that are not dominated by any other.

Here is how we might model this as a formal problem. Let a point p in 2-dimensional space be given by its integer coordinates, $p = (p.x, p.y)$. A point p is said to *dominated by* point q if $p.x \leq q.x$ and $p.y \leq q.y$. Given a set of n points, $P = \{p_1, p_2, \ldots, p_n\}$ in 2-space a point is said to be *maximal* if it is not dominated by any other point in P.

The car selection problem can be modeled in this way. If for each car we associated (x, y) values where x is the speed of the car, and y is the negation of the price (thus high y values mean cheap cars), then the maximal points correspond to the fastest and cheapest cars.

2-dimensional Maxima: Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in 2-space, each represented by its x and y integer coordinates, output the set of the maximal points of P, that is, those points p_i, such that p_i is not dominated by any other point of P. (See the figure below.)

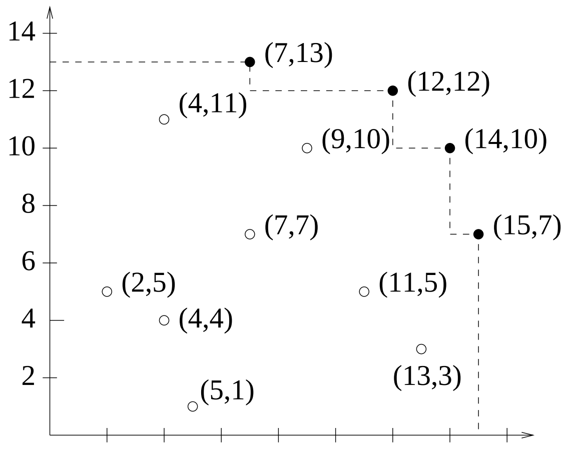


Figure 1: Maximal Points.

Observe that our description of the problem so far has been at a fairly mathematical level. For example, we have intentionally not discussed issues as to how points are represented (e.g., using a structure with records for the x and y coordinates, or a 2-dimensional array) nor have we discussed input and output formats. These would normally be important in a software specification. However, we would like to keep as many of the messy issues out since they would just clutter up the algorithm.

Brute Force Algorithm: To get the ball rolling, let's just consider a simple brute-force algorithm, with no thought to efficiency. Here is the simplest one that I can imagine. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the initial set of points. For each point p_i, test it against all other points p_j. If p_i is not dominated by any other point, then output it.

This English description is clear enough that any (competent) programmer should be able to implement it. However, if you want to be a bit more formal, it could be written in pseudocode as follows:

Brute Force Maxima

```
Maxima(int n, Point P[1..n]) {                  // output maxima of P[0..n-1]
    for i = 1 to n {
        maximal = true;                         // P[i] is maximal by default
        for j = 1 to n {
            if (i != j) and (P[i].x <= P[j].x) and (P[i].y <= P[j].y) {
                maximal = false;                // P[i] is dominated by P[j]
                break;
            }
        }
        if (maximal) output P[i];               // no one dominated...output
    }
}
```

There are no formal rules to the syntax of this pseudocode. In particular, do not assume that more detail is better. For example, I omitted type specifications for the procedure `Maxima` and the variable `maximal`, and I never defined what a `Point` data type is, since I felt that these are pretty clear from context or just unimportant details. Of course, the appropriate level of detail is a judgement call. Remember, algorithms are to be read by people, and so the level of detail depends on your intended audience. When writing pseudocode, you should omit details that detract from the main ideas of the algorithm, and just go with the essentials.

You might also notice that I did not insert any checking for consistency. For example, I assumed that the points in P are all distinct. If there is a duplicate point then the algorithm may fail to output even a single point. (Can you see why?) Again, these are important considerations for implementation, but we will often omit error checking because we want to see the algorithm in its simplest form.

Correctness: Whenever you present an algorithm, you should also present a short argument for its correctness. If the algorithm is tricky, then this proof should contain the explanations of why the tricks works. In a simple case like the one above, there almost nothing that needs to be said. We simply implemented the definition: a point is maximal if no other point dominates it.

Running Time Analysis: The main purpose of our mathematical analyses will be be measure the execution time (and sometimes the space) of an algorithm. Obviously the running time of an implementation of the algorithm would depend on the speed of the machine, optimizations of the compiler, etc. Since we want to avoid these technology issues and treat algorithms as mathematical objects, we will only focus on the pseudocode itself. This implies that we cannot really make distinctions between algorithms whose running times differ by a small constant factor, since these algorithms may be faster or slower depending on how well they exploit the particular machine and compiler. How small is small? To make matters mathematically clean, let us just ignore all constant factors in analyzing running times. We'll see later why even with this big assumption, we can still make meaningful comparisons between algorithms.

In this case we might measure running time by counting the number of steps of pseudocode that are executed, or the number of times that an element of P is accessed, or the number of comparisons that are performed.

Running time depends on input size. So we will define running time in terms of a function of input size. Formally, the *input size* is defined to be the number of characters in the input file, assuming some reasonable encoding of inputs (e.g. numbers are represented in base 10 and separated by a space). However, we will usually make the simplifying assumption that each number is of some constant maximum length (after all, it must fit into one computer word), and so the input size can be estimated up to constant factor by the parameter n, that is, the length of the array P.

Also, different inputs of the same size may generally result in different execution times. (For example, in this problem, the number of times we execute the inner loop before breaking out depends not only on the size of the input, but the structure of the input.) There are two common criteria used in measuring running times:

Worst-case time: is the maximum running time over all (legal) inputs of size n? Let I denote a legal input instance, and let $|I|$ denote its length, and let $T(I)$ denote the running time of the algorithm on input I.

$$T_{worst}(n) = \max_{|I|=n} T(I).$$

Average-case time: is the average running time over all inputs of size n? More generally, for each input I, let $p(I)$ denote the probability of seeing this input. The average-case running time is the weight sum of running times, with the probability being the weight.

$$T_{avg}(n) = \sum_{|I|=n} p(I)T(I).$$

We will almost always work with worst-case running time. This is because for many of the problems we will work with, average-case running time is just too difficult to compute, and it is difficult to specify a natural probability distribution on inputs that are really meaningful for all applications. It turns out that for most of the algorithms we will consider, there will be only a constant factor difference between worst-case and average-case times.

Running Time of the Brute Force Algorithm: Let us agree that the input size is n, and for the running time we will count the number of time that any element of P is accessed. Clearly we go through the outer loop n times, and for each time through this loop, we go through the inner loop n times as well. The condition in the if-statement makes four accesses to P. (Under C semantics, not all four need be evaluated, but let's ignore this since it will just complicate matters). The output statement makes two accesses (to $P[i].x$ and $P[i].y$) for each point that is output. In the worst case every point is maximal (can you see how to generate such an example?) so these two access are made for each time through the outer loop.

Thus we might express the worst-case running time as a pair of nested summations, one for the i-loop and the other for the j-loop:

$$T(n) = \sum_{i=1}^{n} \left(2 + \sum_{j=1}^{n} 4 \right).$$

These are not very hard summations to solve. $\sum_{j=1}^{n} 4$ is just $4n$, and so

$$T(n) = \sum_{i=1}^{n} (4n + 2) = (4n + 2)n = 4n^2 + 2n.$$

As mentioned before we will not care about the small constant factors. Also, we are most interested in what happens as n gets large. Why? Because when n is small, almost any algorithm is fast enough. It is only for large values of n that running time becomes an important issue. When n is large, the n^2 term will be much larger than the n term, and so it will dominate the running time. We will sum this analysis up by simply saying that the worst-case running time of the brute force algorithm is $\Theta(n^2)$. This is called the *asymptotic growth rate* of the function. Later we will discuss more formally what this notation means.

Summations: (This is covered in Chapter 3 of CLR.) We saw that this analysis involved computing a summation. Summations should be familiar from CMSC 150, but let's review a bit here. Given a finite sequence of values $a_1, a_2, \ldots, a_n$, their sum $a_1 + a_2 + \cdots + a_n$ can be expressed in *summation notation* as

$$\sum_{i=1}^{n} a_i.$$

If $n = 0$, then the value of the sum is the additive identity, 0. There are a number of simple algebraic facts about sums. These are easy to verify by simply writing out the summation and applying simple high school algebra. If c is a constant (does not depend on the summation index i) then

$$\sum_{i=1}^{n} ca_i = c\sum_{i=1}^{n} a_i \qquad \text{and} \qquad \sum_{i=1}^{n}(a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i.$$

There are some particularly important summations, which you should probably commit to memory (or at least remember their asymptotic growth rates). If you want some practice with induction, the first two are easy to prove by induction.

Arithmetic Series: For $n \geq 0$,

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \Theta(n^2).$$

Geometric Series: Let $x \neq 1$ be any constant (independent of i), then for $n \geq 0$,

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}.$$

If $0 < x < 1$ then this is $\Theta(1)$, and if $x > 1$, then this is $\Theta(x^n)$.

Harmonic Series: This arises often in probabilistic analyses of algorithms. For $n \geq 0$,

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln n = \Theta(\ln n).$$

Lecture 3: Summations and Analyzing Programs with Loops

(Tuesday, Feb 3, 1998)
Read: Chapt. 3 in CLR.

Recap: Last time we presented an algorithm for the 2-dimensional maxima problem. Recall that the algorithm consisted of two nested loops. It looked something like this:

Brute Force Maxima

```
Maxima(int n, Point P[1..n]) {
    for i = 1 to n {
        ...
        for j = 1 to n {
            ...
        ...
    }
}
```

We were interested in measuring the worst-case running time of this algorithm as a function of the input size, n. The stuff in the "..." has been omitted because it is unimportant for the analysis.

Last time we counted the number of times that the algorithm accessed a coordinate of any point. (This was only one of many things that we could have chosen to count.) We showed that as a function of n in the worst case this quantity was

$$T(n) = 4n^2 + 2n.$$

We were most interested in the growth rate for large values of n (since almost all algorithms run fast for small values of n), so we were most interested in the $4n^2$ term, which determines how the function grows asymptotically for large n. Also, we do not care about constant factors (because we wanted simplicity and machine independence, and figured that the constant factors were better measured by implementing the algorithm). So we can ignored the factor 4 and simply say that the algorithm's worst-case running time grows asymptotically as n^2, which we wrote as $\Theta(n^2)$.

In this and the next lecture we will consider the questions of (1) how is it that one goes about analyzing the running time of an algorithm as function such as $T(n)$ above, and (2) how does one arrive at a simple asymptotic expression for that running time.

A Harder Example: Let's consider another example. Again, we will ignore stuff that takes constant time (expressed as "..." in the code below).

A Not-So-Simple Example:

```
for i = 1 to n {                      // assume that n is input size
    ...
    for j = 1 to 2*i {
        ...
        k = j;
        while (k >= 0) {
            ...
            k = k - 1;
        }
    }
}
```

How do we analyze the running time of an algorithm that has many complex nested loops? The answer is that we write out the loops as summations, and then try to solve the summations. Let $I()$, $M()$, $T()$ be the running times for (one full execution of) the inner loop, middle loop, and the entire program. To convert the loops into summations, we work from the inside-out. Let's consider one pass through the innermost loop. The number of passes through the loop depends on j. It is executed for $k = j, j-1, j-2, \ldots, 0$, and the time spent inside the loop is a constant, so the total time is just $j+1$. We could attempt to arrive at this more formally by expressing this as a summation:

$$I(j) = \sum_{k=0}^{j} 1 = j+1$$

Why the "1"? Because the stuff inside this loop takes constant time to execute. Why did we count up from 0 to j (and not down as the loop does?) The reason is that the mathematical notation for summations always goes from low index to high, and since addition is commutative it does not matter in which order we do the addition.

Now let us consider one pass through the middle loop. It's running time is determined by i. Using the summation we derived above for the innermost loop, and the fact that this loop is executed for j running from 1 to $2i$, it follows that the execution time is

$$M(i) = \sum_{j=1}^{2i} I(j) = \sum_{j=1}^{2i} (j+1).$$

Last time we gave the formula for the arithmetic series:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Our sum is not quite of the right form, but we can split it into two sums:

$$M(i) = \sum_{j=1}^{2i} j + \sum_{j=1}^{2i} 1.$$

The latter sum is clearly just $2i$. The former is an arithmetic series, and so we find can plug in $2i$ for n, and j for i in the formula above to yield the value:

$$M(i) = \frac{2i(2i+1)}{2} + 2i = \frac{4i^2 + 2i + 4i}{2} = 2i^2 + 3i.$$

Now, for the outermost sum and the running time of the entire algorithm we have

$$T(n) = \sum_{i=1}^{n} (2i^2 + 3i).$$

Splitting this up (by the linearity of addition) we have

$$T(n) = 2\sum_{i=1}^{n} i^2 + 3\sum_{i=1}^{n} i.$$

The latter sum is another arithmetic series, which we can solve by the formula above as $n(n+1)/2$. The former summation $\sum_{i=1}^{n} i^2$ is not one that we have seen before. Later, we'll show the following.

Quadratic Series: For $n \geq 0$.

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \cdots + n^2 = \frac{2n^3 + 3n^2 + n}{6}.$$

Assuming this fact for now, we conclude that the total running time is:

$$T(n) = 2\frac{2n^3 + 3n^2 + n}{6} + 3\frac{n(n+1)}{2},$$

which after some algebraic manipulations gives

$$T(n) = \frac{4n^3 + 15n^2 + 11n}{6}.$$

As before, we ignore all but the fastest growing term $4n^3/6$, and ignore constant factors, so the total running time is $\Theta(n^3)$.

Solving Summations: In the example above, we saw an unfamiliar summation, $\sum_{i=1}^{n} i^2$, which we claimed could be solved in closed form as:

$$\sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6}.$$

Solving a summation in *closed-form* means that you can write an exact formula for the summation without any embedded summations or asymptotic terms. In general, when you are presented with an unfamiliar summation, how do you approach solving it, or if not solving it in closed form, at least getting an asymptotic approximation. Here are a few ideas.

Use crude bounds: One of the simples approaches, that usually works for arriving at asymptotic bounds is to replace every term in the summation with a simple upper bound. For example, in $\sum_{i=1}^{n} i^2$ we could replace every term of the summation by the largest term. This would give

$$\sum_{i=1}^{n} i^2 \leq \sum_{i=1}^{n} n^2 = n^3.$$

Notice that this is asymptotically equal to the formula, since both are $\Theta(n^3)$.

This technique works pretty well with relatively slow growing functions (e.g., anything growing more slowly than than a polynomial, that is, i^c for some constant c). It does not give good bounds with faster growing functions, such as an exponential function like 2^i.

Approximate using integrals: Integration and summation are closely related. (Integration is in some sense a continuous form of summation.) Here is a handy formula. Let $f(x)$ be any *monotonically increasing function* (the function increases as x increases).

$$\int_0^n f(x)dx \leq \sum_{i=1}^{n} f(i) \leq \int_1^{n+1} f(x)dx.$$

Most running times are increasing functions of input size, so this formula is useful in analyzing algorithm running times.

Using this formula, we can approximate the above quadratic sum. In this case, $f(x) = x^2$.

$$\sum_{i=1}^{n} i^2 \leq \int_1^{n+1} x^2 dx = \left.\frac{x^3}{3}\right|_{x=1}^{n+1} = \frac{(n+1)^3}{3} - \frac{1}{3} = \frac{n^3 + 3n^2 + 3n}{3}.$$

Note that the constant factor on the leading term of $n^3/3$ is equal to the exact formula.

You might say, why is it easier to work with integrals than summations? The main reason is that most people have more experience in calculus than in discrete math, and there are many mathematics handbooks with lots of solved integrals.

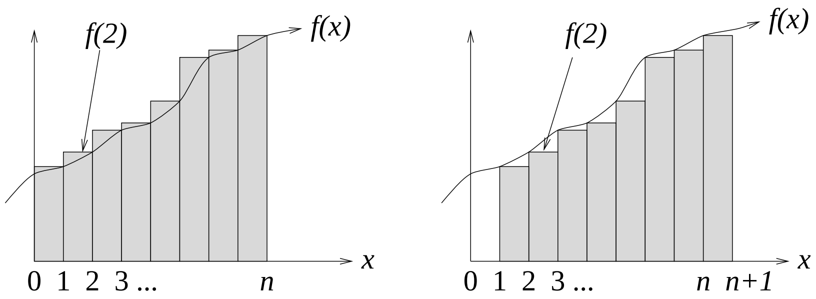


Figure 2: Approximating sums by integrals.

Use constructive induction: This is a fairly good method to apply whenever you can guess the general form of the summation, but perhaps you are not sure of the various constant factors. In this case, the integration formula suggests a solution of the form:

$$\sum_{i=1}^{n} i^2 = an^3 + bn^2 + cn + d,$$

but we do not know what a, b, c, and d are. However, we believe that they are constants (i.e., they are independent of n).

Let's try to prove this formula by induction on n, and as the proof proceeds, we should gather information about what the values of a, b, c, and d are.

Since this is the first induction proof we have done, let us recall how induction works. Basically induction proofs are just the mathematical equivalents of loops in programming. Let n be the integer variable on which we are performing the induction. The theorem or formula to be proved, called the *induction hypothesis* is a function of n, denote $IH(n)$. There is some smallest value n_0 for which $IH(n_0)$ is suppose to hold. We prove $IH(n_0)$, and then we work up to successively larger value of n, each time we may make use of the induction hypothesis, as long as we apply it to strictly smaller values of n.

```
Prove IH(n0);
for n = n0+1 to infinity do
    Prove IH(n), assuming that IH(n') holds for all n' < n;
```

This is sometimes called *strong induction*, because we assume that the hypothesis holds for all $n' < n$. Usually we only need to assume the induction hypothesis for the next smaller value of n, namely $n - 1$.

Basis Case: ($n = 0$) Recall that an empty summation is equal to the additive identity, 0. In this case we want to prove that $0 = a \cdot 0^3 + b \cdot 0^2 + c \cdot 0 + d$. For this to be true, we must have $d = 0$.

Induction Step: Let us assume that $n > 0$, and that the formula holds for all values $n' < n$, and from this we will show that the formula holds for the value n itself.

The structure of proving summations by induction is almost always the same. First, write the summation for i running up to n, then strip off the last term, apply the induction hypothesis on the summation running up to $n - 1$, and then combine everything algebraically. Here we go.

$$\begin{aligned}\sum_{i=1}^{n} i^2 &= \left(\sum_{i=1}^{n-1} i^2\right) + n^2 \\ &= a(n-1)^3 + b(n-1)^2 + c(n-1) + d + n^2\end{aligned}$$

$$\begin{aligned} &= (an^3 - 3an^2 + 3an - a) + (bn^2 - 2bn + b) + (cn - c) + d + n^2 \\ &= an^3 + (-3a + b + 1)n^2 + (3a - 2b + c)n + (-a + b - c + d). \end{aligned}$$

To complete the proof, we want this is equal to $an^3 + bn^2 + cn + d$. Since this should be true for all n, this means that each power of n must match identically. This gives us the following constraints

$$a = a, \quad b = -3a + b + 1, \quad c = 3a - 2b + c, \quad d = -a + b - c + d.$$

We already know that $d = 0$ from the basis case. From the second constraint above we can cancel b from both sides, implying that $a = 1/3$. Combining this with the third constraint we have $b = 1/2$. Finally from the last constraint we have $c = -a+b = 1/6$. This gives the final formula

$$\sum_{i=1}^{n} i^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} = \frac{2n^3 + 3n^2 + n}{6}.$$

As desired, all of the values a through d are constants, independent of n. If we had chosen the wrong general form, then either we would find that some of these "constants" depended on n, or we might get a set of constraints that could not be satisfied.

Notice that constructive induction gave us the exact formula for the summation. The only tricky part is that we had to "guess" the general structure of the solution.

In summary, there is no one way to solve a summation. However, there are many tricks that can be applied to either find asymptotic approximations or to get the exact solution. The ultimate goal is to come up with a close-form solution. This is not always easy or even possible, but for our purposes asymptotic bounds will usually be good enough.

Lecture 4: 2-d Maxima Revisited and Asymptotics

(Thursday, Feb 5, 1998)

Read: Chapts. 2 and 3 in CLR.

2-dimensional Maxima Revisited: Recall the max-dominance problem from the previous lectures. A point p is said to *dominated by* point q if $p.x \le q.x$ and $p.y \le q.y$. Given a set of n points, $P = \{p_1, p_2, \ldots, p_n\}$ in 2-space a point is said to be *maximal* if it is not dominated by any other point in P. The problem is to output all the maximal points of P.

So far we have introduced a simple brute-force algorithm that ran in $\Theta(n^2)$ time, which operated by comparing all pairs of points. The question we consider today is whether there is an approach that is significantly better?

The problem with the brute-force algorithm is that uses no intelligence in pruning out decisions. For example, once we know that a point p_i is dominated by another point p_j, then we we do not need to use p_i for eliminating other points. Any point that p_i dominates will also be dominated by p_j. (This follows from the fact that the domination relation is *transitive*, which can easily be verified.) This observation by itself, does not lead to a significantly faster algorithm though. For example, if all the points are maximal, which can certainly happen, then this optimization saves us nothing.

Plane-sweep Algorithm: The question is whether we can make an significant improvement in the running time? Here is an idea for how we might do it. We will sweep a vertical line across the plane from left to right. As we sweep this line, we will build a structure holding the maximal points lying to the left of the sweep line. When the sweep line reaches the rightmost point of P, then we will have constructed the complete set of maxima. This approach of solving geometric problems by sweeping a line across the plane is called *plane sweep*.

Although we would like to think of this as a continuous process, we need some way to perform the plane sweep in discrete steps. To do this, we will begin by sorting the points in increasing order of their x-coordinates. For simplicity, let us assume that no two points have the same y-coordinate. (This limiting assumption is actually easy to overcome, but it is good to work with the simpler version, and save the messy details for the actual implementation.) Then we will advance the sweep-line from point to point in n discrete steps. As we encounter each new point, we will update the current list of maximal points.

First off, how do we sort the points? We will leave this problem for later in the semester. But the bottom line is that there exist any number of good sorting algorithms whose running time to sort n values is $\Theta(n \log n)$. We will just assume that they exist for now.

So the only remaining problem is, how do we store the existing maximal points, and how do we update them when a new point is processed? We claim that as each new point is added, it must be maximal for the current set. (Why? Beacuse its x-coordinate is larger than all the x-coordinates of all the existing points, and so it cannot be dominated by any of the existing points.) However, this new point may dominate some of the existing maximal points, and so we may need to delete them from the list of maxima. (Notice that once a point is deleted as being nonmaximal, it will never need to be added back again.) Consider the figure below.

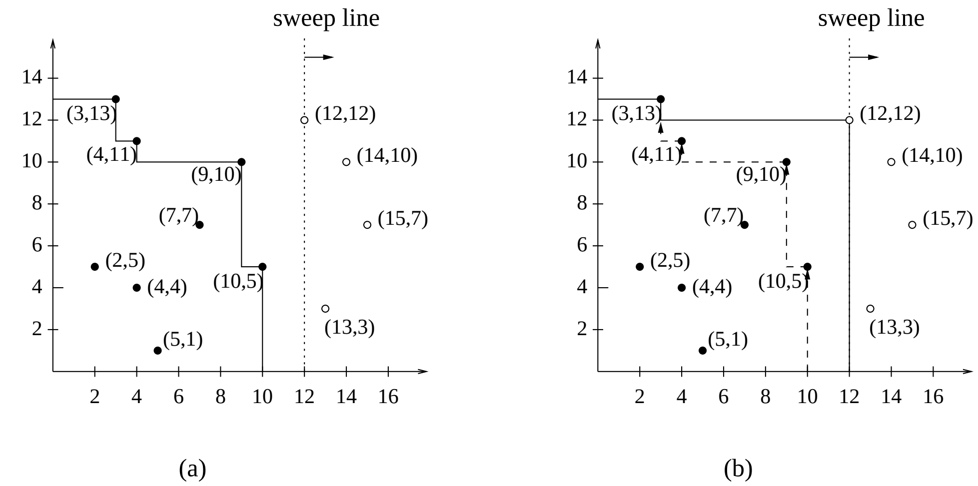


Figure 3: Plane sweep algorithm for 2-d maxima.

Let p_i denote the current point being considered. Notice that since the p_i has greater x-coordinate than all the existing points, it dominates an existing point if and only if its y-coordinate is also larger (or equal). Thus, among the existing maximal points, we want to find those having smaller (or equal) y-coordinate, and eliminate them.

At this point, we need to make an important observation about how maximal points are ordered with respect to the x- and y-coordinates. As we read maximal points from left to right (in order of increasing x-coordinates) the y-coordinates appear in decreasing order. Why is this so? Suppose to the contrary, that we had two maximal points p and q, with $p.x \geq q.x$ but

$p.y \geq q.y$. Then it would follow that q is dominated by p, and hence it is not maximal, a contradiction.

This is nice, because it implies that if we store the existing maximal points in a list, the points that p_i dominates (if any) will all appear at the end of this list. So we have to scan this list to find the breakpoint between the maximal and dominated points. The question is how do we do this?

I claim that we can simply scan the list linearly. But we must do the scan in the proper direction for the algorithm to be efficient. Which direction should we scan the list of current maxima? From left to right, until finding the first point that is not dominated, or from right to left, until finding the first point that is dominated? Stop here and think about it for a moment. If you can answer this question correctly, then it says something about your intuition for designing efficient algorithms. Let us assume that we are trying to optimize worst-case performance.

The correct answer is to scan the list from left to right. Here is why. If you only encounter one point in the scan, then the scan will always be very efficient. The danger is that you may scan many points before finding the proper breakpoint. If we scan the list from left to right, then every point that we encounter whose y-coordinate is less than p_i's will be dominated, and hence it will be eliminated from the computation forever. We will never have to scan this point again. On the other hand, if we scan from left to right, then in the worst case (consider when all the points are maximal) we may rescan the same points over and over again. This will lead to an $\Theta(n^2)$ algorithm

Now we can give the pseudocode for the final plane sweep algorithm. Since we add maximal points onto the end of the list, and delete them from the end of the list, we can use a stack to store the maximal points, where the top of the stack contains the point with the highest x-coordinate. Let S denote this stack. The top element of the stack is denoted $S.top$. Popping the stack means removing the top element.

Plane Sweep Maxima

```
Maxima2(int n, Point P[1..n]) {
    Sort P in ascending order by x-coordinate;
    S = empty;                             // initialize stack of maxima
    for i = 1 to n do {                    // add points in order of x-coordinate
        while (S is not empty and S.top.y <= P[i].y)
            Pop(S);                        // remove points that P[i] dominates
        Push(S, P[i]);                     // add P[i] to stack of maxima
    }
    output the contents of S;
}
```

Why is this algorithm correct? The correctness follows from the discussion up to now. The most important element was that since the current maxima appear on the stack in decreasing order of x-coordinates (as we look down from the top of the stack), they occur in increasing order of y-coordinates. Thus, as soon as we find the last undominated element in the stack, it follows that everyone else on the stack is undominated.

Analysis: This is an interesting program to analyze, primarily because the techniques that we discussed in the last lecture do *not* apply readily here. I claim that after the sorting (which we mentioned takes $\Theta(n \log n)$ time), the rest of the algorithm only takes $\Theta(n)$ time. In particular, we have two nested loops. The outer loop is clearly executed n times. The inner while-loop could be iterated up to $n-1$ times in the worst case (in particular, when the last point added dominates all the others). So, it seems that though we have $n(n-1)$ for a total of $\Theta(n^2)$.

However, this is a good example of how not to be fooled by analyses that are too simple minded. Although it is true that the inner while-loop could be executed as many as $n-1$ times any one time through the outer loop, over the entire course of the algorithm we claim that it cannot be executed more than n times. Why is this? First observe that the total number of elements that have ever been pushed onto the stack is at most n, since we execute exactly one Push for each time through the outer for-loop. Also observe that every time we go through the inner while-loop, we must pop an element off the stack. It is impossible to pop more elements off the stack than are ever pushed on. Therefore, the inner while-loop cannot be executed more than n times over the entire course of the algorithm. (Make sure that you believe the argument before going on.)

Therefore, since the total number of iterations of the inner while-loop is n, and since the total number of iterations in the outer for-loop is n, the total running time of the algorithm is $\Theta(n)$.

Is this really better? How much of an improvement is this plane-sweep algorithm over the brute-force algorithm? Probably the most accurate way to find out would be to code the two up, and compare their running times. But just to get a feeling, let's look at the ratio of the running times. (We have ignored constant factors, but we'll see that they cannot play a very big role.)

We have argued that the brute-force algorithm runs in $\Theta(n^2)$ time, and the improved plane-sweep algorithm runs in $\Theta(n \log n)$ time. What is the base of the logarithm? It turns out that it will not matter for the asymptotics (we'll show this later), so for concreteness, let's assume logarithm base 2, which we'll denote as $\lg n$. The ratio of the running times is:

$$\frac{n^2}{n \lg n} = \frac{n}{\lg n}.$$

For relatively small values of n (e.g. less than 100), both algorithms are probably running fast enough that the difference will be practically negligible. On larger inputs, say, $n = 1,000$, the ratio of n to $\lg n$ is about $1000/10 = 100$, so there is a 100-to-1 ratio in running times. Of course, we have not considered the constant factors. But since neither algorithm makes use of very complex constructs, it is hard to imagine that the constant factors will differ by more than, say, a factor of 10. For even larger inputs, say, $n = 1,000,000$, we are looking at a ratio of roughly $1,000,000/20 = 50,000$. This is quite a significant difference, irrespective of the constant factors.

For example, suppose that there was a constant factor difference of 10 to 1, in favor of the brute-force algorithm. The plane-sweep algorithm would still be 5,000 times faster. If the plane-sweep algorithm took, say 10 seconds to execute, then the brute-force algorithm would take 14 hours.

From this we get an idea about the importance of asymptotic analysis. It tells us which algorithm is better for large values of n. As we mentioned before, if n is not very large, then almost any algorithm will be fast. But efficient algorithm design is most important for large inputs, and the general rule of computing is that input sizes continue to grow until people can no longer tolerate the running times. Thus, by designing algorithms efficiently, you make it possible for the user to run large inputs in a reasonable amount of time.

Asymptotic Notation: We continue to use the notation $\Theta()$ but have never defined it. Let's remedy this now.

Definition: Given any function $g(n)$, we define $\Theta(g(n))$ to be a set of functions that are *asymptotically equivalent* to $g(n)$, or put formally:

$$\Theta(g(n)) = \{f(n) \mid \text{there exist positive constants } c_1,\ c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$

Your response at this point might be, "I'm sorry that I asked". It seems that the reasonably simple concept of "throw away all but the fastest growing term, and ignore constant factors" should have a simpler and more intuitive definition than this. Unfortunately, it does not (although later we will see that there is somewhat easier, and nearly equivalent definition).

First off, we can see that we have been misusing the notation. We have been saying things like $T(n) = \Theta(n^2)$. This cannot be true. The left side is a function, and right side is a set of functions. This should properly be written as $T(n) \in \Theta(n^2)$. However, this abuse of notation is so common in the field of algorithm design, that no one notices it.

Going back to an earlier lecture, recall that we argued that the brute-force algorithm for 2-d maxima had a running time of $T(n) = 4n^2 + 2n$, which we claimed was $\Theta(n^2)$. Let's verify that this is so. In this case $g(n) = n^2$. We want to show that $f(n) = 4n^2 + 2n$ is a member of this set, which means that we must argue that there exist constants c_1, c_2, and n_0 such that

$$0 \le c_1 n^2 \le (4n^2 + 2n) \le c_2 n^2 \qquad \text{for all } n \ge n_0.$$

There are really three inequalities here. The constraint $0 \le c_1 n^2$ is no problem, since we will always be dealing with positive n and positive constants. The next is:

$$c_1 n^2 \le 4n^2 + 2n.$$

If we set $c_1 = 4$, then we have $0 \le 4n^2 \le 4n^2 + 2n$, which is clearly true as long as $n \ge 0$. The other inequality is

$$4n^2 + 2n \le c_2 n^2.$$

If we select $c_2 = 6$, and assume that $n \ge 1$, then we have $n^2 \ge n$, implying that

$$4n^2 + 2n \le 4n^2 + 2n^2 = 6n^2 = c_2 n^2.$$

We have two constraints on n, $n \ge 0$ and $n \ge 1$. So let us make $n_0 = 1$, which will imply that we as long as $n \ge n_0$, we will satisfy both of these constraints.

Thus, we have given a formal proof that $4n^2 + 2n \in \Theta(n^2)$, as desired. Next time we'll try to give some of the intuition behind this definition.

Lecture 5: Asymptotics

(Tuesday, Feb 10, 1998)

Read: Chapt. 3 in CLR. The Limit Rule is not really covered in the text. Read Chapt. 4 for next time.

Asymptotics: We have introduced the notion of $\Theta()$ notation, and last time we gave a formal definition. Today, we will explore this and other asymptotic notations in greater depth, and hopefully give a better understanding of what they mean.

Θ-Notation: Recall the following definition from last time.

Definition: Given any function $g(n)$, we define $\Theta(g(n))$ to be a set of functions:

$$\Theta(g(n)) = \{f(n) \mid \text{there exist strictly positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$

Let's dissect this definition. Intuitively, what we want to say with "$f(n) \in \Theta(g(n))$" is that $f(n)$ and $g(n)$ are *asymptotically equivalent*. This means that they have essentially the same growth rates for large n. For example, functions like $4n^2$, $(8n^2 + 2n - 3)$, $(n^2/5 + \sqrt{n} - 10 \log n)$, and $n(n-3)$ are all intuitively asymptotically equivalent, since as n becomes large, the dominant (fastest growing) term is some constant times n^2. In other words, they all grow *quadratically* in n. The portion of the definition that allows us to select c_1 and c_2 is essentially saying "the constants do not matter because you may pick c_1 and c_2 however you like to satisfy these conditions." The portion of the definition that allows us to select n_0 is essentially saying "we are only interested in large n, since you only have to satisfy the condition for all n bigger than n_0, and you may make n_0 as big a constant as you like."

An example: Consider the function $f(n) = 8n^2 + 2n - 3$. Our informal rule of keeping the largest term and throwing away the constants suggests that $f(n) \in \Theta(n^2)$ (since f grows quadratically). Let's see why the formal definition bears out this informal observation.

We need to show two things: first, that $f(n)$ does grows asymptotically at least as fast as n^2, and second, that $f(n)$ grows no faster asymptotically than n^2. We'll do both very carefully.

Lower bound: $f(n)$ grows asymptotically at least as fast as n^2: This is established by the portion of the definition that reads: (paraphrasing): "there exist positive constants c_1 and n_0, such that $f(n) \geq c_1 n^2$ for all $n \geq n_0$." Consider the following (almost correct) reasoning:

$$f(n) = 8n^2 + 2n - 3 \geq 8n^2 - 3 = 7n^2 + (n^2 - 3) \geq 7n^2 = 7n^2.$$

Thus, if we set $c_1 = 7$, then we are done. But in the above reasoning we have implicitly made the assumptions that $2n \geq 0$ and $n^2 - 3 \geq 0$. These are not true for all n, but they are true for all sufficiently large n. In particular, if $n \geq \sqrt{3}$, then both are true. So let us select $n_0 \geq \sqrt{3}$, and now we have $f(n) \geq c_1 n^2$, for all $n \geq n_0$, which is what we need.

Upper bound: $f(n)$ grows asymptotically no faster than n^2: This is established by the portion of the definition that reads "there exist positive constants c_2 and n_0, such that $f(n) \leq c_2 n^2$ for all $n \geq n_0$." Consider the following reasoning (which is almost correct):

$$f(n) = 8n^2 + 2n - 3 \leq 8n^2 + 2n \leq 8n^2 + 2n^2 = 10n^2.$$

This means that if we let $c_2 = 10$, then we are done. We have implicitly made the assumption that $2n \leq 2n^2$. This is not true for all n, but it is true for all $n \geq 1$. So, let us select $n_0 \geq 1$, and now we have $f(n) \leq c_2 n^2$ for all $n \geq n_0$, which is what we need.

From the lower bound, we have $n_0 \geq \sqrt{3}$ and from the upper bound we have $n_0 \geq 1$, and so combining these we let n_0 be the larger of the two: $n_0 = \sqrt{3}$. Thus, in conclusion, if we let $c_1 = 7$, $c_2 = 10$, and $n_0 = \sqrt{3}$, then we have

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \text{for all } n \geq n_0,$$

and this is exactly what the definition requires. Since we have shown (by construction) the existence of constants c_1, c_2, and n_0, we have established that $f(n) \in n^2$. (Whew! That was a lot more work than just the informal notion of throwing away constants and keeping the largest term, but it shows how this informal notion is implemented formally in the definition.)

Now let's show why $f(n)$ is not in some other asymptotic class. First, let's show that $f(n) \notin \Theta(n)$. If this were true, then we would have to satisfy both the upper and lower bounds. It turns out that the lower bound is satisfied (because $f(n)$ grows at least as fast asymptotically as n). But the upper bound is false. In particular, the upper bound requires us to show "there exist positive constants c_2 and n_0, such that $f(n) \le c_2 n$ for all $n \ge n_0$." Informally, we know that as n becomes large enough $f(n) = 8n^2 + 2n - 3$ will eventually exceed $c_2 n$ no matter how large we make c_2 (since $f(n)$ is growing quadratically and $c_2 n$ is only growing linearly). To show this formally, suppose towards a contradiction that constants c_2 and n_0 did exist, such that $8n^2 + 2n - 3 \le c_2 n$ for all $n \ge n_0$. Since this is true for all sufficiently large n then it must be true in the limit as n tends to infinity. If we divide both side by n we have:

$$\lim_{n\to\infty}\left(8n + 2 - \frac{3}{n}\right) \le c_2.$$

It is easy to see that in the limit the left side tends to ∞, and so no matter how large c_2 is, this statement is violated. This means that $f(n) \notin \Theta(n)$.

Let's show that $f(n) \notin \Theta(n^3)$. Here the idea will be to violate the lower bound: "there exist positive constants c_1 and n_0, such that $f(n) \ge c_1 n^3$ for all $n \ge n_0$." Informally this is true because $f(n)$ is growing quadratically, and eventually any cubic function will exceed it. To show this formally, suppose towards a contradiction that constants c_1 and n_0 did exist, such that $8n^2 + 2n - 3 \ge c_1 n^3$ for all $n \ge n_0$. Since this is true for all sufficiently large n then it must be true in the limit as n tends to infinity. If we divide both side by n^3 we have:

$$\lim_{n\to\infty}\left(\frac{8}{n} + \frac{2}{n^2} - \frac{3}{n^3}\right) \ge c_1.$$

It is easy to see that in the limit the left side tends to 0, and so the only way to satisfy this requirement is to set $c_1 = 0$, but by hypothesis c_1 is positive. This means that $f(n) \notin \Theta(n^3)$.

O-notation and Ω-notation: We have seen that the definition of Θ-notation relies on proving both a lower and upper asymptotic bound. Sometimes we are only interested in proving one bound or the other. The O-notation allows us to state asymptotic upper bounds and the Ω-notation allows us to state asymptotic lower bounds.

Definition: Given any function $g(n)$,

$$\begin{aligned} O(g(n)) = \{f(n) \quad | \quad & \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ & 0 \le f(n) \le c g(n) \text{ for all } n \ge n_0\}. \end{aligned}$$

Definition: Given any function $g(n)$,

$$\begin{aligned} \Omega(g(n)) = \{f(n) \quad | \quad & \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ & 0 \le c g(n) \le f(n) \text{ for all } n \ge n_0\}. \end{aligned}$$

Compare this with the definition of Θ. You will see that O-notation only enforces the upper bound of the Θ definition, and Ω-notation only enforces the lower bound. Also observe that $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$. Intuitively, $f(n) \in O(g(n))$ means that $f(n)$ grows asymptotically at the same rate or slower than $g(n)$. Whereas, $f(n) \in O(g(n))$ means that $f(n)$ grows asymptotically at the same rate or faster than $g(n)$.

For example $f(n) = 3n^2 + 4n \in \Theta(n^2)$ but it is not in $\Theta(n)$ or $\Theta(n^3)$. But $f(n) \in O(n^2)$ and in $O(n^3)$ but not in $O(n)$. Finally, $f(n) \in \Omega(n^2)$ and in $\Omega(n)$ but not in $\Omega(n^3)$.

The Limit Rule for Θ: The previous examples which used limits suggest alternative way of showing that $f(n) \in \Theta(g(n))$.

Limit Rule for Θ-notation: Given positive functions $f(n)$ and $g(n)$, if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,$$

for some constant $c > 0$ (strictly positive but not infinity), then $f(n) \in \Theta(g(n))$.

Limit Rule for O-notation: Given positive functions $f(n)$ and $g(n)$, if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,$$

for some constant $c \geq 0$ (nonnegative but not infinite), then $f(n) \in O(g(n))$.

Limit Rule for Ω-notation: Given positive functions $f(n)$ and $g(n)$, if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$$

(either a strictly positive constant or infinity) then $f(n) \in \Omega(g(n))$.

This limit rule can be applied in almost every instance (that I know of) where the formal definition can be used, and it is almost always easier to apply than the formal definition. The only exceptions that I know of are strange instances where the limit does not exist (e.g. $f(n) = n^{(1+\sin n)}$). But since most running times are fairly well-behaved functions this is rarely a problem.

You may recall the important rules from calculus for evaluating limits. (If not, dredge out your old calculus book to remember.) Most of the rules are pretty self evident (e.g., the limit of a finite sum is the sum of the individual limits). One important rule to remember is the following:

L'Hôpital's rule: If $f(n)$ and $g(n)$ both approach 0 or both approach ∞ in the limit, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)},$$

where $f'(n)$ and $g'(n)$ denote the derivatives of f and g relative to n.

Polynomial Functions: Using the Limit Rule it is quite easy to analyze polynomial functions.

Lemma: Let $f(n) = 2n^4 - 5n^3 - 2n^2 + 4n - 7$. Then $f(n) \in \Theta(n^4)$.

Proof: This would be quite tedious to do by the formal definition. Using the limit rule we have:

$$\lim_{n\to\infty} \frac{f(n)}{n^4} = \lim_{n\to\infty} \left(2 - \frac{5}{n} - \frac{2}{n^2} + \frac{4}{n^3} - \frac{7}{n^4}\right) = 2 - 0 - 0 + 0 - 0 = 2.$$

Since 2 is a strictly positive constant it follows from the limit rule that $f(n) \in \Theta(n^2)$.

In fact, it is easy to generalize this to arbitrary polynomials.

Theorem: Consider any asymptotically positive polynomial of degree $p(n) = \sum_{i=0}^{d} a_i n^i$, where $a_d > 0$. Then $p(n) \in \Theta(n^d)$.

From this, the informal rule of "keep the largest term and throw away the constant factors" is now much more evident.

Exponentials and Logarithms: Exponentials and logarithms are very important in analyzing algorithms. The following are nice to keep in mind. The terminology $\lg^b n$ means $(\lg n)^b$.

Lemma: Given any positive constants $a > 1$, b, and c:

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0 \qquad\qquad \lim_{n\to\infty} \frac{\lg^b n}{n^c} = 0.$$

We won't prove these, but they can be shown by taking appropriate powers, and then applying L'Hôpital's rule. The important bottom line is that polynomials always grow more slowly than exponentials whose base is greater than 1. For example:

$$n^{500} \in O(2^n).$$

For this reason, we will try to avoid exponential running times at all costs. Conversely, logarithmic powers (sometimes called *polylogarithmic functions*) grow more slowly than any polynomial. For example:

$$\lg^{500} n \in O(n).$$

For this reason, we will usually be happy to allow any number of additional logarithmic factors, if it means avoiding any additional powers of n.

At this point, it should be mentioned that these last observations are really asymptotic results. They are true in the limit for large n, but you should be careful just how high the crossover point is. For example, by my calculations, $\lg^{500} n \le n$ only for $n > 2^{6000}$ (which is much larger than input size you'll ever see). Thus, you should take this with a grain of salt. But, for small powers of logarithms, this applies to all reasonably large input sizes. For example $\lg^2 n \le n$ for all $n \ge 16$.

Asymptotic Intuition: To get a intuitive feeling for what common asymptotic running times map into in terms of practical usage, here is a little list.

- $\Theta(1)$: Constant time; you can't beat it!
- $\Theta(\log n)$: This is typically the speed that most efficient data structures operate in for a single access. (E.g., inserting a key into a balanced binary tree.) Also it is the time to find an object in a sorted list of length n by binary search.
- $\Theta(n)$: This is about the fastest that an algorithm can run, given that you need $\Theta(n)$ time just to read in all the data.
- $\Theta(n \log n)$: This is the running time of the best sorting algorithms. Since many problems require sorting the inputs, this is still considered quite efficient.
- $\Theta(n^2), \Theta(n^3), \ldots$.: Polynomial time. These running times are acceptable either when the exponent is small or when the data size is not too large (e.g. $n \le 1,000$).
- $\Theta(2^n), \Theta(3^n)$: Exponential time. This is only acceptable when either (1) your know that you inputs will be of very small size (e.g. $n \le 50$), or (2) you know that this is a worst-case running time that will rarely occur in practical instances. In case (2), it would be a good idea to try to get a more accurate average case analysis.
- $\Theta(n!), \Theta(n^n)$: Acceptable only for really small inputs (e.g. $n \le 20$).

Are their even bigger functions. You betcha! For example, if you want to see a function that grows inconceivably fast, look up the definition of Ackerman's function in our book.

Lecture 6: Divide and Conquer and MergeSort

(Thursday, Feb 12, 1998)
Read: Chapt. 1 (on MergeSort) and Chapt. 4 (on recurrences).

Divide and Conquer: The ancient Roman politicians understood an important principle of good algorithm design (although they were probably not thinking about algorithms at the time). You divide your enemies (by getting them to distrust each other) and then conquer them piece by piece. This is called *divide-and-conquer*. In algorithm design, the idea is to take a problem on a large input, break the input into smaller pieces, solve the problem on each of the small pieces, and then combine the piecewise solutions into a global solution. But once you have broken the problem into pieces, how do you solve these pieces? The answer is to apply divide-and-conquer to them, thus further breaking them down. The process ends when you are left with such tiny pieces remaining (e.g. one or two items) that it is trivial to solve them.

Summarizing, the main elements to a divide-and-conquer solution are

- Divide (the problem into a small number of pieces),
- Conquer (solve each piece, by applying divide-and-conquer recursively to it), and
- Combine (the pieces together into a global solution).

There are a huge number computational problems that can be solved efficiently using divide-and-conquer. In fact the technique is so powerful, that when someone first suggests a problem to me, the first question I usually ask (after what is the brute-force solution) is "does there exist a divide-and-conquer solution for this problem?"

Divide-and-conquer algorithms are typically recursive, since the conquer part involves invoking the same technique on a smaller subproblem. Analyzing the running times of recursive programs is rather tricky, but we will show that there is an elegant mathematical concept, called a *recurrence*, which is useful for analyzing the sort of recursive programs that naturally arise in divide-and-conquer solutions. For the next couple of lectures we will discuss some examples of divide-and-conquer algorithms, and how to analyze them using recurrences.

MergeSort: The first example of a divide-and-conquer algorithm which we will consider is perhaps the best known. This is a simple and very efficient algorithm for sorting a list of numbers, called *MergeSort*. We are given an sequence of n numbers A, which we will assume is stored in an array $A[1 \ldots n]$. The objective is to output a permutation of this sequence, sorted in increasing order. This is normally done by permuting the elements within the array A.

How can we apply divide-and-conquer to sorting? Here are the major elements of the MergeSort algorithm.

Divide: Split A down the middle into two subsequences, each of size roughly $n/2$.

Conquer: Sort each subsequence (by calling MergeSort recursively on each).

Combine: Merge the two sorted subsequences into a single sorted list.

The dividing process ends when we have split the subsequences down to a single item. An sequence of length one is trivially sorted. The key operation where all the work is done is in the combine stage, which merges together two sorted lists into a single sorted list. It turns out that the merging process is quite easy to implement.

The following figure gives a high-level view of the algorithm. The "divide" phase is shown on the left. It works top-down splitting up the list into smaller sublists. The "conquer and combine" phases are shown on the right. They work bottom-up, merging sorted lists together into larger sorted lists.

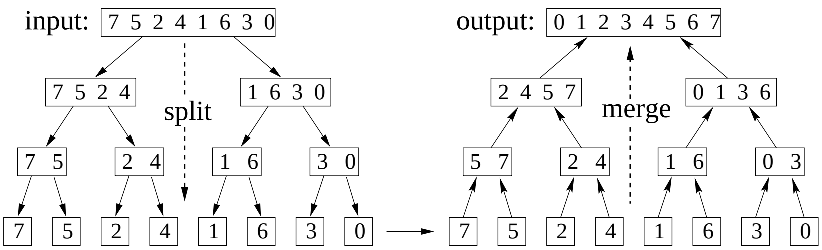


Figure 4: MergeSort.

MergeSort: Let's design the algorithm top-down. We'll assume that the procedure that merges two sorted list is available to us. We'll implement it later. Because the algorithm is called recursively on sublists, in addition to passing in the array itself, we will pass in two indices, which indicate the first and last indices of the subarray that we are to sort. The call `MergeSort(A, p, r)` will sort the subarray $A[p..r]$ and return the sorted result in the same subarray.

Here is the overview. If $r = p$, then this means that there is only one element to sort, and we may return immediately. Otherwise (if $p < r$) there are at least two elements, and we will invoke the divide-and-conquer. We find the index q, midway between p and r, namely $q = (p + r)/2$ (rounded down to the nearest integer). Then we split the array into subarrays $A[p..q]$ and $A[q+1..r]$. (We need to be careful here. Why would it be wrong to do $A[p..q-1]$ and $A[q..r]$? Suppose $r = p + 1$.) Call MergeSort recursively to sort each subarray. Finally, we invoke a procedure (which we have yet to write) which merges these two subarrays into a single sorted array.

MergeSort

```
MergeSort(array A, int p, int r) {
    if (p < r) {                          // we have at least 2 items
        q = (p + r)/2
        MergeSort(A, p, q)                // sort A[p..q]
        MergeSort(A, q+1, r)              // sort A[q+1..r]
        Merge(A, p, q, r)                 // merge everything together
    }
}
```

Merging: All that is left is to describe the procedure that merges two sorted lists. `Merge(A, p, q, r)` assumes that the left subarray, $A[p..q]$, and the right subarray, $A[q+1..r]$, have already been sorted. We merge these two subarrays by copying the elements to a temporary working array called B. For convenience, we will assume that the array B has the same index range A, that is, $B[p..r]$. (One nice thing about pseudocode, is that we can make these assumptions, and leave them up to the programmer to figure out how to implement it.) We have to indices i and j, that point to the current elements of each subarray. We move the smaller element into the next position of B (indicated by index k) and then increment the corresponding index (either i or j). When we run out of elements in one array, then we just copy the rest of the other array into B. Finally, we copy the entire contents of B back into A. (The use of the temporary array is a bit unpleasant, but this is impossible to overcome entirely. It is one of the shortcomings of MergeSort, compared to some of the other efficient sorting algorithms.)

In case you are not aware of C notation, the operator `i++` returns the current value of i, and then increments this variable by one.

Merge

```
Merge(array A, int p, int q, int r) {        // merges A[p..q] with A[q+1..r]
    array B[p..r]
    i = k = p                                // initialize pointers
    j = q+1
    while (i <= q and j <= r) {              // while both subarrays are nonempty
        if (A[i] <= A[j]) B[k++] = A[i++]    // copy from left subarray
        else              B[k++] = A[j++]    // copy from right subarray
    }
    while (i <= q) B[k++] = A[i++]           // copy any leftover to B
    while (j <= r) B[k++] = A[j++]
    for i = p to r do A[i] = B[i]            // copy B back to A
}
```

This completes the description of the algorithm. Observe that of the last two while-loops in the Merge procedure, only one will be executed. (Do you see why?)

If you find the recursion to be a bit confusing. Go back and look at the earlier figure. Convince yourself that as you unravel the recursion you are essentially walking through the tree (the *recursion tree*) shown in the figure. As calls are made you walk down towards the leaves, and as you return you are walking up towards the root. (We have drawn two trees in the figure, but this is just to make the distinction between the inputs and outputs clearer.)

Discussion: One of the little tricks in improving the running time of this algorithm is to avoid the constant copying from A to B and back to A. This is often handled in the implementation by using two arrays, both of equal size. At odd levels of the recursion we merge from subarrays of A to a subarray of B. At even levels we merge from from B to A. If the recursion has an odd number of levels, we may have to do one final copy from B back to A, but this is faster than having to do it at every level. Of course, this only improves the constant factors; it does not change the asymptotic running time.

Another implementation trick to speed things by a constant factor is that rather than driving the divide-and-conquer all the way down to subsequences of size 1, instead stop the dividing process when the sequence sizes fall below constant, e.g. 20. Then invoke a simple $\Theta(n^2)$ algorithm, like insertion sort on these small lists. Often brute force algorithms run faster on small subsequences, because they do not have the added overhead of recursion. Note that since they are running on subsequences of size at most 20, the running times is $\Theta(20^2) = \Theta(1)$. Thus, this will not affect the overall asymptotic running time.

It might seem at first glance that it should be possible to merge the lists "in-place", without the need for additional temporary storage. The answer is that it is, but it no one knows how to do it without destroying the algorithm's efficiency. It turns out that there are faster ways to sort numbers in-place, e.g. using either HeapSort or QuickSort.

Here is a subtle but interesting point to make regarding this sorting algorithm. Suppose that in the if-statement above, we have $A[i] = A[j]$. Observe that in this case we copy from the left sublist. Would it have mattered if instead we had copied from the right sublist? The simple answer is no—since the elements are equal, they can appear in either order in the final sublist. However there is a subtler reason to prefer this particular choice. Many times we are sorting data that does not have a single attribute, but has many attributes (name, SSN, grade, etc.) Often the list may already have been sorted on one attribute (say, name). If we sort on a second attribute (say, grade), then it would be nice if people with same grade are still sorted by name. A sorting algorithm that has the property that equal items will appear in the final sorted list in the same relative order that they appeared in the initial input is called a *stable* *sorting algorithm.* This is a nice property for a sorting algorithm to have. By favoring elements from the left sublist over the right, we will be preserving the relative order of elements. It can be shown that as a result, MergeSort is a stable sorting algorithm. (This is not immediate, but it can be proved by induction.)

Analysis: What remains is to analyze the running time of MergeSort. First let us consider the running time of the procedure `Merge(A, p, q, r)`. Let $n = r - p + 1$ denote the total length of both the left and right subarrays. What is the running time of Merge as a function of n? The algorithm contains four loops (none nested in the other). It is easy to see that each loop can be executed at most n times. (If you are a bit more careful you can actually see that all the while-loops together can only be executed n times in total, because each execution copies one new element to the array B, and B only has space for n elements.) Thus the running time to Merge n items is $\Theta(n)$. Let us write this without the asymptotic notation, simply as n. (We'll see later why we do this.)

Now, how do we describe the running time of the entire MergeSort algorithm? We will do this through the use of a *recurrence*, that is, a function that is defined recursively in terms of itself. To avoid circularity, the recurrence for a given value of n is defined in terms of values that are strictly smaller than n. Finally, a recurrence has some basis values (e.g. for $n = 1$), which are defined explicitly.

Let's see how to apply this to MergeSort. Let $T(n)$ denote the worst case running time of MergeSort on an array of length n. For concreteness we could count whatever we like: number of lines of pseudocode, number of comparisons, number of array accesses, since these will only differ by a constant factor. Since all of the real work is done in the Merge procedure, we will count the total time spent in the Merge procedure.

First observe that if we call MergeSort with a list containing a single element, then the running time is a constant. Since we are ignoring constant factors, we can just write $T(n) = 1$. When we call MergeSort with a list of length $n > 1$, e.g. `Merge(A, p, r)`, where $r - p + 1 = n$, the algorithm first computes $q = \lfloor (p + r)/2 \rfloor$. The subarray $A[p..q]$, which contains $q - p + 1$ elements. You can verify (by some tedious floor-ceiling arithmetic, or simpler by just trying an odd example and an even example) that is of size $\lceil n/2 \rceil$. Thus the remaining subarray $A[q + 1..r]$ has $\lfloor n/2 \rfloor$ elements in it. How long does it take to sort the left subarray? We do not know this, but because $\lceil n/2 \rceil < n$ for $n > 1$, we can express this as $T(\lceil n/2 \rceil)$. Similarly, we can express the time that it takes to sort the right subarray as $T(\lfloor n/2 \rfloor)$. Finally, to merge both sorted lists takes n time, by the comments made above. In conclusion we have

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & \text{otherwise.} \end{cases}$$

Lecture 7: Recurrences

(Tuesday, Feb 17, 1998)

Read: Chapt. 4 on recurrences. Skip Section 4.4.

Divide and Conquer and Recurrences: Last time we introduced divide-and-conquer as a basic technique for designing efficient algorithms. Recall that the basic steps in divide-and-conquer solution are (1) divide the problem into a small number of subproblems, (2) solve each subproblem recursively, and (3) combine the solutions to the subproblems to a global solution. We also described MergeSort, a sorting algorithm based on divide-and-conquer.

Because divide-and-conquer is an important design technique, and because it naturally gives rise to recursive algorithms, it is important to develop mathematical techniques for solving recurrences, either exactly or asymptotically. To do this, we introduced the notion of a *recurrence*, that is, a recursively defined function. Today we discuss a number of techniques for solving recurrences.

MergeSort Recurrence: Here is the recurrence we derived last time for MergeSort. Recall that $T(n)$ is the time to run MergeSort on a list of size n. We argued that if the list is of length 1, then the total sorting time is a constant $\Theta(1)$. If $n > 1$, then we must recursively sort two sublists, one of size $\lceil n/2 \rceil$ and the other of size $\lfloor n/2 \rfloor$, and the nonrecursive part took $\Theta(n)$ time for splitting the list (constant time) and merging the lists ($\Theta(n)$ time). Thus, the total running time for MergeSort could be described by the following recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & \text{otherwise.} \end{cases}$$

Notice that we have dropped the $\Theta()$'s, replacing $\Theta(1)$ and $\Theta(n)$ by just 1 and n, respectively. This is done to make the recurrence more concrete. If we had wanted to be more precise, we could have replaced these with more exact functions, e.g., c_1 and $c_2 n$ for some constants c_1 and c_2. The analysis would have been a bit more complex, but we would arrive at the same asymptotic formula.

Getting a feel: We could try to get a feeling for what this means by plugging in some values and expanding the definition.

$$\begin{aligned}
T(1) &= 1 \qquad\qquad\qquad\qquad\qquad \text{(by the basis.)} \\
T(2) &= T(1) + T(1) + 2 = 1 + 1 + 2 = 4 \\
T(3) &= T(2) + T(1) + 3 = 4 + 1 + 3 = 8 \\
T(4) &= T(2) + T(2) + 4 = 4 + 4 + 4 = 12 \\
T(5) &= T(3) + T(2) + 5 = 8 + 4 + 5 = 17 \\
\ldots & \\
T(8) &= T(4) + T(4) + 8 = 12 + 12 + 8 = 32 \\
\ldots & \\
T(16) &= T(8) + T(8) + 16 = 32 + 32 + 16 = 80 \\
\ldots & \\
T(32) &= T(16) + T(16) + 32 = 80 + 80 + 32 = 192.
\end{aligned}$$

It's hard to see much of a pattern here, but here is a trick. Since the recurrence divides by 2 each time, let's consider powers of 2, since the function will behave most regularly for these values. If we consider the ratios $T(n)/n$ for powers of 2 and interesting pattern emerges:

$$\begin{aligned}
T(1)/1 &= 1 & T(8)/8 &= 4 \\
T(2)/2 &= 2 & T(16)/16 &= 5 \\
T(4)/4 &= 3 & T(32)/32 &= 6.
\end{aligned}$$

This suggests that for powers of 2, $T(n)/n = (\lg n) + 1$, or equivalently, $T(n) = (n \lg n) + n$ which is $\Theta(n \log n)$. This is not a proof, but at least it provides us with a starting point.

Logarithms in Θ-notation: Notice that I have broken away from my usual convention of say $\lg n$ and just said $\log n$ inside the $\Theta()$. The reason is that the base really does not matter when it is inside the Θ. Recall the change of base formula:

$$\log_b n = \frac{\log_a n}{\log_a b}.$$

If a and b are constants the $\log_a b$ is a constant. Consequently $\log_b n$ and $\log_a n$ differ only by a constant factor. Thus, inside the $\Theta()$ we do not need to differentiate between them. Henceforth, I will not be fussy about the bases of logarithms if asymptotic results are sufficient.

Eliminating Floors and Ceilings: One of the nasty things about recurrences is that floors and ceilings are a pain to deal with. So whenever it is reasonable to do so, we will just forget about them, and make whatever simplifying assumptions we like about n to make things work out. For this case, we will make the simplifying assumption that n is a power of 2. Notice that this means that our analysis will only be correct for a very limited (but infinitely large) set of values of n, but it turns out that as long as the algorithm doesn't act significantly different for powers of 2 versus other numbers, the asymptotic analysis will hold for all n as well. So let us restate our recurrence under this assumption:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise.} \end{cases}$$

Verification through Induction: We have just generated a guess for the solution to our recurrence. Let's see if we can verify its correctness formally. The proof will be by strong induction on n. Because n is limited to powers of 2, we cannot do the usual n to $n+1$ proof (because if n is a power of 2, $n+1$ will generally not be a power of 2). Instead we use strong induction.

Claim: For all $n \geq 1$, n a power of 2, $T(n) = (n \lg n) + n$.

Proof: (By strong induction on n.)

Basis case: ($n = 1$) In this case $T(1) = 1$ by definition and the formula gives $1 \lg 1 + 1 = 1$, which matches.

Induction step: Let $n > 1$, and assume that the formula $T(n') = (n' \lg n') + n'$, holds whenever $n' < n$. We want to prove the formula holds for n itself. To do this, we need to express $T(n)$ in terms of smaller values. To do this, we apply the definition:

$$T(n) = 2T(n/2) + n.$$

Now, $n/2 < n$, so we can apply the induction hypothesis here, yielding $T(n/2) = (n/2)\lg(n/2) + (n/2)$. Plugging this in gives

$$\begin{aligned} T(n) &= 2((n/2)\lg(n/2) + (n/2)) + n \\ &= (n \lg(n/2) + n) + n \\ &= n(\lg n - \lg 2) + 2n \\ &= (n \lg n - n) + 2n \\ &= n \lg n + n, \end{aligned}$$

which is exactly what we want to prove.

The Iteration Method: The above method of "guessing" a solution and verifying through induction works fine as long as your recurrence is simple enough that you can come up with a good guess. But if the recurrence is at all messy, there may not be a simple formula. The following method is quite powerful. When it works, it allows you to convert a recurrence into a summation. By in large, summations are easier to solve than recurrences (and if nothing else, you can usually approximate them by integrals).

The method is called *iteration*. Let's start expanding out the definition until we see a pattern developing. We first write out the definition $T(n) = 2T(n/2) + n$. This has a recursive formula inside $T(n/2)$ which we can expand, by filling in the definition but this time with the argument

$n/2$ rather than n. Plugging in we get $T(n) = 2(2T(n/4) + n/2) + n$. We then simplify and repeat. Here is what we get when we repeat this.

$$\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(2T(n/4) + n/2) + n \;=\; 4T(n/4) + n + n \\
&= 4(2T(n/8) + n/4) + n + n \;=\; 8T(n/8) + n + n + n \\
&= 8(2T(n/16) + n/8) + n + n + n \;=\; 16T(n/16) + n + n + n + n \\
&= \ldots
\end{aligned}$$

At this point we can see a general pattern emerging.

$$\begin{aligned}
T(n) &= 2^k T(n/(2^k)) + (n + n + \cdots + n) \qquad (k \text{ times}) \\
&= 2^k T(n/(2^k)) + kn.
\end{aligned}$$

Now, we have generated alot of equations, but we still haven't gotten anywhere, because we need to get rid of the $T()$ from the right-hand side. Here's how we can do that. We *know* that $T(1) = 1$. Thus, let us select k to be a value which forces the term $n/(2^k) = 1$. This means that $n = 2^k$, implying that $k = \lg n$. If we substitute this value of k into the equation we get

$$\begin{aligned}
T(n) &= 2^{(\lg n)} T(n/(2^{(\lg n)})) + (\lg n)n \\
&= 2^{(\lg n)} T(1) + n \lg n \;=\; 2^{(\lg n)} + n \lg n \;=\; n + n \lg n.
\end{aligned}$$

In simplifying this, we have made use of the formula from the first homework, $a^{\log_b n} = n^{\log_b a}$, where $a = b = 2$. Thus we have arrived at the same conclusion, but this time no guesswork was involved.

The Iteration Method (a Messier Example): That one may have been a bit too easy to see the general form. Let's try a messier recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 3T(n/4) + n & \text{otherwise.} \end{cases}$$

To avoid problems with floors and ceilings, we'll make the simplifying assumption here that n is a power of 4. As before, the idea is to repeatedly apply the definition, until a pattern emerges.

$$\begin{aligned}
T(n) &= 3T(n/4) + n \\
&= 3(3T(n/16) + n/4) + n \;=\; 9T(n/16) + 3(n/4) + n \\
&= 9(3T(n/64) + n/16) + 3(n/4) + n \;=\; 27T(n/64) + 9(n/16) + 3(n/4) + n \\
&= \ldots \\
&= 3^k T\left(\frac{n}{4^k}\right) + 3^{k-1}(n/4^{k-1}) + \cdots + 9(n/16) + 3(n/4) + n \\
&= 3^k T\left(\frac{n}{4^k}\right) + \sum_{i=0}^{k-1} \frac{3^i}{4^i} n.
\end{aligned}$$

As before, we have the recursive term $T(n/4^k)$ still floating around. To get rid of it we recall that we know the value of $T(1)$, and so we set $n/4^k = 1$ implying that $4^k = n$, that is,

$k = \log_4 n$. So, plugging this value in for k we get:

$$\begin{aligned}
T(n) &= 3^{\log_4 n} T(1) + \sum_{i=0}^{(\log_4 n)-1} \frac{3^i}{4^i} n \\
&= n^{\log_4 3} + \sum_{i=0}^{(\log_4 n)-1} \frac{3^i}{4^i} n.
\end{aligned}$$

Again, in the last step we used the formula $a^{\log_b n} = n^{\log_b a}$ where $a = 3$ and $b = 4$, and the fact that $T(1) = 1$. (Why did we write it this way? This emphasizes that the function is of the form n^c for some constant c.) By the way, $\log_4 3 = 0.7925\ldots \approx 0.79$, so $n^{\log_4 3} \approx n^{0.79}$.

We have this messy summation to solve though. First observe that the value n remains constant throughout the sum, and so we can pull it out front. Also note that we can write $3^i/4^i$ and $(3/4)^i$.

$$T(n) = n^{\log_4 3} + n \sum_{i=0}^{(\log_4 n)-1} \left(\frac{3}{4}\right)^i.$$

Note that this is a geometric series. We may apply the formula for the geometric series, which gave in an earlier lecture. For $x \neq 1$:

$$\sum_{i=0}^{m} x^i = \frac{x^{m+1} - 1}{x - 1}.$$

In this case $x = 3/4$ and $m = \log_4 n - 1$. We get

$$T(n) = n^{\log_4 3} + n \frac{(3/4)^{\log_4 n} - 1}{(3/4) - 1}.$$

Applying our favorite log identity once more to the expression in the numerator (with $a = 3/4$ and $b = 4$) we get

$$(3/4)^{\log_4 n} = n^{\log_4 (3/4)} = n^{(\log_4 3 - \log_4 4)} = n^{(\log_4 3 - 1)} = \frac{n^{\log_4 3}}{n}.$$

If we plug this back in, we have

$$\begin{aligned}
T(n) &= n^{\log_4 3} + n \frac{\frac{n^{\log_4 3}}{n} - 1}{(3/4) - 1} \\
&= n^{\log_4 3} + \frac{n^{\log_4 3} - n}{-1/4} \\
&= n^{\log_4 3} - 4(n^{\log_4 3} - n) \\
&= n^{\log_4 3} + 4(n - n^{\log_4 3}) \\
&= 4n - 3n^{\log_4 3}.
\end{aligned}$$

So the final result (at last!) is

$$T(n) = 4n - 3n^{\log_4 3} \approx 4n - 3n^{0.79} \in \Theta(n).$$

It is interesting to note the unusual exponent of $\log_4 3 \approx 0.79$. We have seen that two nested loops typically leads to $\Theta(n^2)$ time, and three nested loops typically leads to $\Theta(n^3)$ time, so it seems remarkable that we could generate a strange exponent like 0.79 as part of a running time. However, as we shall see, this is often the case in divide-and-conquer recurrences.

Lecture 8: More on Recurrences

(Thursday, Feb 19, 1998)
Read: Chapt. 4 on recurrences, skip Section 4.4.

Recap: Last time we discussed recurrences, that is, functions that are defined recursively. We discussed their importance in analyzing divide-and-conquer algorithms. We also discussed two methods for solving recurrences, namely guess-and-verify (by induction), and iteration. These are both very powerful methods, but they are quite "mechanical", and it is difficult to get a quick and intuitive sense of what is going on in the recurrence. Today we will discuss two more techniques for solving recurrences. The first provides a way of visualizing recurrences and the second, called the Master Theorem, is a method of solving many recurrences that arise in divide-and-conquer applications.

Visualizing Recurrences Using the Recursion Tree: Iteration is a very powerful technique for solving recurrences. But, it is easy to get lost in all the symbolic manipulations and lose sight of what is going on. Here is a nice way to visualize what is going on in iteration. We can describe any recurrence in terms of a tree, where each expansion of the recurrence takes us one level deeper in the tree.

Recall that the recurrence for MergeSort (which we simplified by assuming that n is a power of 2, and hence could drop the floors and ceilings)

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise.} \end{cases}$$

Suppose that we draw the recursion tree for MergeSort, but each time we merge two lists, we label that node of the tree with the time it takes to perform the associated (nonrecursive) merge. Recall that to merge two lists of size $m/2$ to a list of size m takes $\Theta(m)$ time, which we will just write as m. Below is an illustration of the resulting recursion tree.

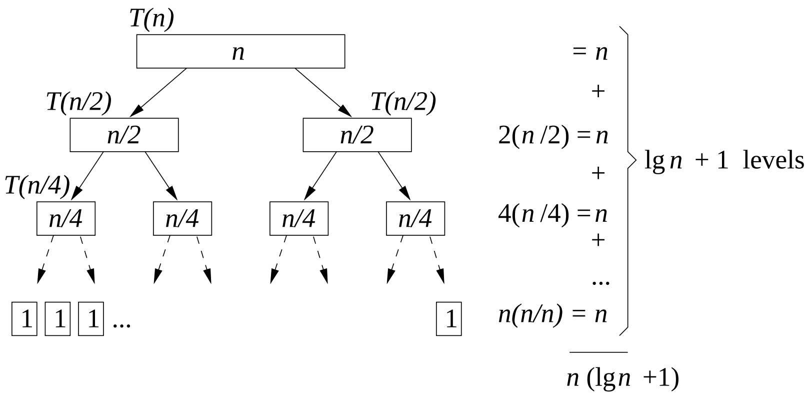


Figure 5: Using the recursion tree to visualize a recurrence.

Observe that the total work at the topmost level of the recursion is $\Theta(n)$ (or just n for short). At the second level we have two merges, each taking $n/2$ time, for a total of $2(n/2) = n$. At the third level we have 4 merges, each taking $n/4$ time, for a total of $4(n/4) = n$. This continues until the bottommost level of the tree. Since the tree exactly $\lg n + 1$ levels $(0, 1, 2, \ldots, \lg n)$, and each level contributes a total of n time, the total running time is $n(\lg n + 1) = n \lg n + n$. This is exactly what we got by the iteration method.

This can be used for a number of simple recurrences. For example, let's try it on the following recurrence. The tree is illustrated below.

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

Again, we label each node with the amount of work at that level. In this case the work for $T(m)$ is m^2. For the top level (or 0th level) the work is n^2. At level 1 we have three nodes whose work is $(n/2)^2$ each, for a total of $3(n/2)^2$. This can be written as $n^2(3/4)$. At the level 2 the work is $9(n/4)^2$, which can be written as $n^2(9/16)$. In general it is easy to extrapolate to see that at the level i, we have 3^i nodes, each involving $(n/2^i)^2$ work, for a total of $3^i(n/2^i)^2 = n^2(3/4)^i$.

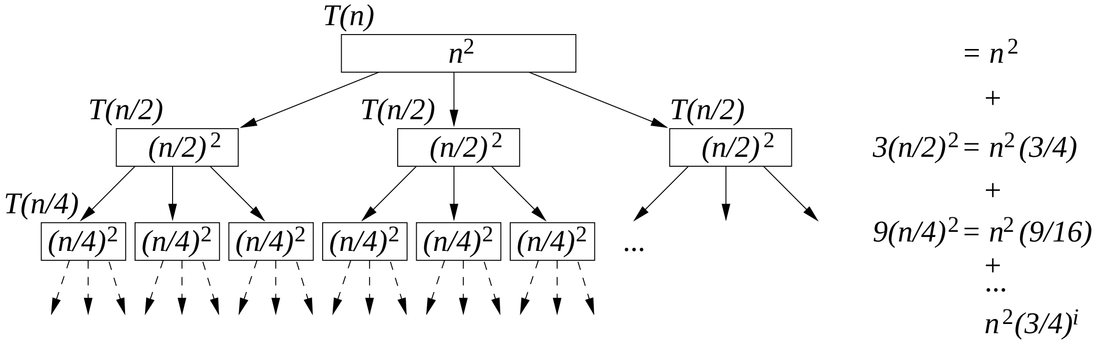


Figure 6: Another recursion tree example.

This leads to the following summation. Note that we have not determined where the tree bottoms out, so we have left off the upper bound on the sum.

$$T(n) = n^2 \sum_{i=0}^{?} \left(\frac{3}{4}\right)^i.$$

If all we wanted was an asymptotic expression, then are essentially done at this point. Why? The summation is a geometric series, and the base (3/4) is less than 1. This means that this series converges to some nonzero constant (even if we ran the sum out to ∞). Thus the running time is $\Theta(n^2)$.

To get a more exact result, observe that the recursion bottoms out when we get down to single items, and since the sizes of the inputs are cut by half at each level, it is not hard to see that the final level is level $\lg n$. (It is easy to be off by ± 1 here, but this sort of small error will not affect the asymptotic result. In this case we happen to be right.) So, we can plug in $\lg n$ for the "?" in the above summation.

$$T(n) = n^2 \sum_{i=0}^{\lg n} \left(\frac{3}{4}\right)^i.$$

If we wanted to get a more exact answer, we could plug the summation into the formula for the geometric series and simplify. This would lead to an expression like

$$T(n) = n^2 \frac{(3/4)^{\lg n+1} - 1}{(3/4) - 1}.$$

This will take some work to simplify, but at this point it is all just tedious algebra to get the formula into simple form. (This sort of algebraic is typical of algorithm analysis, so be sure that you follow each step.)

$$\begin{aligned}
T(n) &= n^2 \frac{(3/4)^{\lg n+1} - 1}{(3/4) - 1} \;=\; -4n^2((3/4)^{\lg n+1} - 1) \\
&= 4n^2(1 - (3/4)^{\lg n+1}) \;=\; 4n^2(1 - (3/4)(3/4)^{\lg n}) \\
&= 4n^2(1 - (3/4)n^{\lg(3/4)}) \;=\; 4n^2(1 - (3/4)n^{\lg 3 - \lg 4}) \\
&= 4n^2(1 - (3/4)n^{\lg 3 - 2}) \;=\; 4n^2(1 - (3/4)(n^{\lg 3}/n^2)) \\
&= 4n^2 - 3n^{\lg 3}.
\end{aligned}$$

Note that $\lg 3 \approx 1.58$, so the whole expression is $\Theta(n^2)$.

In conclusion, the technique of drawing the recursion tree is a somewhat more visual way of analyzing summations, but it is really equivalent to the method of iteration.

(Simplified) Master Theorem: If you analyze many divide-and-conquer algorithms, you will see that the same general type of recurrence keeps popping up. In general you are breaking a problem into a subproblems, where each subproblem is roughly a factor of $1/b$ of the original problem size, and the time it takes to do the splitting and combining on an input of size n is $\Theta(n^k)$. For example, in MergeSort, $a = 2$, $b = 2$, and $k = 1$.

Rather than doing every such recurrence from scratch, can we just come up with a general solution? The answer is that you can if all you need is an asymptotic expression. This result is called the *Master Theorem*, because it can be used to "master" so many different types of recurrence. Our text gives a fairly complicated version of this theorem. We will give a simpler version, which is general enough for most typical situations. In cases where this doesn't apply, try the one from the book. If the one from the book doesn't apply, then you will probably need iteration, or some other technique.

Theorem: (Simplified Master Theorem) Let $a \geq 1$, $b > 1$ be constants and let $T(n)$ be the recurrence

$$T(n) = aT(n/b) + n^k,$$

defined for $n \geq 0$. (As usual let us assume that n is a power of b. The basis case, $T(1)$ can be any constant value.) Then

Case 1: if $a > b^k$ then $T(n) \in \Theta(n^{\log_b a})$.

Case 2: if $a = b^k$ then $T(n) \in \Theta(n^k \log n)$.

Case 3: if $a < b^k$ then $T(n) \in \Theta(n^k)$.

Using this version of the Master Theorem we can see that in the MergeSort recurrence $a = 2$, $b = 2$, and $k = 1$. Thus, $a = b^k$ $(2 = 2^1)$ and so Case 2 applies. From this we have $T(n) \in \Theta(n \log n)$.

In the recurrence above, $T(n) = 3T(n/2) + n^2$, we have $a = 3$, $b = 2$ and $k = 2$. We have $a < b^k$ $(3 < 2^2)$ in this case, and so Case 3 applies. From this we have $T(n) \in \Theta(n^2)$.

Finally, consider the recurrence $T(n) = 4T(n/3) + n$, in which we have $a = 4$, $b = 3$ and $k = 1$. In this case we have $a > b^k$ $(4 > 3^1)$, and so Case 1 applies. From this we have $T(n) \in \Theta(n^{\log_3 4}) \approx \Theta(n^{1.26})$. This may seem to be a rather strange running time (a non-integer exponent), but this not uncommon for many divide-and-conquer solutions.

There are many recurrences that cannot be put into this form. For example, if the splitting and combining steps involve sorting, we might have seen a recurrence of the form

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n \log n & \text{otherwise.} \end{cases}$$

This solves to $T(n) = \Theta(n \log^2 n)$, but the Master Theorem (neither this form nor the one in CLR) will tell you this. However, iteration works just fine here.

Recursion Trees Revisited: The recursion trees offer some intuition about why it is that there are three cases in the Master Theorem. Generally speaking the question is where is most of the work done: at the top of the tree (the root level), at the bottom of the tree (the leaf level), or spread equally throughout the entire tree.

For example, in the MergeSort recurrence (which corresponds to Case 2 in the Master Theorem) every level of the recursion tree provides the same total work, namely n. For this reason the total work is equal to this value times the height of the tree, namely $\Theta(\log n)$, for a total of $\Theta(n \log n)$.

Next consider the earlier recurrence $T(n) = 3T(n/2) + n^2$ (which corresponds to Case 3 in the Master Theorem). In this instance most of the work was concentrated at the root of the tree. Each level of the tree provided a smaller fraction of work. By the nature of the geometric series, it did not matter how many levels the tree had at all. Even with an infinite number of levels, the geometric series that result will converge to a constant value. This is an important observation to make. A common way to design the most efficient divide-and-conquer algorithms is to try to arrange the recursion so that most of the work is done at the root, and at each successive level of the tree the work at this level reduces (by some constant factor). As long as this is the case, Case 3 will apply.

Finally, in the recurrence $T(n) = 4T(n/3) + n$ (which corresponds to Case 1), most of the work is done at the leaf level of the recursion tree. This can be seen if you perform iteration on this recurrence, the resulting summation is

$$n \sum_{i=0}^{\log_3 n} \left(\frac{4}{3}\right)^i .$$

(You might try this to see if you get the same result.) Since $4/3 > 1$, as we go deeper into the levels of the tree, that is deeper into the summation, the terms are growing successively larger. The largest contribution will be from the leaf level.

Lecture 9: Medians and Selection

(Tuesday, Feb 24, 1998)

Read: Todays material is covered in Sections 10.2 and 10.3. You are not responsible for the randomized analysis of Section 10.2. Our presentation of the partitioning algorithm and analysis are somewhat different from the ones in the book.

Selection: In the last couple of lectures we have discussed recurrences and the divide-and-conquer method of solving problems. Today we will give a rather surprising (and very tricky) algorithm which shows the power of these techniques.

The problem that we will consider is very easy to state, but surprisingly difficult to solve optimally. Suppose that you are given a set of n numbers. Define the *rank* of an element to be one plus the number of elements that are smaller than this element. Since duplicate elements make our life more complex (by creating multiple elements of the same rank), we will make the simplifying assumption that all the elements are distinct for now. It will be easy to get around this assumption later. Thus, the rank of an element is its final position if the set is sorted. The minimum is of rank 1 and the maximum is of rank n.

Of particular interest in statistics is the *median*. If n is odd then the median is defined to be the element of rank $(n+1)/2$. When n is even there are two natural choices, namely the elements of ranks $n/2$ and $(n/2)+1$. In statistics it is common to return the average of these two elements. We will define the median to be either of these elements.

Medians are useful as measures of the *central tendency* of a set, especially when the distribution of values is highly skewed. For example, the median income in a community is likely to be more meaningful measure of the central tendency than the average is, since if Bill Gates lives in your community then his gigantic income may significantly bias the average, whereas it cannot have a significant influence on the median. They are also useful, since in divide-and-conquer applications, it is often desirable to partition a set about its median value, into two sets of roughly equal size. Today we will focus on the following generalization, called the *selection problem*.

Selection: Given a set A of n distinct numbers and an integer k, $1 \le k \le n$, output the element of A of rank k.

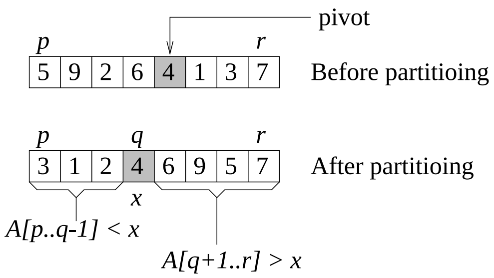
The selection problem can easily be solved in $\Theta(n \log n)$ time, simply by sorting the numbers of A, and then returning $A[k]$. The question is whether it is possible to do better. In particular, is it possible to solve this problem in $\Theta(n)$ time? We will see that the answer is yes, and the solution is far from obvious.

The Sieve Technique: The reason for introducing this algorithm is that it illustrates a very important special case of divide-and-conquer, which I call the *sieve technique*. We think of divide-and-conquer as breaking the problem into a small number of smaller subproblems, which are then solved recursively. The sieve technique is a special case, where the number of subproblems is just 1.

The sieve technique works in phases as follows. It applies to problems where we are interested in finding a single item from a larger set of n items. We do not know which item is of interest, however after doing some amount of analysis of the data, taking say $\Theta(n^k)$ time, for some constant k, we find that we do not know what the desired the item is, but we can identify a large enough number of elements that *cannot* be the desired value, and can be eliminated from further consideration. In particular "large enough" means that the number of items is at least some fixed constant fraction of n (e.g. $n/2$, $n/3$, $0.0001n$). Then we solve the problem recursively on whatever items remain. Each of the resulting recursive solutions then do the same thing, eliminating a constant fraction of the remaining set.

Applying the Sieve to Selection: To see more concretely how the sieve technique works, let us apply it to the selection problem. Recall that we are given an array $A[1..n]$ and an integer k, and want to find the k-th smallest element of A. Since the algorithm will be applied inductively, we will assume that we are given a subarray $A[p..r]$ as we did in MergeSort, and we want to find the kth smallest item (where $k \le r-p+1$). The initial call will be to the entire array $A[1..n]$.

There are two principal algorithms for solving the selection problem, but they differ only in one step, which involves judiciously choosing an item from the array, called the *pivot element*, which we will denote by x. Later we will see how to choose x, but for now just think of it as a random element of A. We then partition A into three parts. $A[q]$ contains the element x, subarray $A[p..q-1]$ will contain all the elements that are less than x, and $A[q+1..r]$, will contain all the element that are greater than x. (Recall that we assumed that all the elements are distinct.) Within each subarray, the items may appear in any order. This is illustrated below.



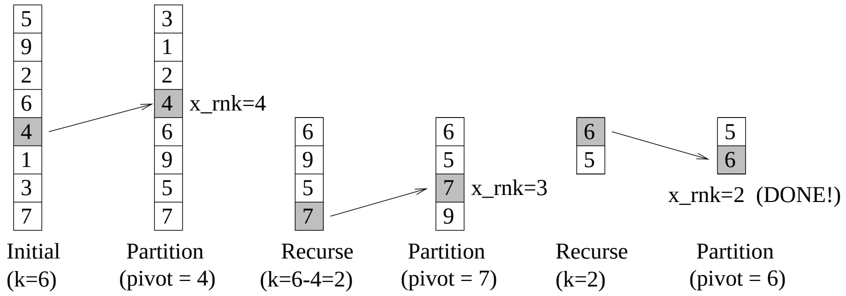


Figure 7: Selection Algorithm.

It is easy to see that the rank of the pivot x is $q-p+1$ in $A[p..r]$. Let $x\_rnk = q - p + 1$. If $k = x\_rnk$, then the pivot is the kth smallest, and we may just return it. If $k < x\_rnk$, then we know that we need to recursively search in $A[p..q-1]$ and if $k > x\_rnk$ then we need to recursively search $A[q+1..r]$. In this latter case we have eliminated q smaller elements, so we want to find the element of rank $k-q$. Here is the complete pseudocode.

Selection

```
Select(array A, int p, int r, int k) {      // return kth smallest of A[p..r]
    if (p == r) return A[p]                 // only 1 item left, return it
    else {
        x = Choose_Pivot(A, p, r)           // choose the pivot element
        q = Partition(A, p, r, x)           // partition <A[p..q-1], x, A[q+1..r]>
        x_rnk = q - p + 1                   // rank of the pivot
        if (k == x_rnk) return x            // the pivot is the kth smallest
        else if (k < x_rnk)
            return Select(A, p, q-1, k)     // select from left subarray
        else
            return Select(A, q+1, r, k-q)   // select from right subarray
    }
}
```

Notice that this algorithm satisfies the basic form of a sieve algorithm. It analyzes the data (by choosing the pivot element and partitioning) and it eliminates some part of the data set, and recurses on the rest. When $k = x\_rnk$ then we get lucky and eliminate everything. Otherwise we either eliminate the pivot and the right subarray or the pivot and the left subarray.

We will discuss the details of choosing the pivot and partitioning later, but assume for now that they both take $\Theta(n)$ time. The question that remains is how many elements did we succeed in eliminating? If x is the largest or smallest element in the array, then we may only succeed in eliminating one element with each phase. In fact, if x is one of the smallest elements of A or one of the largest, then we get into trouble, because we may only eliminate it and the few smaller or larger elements of A. Ideally x should have a rank that is neither too large nor too small.

Let us suppose for now (optimistically) that we are able to design the procedure `Choose_Pivot` in such a way that is eliminates exactly half the array with each phase, meaning that we recurse on the remaining $n/2$ elements. This would lead to the following recurrence.

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(n/2) + n & \text{otherwise.} \end{cases}$$

We can solve this either by expansion (iteration) or the Master Theorem. If we expand this recurrence level by level we see that we get the summation

$$T(n) \;=\; n + \frac{n}{2} + \frac{n}{4} + \cdots \;\le\; \sum_{i=0}^{\infty} \frac{n}{2^i} \;=\; n \sum_{i=0}^{\infty} \frac{1}{2^i}.$$

Recall the formula for the infinite geometric series. For any c such that $|c| < 1$, $\sum_{i=0}^{\infty} c^i = 1/(1-c)$. Using this we have

$$T(n) \le 2n \in O(n).$$

(This only proves the upper bound on the running time, but it is easy to see that it takes at least $\Omega(n)$ time, so the total running time is $\Theta(n)$.)

This is a bit counterintuitive. Normally you would think that in order to design a $\Theta(n)$ time algorithm you could only make a single, or perhaps a constant number of passes over the data set. In this algorithm we make many passes (it could be as many as $\lg n$). However, because we eliminate a constant fraction of elements with each phase, we get this convergent geometric series in the analysis, which shows that the total running time is indeed linear in n. This lesson is well worth remembering. It is often possible to achieve running times in ways that you would not expect.

Note that the assumption of eliminating half was not critical. If we eliminated even one per cent, then the recurrence would have been $T(n) = T(99n/100) + n$, and we would have gotten a geometric series involving $99/100$, which is still less than 1, implying a convergent series. Eliminating *any* constant fraction would have been good enough.

Choosing the Pivot: There are two issues that we have left unresolved. The first is how to choose the pivot element, and the second is how to partition the array. Both need to be solved in $\Theta(n)$ time. The second problem is a rather easy programming exercise. Later, when we discuss QuickSort, we will discuss partitioning in detail.

For the rest of the lecture, let's concentrate on how to choose the pivot. Recall that before we said that we might think of the pivot as a random element of A. Actually this is not such a bad idea. Let's see why.

The key is that we want the procedure to eliminate at least some constant fraction of the array after each partitioning step. Let's consider the top of the recurrence, when we are given $A[1..n]$. Suppose that the pivot x turns out to be of rank q in the array. The partitioning algorithm will split the array into $A[1..q-1] < x$, $A[q] = x$ and $A[q+1..n] > x$. If $k = q$, then we are done. Otherwise, we need to search one of the two subarrays. They are of sizes $q-1$ and $n-q$, respectively. The subarray that contains the kth smallest element will generally depend on what k is, so in the worst case, k will be chosen so that we have to recurse on the larger of the two subarrays. Thus if $q > n/2$, then we may have to recurse on the left subarray of size $q-1$, and if $q < n/2$, then we may have to recurse on the right subarray of size $n-q$. In either case, we are in trouble if q is very small, or if q is very large.

If we could select q so that it is roughly of middle rank, then we will be in good shape. For example, if $n/4 \le q \le 3n/4$, then the larger subarray will never be larger than $3n/4$. Earlier we said that we might think of the pivot as a random element of the array A. Actually this works pretty well in practice. The reason is that roughly half of the elements lie between ranks $n/4$ and $3n/4$, so picking a random element as the pivot will succeed about half the time to eliminate at least $3n/4$. Of course, we might be continuously unlucky, but a careful analysis will show that the expected running time is still $\Theta(n)$. We will return to this later.

Instead, we will describe a rather complicated method for computing a pivot element that achieves the desired properties. Recall that we are given an array $A[1..n]$, and we want to compute an element x whose rank is (roughly) between $n/4$ and $3n/4$. We will have to describe this algorithm at a very high level, since the details are rather involved. Here is the description for Select_Pivot:

Groups of 5: Partition A into groups of 5 elements, e.g. $A[1..5]$, $A[6..10]$, $A[11..15]$, etc. There will be exactly $m = \lceil n/5 \rceil$ such groups (the last one might have fewer than 5 elements). This can easily be done in $\Theta(n)$ time.

Group medians: Compute the median of each group of 5. There will be m group medians. We do not need an intelligent algorithm to do this, since each group has only a constant number of elements. For example, we could just BubbleSort each group and take the middle element. Each will take $\Theta(1)$ time, and repeating this $\lceil n/5 \rceil$ times will give a total running time of $\Theta(n)$. Copy the group medians to a new array B.

Median of medians: Compute the median of the group medians. For this, we will have to call the selection algorithm recursively on B, e.g. `Select(B, 1, m, k)`, where $m = \lceil n/5 \rceil$, and $k = \lfloor (m+1)/2 \rfloor$. Let x be this median of medians. Return x as the desired pivot.

The algorithm is illustrated in the figure below. To establish the correctness of this procedure, we need to argue that x satisfies the desired rank properties.

Lemma: The element x is of rank at least $n/4$ and at most $3n/4$ in A.

Proof: We will show that x is of rank at least $n/4$. The other part of the proof is essentially symmetrical. To do this, we need to show that there are at least $n/4$ elements that are less than or equal to x. This is a bit complicated, due to the floor and ceiling arithmetic, so to simplify things we will assume that n is evenly divisible by 5. Consider the groups shown in the tabular form above. Observe that at least half of the group medians are less than or equal to x. (Because x is their median.) And for each group median, there are three elements that are less than or equal to this median within its group (because it is the median of its group). Therefore, there are at least $3((n/5)/2 = 3n/10 \ge n/4$ elements that are less than or equal to x in the entire array.

Analysis: The last order of business is to analyze the running time of the overall algorithm. We achieved the main goal, namely that of eliminating a constant fraction (at least 1/4) of the remaining list at each stage of the algorithm. The recursive call in `Select()` will be made to list no larger than $3n/4$. However, in order to achieve this, within `Select_Pivot()` we needed to make a recursive call to `Select()` on an array B consisting of $\lceil n/5 \rceil$ elements. Everything else took only $\Theta(n)$ time. As usual, we will ignore floors and ceilings, and write the $\Theta(n)$ as n for concreteness. The running time is

$$T(n) \le \begin{cases} 1 & \text{if } n = 1, \\ T(n/5) + T(3n/4) + n & \text{otherwise.} \end{cases}$$

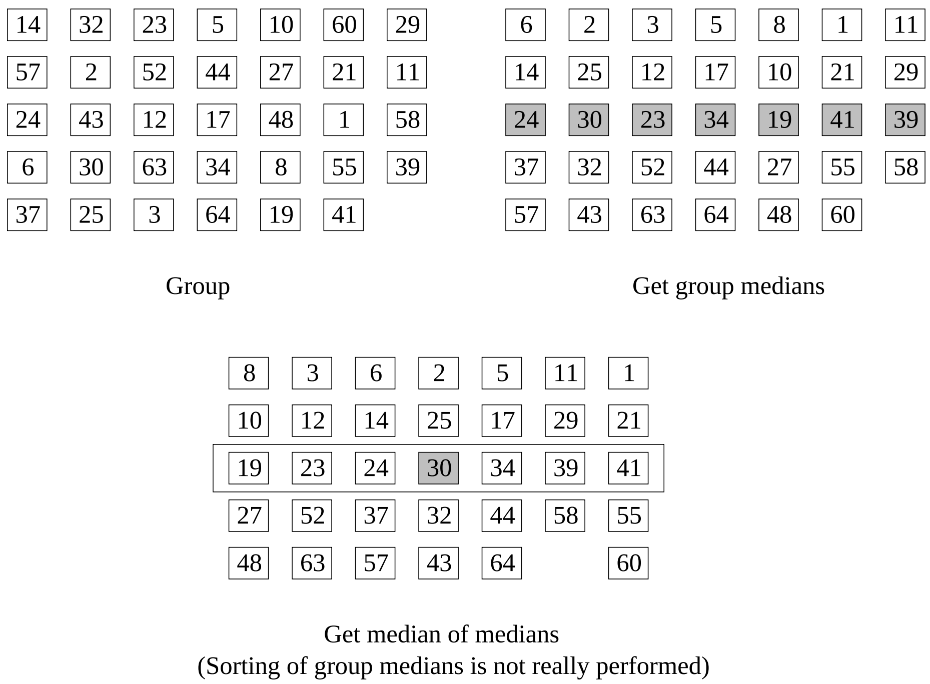


Figure 8: Choosing the Pivot. 30 is the final pivot.

This is a very strange recurrence because it involves a mixture of different fractions ($n/5$ and $3n/4$). This mixture will make it impossible to use the Master Theorem, and difficult to apply iteration. However, this is a good place to apply constructive induction. We know we want an algorithm that runs in $\Theta(n)$ time.

Theorem: There is a constant c, such that $T(n) \le cn$.

Proof: (by strong induction on n)

Basis: ($n = 1$) In this case we have $T(n) = 1$, and so $T(n) \le cn$ as long as $c \ge 1$.

Step: We assume that $T(n') \le cn'$ for all $n' < n$. We will then show that $T(n) \le cn$. By definition we have

$$T(n) = T(n/5) + T(3n/4) + n.$$

Since $n/5$ and $3n/4$ are both less than n, we can apply the induction hypothesis, giving

$$\begin{aligned} T(n) &\le c\frac{n}{5} + c\frac{3n}{4} + n \;=\; cn\left(\frac{1}{5} + \frac{3}{4}\right) + n \\ &= cn\frac{19}{20} + n \;=\; n\left(\frac{19c}{20} + 1\right). \end{aligned}$$

This last expression will be $\le cn$, provided that we select c such that $c \le (19c/20)+1$. Solving for c we see that this is true provided that $c \ge 20$.

Combining the constraints that $c \ge 1$, and $c \ge 20$, we see that by letting $c = 20$, we are done.

A natural question is why did we pick groups of 5? If you look at the proof above, you will see that it works for any value that is strictly greater than 4. (You might try it replacing the 5 with 3, 4, or 6 and see what happens.)

Lecture 10: Long Integer Multiplication

(Thursday, Feb 26, 1998)

Read: Todays material on integer multiplication is not covered in CLR.

Office hours: The TA, Kyongil, will have extra office hours on Monday before the midterm, from 1:00-2:00. I'll have office hours from 2:00-4:00 on Monday.

Long Integer Multiplication: The following little algorithm shows a bit more about the surprising applications of divide-and-conquer. The problem that we want to consider is how to perform arithmetic on long integers, and multiplication in particular. The reason for doing arithmetic on long numbers stems from cryptography. Most techniques for encryption are based on number-theoretic techniques. For example, the character string to be encrypted is converted into a sequence of numbers, and encryption keys are stored as long integers. Efficient encryption and decryption depends on being able to perform arithmetic on long numbers, typically containing hundreds of digits.

Addition and subtraction on large numbers is relatively easy. If n is the number of digits, then these algorithms run in $\Theta(n)$ time. (Go back and analyze your solution to the problem on Homework 1). But the standard algorithm for multiplication runs in $\Theta(n^2)$ time, which can be quite costly when lots of long multiplications are needed.

This raises the question of whether there is a more efficient way to multiply two very large numbers. It would seem surprising if there were, since for centuries people have used the same algorithm that we all learn in grade school. In fact, we will see that it is possible.

Divide-and-Conquer Algorithm: We know the basic grade-school algorithm for multiplication. We normally think of this algorithm as applying on a digit-by-digit basis, but if we partition an n digit number into two "super digits" with roughly $n/2$ each into longer sequences, the same multiplication rule still applies.

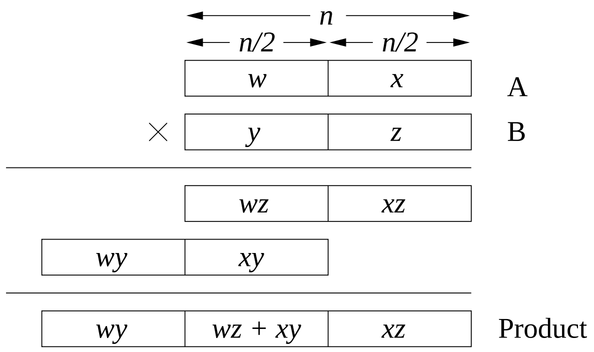


Figure 9: Long integer multiplication.

To avoid complicating things with floors and ceilings, let's just assume that the number of digits n is a power of 2. Let A and B be the two numbers to multiply. Let $A[0]$ denote the least significant digit and let $A[n-1]$ denote the most significant digit of A. Because of the way we write numbers, it is more natural to think of the elements of A as being indexed in decreasing order from left to right as $A[n-1..0]$ rather than the usual $A[0..n-1]$.

Let $m = n/2$. Let

$$\begin{array}{rclcrcll} w &=& A[n-1..m] & \quad & x &=& A[m-1..0] & \text{and} \\ y &=& B[n-1..m] & & z &=& B[m-1..0]. & \end{array}$$

If we think of w, x, y and z as $n/2$ digit numbers, we can express A and B as

$$\begin{aligned} A &= w \cdot 10^m + x \\ B &= y \cdot 10^m + z, \end{aligned}$$

and their product is

$$\mathit{mult}(A, B) = \mathit{mult}(w, y)10^{2m} + (\mathit{mult}(w, z) + \mathit{mult}(x, y))10^m + \mathit{mult}(x, z).$$

The operation of multiplying by 10^m should be thought of as simply shifting the number over by m positions to the right, and so is not really a multiplication. Observe that all the additions involve numbers involving roughly $n/2$ digits, and so they take $\Theta(n)$ time each. Thus, we can express the multiplication of two long integers as the result of 4 products on integers of roughly half the length of the original, and a constant number of additions and shifts, each taking $\Theta(n)$ time. This suggests that if we were to implement this algorithm, its running time would be given by the following recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 4T(n/2) + n & \text{otherwise.} \end{cases}$$

If we apply the Master Theorem, we see that $a = 4$, $b = 2$, $k = 1$, and $a > b^k$, implying that Case 1 holds and the running time is $\Theta(n^{\lg 4}) = \Theta(n^2)$. Unfortunately, this is no better than the standard algorithm.

Faster Divide-and-Conquer Algorithm: Even though the above exercise appears to have gotten us nowhere, it actually has given us an important insight. It shows that the critical element is the number of multiplications on numbers of size $n/2$. The number of additions (as long as it is a constant) does not affect the running time. So, if we could find a way to arrive at the same result algebraically, but by trading off multiplications in favor of additions, then we would have a more efficient algorithm. (Of course, we cannot simulate multiplication through repeated additions, since the number of additions must be a constant, independent of n.)

The key turns out to be a algebraic "trick". The quantities that we need to compute are $C = wy$, $D = xz$, and $E = (wz + xy)$. Above, it took us four multiplications to compute these. However, observe that if instead we compute the following quantities, we can get everything we want, using only three multiplications (but with more additions and subtractions).

$$\begin{aligned} C &= \mathit{mult}(w, y) \\ D &= \mathit{mult}(x, z) \\ E &= \mathit{mult}((w + x), (y + z)) - C - D \;=\; (wy + wz + xy + xz) - wy - xz \;=\; (wz + xy). \end{aligned}$$

Finally we have

$$\mathit{mult}(A, B) = C \cdot 10^{2m} + E \cdot 10^m + D.$$

Altogether we perform 3 multiplications, 4 additions, and 2 subtractions all of numbers with $n/2$ digitis. We still need to shift the terms into their proper final positions. The additions, subtractions, and shifts take $\Theta(n)$ time in total. So the total running time is given by the recurrence:

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

Now when we apply the Master Theorem, we have $a = 3$, $b = 2$ and $k = 1$, yielding $T(n) \in \Theta(n^{\lg 3}) \approx \Theta(n^{1.585})$.

Is this really an improvement? This algorithm carries a larger constant factor because of the overhead of recursion and the additional arithmetic operations. But asymptotics says that if n is large enough, then this algorithm will be superior. For example, if we assume that the clever algorithm has overheads that are 5 times greater than the simple algorithm (e.g. $5n^{1.585}$ versus n^2) then this algorithm beats the simple algorithm for $n \geq 50$. If the overhead was 10 times larger, then the crossover would occur for $n \geq 260$.

Review for the Midterm: Here is a list topics and readings for the first midterm exam. Generally you are responsible for anything discussed in class, and anything appearing on homeworks. It is a good idea to check out related chapters in the book, because this is where I often look for ideas on problems.

Worst-case, Average-case: Recall that a worst-case means that we consider the highest running time over all inputs of size n, average case means that we average running times over all inputs of size n (and generally weighting each input by its probability of occuring). (Chapt 1 of CLR.)

General analysis methods: Be sure you understand the induction proofs given in class and on the homeworks. Also be sure you understand how the constructive induction proofs worked.

Summations: Write down (and practice recognizing) the basic formulas for summations. These include the arithmetic series $\sum_i i$, the quadratic series, $\sum_i i^2$, the geometric series $\sum_i x^i$, and the harmonic series $\sum_i 1/i$. Practice with simplifying summations. For example, be sure that you can take something like

$$\sum_i 3^i \left(\frac{n}{2^i}\right)^2$$

and simplify it to a geometric series

$$n^2 \sum_i (3/4)^i.$$

Also be sure you can apply the integration rule to summations. (Chapt. 3 of CLR.)

Asymptotics: Know the formal definitions for Θ, O, and Ω, as well as how to use the limit-rule. Know the what the other forms, o and ω, mean informally. There are a number of good sample problems in the book. I'll be happy to check any of your answers. Also be able to rank functions in asymptotic order. For example which is larger $\lg \sqrt{n}$ or $\sqrt{\lg n}$? (It is the former, can you see why?) Remember the following rule and know how to use it.

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0 \qquad \lim_{n \to \infty} \frac{\lg^b n}{n^c} = 0.$$

(Chapt. 2 of CLR.)

Recurrences: Know how to analyze the running time of a recursive program by expressing it as a recurrence. Review the basic techniques for solving recurrences: guess and verify by induction (I'll provide any guesses that you need on the exam), constructive induction, iteration, and the (simplified) Master Theorem. (You are NOT responsible for the more complex version given in the text.) (Chapt 4, Skip 4.4.)

Divide-and-conquer: Understand how to design algorithms by divide-and-conquer. Understand the divide-and-conquer algorithm for MergeSort, and be able to work an example by hand. Also understand how the sieve technique works, and how it was used in the selection problem. (Chapt 10 on Medians; skip the randomized analysis. The material on the 2-d maxima and long integer multiplication is not discussed in CLR.)

Lecture 11: First Midterm Exam

(Tuesday, March 3, 1998)
First midterm exam today. No lecture.

Lecture 12: Heaps and HeapSort

(Thursday, Mar 5, 1998)
Read: Chapt 7 in CLR.

Sorting: For the next series of lectures we will focus on sorting algorithms. The reasons for studying sorting algorithms in details are twofold. First, sorting is a very important algorithmic problem. Procedures for sorting are parts of many large software systems, either explicitly or implicitly. Thus the design of efficient sorting algorithms is important for the overall efficiency of these systems. The other reason is more pedagogical. There are many sorting algorithms, some slow and some fast. Some possess certain desirable properties, and others do not. Finally sorting is one of the few problems where there provable lower bounds on how fast you can sort. Thus, sorting forms an interesting case study in algorithm theory.

In the sorting problem we are given an array $A[1..n]$ of n numbers, and are asked to reorder these elements into increasing order. More generally, A is of an array of records, and we choose one of these records as the *key value* on which the elements will be sorted. The key value need not be a number. It can be any object from a *totally ordered* domain. Totally ordered means that for any two elements of the domain, x, and y, either $x < y$, $x =$, or $x > y$.

There are some domains that can be partially ordered, but not totally ordered. For example, sets can be partially ordered under the subset relation, $\subset$, but this is not a total order, it is not true that for any two sets either $x \subset y$, $x = y$ or $x \supset y$. There is an algorithm called *topological sorting* which can be applied to "sort" partially ordered sets. We may discuss this later.

Slow Sorting Algorithms: There are a number of well-known slow sorting algorithms. These include the following:

Bubblesort: Scan the array. Whenever two consecutive items are found that are out of order, swap them. Repeat until all consecutive items are in order.

Insertion sort: Assume that $A[1..i-1]$ have already been sorted. Insert $A[i]$ into its proper position in this subarray, by shifting all larger elements to the right by one to make space for the new item.

Selection sort: Assume that $A[1..i-1]$ contain the $i-1$ smallest elements in sorted order. Find the smallest element in $A[i..n]$, and then swap it with $A[i]$.

These algorithms are all easy to implement, but they run in $\Theta(n^2)$ time in the worst case. We have already seen that MergeSort sorts an array of numbers in $\Theta(n \log n)$ time. We will study two others, HeapSort and QuickSort.

Priority Queues: The heapsort algorithm is based on a very nice data structure, called a *heap*. A heap is a concrete implementation of an abstract data type called a *priority queue*. A priority queue stores elements, each of which is associated with a numeric key value, called its *priority*. A simple priority queue supports three basic operations:

Create: Create an empty queue.

Insert: Insert an element into a queue.

ExtractMax: Return the element with maximum key value from the queue. (Actually it is more common to extract the minimum. It is easy to modify the implementation (by reversing $<$ and $>$ to do this.)

Empty: Test whether the queue is empty.

Adjust Priority: Change the priority of an item in the queue.

It is common to support a number of additional operations as well, such as building a priority queue from an initial set of elements, returning the largest element without deleting it, and changing the priority of an element that is already in the queueu (either decreasing or increasing it).

Heaps: A heap is a data structure that supports the main priority queue operations (insert and delete max) in $\Theta(\log n)$ time. For now we will describe the heap in terms of a binary tree implementation, but we will see later that heaps can be stored in arrays.

By a *binary tree* we mean a data structure which is either empty or else it consists of three things: a root node, a left subtree and a right subtree. The left subtree and right subtrees are each binary trees. They are called the *left child* and *right child* of the root node. If both the left and right children of a node are empty, then this node is called a *leaf node*. A nonleaf node is called an *internal* node.

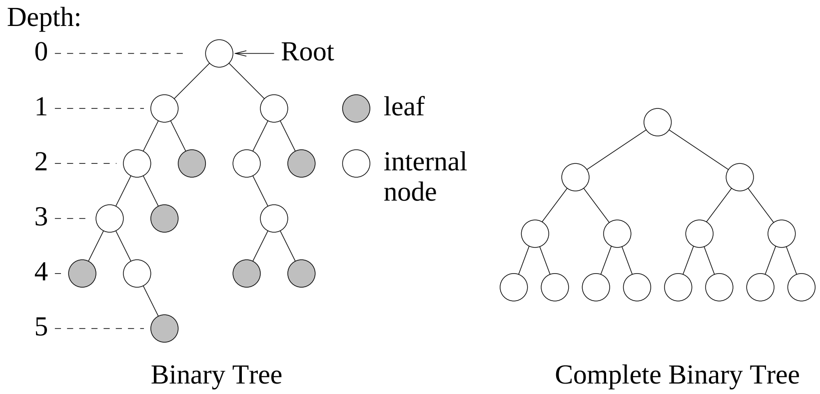


Figure 10: Binary trees.

The *depth* of a node in a binary tree is its distance from the root. The root is at depth 0, its children at depth 1, its grandchildren at depth 2, and so on. The *height* of a binary tree is its maximum depth. Binary tree is said to be *complete* if all internal nodes have two (nonempty) children, and all leaves have the same depth. An important fact about a complete binary trees is that a complete binary tree of height h has

$$n = 1 + 2 + \ldots + 2^h = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

nodes altogether. If we solve for h in terms of n, we see that the the height of a complete binary tree with n nodes is $h = (\lg(n+1)) - 1 \approx \lg n \in \Theta(\log n)$.

A heap is represented as an *left-complete* binary tree. This means that all the levels of the tree are full except the bottommost level, which is filled from left to right. An example is shown below. The keys of a heap are stored in something called *heap order*. This means that for each node u, other than the root, $key(Parent(u)) \geq key(u)$. This implies that as you follow any path from a leaf to the root the keys appear in (nonstrict) increasing order. Notice that this implies that the root is necessarily the largest element.

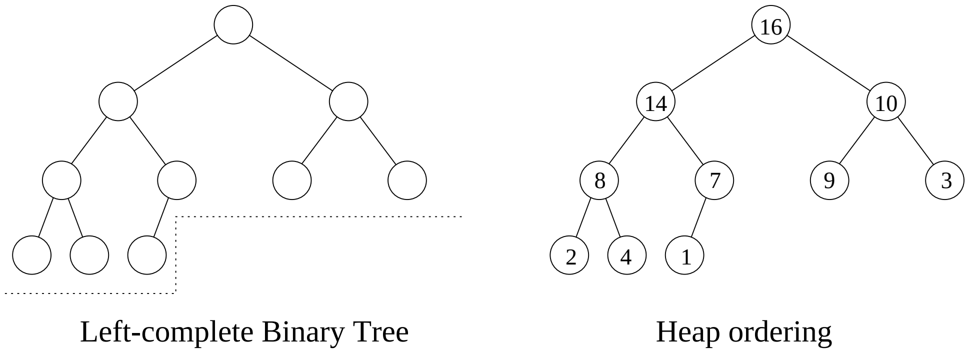


Figure 11: Heap.

Next time we will show how the priority queue operations are implemented for a heap.

Lecture 13: HeapSort

(Tuesday, Mar 10, 1998)
Read: Chapt 7 in CLR.

Heaps: Recall that a heap is a data structure that supports the main priority queue operations (insert and extract max) in $\Theta(\log n)$ time each. It consists of a left-complete binary tree (meaning that all levels of the tree except possibly the bottommost) are full, and the bottommost level is filled from left to right. As a consequence, it follows that the depth of the tree is $\Theta(\log n)$ where n is the number of elements stored in the tree. The keys of the heap are stored in the tree in what is called *heap order*. This means that for each (nonroot) node its parent's key is at least as large as its key. From this it follows that the largest key in the heap appears at the root.

Array Storage: Last time we mentioned that one of the clever aspects of heaps is that they can be stored in arrays, without the need for using pointers (as would normally be needed for storing binary trees). The reason for this is the left-complete nature of the tree.

This is done by storing the heap in an array $A[1..n]$. Generally we will not be using all of the array, since only a portion of the keys may be part of the current heap. For this reason, we maintain a variable $m \leq n$ which keeps track of the current number of elements that are actually stored actively in the heap. Thus the heap will consist of the elements stored in elements $A[1..m]$.

We store the heap in the array by simply unraveling it level by level. Because the binary tree is left-complete, we know exactly how many elements each level will supply. The root level supplies 1 node, the next level 2, then 4, then 8, and so on. Only the bottommost level may supply fewer than the appropriate power of 2, but then we can use the value of m to determine where the last element is. This is illustrated below.

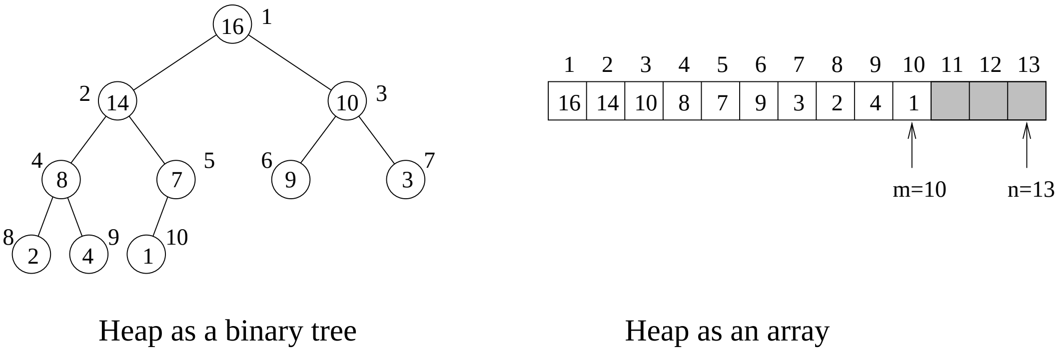


Figure 12: Storing a heap in an array.

We should emphasize that this *only works* because the tree is left-complete. This cannot be used for general trees.

We claim that to access elements of the heap involves simple arithmetic operations on the array indices. In particular it is easy to see the following.

$Left(i)$: return $2i$.

$Right(i)$: return $2i + 1$.

$Parent(i)$: return $\lfloor i/2 \rfloor$.

$IsLeaf(i)$: return $Left(i) > m$. (That is, if i's left child is not in the tree.)

$IsRoot(i)$: return $i == 1$.

For example, the heap ordering property can be stated as "for all i, $1 \le i \le n$, if (not $IsRoot(i)$) then $A[Parent(i)] \ge A[i]$".

So is a heap a binary tree or an array? The answer is that from a conceptual standpoint, it is a binary tree. However, it is implemented (typically) as an array for space efficiency.

Maintaining the Heap Property: There is one principal operation for maintaining the heap property. It is called `Heapify`. (In other books it is sometimes called *sifting down*.) The idea is that we are given an element of the heap which we suspect may not be in valid heap order, but we assume that all of other the elements in the subtree rooted at this element are in heap order. In particular this root element may be too small. To fix this we "sift" it down the tree by swapping it with one of its children. Which child? We should take the larger of the two children to satisfy the heap ordering property. This continues recursively until the element is either larger than both its children or until its falls all the way to the leaf level. Here is the pseudocode. It is given the heap in the array A, and the index i of the suspected element, and m the current active size of the heap. The element $A[max]$ is set to the maximum of $A[i]$ and it two children. If $max \ne i$ then we swap $A[i]$ and $A[max]$ and then recurse on $A[max]$.

Heapify

```
Heapify(array A, int i, int m) {           // sift down A[i] in A[1..m]
    l = Left(i)                            // left child
    r = Right(i)                           // right child
    max = i
    if (l <= m and A[l] > A[max]) max = l  // left child exists and larger
```

```
    if (r <= m and A[r] > A[max]) max = r       // right child exists and larger
    if (max != i) {                             // if either child larger
        swap A[i] with A[max]                   // swap with larger child
        Heapify(A, max, m)                      // and recurse
    }
}
```

See Figure 7.2 on page 143 of CLR for an example of how Heapify works (in the case where $m = 10$). We show the execution on a tree, rather than on the array representation, since this is the most natural way to conceptualize the heap. You might try simulating this same algorithm on the array, to see how it works at a finer details.

Note that the recursive implementation of Heapify is not the most efficient. We have done so because many algorithms on trees are most naturally implemented using recursion, so it is nice to practice this here. It is possible to write the procedure iteratively. This is left as an exercise.

The HeapSort algorithm will consist of two major parts. First building a heap, and then extracting the maximum elements from the heap, one by one. We will see how to use Heapify to help us do both of these.

How long does Hepify take to run? Observe that we perform a constant amount of work at each level of the tree until we make a call to Heapify at the next lower level of the tree. Thus we do $O(1)$ work for each level of the tree which we visit. Since there are $\Theta(\log n)$ levels altogether in the tree, the total time for Heapify is $O(\log n)$. (It is not $\Theta(\log n)$ since, for example, if we call Heapify on a leaf, then it will terminate in $\Theta(1)$ time.)

Building a Heap: We can use Heapify to build a heap as follows. First we start with a heap in which the elements are not in heap order. They are just in the same order that they were given to us in the array A. We build the heap by starting at the leaf level and then invoke Heapify on each node. (Note: We cannot start at the top of the tree. Why not? Because the precondition which Heapify assumes is that the entire tree rooted at node i is already in heap order, except for i.) Actually, we can be a bit more efficient. Since we know that each leaf is already in heap order, we may as well skip the leaves and start with the first nonleaf node. This will be in position $\lfloor n/2 \rfloor$. (Can you see why?)

Here is the code. Since we will work with the entire array, the parameter m for Heapify, which indicates the current heap size will be equal to n, the size of array A, in all the calls.

BuildHeap

```
BuildHeap(int n, array A[1..n]) {               // build heap from A[1..n]
    for i = n/2 downto 1 {
        Heapify(A, i, n)
    }
}
```

An example of Heapify is shown in Figure 7.3 on page 146 of CLR. Since each call to Heapify takes $O(\log n)$ time, and we make roughly $n/2$ calls to it, the total running time is $O((n/2)\log n) = O(n \log n)$. Next time we will show that this actually runs faster, and in fact it runs in $\Theta(n)$ time.

HeapSort: We can now give the HeapSort algorithm. The idea is that we need to repeatedly extract the maximum item from the heap. As we mentioned earlier, this element is at the root of the heap. But once we remove it we are left with a hole in the tree. To fix this we will replace it with the last leaf in the tree (the one at position $A[m]$). But now the heap order will very likely be destroyed. So we will just apply Heapify to the root to fix everything back up.

HeapSort

```
HeapSort(int n, array A[1..n]) {         // sort A[1..n]
    BuildHeap(n, A)                      // build the heap
    m = n                                // initially heap contains all
    while (m >= 2) {
        swap A[1] with A[m]              // extract the m-th largest
        m = m-1                          // unlink A[m] from heap
        Heapify(A, 1, m)                 // fix things up
    }
}
```

An example of HeapSort is shown in Figure 7.4 on page 148 of CLR. We make $n-1$ calls to Heapify, each of which takes $O(\log n)$ time. So the total running time is $O((n-1)\log n) = O(n \log n)$.

Lecture 14: HeapSort Analysis and Partitioning

(Thursday, Mar 12, 1998)

Read: Chapt 7 and 8 in CLR. The algorithm we present for partitioning is different from the texts.

HeapSort Analysis: Last time we presented HeapSort. Recall that the algorithm operated by first building a heap in a bottom-up manner, and then repeatedly extracting the maximum element from the heap and moving it to the end of the array. One clever aspect of the data structure is that it resides inside the array to be sorted.

We argued that the basic heap operation of Heapify runs in $O(\log n)$ time, because the heap has $O(\log n)$ levels, and the element being sifted moves down one level of the tree after a constant amount of work.

Based on this we can see that (1) that it takes $O(n \log n)$ time to build a heap, because we need to apply Heapify roughly $n/2$ times (to each of the internal nodes), and (2) that it takes $O(n \log n)$ time to extract each of the maximum elements, since we need to extract roughly n elements and each extraction involves a constant amount of work and one Heapify. Therefore the total running time of HeapSort is $O(n \log n)$.

Is this tight? That is, is the running time $\Theta(n \log n)$? The answer is yes. In fact, later we will see that it is not possible to sort faster than $\Omega(n \log n)$ time, assuming that you use comparisons, which HeapSort does. However, it turns out that the first part of the analysis is not tight. In particular, the BuildHeap procedure that we presented actually runs in $\Theta(n)$ time. Although in the wider context of the HeapSort algorithm this is not significant (because the running time is dominated by the $\Theta(n \log n)$ extraction phase).

Nonetheless there are situations where you might not need to sort all of the elements. For example, it is common to extract some unknown number of the smallest elements until some criterion (depending on the particular application) is met. For this reason it is nice to be able to build the heap quickly since you may not need to extract all the elements.

BuildHeap Analysis: Let us consider the running time of BuildHeap more carefully. As usual, it will make our lives simple by making some assumptions about n. In this case the most convenient assumption is that n is of the form $n = 2^{h+1} - 1$, where h is the height of the tree. The reason is that a left-complete tree with this number of nodes is a complete tree, that is, its bottommost level is full. This assumption will save us from worrying about floors and ceilings.

With this assumption, level 0 of the tree has 1 node, level 1 has 2 nodes, and up to level h, which has 2^h nodes. All the leaves reside on level h.

Recall that when Heapify is called, the running time depends on how far an element might sift down before the process terminates. In the worst case the element might sift down all the way to the leaf level. Let us count the work done level by level.

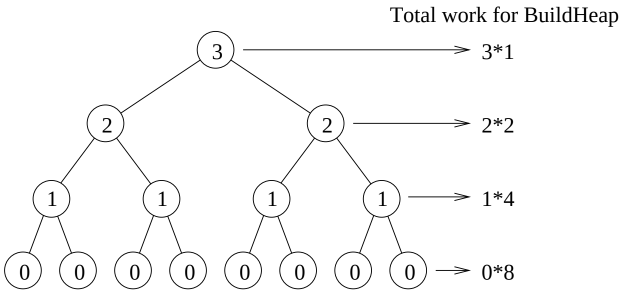


Figure 13: Analysis of BuildHeap.

At the bottommost level there are 2^h nodes, but we do not call Heapify on any of these so the work is 0. At the next to bottommost level there are 2^{h-1} nodes, and each might sift down 1 level. At the 3rd level from the bottom there are 2^{h-2} nodes, and each might sift down 2 levels. In general, at level j from the bottom there are 2^{h-j} nodes, and each might sift down j levels. So, if we count from bottom to top, level-by-level, we see that the total time is proportional to

$$T(n) = \sum_{j=0}^{h} j2^{h-j} = \sum_{j=0}^{h} j\frac{2^h}{2^j}.$$

If we factor out the 2^h term, we have

$$T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j}.$$

This is a sum that we have never seen before. We could try to approximate it by an integral, which would involve integration by parts, but it turns out that there is a very cute solution to this particular sum. We'll digress for a moment to work it out. First, write down the infinite general geometric series, for any constant $x < 1$.

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}.$$

Then take the derivative of both sides with respect to x, and multiply by x giving:

$$\sum_{j=0}^{\infty} jx^{j-1} = \frac{1}{(1-x)^2} \qquad\qquad \sum_{j=0}^{\infty} jx^j = \frac{x}{(1-x)^2},$$

and if we plug $x = 1/2$, then voila! we have the desired formula:

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{1/2}{(1-(1/2))^2} = \frac{1/2}{1/4} = 2.$$

In our case we have a bounded sum, but since the infinite series is bounded, we can use it instead as an easy approximation.

Using this we have

$$T(n) \;=\; 2^h \sum_{j=0}^{h} \frac{j}{2^j} \;\leq\; 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \;\leq\; 2^h \cdot 2 \;=\; 2^{h+1}.$$

Now recall that $n = 2^{h+1} - 1$, so we have $T(n) \leq n + 1 \in O(n)$. Clearly the algorithm takes at least $\Omega(n)$ time (since it must access every element of the array at least once) so the total running time for BuildHeap is $\Theta(n)$.

It is worthwhile pausing here a moment. This is the second time we have seen a relatively complex structured algorithm, with doubly nested loops, come out with a running time of $\Theta(n)$. (The other example was the median algorithm, based on the sieve technique. Actually if you think deeply about this, there is a sense in which a parallel version of BuildHeap can be viewed as operating like a sieve, but maybe this is getting too philosophical.) Perhaps a more intuitive way to describe what is happening here is to observe an important fact about binary trees. This is that the vast majority of nodes are at the lowest level of the tree. For example, in a complete binary tree of height h there is a total of $n \approx 2^{h+1}$ nodes in total, and the number of nodes in the bottom 3 levels alone is

$$2^h + 2^{h-1} + 2^{h-2} \;=\; \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \;=\; \frac{7n}{8} = 0.875n.$$

That is, almost 90% of the nodes of a complete binary tree reside in the 3 lowest levels. Thus the lesson to be learned is that when designing algorithms that operate on trees, it is important to be most efficient on the bottommost levels of the tree (as BuildHeap is) since that is where most of the weight of the tree resides.

Partitioning: Our next sorting algorithm is QuickSort. QuickSort is interesting in a number of respects. First off, (as we will present it) it is a *randomized algorithm*, which means that it makes use of a random number generator. We will show that in the worst case its running time is $O(n^2)$, its expected case running time is $O(n \log n)$. Moreover, this expected case running time occurs with *high probability*, in that the probability that the algorithm takes significantly more than $O(n \log n)$ time is rapidly decreasing function of n. In addition, QuickSort has a better locality-of-reference behavior than either MergeSort or HeapSort, and thus it tends to run fastest of all three algorithms. This is how it got its name. QuickSort (and its variants) are considered the methods of choice for most standard library sorting algorithms.

Next time we will discuss QuickSort. Today we will discuss one aspect of QuickSort, namely the partitioning algorithm. This is the same partitioning algorithm which we discussed when we talked about the selection (median) problem. We are given an array $A[p..r]$, and a pivot element x chosen from the array. Recall that the partitioning algorithm is suppose to partition A into three subarrays: $A[p..q-1]$ whose elements are all less than or equal to x, $A[q] = x$, and $A[q+1..r]$ whose elements are greater than or equal to x. We will assume that x is the first element of the subarray, that is, $x = A[p]$. If a different rule is used for selecting x, this is easily handled by swapping this element with $A[p]$ before calling this procedure.

We will present a different algorithm from the one given in the text (in Section 8.1). This algorithm is a little easier to verify the correctness, and a little easier to analyze. (But I suspect that the one in the text is probably a bit for efficient for actual implementation.)

This algorithm works by maintaining the following *invariant condition*. The subarray is broken into four segments. The boundaries between these items are indicated by the indices p, q, s, and r.

(1) $A[p] = x$ is the pivot value,

(2) $A[p+1..q]$ contains items that are less than x,

(3) $A[q+1..s-1]$ contains items that are greater than or equal to x, and

(4) $A[s..r]$ contains items whose values are currently unknown.

This is illustrated below.

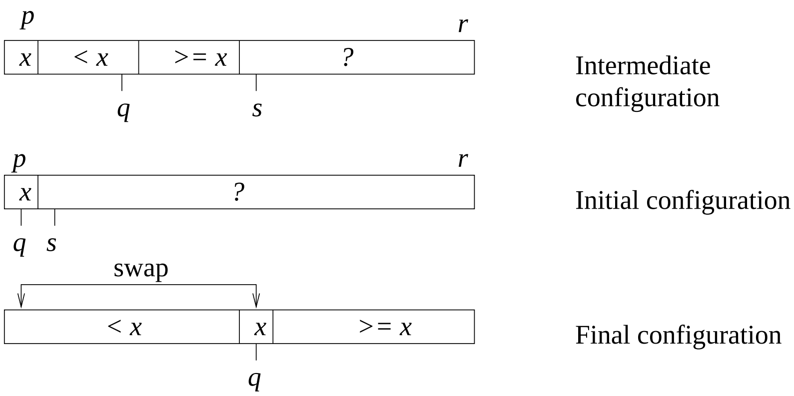


Figure 14: Partitioning intermediate structure.

The algorithm begins by setting $q = p$ and $s = p+1$. With each step through the algorithm we test the value of $A[s]$ against x. If $A[s] \geq x$, then we can simply increment s. Otherwise we increment q, swap $A[s]$ with $A[q]$, and then increment s. Notice that in either case, the invariant is still maintained. In the first case this is obvious. In the second case, $A[q]$ now holds a value that is less than x, and $A[s-1]$ now holds a value that is greater than or equal to x. The algorithm ends when $s = r$, meaning that all of the elements have been processed. To finish things off we swap $A[p]$ (the pivot) with $A[q]$, and return the value of q. Here is the complete code:

Partition

```
Partition(int p, int r, array A) {            // 3-way partition of A[p..r]
    x = A[p]                                  // pivot item in A[p]
    q = p
    for s = p+1 to r do {
        if (A[s] < x) {
            q = q+1
            swap A[q] with A[s]
        }
    }
    swap A[p] with A[q]                       // put the pivot into final position
    return q                                  // return location of pivot
}
```

An example is shown below.

Lecture 15: QuickSort

(Tuesday, Mar 17, 1998)

Revised: March 18. Fixed a bug in the analysis.

Read: Chapt 8 in CLR. My presentation and analysis are somewhat different than the text's.

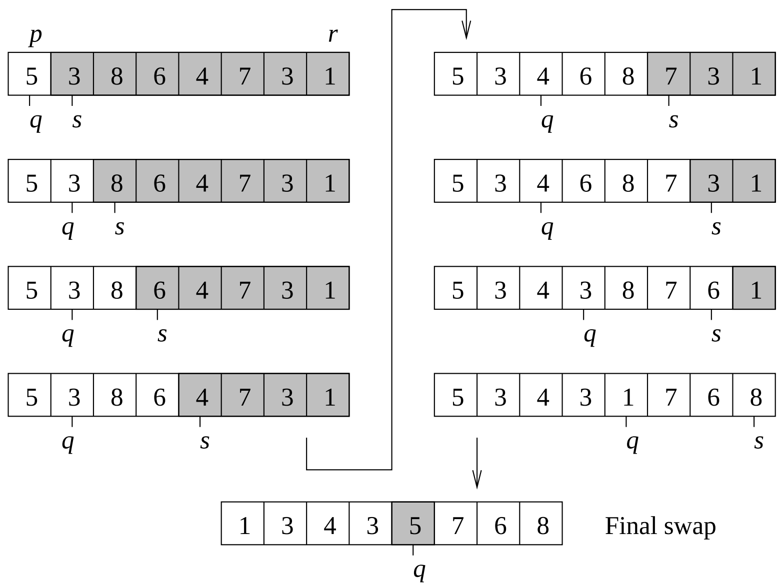


Figure 15: Partitioning example.

QuickSort and Randomized Algorithms: Early in the semester we discussed the fact that we usually study the worst-case running times of algorithms, but sometimes average-case is a more meaningful measure. Today we will study QuickSort. It is a worst-case $\Theta(n^2)$ algorithm, whose expected-case running time is $\Theta(n \log n)$.

We will present QuickSort as a *randomized* algorithm, that is, an algorithm which makes random choices. There are two common types of randomized algorithms:

Monte Carlo algorithms: These algorithms may produce the wrong result, but the probability of this occurring can be made arbitrarily small by the user. Usually the lower you make this probability, the longer the algorithm takes to run.

Las Vegas algorithms: These algorithms always produce the correct result, but the running time is a random variable. In these cases the expected running time, averaged over all possible random choices is the measure of the algorithm's running time.

The most well known Monte Carlo algorithm is one for determining whether a number is prime. This is an important problem in cryptography. The QuickSort algorithm that we will discuss today is an example of a Las Vegas algorithm. Note that QuickSort does not need to be implemented as a randomized algorithm, but as we shall see, this is generally considered the safest implementation.

QuickSort Overview: QuickSort is also based on the divide-and-conquer design paradigm. Unlike MergeSort where most of the work is done after the recursive call returns, in QuickSort the work is done before the recursive call is made. Here is an overview of QuickSort. Note the similarity with the selection algorithm, which we discussed earlier. Let $A[p..r]$ be the (sub)array to be sorted. The initial call is to $A[1..n]$.

Basis: If the list contains 0 or 1 elements, then return.

Select pivot: Select a random element x from the array, called the *pivot*.

Partition: Partition the array in three subarrays, those elements $A[1..q-1] \le x$, $A[q] = x$, and $A[q+1..n] \ge x$.

Recurse: Recursively sort $A[1..q-1]$ and $A[q+1..n]$.

The pseudocode for QuickSort is given below. The initial call is QuickSort(1, n, A). The Partition routine was discussed last time. Recall that Partition assumes that the pivot is stored in the first element of A. Since we want a random pivot, we pick a random index i from p to r, and then swap $A[i]$ with $A[p]$.

QuickSort

```
QuickSort(int p, int r, array A) {              // Sort A[p..r]
    if (r <= p) return                          // 0 or 1 items, return
    i = a random index from [p..r]              // pick a random element
    swap A[i] with A[p]                         // swap pivot into A[p]
    q = Partition(p, r, A)                      // partition A about pivot
    QuickSort(p, q-1, A)                        // sort A[p..q-1]
    QuickSort(q+1, r, A)                        // sort A[q+1..r]
}
```

QuickSort Analysis: The correctness of QuickSort should be pretty obvious. However its analysis is not so obvious. It turns out that the running time of QuickSort depends heavily on how good a job we do in selecting the pivot. In particular, if the rank of the pivot (recall that this means its position in the final sorted list) is very large or very small, then the partition will be unbalanced. We will see that unbalanced partitions (like unbalanced binary trees) are bad, and result is poor running times. However, if the rank of the pivot is anywhere near the middle portion of the array, then the split will be reasonably well balanced, and the overall running time will be good. Since the pivot is chosen at random by our algorithm, we may do well most of the time and poorly occasionally. We will see that the expected running time is $O(n \log n)$.

Worst-case Analysis: Let's begin by considering the worst-case performance, because it is easier than the average case. Since this is a recursive program, it is natural to use a recurrence to describe its running time. But unlike MergeSort, where we had control over the sizes of the recursive calls, here we do not. It depends on how the pivot is chosen. Suppose that we are sorting an array of size n, $A[1..n]$, and further suppose that the pivot that we select is of rank q, for some q in the range 1 to n. It takes $\Theta(n)$ time to do the partitioning and other overhead, and we make two recursive calls. The first is to the subarray $A[1..q-1]$ which has $q-1$ elements, and the other is to the subarray $A[q+1..n]$ which has $r-(q+1)+1 = r-q$ elements. So if we ignore the Θ (as usual) we get the recurrence:

$$T(n) = T(q-1) + T(n-q) + n.$$

This depends on the value of q. To get the worst case, we maximize over all possible values of q. As a basis we have that $T(0) = T(1) = \Theta(1)$. Putting this together we have

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \max_{1 \leq q \leq n}(T(q-1) + T(n-q) + n) & \text{otherwise.} \end{cases}$$

Recurrences that have max's and min's embedded in them are very messy to solve. The key is determining which value of q gives the maximum. (A rule of thumb of algorithm analysis is that the worst cases tends to happen either at the extremes or in the middle. So I would plug in the value $q = 1$, $q = n$, and $q = n/2$ and work each out.) In this case, the worst case happens at either of the extremes (but see the book for a more careful analysis based on an analysis of the second derivative). If we expand the recurrence in the case $q = 1$ we get:

$$\begin{aligned}
T(n) &\le T(0) + T(n-1) + n \;=\; 1 + T(n-1) + n \\
&= T(n-1) + (n+1) \\
&= T(n-2) + n + (n+1) \\
&= T(n-3) + (n-1) + n + (n+1) \\
&= T(n-4) + (n-2) + (n-1) + n + (n+1) \\
&= \ldots \\
&= T(n-k) + \sum_{i=-1}^{k-2} (n-i).
\end{aligned}$$

For the basis, $T(1) = 1$ we set $k = n-1$ and get

$$\begin{aligned}
T(n) &\le T(1) + \sum_{i=-1}^{n-3} (n-i) \\
&= 1 + (3 + 4 + 5 + \ldots + (n-1) + n + (n+1)) \\
&\le \sum_{i=1}^{n+1} i \;=\; \frac{(n+1)(n+2)}{2} \in O(n^2).
\end{aligned}$$

In fact, a more careful analysis reveals that it is $\Theta(n^2)$ in this case.

Average-case Analysis: Next we show that in the average case QuickSort runs in $\Theta(n \log n)$ time. When we talked about average-case analysis at the beginning of the semester, we said that it depends on some assumption about the distribution of inputs. However, in this case, the analysis does not depend on the input distribution at all—it only depends on the random choices that the algorithm makes. This is good, because it means that the analysis of the algorithm's performance is the same for all inputs. In this case the average is computed over all possible random choices that the algorithm might make for the choice of the pivot index in the second step of the QuickSort procedure above.

To analyze the average running time, we let $T(n)$ denote the average running time of QuickSort on a list of size n. It will simplify the analysis to assume that all of the elements are distinct. The algorithm has n random choices for the pivot element, and each choice has an equal probability of $1/n$ of occuring. So we can modify the above recurrence to compute an average rather than a max, giving:

$$T(n) = \begin{cases} 1 & \text{if } n \le 1 \\ \frac{1}{n}\sum_{q=1}^{n}(T(q-1) + T(n-q) + n) & \text{otherwise.} \end{cases}$$

This is not a standard recurrence, so we cannot apply the Master Theorem. Expansion is possible, but rather tricky. Instead, we will attempt a constructive induction to solve it. We know that we want a $\Theta(n \log n)$ running time, so let's try $T(n) \le an \lg n + b$. (Properly we should write $\lceil \lg n \rceil$ because unlike MergeSort, we cannot assume that the recursive calls will be made on array sizes that are powers of 2, but we'll be sloppy because things will be messy enough anyway.)

Theorem: There exist a constant c such that $T(n) \le cn \ln n$, for all $n \ge 2$. (Notice that we have replaced $\lg n$ with $\ln n$. This has been done to make the proof easier, as we shall see.)

Proof: The proof is by constructive induction on n. For the basis case $n = 2$ we have

$$\begin{aligned} T(2) &= \frac{1}{2}\sum_{q=1}^{2}(T(q-1)+T(2-q)+2) \\ &= \frac{1}{2}((T(0)+T(1)+2)+(T(1)+T(0)+2) = \frac{8}{2} = 4. \end{aligned}$$

We want this to be at most $c2\ln 2$ implying that $c \geq 4/(2\ln 2) \approx 2.88$.

For the induction step, we assume that $n \geq 3$, and the induction hypothesis is that for any $n' < n$, we have $T(n') \leq cn' \ln n'$. We want to prove it is true for $T(n)$. By expanding the definition of $T(n)$, and moving the factor of n outside the sum we have:

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{q=1}^{n}(T(q-1)+T(n-q)+n) \\ &= \frac{1}{n}\sum_{q=1}^{n}(T(q-1)+T(n-q))+n. \end{aligned}$$

Observe that if we split the sum into two sums, they both add the same values $T(0) + T(1) + \ldots + T(n-1)$, just that one counts up and the other counts down. Thus we can replace this with $2\sum_{q=0}^{n-1} T(q)$. Because they don't follow the formula, we'll extract $T(0)$ and $T(1)$ and treat them specially. If we make this substitution and apply the induction hypothesis to the remaining sum we have (which we can because $q < n$) we have

$$\begin{aligned} T(n) &= \frac{2}{n}\left(\sum_{q=0}^{n-1}T(q)\right)+n \;=\; \frac{2}{n}\left(T(0)+T(1)+\sum_{q=2}^{n-1}T(q)\right)+n \\ &\leq \frac{2}{n}\left(1+1+\sum_{q=2}^{n-1}(cq\lg q)\right)+n \\ &= \frac{2c}{n}\left(\sum_{q=2}^{n-1}(cq\ln q)\right)+n+\frac{4}{n}. \end{aligned}$$

We have never seen this sum before. Later we will show that

$$S(n) = \sum_{q=2}^{n-1} q\ln q \leq \frac{n^2}{2}\ln n - \frac{n^2}{4}.$$

Assuming this for now, we have

$$\begin{aligned} T(n) &= \frac{2c}{n}\left(\frac{n^2}{2}\ln n - \frac{n^2}{4}\right)+n+\frac{4}{n} \\ &= cn\ln n - \frac{cn}{2}+n+\frac{4}{n} \\ &= cn\ln n + n\left(1-\frac{c}{2}\right)+\frac{4}{n}. \end{aligned}$$

To finish the proof, we want all of this to be at most $cn\ln n$. If we cancel the common $cn\ln n$ we see that this will be true if we select c such that

$$n\left(1-\frac{c}{2}\right)+\frac{c}{n} \leq 0.$$

We claim that if we select $c = 3$, this will be true. In particular, if we plug this value in for c, simplify, and use the fact that $n \geq 3$, we have:

$$n\left(1 - \frac{3}{2}\right) + \frac{3}{n} \;=\; \frac{3}{n} - \frac{n}{2} \;\leq\; 1 - \frac{3}{2} \;=\; -\frac{1}{2} \;\leq\; 0,$$

as desired. From the basis case we have $c \geq 2.88$, so we may choose $c = 3$ to satisfy all the constraints.

The Leftover Sum: The only missing element to the proof is dealing with the sum

$$S(n) = \sum_{q=2}^{n-1} q \ln q.$$

To bound this, recall the integration formula for bounding summations (which we paraphrase here). For any monotonically increasing function $f(x)$

$$\sum_{i=a}^{b-1} f(i) \leq \int_a^b f(x)dx.$$

The function $f(x) = x \ln x$ is monotonically increasing, and so we have

$$S(n) \leq \int_2^n x \ln x dx.$$

If you are a calculus macho man, then you can integrate this by parts, and if you are a calculus wimp (like me) then you can look it up in a book of integrals

$$\int_2^n x \ln x dx = \frac{x^2}{2}\ln x - \frac{x^2}{4}\bigg|_{x=2}^{n} = \left(\frac{n^2}{2}\ln n - \frac{n^2}{4}\right) - (2\ln 2 - 1) \leq \frac{n^2}{2}\ln n - \frac{n^2}{4}.$$

This completes the summation bound, and hence the entire proof.

Summary: So even though the worst-case running time of QuickSort is $\Theta(n^2)$, the average-case running time is $\Theta(n \log n)$. Although we did not show it, it turns out that this doesn't just happen much of the time. For large values of n, the running time is $\Theta(n \log n)$ with high probability. In order to get $\Theta(n^2)$ time the algorithm must make poor choices for the pivot at virtually every step. Poor choices are rare, and so continuously making poor choices are very rare. You might ask, could we make QuickSort deterministic $\Theta(n \log n)$ by calling the selection algorithm to use the median as the pivot. The answer is that this would work, but the resulting algorithm would be so slow practically that no one would ever use it.

QuickSort (like MergeSort) is not formally an in-place sorting algorithm, because it does make use of a recursion stack. In MergeSort and in the expected case for QuickSort, the size of the stack is $O(\log n)$, so this is not really a problem.

QuickSort is the most popular algorithm for implementation because its actual performance (on typical modern architectures) is so good. The reason for this stems from the fact that (unlike Heapsort) which can make large jumps around in the array, the main work in QuickSort (in partitioning) spends most of its time accessing elements that are close to one another. The reason it tends to outperform MergeSort (which also has good locality of reference) is that most comparisons are made against the pivot element, which can be stored in a register. In MergeSort we are always comparing two array elements against each other. The most efficient versions of QuickSort uses the recursion for large subarrays, but once the sizes of the subarray falls below some minimum size (e.g. 20) it switches to a simple iterative algorithm, such as selection sort.

Lecture 16: Lower Bounds for Sorting

(Thursday, Mar 19, 1998)
Read: Chapt. 9 of CLR.

Review of Sorting: So far we have seen a number of algorithms for sorting a list of numbers in ascending order. Recall that an *in-place* sorting algorithm is one that uses no additional array storage (however, we allow QuickSort to be called in-place even though they need a stack of size $O(\log n)$ for keeping track of the recursion). A sorting algorithm is *stable* if duplicate elements remain in the same relative position after sorting.

Slow Algorithms: Include BubbleSort, InsertionSort, and SelectionSort. These are all simple $\Theta(n^2)$ in-place sorting algorithms. BubbleSort and InsertionSort can be implemented as stable algorithms, but SelectionSort cannot (without significant modifications).

Mergesort: Mergesort is a stable $\Theta(n \log n)$ sorting algorithm. The downside is that MergeSort is the only algorithm of the three that requires additional array storage, implying that it is not an in-place algorithm.

Quicksort: Widely regarded as the *fastest* of the fast algorithms. This algorithm is $O(n \log n)$ in the *expected case*, and $O(n^2)$ in the worst case. The probability that the algorithm takes asymptotically longer (assuming that the pivot is chosen randomly) is extremely small for large n. It is an in-place sorting algorithm but is not stable.

Heapsort: Heapsort is based on a nice data structure, called a *heap*, which is a fast priority queue. Elements can be inserted into a heap in $O(\log n)$ time, and the largest item can be extracted in $O(\log n)$ time. (It is also easy to set up a heap for extracting the smallest item.) If you only want to extract the k largest values, a heap can allow you to do this is $O(n + k \log n)$ time. It is an in-place algorithm, but it is not stable.

Lower Bounds for Comparison-Based Sorting: Can we sort faster than $O(n \log n)$ time? We will give an argument that if the sorting algorithm is based solely on making comparisons between the keys in the array, then it is impossible to sort more efficiently than $\Omega(n \log n)$ time. Such an algorithm is called a *comparison-based* sorting algorithm, and includes all of the algorithms given above.

Virtually all known general purpose sorting algorithms are based on making comparisons, so this is not a very restrictive assumption. This does not preclude the possibility of a sorting algorithm whose actions are determined by other types of operations, for example, consulting the individual bits of numbers, performing arithmetic operations, indexing into an array based on arithmetic operations on keys.

We will show that any *comparison-based* sorting algorithm for a input sequence $\langle a_1, a_2, \ldots, a_n \rangle$ must make at least $\Omega(n \log n)$ comparisons in the worst-case. This is still a difficult task if you think about it. It is easy to show that a problem *can* be solved fast (just give an algorithm). But to show that a problem *cannot* be solved fast you need to reason in some way about all the possible algorithms that might ever be written. In fact, it seems surprising that you could even hope to prove such a thing. The catch here is that we are limited to using comparison-based algorithms, and there is a clean mathematical way of characterizing all such algorithms.

Decision Tree Argument: In order to prove lower bounds, we need an abstract way of modeling "any possible" comparison-based sorting algorithm, we model such algorithms in terms of an abstract model called a *decision tree*.

In a *comparison-based* sorting algorithm only comparisons between the keys are used to determine the action of the algorithm. Let $\langle a_1, a_2, \ldots, a_n \rangle$ be the input sequence. Given two elements, a_i and a_j, their relative order can only be determined by the results of comparisons like $a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, and $a_i > a_j$.

A decision tree is a mathematical representation of a sorting algorithm (for a fixed value of n). Each node of the decision tree represents a comparison made in the algorithm (e.g., $a_4 : a_7$), and the two branches represent the possible results, for example, the left subtree consists of the remaining comparisons made under the assumption that $a_4 \le a_7$ and the right subtree for $a_4 > a_7$. (Alternatively, one might be labeled with $a_4 < a_7$ and the other with $a_4 \ge a_7$.)

Observe that once we know the value of n, then the "action" of the sorting algorithm is completely determined by the results of its comparisons. This action may involve moving elements around in the array, copying them to other locations in memory, performing various arithmetic operations on non-key data. But the bottom-line is that at the end of the algorithm, the keys will be permuted in the final array in some way. Each leaf in the decision tree is labeled with the final permutation that the algorithm generates after making all of its comparisons.

To make this more concrete, let us assume that $n = 3$, and let's build a decision tree for SelectionSort. Recall that the algorithm consists of two phases. The first finds the smallest element of the entire list, and swaps it with the first element. The second finds the smaller of the remaining two items, and swaps it with the second element. Here is the decision tree (in outline form). The first comparison is between a_1 and a_2. The possible results are:

$a_1 \le a_2$: Then a_1 is the current minimum. Next we compare a_1 with a_3 whose results might be either:

- **$a_1 \le a_3$:** Then we know that a_1 is the minimum overall, and the elements remain in their original positions. Then we pass to phase 2 and compare a_2 with a_3. The possible results are:
  - **$a_2 \le a_3$:** Final output is $\langle a_1, a_2, a_3 \rangle$.
  - **$a_2 > a_3$:** These two are swapped and the final output is $\langle a_1, a_3, a_2 \rangle$.
- **$a_1 > a_3$:** Then we know that a_3 is the minimum is the overall minimum, and it is swapped with a_1. The we pass to phase 2 and compare a_2 with a_1 (which is now in the third position of the array) yielding either:
  - **$a_2 \le a_1$:** Final output is $\langle a_3, a_2, a_1 \rangle$.
  - **$a_2 > a_1$:** These two are swapped and the final output is $\langle a_3, a_1, a_2 \rangle$.

$a_1 > a_2$: Then a_2 is the current minimum. Next we compare a_2 with a_3 whose results might be either:

- **$a_2 \le a_3$:** Then we know that a_2 is the minimum overall. We swap a_2 with a_1, and then pass to phase 2, and compare the remaining items a_1 and a_3. The possible results are:
  - **$a_1 \le a_3$:** Final output is $\langle a_2, a_1, a_3 \rangle$.
  - **$a_1 > a_3$:** These two are swapped and the final output is $\langle a_2, a_3, a_1 \rangle$.
- **$a_2 > a_3$:** Then we know that a_3 is the minimum is the overall minimum, and it is swapped with a_1. We pass to phase 2 and compare a_2 with a_1 (which is now in the third position of the array) yielding either:
  - **$a_2 \le a_1$:** Final output is $\langle a_3, a_2, a_1 \rangle$.
  - **$a_2 > a_1$:** These two are swapped and the final output is $\langle a_3, a_1, a_2 \rangle$.

The final decision tree is shown below. Note that there are some nodes that are unreachable. For example, in order to reach the fourth leaf from the left it must be that $a_1 \le a_2$ and $a_1 > a_2$, which cannot both be true. Can you explain this? (The answer is that virtually all sorting algorithms, especially inefficient ones like selection sort, may make comparisons that are redundant, in the sense that their outcome has already been determined by earlier comparisons.) As you can see, converting a complex sorting algorithm like HeapSort into a decision tree for a large value of n will be very tedious and complex, but I hope you are convinced by this exercise that it can be done in a simple mechanical way.

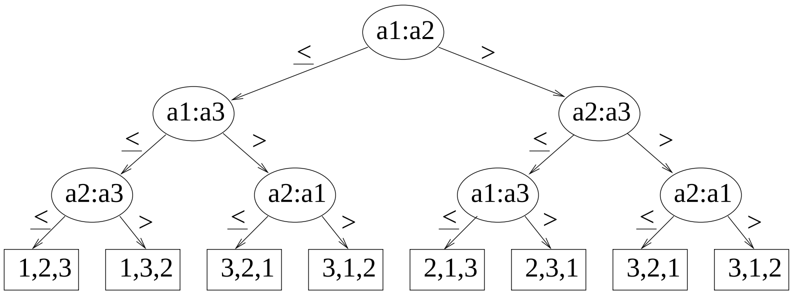


Figure 16: Decision Tree for SelectionSort on 3 keys.

Using Decision Trees for Analyzing Sorting: Consider any sorting algorithm. Let $T(n)$ be the maximum number of comparisons that this algorithm makes on any input of size n. Notice that the running time fo the algorithm must be at least as large as $T(n)$, since we are not counting data movement or other computations at all. The algorithm defines a decision tree. Observe that the height of the decision tree is exactly equal to $T(n)$, because any path from the root to a leaf corresponds to a sequence of comparisons made by the algorithm.

As we have seen earlier, any binary tree of height $T(n)$ has at most $2^{T(n)}$ leaves. This means that this sorting algorithm can *distinguish* between at most $2^{T(n)}$ different final actions. Let's call this quantity $A(n)$, for the number of different final actions the algorithm can take. Each action can be thought of as a specific way of permuting the oringinal input to get the sorted output.

How many possible actions must any sorting algorithm distinguish between? If the input consists of n distinct numbers, then those numbers could be presented in any of $n!$ different permutations. For each different permutation, the algorithm must "unscramble" the numbers in an essentially different way, that is it must take a different action, implying that $A(n) \geq n!$. (Again, $A(n)$ is usually not exactly equal to $n!$ because most algorithms contain some redundant unreachable leaves.)

Since $A(n) \leq 2^{T(n)}$ we have $2^{T(n)} \geq n!$, implying that

$$T(n) \geq \lg(n!).$$

We can use *Stirling's approximation* for $n!$ (see page 35 in CLR) yielding:

$$\begin{aligned}
n! &\geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \\
T(n) &\geq \lg\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) \\
&= \lg\sqrt{2\pi n} + n\lg n - n\lg e \;\in\; \Omega(n\log n).
\end{aligned}$$

Thus we have, the following theorem.

Theorem: Any comparison-based sorting algorithm has worst-case running time $\Omega(n\log n)$.

This can be generalized to show that the *average-case* time to sort is also $\Omega(n \log n)$ (by arguing about the average height of a leaf in a tree with at least $n!$ leaves). The lower bound on sorting can be generalized to provide lower bounds to a number of other problems as well.

Lecture 17: Linear Time Sorting

(Tuesday, Mar 31, 1998)
Read: Chapt. 9 of CLR.

Linear Time Sorting: Last time we presented a proof that it is not possible to sort faster than $\Omega(n \log n)$ time assuming that the algorithm is based on making 2-way comparisons. Recall that the argument was based on showing that any comparison-based sorting could be represented as a decision tree, the decision tree must have at least $n!$ leaves, to distinguish between the $n!$ different permutations in which the keys could be input, and hence its height must be at least $\lg(n!) \in \Omega(n \lg n)$.

This lower bound implies that if we hope to sort numbers faster than in $O(n \log n)$ time, we cannot do it by making comparisons alone. Today we consider the question of whether it is possible to sort without the use of comparisons. They answer is yes, but only under very restrictive circumstances. Many applications involve sorting small integers (e.g. sorting characters, exam scores, last four digits of a social security number, etc.). We present three algorithms based on the theme of speeding up sorting in special cases, by *not* making comparisons.

Counting Sort: Counting sort assumes that each input is an integer in the range from 1 to k. The algorithm sorts in $\Theta(n+k)$ time. If k is known to be $\Theta(n)$, then this implies that the resulting sorting algorithm is $\Theta(n)$ time.

The basic idea is to determine, for each element in the input array, its *rank* in the final sorted array. Recall that the rank of a item is the number of elements in the array that are less than or equal to it. Notice that once you know the rank of every element, you sort by simply copying each element to the appropriate location of the final sorted output array. The question is how to find the rank of an element without comparing it to the other elements of the array? Counting sort uses the following three arrays. As usual $A[1..n]$ is the input array. Recall that although we usually think of A as just being a list of numbers, it is actually a list of records, and the numeric value is the *key* on which the list is being sorted. In this algorithm we will be a little more careful to distinguish the entire record $A[j]$ from the key $A[j].key$.

We use three arrays:

$A[1..n]$: Holds the initial input. $A[j]$ is a record. $A[j].key$ is the integer key value on which to sort.

$B[1..n]$: Array of records which holds the sorted output.

$R[1..k]$: An array of integers. $R[x]$ is the rank of x in A, where $x \in [1..k]$.

The algorithm is remarkably simple, but deceptively clever. The algorithm operates by first constructing R. We do this in two steps. First we set $R[x]$ to be the number of elements of $A[j]$ whose key is equal to x. We can do this initializing R to zero, and then for each j, from 1 to n, we increment $R[A[j].key]$ by 1. Thus, if $A[j].key = 5$, then the 5th element of R is incremented, indicating that we have seen one more 5. To determine the number of elements that are less than or equal to x, we replace $R[x]$ with the sum of elements in the subarray $R[1..x]$. This is done by just keeping a running total of the elements of R.

Now $R[x]$ now contains the rank of x. This means that if $x = A[j].key$ then the final position of $A[j]$ should be at position $R[x]$ in the final sorted array. Thus, we set $B[R[x]] = A[j]$. Notice that this copies the entire record, not just the key value. There is a subtlety here however. We need to be careful if there are duplicates, since we do not want them to overwrite the same location of B. To do this, we decrement $R[i]$ after copying.

Counting Sort

```
CountingSort(int n, int k, array A, array B) {  // sort A[1..n] to B[1..n]
    for x = 1 to k do R[x] = 0                  // initialize R
    for j = 1 to n do R[A[j].key]++             // R[x] = #(A[j] == x)
    for x = 2 to k do R[x] += R[x-1]            // R[x] = rank of x
    for j = n downto 1 do {                     // move each element of A to B
        x = A[j].key                            // x = key value
        B[R[x]] = A[j]                          // R[x] is where to put it
        R[x]--                                  // leave space for duplicates
    }
}
```

There are four (unnested) loops, executed k times, n times, $k-1$ times, and n times, respectively, so the total running time is $\Theta(n+k)$ time. If $k = O(n)$, then the total running time is $\Theta(n)$. The figure below shows an example of the algorithm. You should trace through a few examples, to convince yourself how it works.

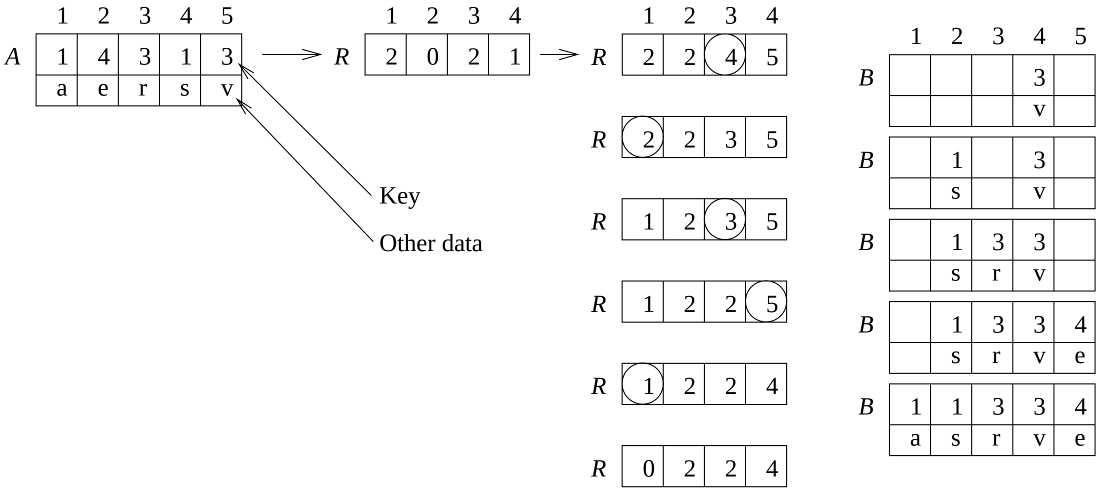


Figure 17: Counting Sort.

Obviously this not an in-place sorting algorithm (we need two additional arrays). However it is a stable sorting algorithm. I'll leave it as an exercise to prove this. (As a hint, notice that the last loop runs down from n to 1. It would not be stable if the loop were running the other way.)

Radix Sort: The main shortcoming of counting sort is that it is only really (due to space requirements) for small integers. If the integers are in the range from 1 to 1 million, we may not want to allocate an array of a million elements. Radix sort provides a nice way around this by sorting numbers one digit at a time. Actually, what constitutes a "digit" is up to the implementor. For example, it is typically more convenient to sort by bytes rather than digits (especially for sorting character strings). There is a tradeoff between the space and time.

The idea is very simple. Let's think of our list as being composed of n numbers, each having d decimal digits (or digits in any base). Let's suppose that we have access to a stable sorting algorithm, like Counting Sort. To sort these numbers we can simply sort repeatedly, starting at the lowest order digit, and finishing with the highest order digit. Since the sorting algorithm is stable, we know that if the numbers are already sorted with respect to low order digits, and then later we sort with respect to high order digits, numbers having the same high order digit will remain sorted with respect to their low order digit. As usual, let $A[1..n]$ be the array to sort, and let d denote the number of digits in A. We will not discuss how it is that A is broken into digits, but this might be done through bit manipulations (shifting and masking off bits) or by accessing elements byte-by-byte, etc.

Radix Sort

```
RadixSort(int n, int d, array A) {             // sort A[1..n] with d digits
    for i = 1 to d do {
        Sort A (stably) with respect to i-th lowest order digit;
    }
}
```

Here is an example.

576		49[4]		9[5]4		[1]76		176
494		19[4]		5[7]6		[1]94		194
194		95[4]		1[7]6		[2]78		278
296	$\Longrightarrow$	57[6]	$\Longrightarrow$	2[7]8	$\Longrightarrow$	[2]96	$\Longrightarrow$	296
278		29[6]		4[9]4		[4]94		494
176		17[6]		1[9]4		[5]76		576
954		27[8]		2[9]6		[9]54		954

The running time is clearly $\Theta(d(n + k))$ where d is the number of digits, n is the length of the list, and k is the number of values a digit can have. This is usually a constant, but the algorithm's running time will be $\Theta(dn)$ as long as $k \in O(n)$.

Notice that we can be quite flexible in the definition of what a "digit" is. It can be any number in the range from 1 to cn for some constant c, and we will still have an $\Theta(n)$ time algorithm. For example, if we have $d = 2$ and set $k = n$, then we can sort numbers in the range $n * n = n^2$ in $\Theta(n)$ time. In general, this can be used to sort numbers in the range from 1 to n^d in $\Theta(dn)$ time.

At this point you might ask, since a computer integer word typically consists of 32 bits (4 bytes), then doesn't this imply that we can sort any array of integers in $O(n)$ time (by applying radix sort on each of the $d = 4$ bytes)? The answer is yes, subject to this word-length restriction. But you should be careful about attempting to make generalizations when the sizes of the numbers are not bounded. For example, suppose you have n keys and there are no duplicate values. Then it follows that you need at least $B = \lceil \lg n \rceil$ bits to store these values. The number of bytes is $d = \lceil B/8 \rceil$. Thus, if you were to apply radix sort in this situation, the running time would be $\Theta(dn) = \Theta(n \log n)$. So there is no real asymptotic savings here. Furthermore, the locality of reference behavior of Counting Sort (and hence of RadixSort) is not as good as QuickSort. Thus, it is not clear whether it is really faster to use RadixSort over QuickSort. This is at a level of similarity, where it would probably be best to implement both algorithms on your particular machine to determine which is really faster.

Lecture 18: Review for Second Midterm

(Thursday, Apr 2, 1998)

General Remarks: Up to now we have covered the basic techniques for analyzing algorithms (asymptotics, summations, recurrences, induction), have discussed some algorithm design techniques (divide-and-conquer in particular), and have discussed sorting algorithm and related topics. Recall that our goal is to provide you with the necessary tools for designing and analyzing efficient algorithms.

Material from Text: You are only responsible for material that has been covered in class or on class assignments. However it is always a good idea to see the text to get a better insight into some of the topics we have covered. The relevant sections of the text are the following.

- Review Chapts 1: InsertionSort and MergeSort.
- Chapt 7: Heaps, HeapSort. Look at Section 7.5 on priority queues, even though we didn't cover it in class.
- Chapt 8: QuickSort. You are responsible for the partitioning algorithm which we gave in class, not the one in the text. Section 8.2 gives some good intuition on the analysis of QuickSort.
- Chapt 9 (skip 9.4): Lower bounds on sorting, CountingSort, RadixSort.
- Chapt. 10 (skip 10.1): Selection. Read the analysis of the average case of selection. It is similar to the QuickSort analysis.

You are also responsible for anything covered in class.

Cheat Sheets: The exam is closed-book, closed-notes, but you are allowed two sheets of notes (front and back). You should already have the cheat sheet from the first exam with basic definitions of asymptotics, important summations, Master theorem. Also add Stirling's approximation (page 35), and the integration formula for summations (page 50). You should be familiar enough with each algorithm presented in class that you could work out an example by hand, without refering back to your cheat sheet. But it is a good idea to write down a brief description of each algorithm. For example, you might be asked to show the result of BuildHeap on an array, or show how to apply the Partition algorithm used in QuickSort.

Keep track of algorithm running times and their limitations. For example, if you need an efficient stable sorting algorithm, MergeSort is fine, but both HeapSort and QuickSort are not stable. You can sort short integers in $\Theta(n)$ time through CountingSort, but you cannot use this algorithm to sort arbitrary numbers, such as reals.

Sorting issues: We discussed the following issues related to sorting.

Slow Algorithms: BubbleSort, InsertionSort, SelectionSort are all simple $\Theta(n^2)$ algorithm. They are fine for small inputs. They are all in-place sorting algorithms (they use no additional array storage), and BubbleSort and InsertionSort are stable sorting algorithms (if implemented carefully).

MergeSort: A divide-and-conquer $\Theta(n \log n)$ algorithm. It is stable, but requires additional array storage for merging, and so it is not an in-place algorithm.

HeapSort: A $\Theta(n \log n)$ algorithm which uses a clever data structure, called a heap. Heaps are a nice way of implementing a priority queue data structure, allowing insertions, and extracting the maximum in $\Theta(\log n)$ time, where n is the number of active elements in the heap. Remember that a heap can be built in $\Theta(n)$ time. HeapSort is not stable, but it is an in-place sorting algorithm.

QuickSort: The algorithm is based on selecting a pivot value. If chosen randomly, then the expected time is $\Theta(n \log n)$, but the worst-case is $\Theta(n^2)$. However the worst-case occurs so rarely that people usually do not worry about it. This algorithm is not stable, but it is considered an in-place sorting algorithm even though it does require some additional array storage. It implicitly requires storage for the recursion stack, but the expected depth of the recursion is $O(\log n)$, so this is not too bad.

Lower bounds: Assuming comparisons are used, you cannot sort faster than $\Omega(n \log n)$ time. This is because any comparison-based algorithm can be written as a decision tree, and because there are $n!$ possible outcomes to sorting, even a perfectly balanced tree would require height of at least $O(\log n!) = O(n \log n)$.

Counting sort: If you are sorting n small integers (in the range of 1 to k) then this algorithm will sort them in $\Theta(n + k)$ time. Recall that the algorithm is based on using the elements as indices to an array. In this way it circumvents the lower bound argument.

Radix sort: If you are sorting n integers that have been broken into d digits (each of constant size), you can sort them in $O(dn)$ time.

What sort of questions might there be? Some will ask you to about the properties of these sorting algorithms, or asking which algorithm would be most appropriate to use in a certain circumstance. Others will ask you to either reason about the internal operations of the algorithms, or ask you to extend these algorithms for other purposes. Finally, there may be problems asking you to devise algorithms to solve some sort of novel sorting problem.

Lecture 19: Second Midterm Exam

(Tuesday, April 7, 1998)
Second midterm exam today. No lecture.

Lecture 20: Introduction to Graphs

(Thursday, April 9, 1998)
Read: Sections 5.4, 5.5.

Graph Algorithms: For the next few weeks, we will be discussing a number of various topics. One involves algorithms on *graphs*. Intuitively, a graph is a collection of vertices or nodes, connected by a collection of edges. Graphs are very important discrete structures because they are a very flexible mathematical model for many application problems. Basically, any time you have a set of objects, and there is some "connection" or "relationship" or "interaction" between pairs of objects, a graph is a good way to model this. Examples of graphs in application include *communication* and *transportation networks*, *VLSI* and other sorts of *logic circuits*, *surface meshes* used for shape description in computer-aided design and geographic information systems, *precedence constraints* in scheduling systems. The list of application is almost too long to even consider enumerating it.

Most of the problems in computational graph theory that we will consider arise because they are of importance to one or more of these application areas. Furthermore, many of these problems form the basic building blocks from which more complex algorithms are then built.

Graphs and Digraphs: A *directed graph* (or digraph) $G = (V, E)$ consists of a finite set of *vertices* V (also called *nodes*) and E is a binary relation on V (i.e. a set of *ordered* pairs from V) called the *edges*.

For example, the figure below (left) shows a directed graph. Observe that *self-loops* are allowed by this definition. Some definitions of graphs disallow this. Multiple edges are not permitted (although the edges (v, w) and (w, v) are distinct). This shows the graph $G = (V, E)$ where $V = \{1, 2, 3\}$ and $E = \{(1, 1), (1, 2), (2, 3), (3, 2), (1, 3)\}$.

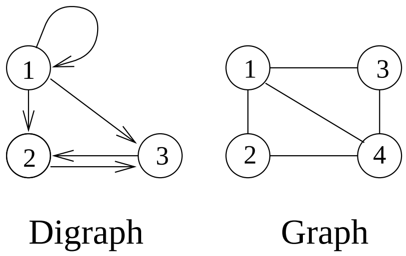


Figure 18: Digraph and graph example.

In an *undirected graph* (or just *graph*) $G = (V, E)$ the edge set consists of unordered pairs of distinct vertices (thus self-loops are not allowed). The figure above (right) shows the graph $G = (V, E)$, where $V = \{1, 2, 3, 4\}$ and the edge set is $E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\}$.

We say that vertex w is *adjacent* to vertex v if there is an edge from v to w. In an undirected graph, we say that an edge is *incident* on a vertex if the vertex is an endpoint of the edge. In a directed graph we will often say that an edge either *leaves* or *enters* a vertex.

A digraph or undirected graph is said to be *weighted* if its edges are labelled with numeric weights. The meaning of the weight is dependent on the application, e.g. distance between vertices or flow capacity through the edge.

Observe that directed graphs and undirected graphs are different (but similar) objects mathematically. Certain notions (such as path) are defined for both, but other notions (such as connectivity) are only defined for one.

In a digraph, the number of edges coming out of a vertex is called the *out-degree* of that vertex, and the number of edges coming in is called the *in-degree*. In an undirected graph we just talk about the *degree* of a vertex, as the number of edges which are *incident* on this vertex. By the *degree* of a graph, we usually mean the maximum degree of its vertices.

In a directed graph, each edge contributes 1 to the in-degree of a vertex and contributes one to the out-degree of each vertex, and thus we have

Observation: For a digraph $G = (V, E)$,

$$\sum_{v \in V} \textit{in-deg}(v) = \sum_{v \in V} \textit{out-deg}(v) = |E|.$$

($|E|$ means the cardinality of the set E, i.e. the number of edges).

In an undirected graph each edge contributes once to the outdegree of two different edges and thus we have

Observation: For an undirected graph $G = (V, E)$,

$$\sum_{v \in V} \textit{deg}(v) = 2|E|.$$

Lecture 21: More on Graphs

(Tuesday, April 14, 1998)
Read: Sections 5.4, 5.5.

Graphs: Last time we introduced the notion of a graph (undirected) and a digraph (directed). We defined vertices, edges, and the notion of degrees of vertices. Today we continue this discussion. Recall that graphs and digraphs both consist of two objects, a set of vertices and a set of edges. For graphs edges are undirected and for graphs they are directed.

Paths and Cycles: Let's concentrate on directed graphs for the moment. A *path* in a directed graph is a sequence of vertices $\langle v_0, v_1, \ldots, v_k \rangle$ such that (v_{i-1}, v_i) is an edge for $i = 1, 2, \ldots, k$. The *length* of the path is the number of edges, k. We say that w is *reachable* from u if there is a path from u to w. Note that every vertex is reachable from itself by a path that uses zero edges. A path is *simple* if all vertices (except possibly the first and last) are distinct.

A *cycle* in a digraph is a path containing at least one edge and for which $v_0 = v_k$. A cycle is *simple* if, in addition, $v_1, \ldots, v_k$ are distinct. (Note: A self-loop counts as a simple cycle of length 1).

In undirected graphs we define path and cycle exactly the same, but for a *simple cycle* we add the requirement that the cycle visit at least three distinct vertices. This is to rule out the degenerate cycle $\langle u, w, u \rangle$, which simply jumps back and forth along a single edge.

There are two interesting classes cycles. A *Hamiltonian cycle* is a cycle that visits every vertex in a graph exactly once. A *Eulerian cycle* is a cycle (not necessarily simple) that visits every edge of a graph exactly once. (By the way, this is pronounced "Oiler-ian" and not "Yooler-ian".) There are also "path" versions in which you need not return to the starting vertex.

One of the early problems which motivated interest in graph theory was the *Königsberg Bridge Problem*. This city sits on the Pregel River as is joined by 7 bridges. The question is whether it is possible to cross all 7 bridges without visiting any bridge twice. Leonard Euler showed that it is not possible, by showing that this question could be posed as a problem of whether the multi-graph shown below has an Eulerian path, and then proving necessary and sufficient conditions for a graph to have such a path.

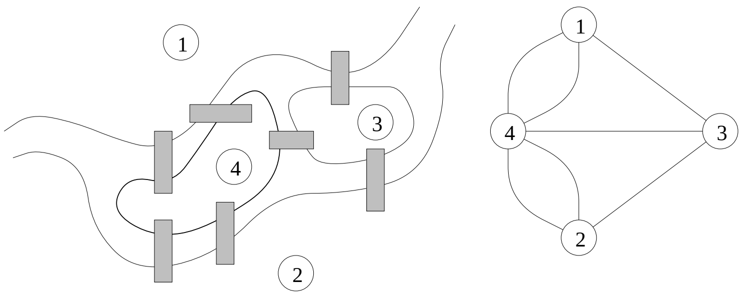


Figure 19: Bridge's at Königsberg Problem.

Euler proved that for a graph to have an Eulerian path, all but at most two of the vertices must have even degree. In this graph, all 4 of the vertices have odd degree.

Connectivity and acyclic graphs: A graph is said to be *acyclic* if it contains no simple cycles. A graph is *connected* if every vertex can reach every other vertex. An acyclic connected graph is called a *free tree* or simply *tree* for short. The term "free" is intended to emphasize the fact that the tree has no root, in contrast to a *rooted tree*, as is usually seen in data structures.

Observe that a free tree is a minimally connected graph in the sense that the removal of any causes the resulting graph to be disconnected. Furthermore, there is a unique path between any two vertices in a free tree. The addition of any edge to a free tree results in the creation of a single cycle.

The "reachability" relation is an equivalence relation on vertices, that is, it is reflexive (a vertex is reachable from itself), symmetric (if u is reachable from v, then v is reachable from u), and transitive (if u is reachable from v and v is reachable from w, then u is reachable from w). This implies that the reachability relation partitions the vertices of the graph in equivalence classes. These are called *connected components*.

A connected graph has a single connected component. An acyclic graph (which is not necessarily connected) consists of many free trees, and is called (what else?) a *forest*.

A digraph is *strongly connected* if for any two vertices u and v, u can reach v and v can reach u. (There is another type of connectivity in digraphs called *weak connectivity*, but we will not consider it.) As with connected components in graphs, strongly connectivity defines an equivalence partition on the vertices. These are called the *strongly connected components* of the digraph.

A directed graph that is acyclic is called a *DAG*, for *directed acyclic graph*. Note that it is different from a directed tree.

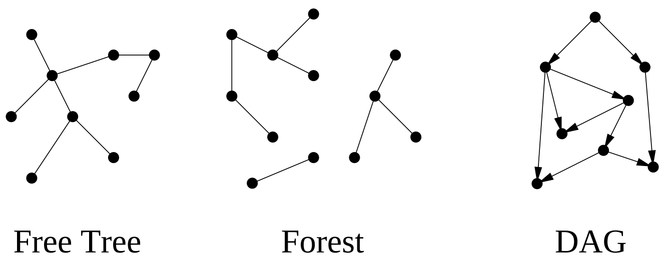


Figure 20: Free trees and forests.

Isomorphism: Two graphs $G = (V, E)$ and $G' = (V', E')$ are said to be *isomorphic* if there is a bijection (that is, a 1–1 and onto) function $f : V \to V'$, such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$. Isomorphic graphs are essentially "equal" except that their vertices have been given different names.

Determining whether graphs are isomorphic is not as easy as it might seem at first. For example, consider the graphs in the figure. Clearly (a) and (b) seem to appear more similar to each other than to (c), but in fact looks are deceiving. Observe that in all three cases all the vertices have degree 3, so that is not much of a help. Observe there are simple cycles of length 4 in (a), but the smallest simple cycles in (b) and (c) are of length 5. This implies that (a) cannot be isomorphic to either (b) or (c). It turns out that (b) and (c) are isomorphic. One possible isomorphism mapping is given below. The notation $(u \to v)$ means that vertex u from graph (b) is mapped to vertex v in graph (c). Check that each edge from (b) is mapped to an edge of (c).

$$\{(1 \to 1), (2 \to 2), (3 \to 3), (4 \to 7), (5 \to 8), (6 \to 5), (7 \to 10), (8 \to 4), (9 \to 6), (10 \to 9)\}.$$

Subgraphs and special graphs: A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Given a subset $V' \subseteq V$, the subgraph *induced* by V' is the graph $G' = (V', E')$ where

$$E' = \{(u, v) \in E \mid u, v \in V'\}.$$

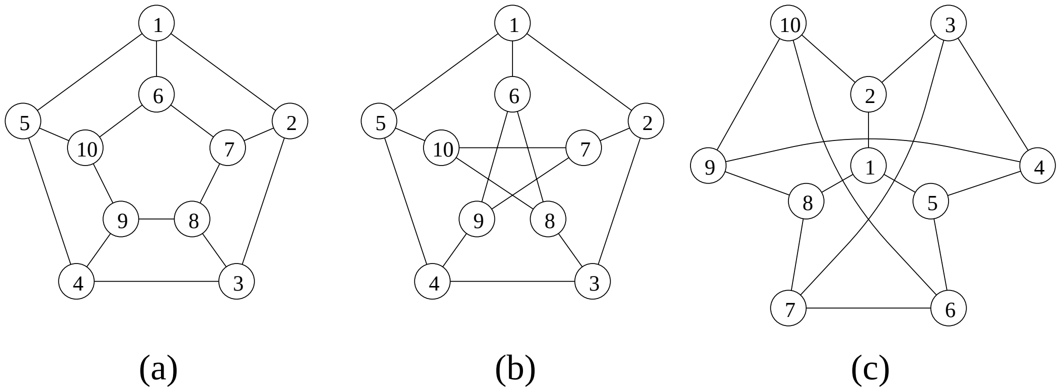


Figure 21: Graph isomorphism.

In other words, take all the edges of G that join pairs of vertices in V'.

An undirected graph that has the maximum possible number of edges is called a *complete graph*. Complete graphs are often denoted with the letter K. For example, K_5 is the complete graph on 5 vertices. Given a graph G, a subset of vertices $V' \subseteq V$ is said to form a *clique* if the subgraph induced by V' is complete. In other words, all the vertices of V' are adjacent to one another. A subset of vertices V' forms an *independent set* if the subgraph induced by V' has no edges. For example, in the figure below (a), the subset $\{1, 2, 4, 6\}$ is a clique, and $\{3, 4, 7, 8\}$ is an independent set.

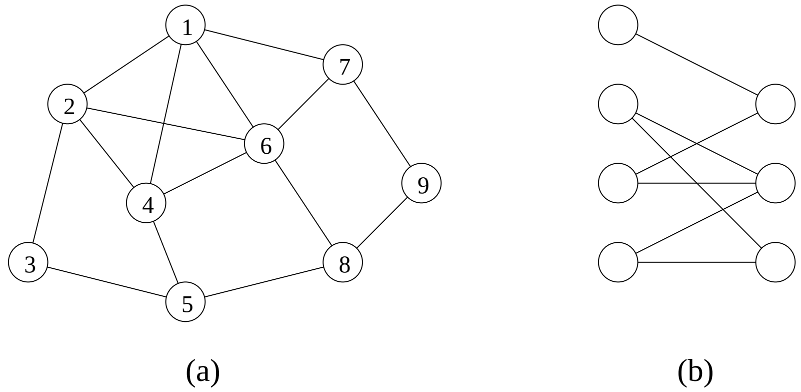


Figure 22: Cliques, Independent set, Bipartite graphs.

A *bipartite graph* is an undirected graph in which the vertices can be partitioned into two sets V_1 and V_2 such that all the edges go between a vertex in V_1 and V_2 (never within the same group). For example, the graph shown in the figure (b), is bipartite.

The *complement* of a graph $G = (V, E)$, often denoted $\bar{G}$ is a graph on the same vertex set, but in which the edge set has been complemented. The *reversal* of a directed graph, often denoted G^R is a graph on the same vertex set in which all the edge directions have been reversed. This may also be called the *transpose* and denoted G^T.

A graph is *planar* if it can be drawn on the plane such that no two edges cross over one another. Planar graphs are important special cases of graphs, since they arise in applications of geographic information systems (as subdivisions of region into smaller subregions), circuits (where wires cannot cross), solid modeling (for modeling complex surfaces as collections of small triangles). In general there may be many different ways to draw a planar graph in the plane. For example, the figure below shows two essentially different drawings of the same graph. Such a drawing is called a *planar embedding.* The *neighbors* of a vertex are the vertices that it is adjacent to. An embedding is determined by the counterclockwise cyclic ordering of the neighbors about all the vertices. For example, in the embedding on the left, the neighbors of vertex 1 in counterclockwise order are $\langle 2,3,4,5\rangle$, but on the right the order is $\langle 2,5,4,3\rangle$. Thus the two embeddings are different.

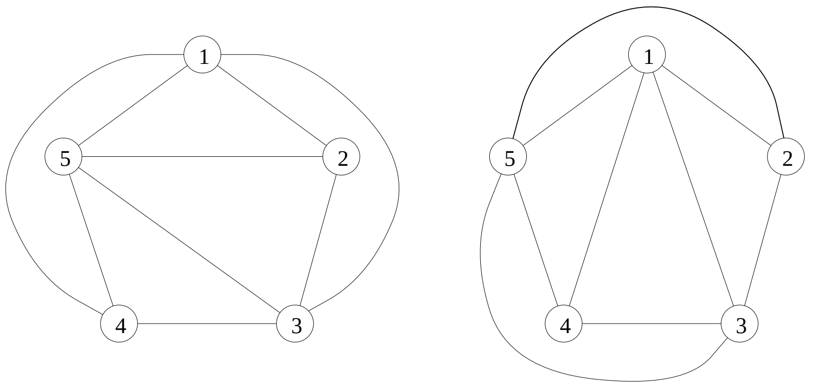


Figure 23: Planar Embeddings.

An important fact about planar embeddings of graphs is that they naturally subdivide the plane into regions, called *faces*. For example, in the figure on the left, the triangular region bounded by vertices $\langle 1,2,5\rangle$ is a face. There is always one face, called the *unbounded face* that surrounds the whole graph. This embedding has 6 faces (including the unbounded face). Notice that the other embedding also has 6 faces. Is it possible that two different embeddings have different numbers of faces? The answer is no. The reason stems from an important observation called *Euler's formula*, which relates the numbers of vertices, edges, and faces in a planar graph.

Euler's Formula: A connected planar embedding of a graph with V vertices, E edges, and F faces, satisfies:

$$V - E + F = 2.$$

In the examples above, both graphs have 5 vertices, and 9 edges, and so by Euler's formula they have $F = 2 - V + E = 2 - 5 + 9 = 6$ faces.

Size Issues: When referring to graphs and digraphs we will always let $n = |V|$ and $e = |E|$. (Our textbook usually uses the abuse of notation $V = |V|$ and $E = |E|$. Beware, the sometimes V is a set, and sometimes it is a number. Some authors use $m = |E|$.)

Because the running time of an algorithm will depend on the size of the graph, it is important to know how n and e relate to one another.

Observation: For a digraph $e \le n^2 = O(n^2)$. For an undirected graph $e \le \binom{n}{2} = n(n-1)/2 = O(n^2)$.

A graph or digraph is allowed to have no edges at all. One interesting question is what the minimum number of edges that a connected graph must have.

We say that a graph is *sparse* if e is much less than n^2.

For example, the important class of *planar graphs* (graphs which can be drawn on the plane so that no two edges cross over one another) $e = O(n)$. In most application areas, very large graphs tend to be sparse. This is important to keep in mind when designing graph algorithms, because when n is really large and $O(n^2)$ running time is often unacceptably large for real-time response.

Lecture 22: Graphs Representations and BFS

(Thursday, April 16, 1998)
Read: Sections 23.1 through 23.3 in CLR.

Representations of Graphs and Digraphs: We will describe two ways of representing graphs and digraphs. First we show how to represent digraphs. Let $G = (V, E)$ be a digraph with $n = |V|$ and let $e = |E|$. We will assume that the vertices of G are indexed $\{1, 2, \ldots, n\}$.

Adjacency Matrix: An $n \times n$ matrix defined for $1 \le v, w \le n$.

$$A[v,w] = \begin{cases} 1 & \text{if } (v,w) \in E \\ 0 & \text{otherwise.} \end{cases}$$

If the digraph has weights we can store the weights in the matrix. For example if $(v, w) \in E$ then $A[v, w] = W(v, w)$ (the weight on edge (v, w)). If $(v, w) \notin E$ then generally $W(v, w)$ need not be defined, but often we set it to some "special" value, e.g. $A(v, w) = -1$, or ∞. (By ∞ we mean (in practice) some number which is larger than any allowable weight. In practice, this might be some machine dependent constant like `MAXINT`.)

Adjacency List: An array $Adj[1 \ldots n]$ of pointers where for $1 \le v \le n$, $Adj[v]$ points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.

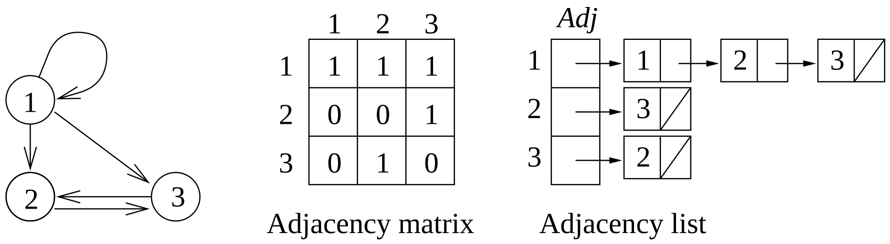


Figure 24: Adjacency matrix and adjacency list for digraphs.

We can represent undirected graphs using exactly the same representation, but we will store each edge twice. In particular, we representing the undirected edge $\{v, w\}$ by the two oppositely directed edges (v, w) and (w, v). Notice that even though we represent undirected graphs in the same way that we represent digraphs, it is important to remember that these two classes of objects are mathematically distinct from one another.

This can cause some complications. For example, suppose you write an algorithm that operates by marking edges of a graph. You need to be careful when you mark edge (v, w) in the representation that you also mark (w, v), since they are both the same edge in reality. When dealing with adjacency lists, it may not be convenient to walk down the entire linked list, so it is common to include *cross links* between corresponding edges.

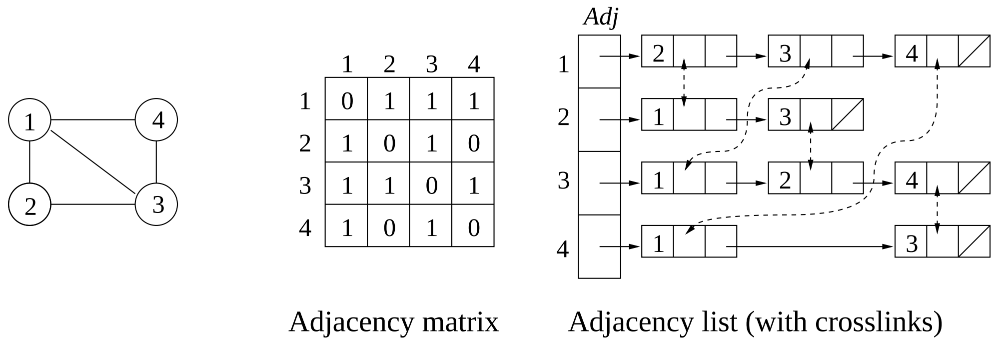


Figure 25: Adjacency matrix and adjacency list for graphs.

An adjacency matrix requires $\Theta(n^2)$ storage and an adjacency list requires $\Theta(n + e)$ storage (one entry for each vertex in *Adj* and each list has *outdeg*(v) entries, which when summed is $\Theta(e)$. For sparse graphs the adjacency list representation is more cost effective.

Shortest Paths: To motivate our first algorithm on graphs, consider the following problem. You are given an undirected graph $G = (V, E)$ (by the way, everything we will be saying can be extended to directed graphs, with only a few small changes) and a *source vertex* $s \in V$. The *length* of a path in a graph (without edge weights) is the number of edges on the path. We would like to find the shortest path from s to each other vertex in G. If there are ties (two shortest paths of the same length) then either path may be chosen arbitrarily.

The final result will be represented in the following way. For each vertex $v \in V$, we will store $d[v]$ which is the *distance* (length of the shortest path) from s to v. Note that $d[s] = 0$. We will also store a *predecessor (or parent) pointer* $\pi[v]$, which indicates the first vertex along the shortest path if we walk from v backwards to s. We will let $\pi[s] = \text{NIL}$.

It may not be obvious at first, but these single predecessor pointers are sufficient to reconstruct the shortest path to any vertex. Why? We make use of a simple fact which is an example of a more general principal of many optimization problems, called the *principal of optimality.* For a path to be a shortest path, every subpath of the path must be a shortest path. (If not, then the subpath could be replaced with a shorter subpath, implying that the original path was not shortest after all.)

Using this observation, we know that the last edge on the shortest path from s to v is the edge (u, v), then the first part of the path *must* consist of a shortest path from s to u. Thus by following the predecessor pointers we will construct the *reverse* of the shortest path from s to v.

Obviously, there is simple brute-force strategy for computing shortest paths. We could simply start enumerating all simple paths starting at s, and keep track of the shortest path arriving at each vertex. However, since there can be as many as $n!$ simple paths in a graph (consider a complete graph), then this strategy is clearly impractical.

Here is a simple strategy that is more efficient. Start with the source vertex s. Clearly, the distance to each of s's neighbors is exactly 1. Label all of them with this distance. Now consider the unvisited neighbors of these neighbors. They will be at distance 2 from s. Next consider the unvisited neighbors of the neighbors of the neighbors, and so on. Repeat this until there are no more unvisited neighbors left to visit. This algorithm can be *visualized* as simulating a wave propagating outwards from s, visiting the vertices in bands at ever increasing distances from s.

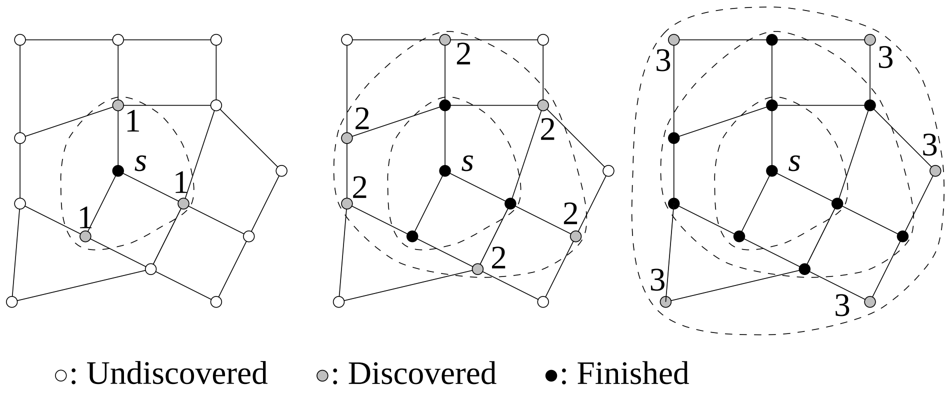


Figure 26: Breadth-first search for shortest paths.

Breadth-first search: Given an graph $G = (V, E)$, breadth-first search starts at some source vertex s and "discovers" which vertices are reachable from s. Define the *distance* between a vertex v and s to be the minimum number of edges on a path from s to v. Breadth-first search discovers vertices in increasing order of distance, and hence can be used as an algorithm for computing shortest paths. At any given time there is a "frontier" of vertices that have been discovered, but not yet processed. Breadth-first search is named because it visits vertices across the entire "breadth" of this frontier.

Initially all vertices (except the source) are colored white, meaning that they have not been seen. When a vertex has first been discovered, it is colored gray (and is part of the frontier). When a gray vertex is processed, then it becomes black.

Breadth-First Search

```
BFS(graph G=(V,E), vertex s) {
    int d[1..size(V)]                           // vertex distances
    int color[1..size(V)]                       // vertex colors
    vertex pred[1..size(V)]                     // predecessor pointers
    queue Q = empty                             // FIFO queue

    for each u in V {                           // initialization
        color[u] = white
        d[u]     = INFINITY
        pred[u]  = NULL
    }
    color[s] = gray                             // initialize source s
    d[s] = 0
    enqueue(Q, s)                               // put source in the queue
    while (Q is nonempty) {
        u = dequeue(Q)                          // u is the next vertex to visit
        for each v in Adj[u] {
            if (color[v] == white) {            // if neighbor v undiscovered
                color[v] = gray                 // ...mark it discovered
                d[v]     = d[u]+1               // ...set its distance
                pred[v]  = u                    // ...and its predecessor
                enqueue(Q, v)                   // ...put it in the queue
            }
        }
        color[u] = black                        // we are done with u
```

```
        }
    }
```

The search makes use of a *queue*, a first-in first-out list, where elements are removed in the same order they are inserted. The first item in the queue (the next to be removed) is called the *head* of the queue. We will also maintain arrays $color[u]$ which holds the color of vertex u (either white, gray or black), $pred[u]$ which points to the predecessor of u (i.e. the vertex who first discovered u, and $d[u]$, the distance from s to u. Only the color is really needed for the search, but the others are useful depending on the application.

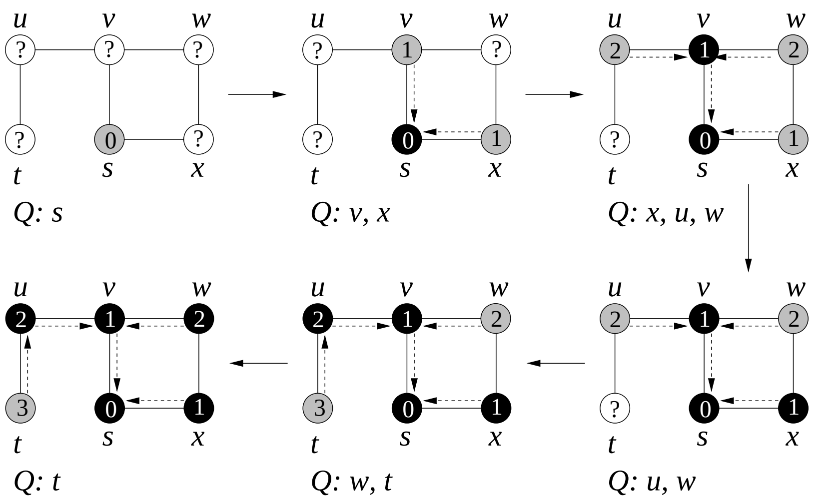


Figure 27: Breadth-first search: Example.

Observe that the predecessor pointers of the BFS search define an inverted tree. If we reverse these edges we get a rooted unordered tree called a *BFS tree* for G. (Note that there are many potential BFS trees for a given graph, depending on where the search starts, and in what order vertices are placed on the queue.) These edges of G are called *tree edges* and the remaining edges of G are called *cross edges*. It is not hard to prove that if G is an undirected graph, then cross edges always go between two nodes that are at most one level apart in the BFS tree. The reason is that if any cross edge spanned two or more levels, then when the vertex at the higher level (closer to the root) was being processed, it would have discovered the other vertex, implying that the other vertex would appear on the very next level of the tree, a contradiction. (In a directed graph cross edges will generally go down at most 1 level, but they may come up an arbitrary number of levels.)

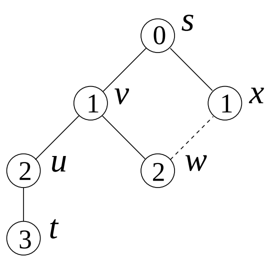


Figure 28: BFS tree.

Analysis: The running time analysis of BFS is similar to the running time analysis of many graph traversal algorithms. Let $n = |V|$ and $e = |E|$. Observe that the initialization portion requires $\Theta(n)$ time. The real meat is in the traversal loop. Since we never visit a vertex twice, the number of times we go through the while loop is at most n (exactly n assuming each vertex is reachable from the source). The number of iterations through the inner for loop is proportional to $deg(u) + 1$. (The $+1$ is because even if $deg(u) = 0$, we need to spend a constant amount of time to set up the loop.) Summing up over all vertices we have the running time

$$T(n) = n + \sum_{u \in V} (deg(u) + 1) = n + \sum_{u \in V} deg(u) + n = 2n + 2e \in \Theta(n + e).$$

For an directed graph the analysis is essentially the same.

Lecture 23: All-Pairs Shortest Paths

(Tuesday, April 21, 1998)
Read: Chapt 26 (up to Section 26.2) in CLR.

All-Pairs Shortest Paths: Last time we showed how to compute shortest paths starting at a designated source vertex, and assuming that there are no weights on the edges. Today we talk about a considerable generalization of this problem. First, we compute shortest paths not from a single vertex, but from every vertex in the graph. Second, we allow edges in the graph to have numeric *weights*.

Let $G = (V, E)$ be a directed graph with edge weights. If (u, v) E, is an edge of G, then the weight of this edge is denoted $W(u, v)$. Intuitively, this weight denotes the distance of the road from u to v, or more generally the cost of traveling from u to v. For now, let us think of the weights as being positive values, but we will see that the algorithm that we are about to present can handle negative weights as well, in special cases. Intuitively a negative weight means that you get paid for traveling from u to v. Given a path $\pi = \langle u_0, u_1, \ldots, u_k \rangle$, the *cost* of this path is the sum of the edge weights:

$$cost(\pi) = W(u_0, u_1) + W(u_1, u_2) + \cdots W(u_{k-1}, u_k) = \sum_{i=1}^{k} W(u_{i-1}, u_i).$$

(We will avoid using the term *length*, since it can be confused with the number of edges on the path.) The *distance* between two vertices is the cost of the minimum cost path between them.

We consider the problem of determining the cost of the shortest path between all pairs of vertices in a weighted directed graph. We will present two algorithms for this problem. The first is a rather naive $\Theta(n^4)$ algorithm, and the second is a $\Theta(n^3)$ algorithm. The latter is called the *Floyd-Warshall algorithm*. Both algorithms is based on a completely different algorithm design technique, called *dynamic programming*.

For these algorithms, we will assume that the digraph is represented as an adjacency matrix, rather than the more common adjacency list. Recall that adjacency lists are generally more efficient for sparse graphs (and large graphs tend to be sparse). However, storing all the distance information between each pair of vertices, will quickly yield a dense digraph (since typically almost every vertex can reach almost every other vertex). Therefore, since the output will be dense, there is no real harm in using the adjacency matrix.

Because both algorithms are matrix-based, we will employ common matrix notation, using i, j and k to denote vertices rather than u, v, and w as we usually do. Let $G = (V, E, w)$ denote the input digraph and its edge weight function. The edge weights may be positive, zero, or negative, but we assume that there are no cycles whose total weight is negative. It is easy to see why this causes problems. If the shortest path ever enters such a cycle, it would never exit. Why? Because by going round the cycle over and over, the cost will become smaller and smaller. Thus, the shortest path would have a weight of $-\infty$, and would consist of an infinite number of edges. Disallowing negative weight cycles will rule out the possibility this absurd situation.

Input Format: The input is an $n \times n$ matrix W of edge weights, which are based on the edge weights in the digraph. We let w_{ij} denote the entry in row i and column j of W.

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ W(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ +\infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

Setting $w_{ij} = \infty$ if there is no edge, intuitively means that there is no direct link between these two nodes, and hence the direct cost is infinite. The reason for setting $w_{ii} = 0$ is that intuitively the cost of getting from any vertex to should be 0, since we have no distance to travel. Note that in digraphs it is possible to have self-loop edges, and so $W(i,j)$ may generally be nonzero. Notice that it cannot be negative (otherwise we would have a negative cost cycle consisting of this single edge). If it is positive, then it never does us any good to follow this edge (since it increases our cost and doesn't take us anywhere new).

The output will be an $n \times n$ distance matrix $D = d_{ij}$ where $d_{ij} = \delta(i,j)$, the shortest path cost from vertex i to j. Recovering the shortest paths will also be an issue. To help us do this, we will also compute an auxiliary matrix $pred[i,j]$. The value of $pred[i,j]$ will be a vertex that is somewhere along the shortest path from i to j. If the shortest path travels directly from i to j without passing through any other vertices, then $pred[i,j]$ will be set to *null*. We will see later than using these values it will be possible to reconstruct any shortest path in $\Theta(n)$ time.

Dynamic Programming for Shortest Paths: The algorithm is based on a technique called *dynamic programming*. Dynamic programming problems are typically optimization problems (find the smallest or largest solution, subject to various constraints). The technique is related to divide-and-conquer, in the sense that it breaks problems down into smaller problems that it solves recursively. However, because of the somewhat different nature of dynamic programming problems, standard divide-and-conquer solutions are not usually efficient. The basic elements that characterize a dynamic programming algorithm are:

Substructure: Decompose the problem into smaller subproblems.

Optimal substructure: Each of the subproblems should be solved optimally.

Bottom-up computation: Combine solutions on small subproblems to solve larger subproblems, and eventually to arrive at a solution to the complete problem.

The question is how to decompose the shortest path problem into subproblems in a meaningful way. There is one very natural way to do this. What is remarkable, is that this does *not* lead to the best solution. First we will introduce the natural decomposition, and later present the Floyd-Warshall algorithm makes use of a different, but more efficient dynamic programming formulation.

Path Length Formulation: We will concentrate just on computing the *cost* of the shortest path, not the path itself. Let us first sketch the natural way to break the problem into subproblems. We want to find some parameter, which constrains the estimates to the shortest path costs. At first the estimates will be crude. As this parameter grows, the shortest paths cost estimates should converge to their correct values. A natural way to do this is to restrict the number of edges that are allowed to be in the shortest path.

For $0 \leq m \leq n-1$, define $d_{ij}^{(m)}$ to be the cost of the shortest path from vertex i to vertex j that contains at most m edges. Let $D^{(m)}$ denote the matrix whose entries are these values. The idea is to compute $D^{(0)}$ then $D^{(1)}$, and so on, up to $D^{(n-1)}$. Since we know that no shortest path can use more than $n-1$ edges (for otherwise it would have to repeat a vertex), we know that $D^{(n-1)}$ is the final distance matrix. This is illustrated in the figure (a) below.

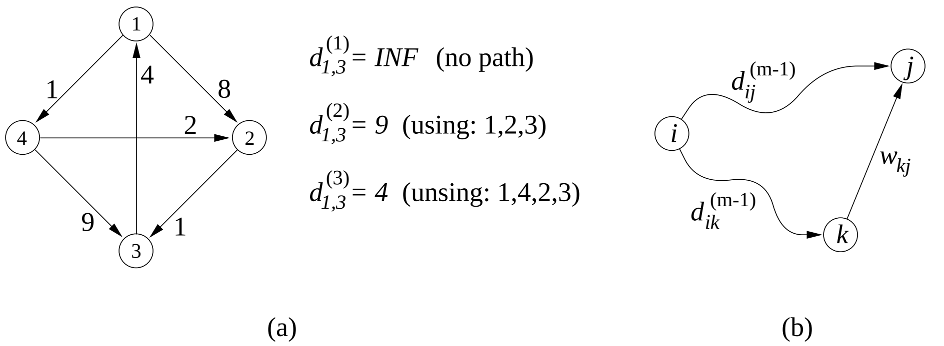


Figure 29: Dynamic Programming Formulation.

The question is, how do we compute these distance matrices? As a basis, we could start with paths of containing 0 edges, $D^{(0)}$ (as our text does). However, observe that $D^{(1)} = W$, since the edges of the digraph are just paths of length 1. It is just as easy to start with $D^{(1)}$, since we are given W as input. So as our basis case we have

$$d_{ij}^{(1)} = w_{ij}.$$

Now, to make the induction go, we claim that it is possible to compute $D^{(m)}$ from $D^{(m-1)}$, for $m \geq 2$. Consider how to compute the quantity $d_{ij}^{(m)}$. This is the length of the shortest path from i to j using at most m edges. There are two cases:

Case 1: If the shortest path uses strictly fewer than m edges, then its cost is just $d_{ij}^{(m-1)}$.

Case 2: If the shortest path uses exactly m edges, then the path uses $m-1$ edges to go from i to some vertex k, and then follows a single edge (k, j) of weight w_{kj} to get to j. The path from i to k should be shortest (by the principle of optimality) so the length of the resulting path is $d_{ik}^{(m-1)} + w_{ij}$. But we do not know what k is. So we minimize over all possible choices.

This is illustrated in the figure (b) above.

This suggests the following rule:

$$d_{ij}^{(m)} = \min \left\{ \begin{array}{l} d_{ij}^{(m-1)} \\ \min_{1 \leq k \leq n} \left(d_{ik}^{(m-1)} + w_{kj} \right) \end{array} \right\}.$$

Notice that the two terms of the main min correspond to the two cases. In the second case, we consider all vertices k, and consider the length of the shortest path from i to k, using $m-1$ edges, and then the single edge length cost from k to j.

We can simplify this formula a bit by observing that since $w_{jj} = 0$, we have $d_{ij}^{(m-1)} = d_{ij}^{(m-1)} + w_{jj}$. This term occurs in the second case (when $k = j$). Thus, the first term is redundant. This gives

$$d_{ij}^{(m)} = \min_{1 \leq k \leq n} \left(d_{ik}^{(m-1)} + w_{kj} \right),$$

The next question is how shall we implement this rule. One way would be to write a recursive procedure to do it. Here is a possible implementation. To compute the shortest path from i to j, the initial call would be $Dist(n-1, i, j)$. The array of weights

Recursive Shortest Paths

```
Dist(int m, int i, int j) {
    if (m == 1) return W[i,j]                         // single edge case
    best = INF
    for k = 1 to n do
        best = min(best, Dist(m-1, i, k) + w[k,j])  // apply the update rule
    return best
}
```

Unfortunately this will be *very slow*. Let $T(m,n)$ be the running time of this algorithm on a graph with n vertices, where the first argument is m. The algorithm makes n calls to itself with the first argument of $m-1$. When $m = 1$, the recursion bottoms out, and we have $T(1,n) = 1$. Otherwise, we make n recursive calls to $T(m-1,n)$. This gives the recurrence:

$$T(m,n) = \begin{cases} 1 & \text{if } m = 1, \\ nT(m-1,n) + 1 & \text{otherwise.} \end{cases}$$

The total running time is $T(n-1,n)$. It is a straightforward to solve this by expansion. The result will be $O(n^n)$, a huge value. It is not hard to see why. If you unravel the recursion, you will see that this algorithm is just blindly trying all possible paths from i to j. There are exponentially many such paths.

So how do we make this faster? The answer is to use *table-lookup*. This is the key to dynamic programming. Observe that there are only $O(n^3)$ different possible numbers $d_{ij}^{(m)}$ that we have to compute. Once we compute one of these values, we will store it in a table. Then if we want this value again, rather than recompute it, we will simply look its value up in the table.

The figure below gives an implementation of this idea. The main procedure $ShortestPath(n, w)$ is given the number of vertices n and the matrix of edge weights W. The matrix $D^{(m)}$ is stored as $D[m]$, for $1 \leq m \leq n-1$. For each m, $D[m]$ is a 2-dimensional matrix, implying that D is a 3-dimensional matrix. We initialize $D^{(1)}$ by copying W. Then each call to $ExtendPaths()$ computes $D^{(m)}$ from $D^{(m-1)}$, from the above formula.

Dynamic Program Shortest Paths

```
ShortestPath(int n, int W[1..n, 1..n]) {
    array D[1..n-1][1..n, 1..n]
    copy W to D[1]                                 // initialize D[1]
    for m = 2 to n-1 do
        D[m] = ExtendPaths(n, D[m-1], W)           // comput D[m] from D[m-1]
    return D[n-1]
}

ExtendShortestPath(int n, int d[1..n, 1..n], int W[1..n, 1..n]) {
    matrix dd[1..n, 1..n] = d[1..n, 1..n]          // copy d to temp matrix
```

```
        for i = 1 to n do                            // start from i
            for j = 1 to n do                        // ...to j
                for k = 1 to n do                    // ...passing through k
                    dd[i,j] = min(dd[i,j], d[i,k] + W[k,j])
        return dd                                    // return matrix of distances
}
```

The procedure *ExtendShortestPath*() consists of 3 nested loops, and so its running time is $\Theta(n^3)$. It is called $n-2$ times by the main procedure, and so the total running time is $\Theta(n^4)$. Next time we will see that we can improve on this. This is illustrated in the figure below.

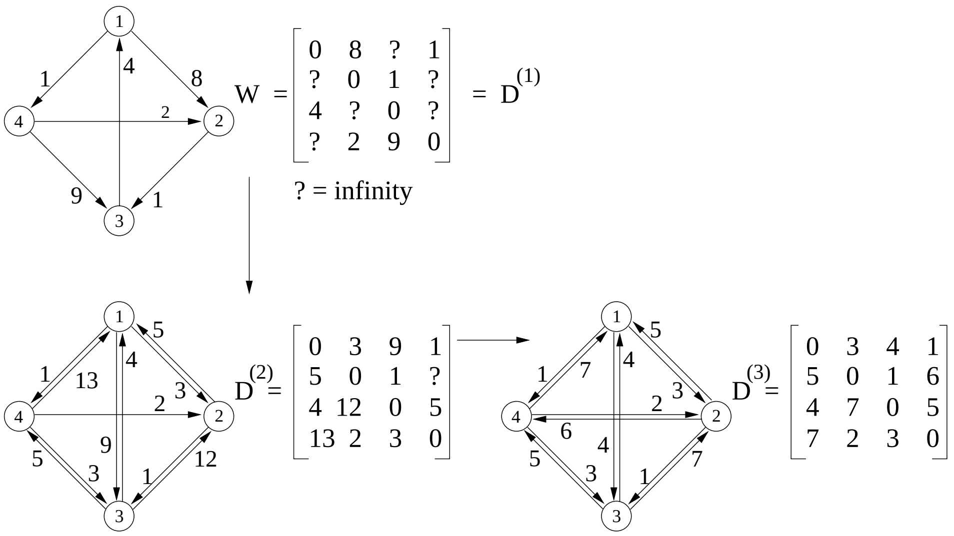


Figure 30: Shortest Path Example.

Lecture 24: Floyd-Warshall Algorithm

(Thursday, April 23, 1998)

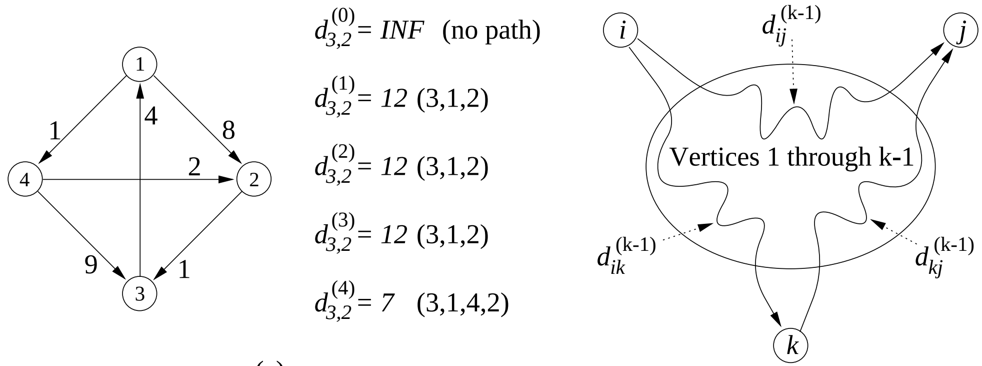
Read: Chapt 26 (up to Section 26.2) in CLR.

Floyd-Warshall Algorithm: We continue discussion of computing shortest paths between all pairs of vertices in a directed graph. The Floyd-Warshall algorithm dates back to the early 60's. Warshall was interested in the weaker question of reachability: determine for each pair of vertices u and v, whether u can reach v. Floyd realized that the same technique could be used to compute shortest paths with only minor variations.

The Floyd-Warshall algorithm improves upon this algorithm, running in $\Theta(n^3)$ time. The genius of the Floyd-Warshall algorithm is in finding a different formulation for the shortest path subproblem than the path length formulation introduced earlier. At first the formulation may seem most unnatural, but it leads to a faster algorithm. As before, we will compute a set of matrices whose entries are $d_{ij}^{(k)}$. We will change the *meaning* of each of these entries.

For a path $p = \langle v_1, v_2, \ldots, v_\ell \rangle$ we say that the vertices $v_2, v_3, \ldots, v_{\ell-1}$ are the *intermediate vertices* of this path. Note that a path consisting of a single edge has no intermediate vertices. We define $d_{ij}^{(k)}$ to be the shortest path from i to j such that any intermediate vertices on the path are chosen from the set $\{1, 2, \ldots, k\}$. In other words, we consider a path from i to j

which either consists of the single edge (i, j), or it visits some intermediate vertices along the way, but these intermediate can only be chosen from $\{1, 2, \ldots, k\}$. The path is free to visit any subset of these vertices, and to do so in any order. Thus, the difference between Floyd's formulation and the previous formulation is that here the superscript (k) restricts the set of vertices that the path is allowed to pass through, and there the superscript (m) restricts the number of edges the path is allowed to use. For example, in the digraph shown in the following figure, notice how the value of $d_{32}^{(k)}$ changes as k varies.



(a)

Figure 31: Floyd-Warshall Formulation.

Floyd-Warshall Update Rule: How do we compute $d_{ij}^{(k)}$ assuming that we have already computed the previous matrix $d^{(k-1)}$? As before, there are two basic cases, depending on the ways that we might get from vertex i to vertex j, assuming that the intermediate vertices are chosen from $\{1, 2, \ldots, k\}$:

We don't go through k at all: Then the shortest path from i to j uses only intermediate vertices $\{1, \ldots, k-1\}$ and hence the length of the shortest path is $d_{ij}^{(k-1)}$.

We do go through k: First observe that a shortest path does not pass through the same vertex twice, so we can assume that we pass through k exactly once. (The assumption that there are no negative cost cycles is being used here.) That is, we go from i to k, and then from k to j. In order for the overall path to be as short as possible we should take the shortest path from i to k, and the shortest path from k to j. (This is the principle of optimality.) Each of these paths uses intermediate vertices only in $\{1, 2, \ldots, k-1\}$. The length of the path is $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

This suggests the following recursive rule for computing $d^{(k)}$:

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

The final answer is $d_{ij}^{(n)}$ because this allows all possible vertices as intermediate vertices. Again, we could write a recursive program to compute $d_{ij}^{(k)}$, but this will be prohibitively slow. Instead, we compute it by storing the values in a table, and looking the values up as we need them. Here is the complete algorithm. We have also included predecessor pointers, $pred[i, j]$ for extracting the final shortest paths. We will discuss them later.

```
Floyd_Warshall(int n, int W[1..n, 1..n]) {
    array d[1..n, 1..n]
    for i = 1 to n do {                             // initialize
        for j = 1 to n do {
            d[i,j] = W[i,j]
            pred[i,j] = null
        }
    }
    for k = 1 to n do                               // use intermediates {1..k}
        for i = 1 to n do                           // ...from i
            for j = 1 to n do                       // ...to j
                if (d[i,k] + d[k,j]) < d[i,j]) {
                    d[i,j] = d[i,k] + d[k,j]        // new shorter path length
                    pred[i,j] = k                   // new path is through k
                }
    return d                                        // matrix of final distances
}
```

Clearly the algorithm's running time is $\Theta(n^3)$. The space used by the algorithm is $\Theta(n^2)$. Observe that we deleted all references to the superscript (k) in the code. It is left as an exercise that this does not affect the correctness of the algorithm. An example is shown in the following figure.

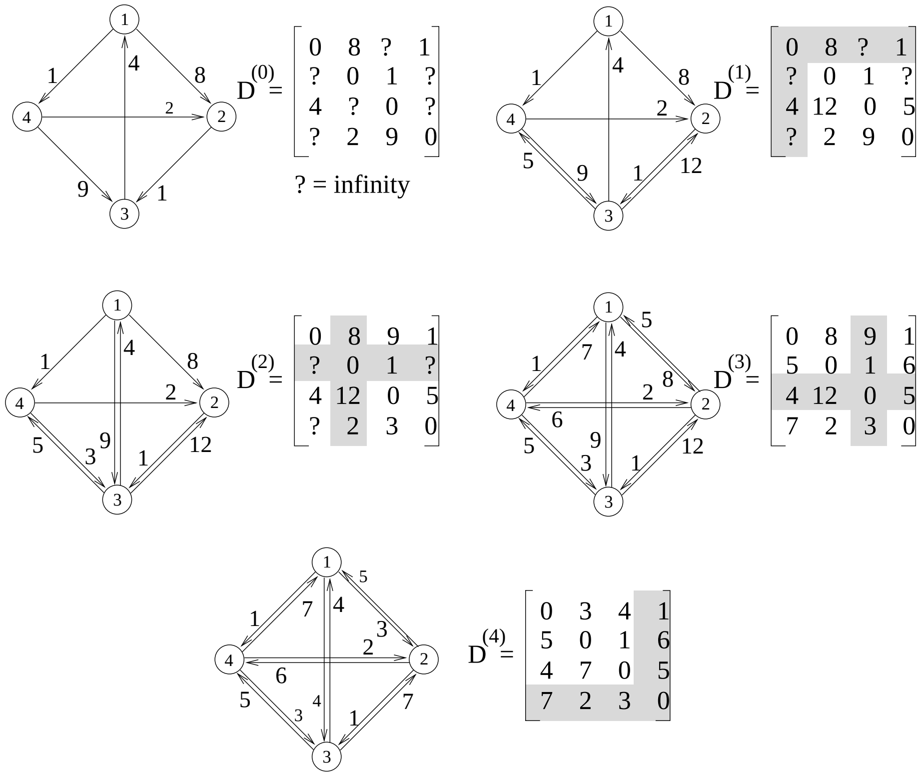


Figure 32: Floyd-Warshall Example.

Extracting Shortest Paths: The predecessor pointers $pred[i, j]$ can be used to extract the final path. Here is the idea, whenever we discover that the shortest path from i to j passes through an intermediate vertex k, we set $pred[i, j] = k$. If the shortest path does not pass through any intermediate vertex, then $pred[i, j] = null$. To find the shortest path from i to j, we consult $pred[i, j]$. If it is *null*, then the shortest path is just the edge (i, j). Otherwise, we recursively compute the shortest path from i to $pred[i, j]$ and the shortest path from $pred[i, j]$ to j.

Printing the Shortest Path

```
Path(i,j) {
    if pred[i,j] = null                     // path is a single edge
        output(i,j)
    else {                                  // path goes through pred
        Path(i, pred[i,j]);                 // print path from i to pred
        Path(pred[i,j], j);                 // print path from pred to j
    }
}
```

Lecture 25: Longest Common Subsequence

(April 28, 1998)
Read: Section 16.3 in CLR.

Strings: One important area of algorithm design is the study of algorithms for character strings. There are a number of important problems here. Among the most important has to do with efficiently searching for a substring or generally a pattern in large piece of text. (This is what text editors and functions like "grep" do when you perform a search.) In many instances you do not want to find a piece of text exactly, but rather something that is "similar". This arises for example in genetics research. Genetic codes are stored as long DNA molecules. The DNA strands can be broken down into a long sequences each of which is one of four basic types: C, G, T, A.

But exact matches rarely occur in biology because of small changes in DNA replication. Exact substring search will only find exact matches. For this reason, it is of interest to compute similarities between strings that do not match exactly. The method of string similarities should be insensitive to random insertions and deletions of characters from some originating string. There are a number of measures of similarity in strings. The first is the *edit distance*, that is, the minimum number of single character insertions, deletions, or transpositions necessary to convert one string into another. The other, which we will study today, is that of determining the length of the longest common subsequence.

Longest Common Subsequence: Let us think of character strings as sequences of characters. Given two sequences $X = \langle x_1, x_2, \ldots, x_m\rangle$ and $Z = \langle z_1, z_2, \ldots, z_k\rangle$, we say that Z is a *subsequence* of X if there is a strictly increasing sequence of k indices $\langle i_1, i_2, \ldots, i_k\rangle$ $(1 \le i_1 < i_2 < \ldots < i_k \le n)$ such that $Z = \langle X_{i_1}, X_{i_2}, \ldots, X_{i_k}\rangle$. For example, let $X = \langle ABRACADABRA\rangle$ and let $Z = \langle AADAA\rangle$, then Z is a subsequence of X.

Given two strings X and Y, the *longest common subsequence* of X and Y is a longest sequence Z which is both a subsequence of X and Y.

For example, let X be as before and let $Y = \langle YABBADABBADOO\rangle$. Then the longest common subsequence is $Z = \langle ABADABA\rangle$.

The Longest Common Subsequence Problem (LCS) is the following. Given two sequences $X = \langle x_1, \ldots, x_m\rangle$ and $Y = \langle y_1, \ldots, y_n\rangle$ determine a longest common subsequence. Note that it is not always unique. For example the LCS of $\langle ABC\rangle$ and $\langle BAC\rangle$ is either $\langle AC\rangle$ or $\langle BC\rangle$.

Dynamic Programming Solution: The simple brute-force solution to the problem would be to try all possible subsequences from one string, and search for matches in the other string, but this is hopelessly inefficient, since there are an exponential number of possible subsequences.

Instead, we will derive a dynamic programming solution. In typical DP fashion, we need to break the problem into smaller pieces. There are many ways to do this for strings, but it turns out for this problem that considering all pairs of *prefixes* will suffice for us. A *prefix* of a sequence is just an initial string of values, $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. X_0 is the empty sequence.

The idea will be to compute the longest common subsequence for every possible pair of prefixes. Let $c[i,j]$ denote the length of the longest common subsequence of X_i and Y_j. Eventually we are interested in $c[m,n]$ since this will be the LCS of the two entire strings. The idea is to compute $c[i,j]$ assuming that we already know the values of $c[i',j']$ for $i' \leq i$ and $j' \leq j$ (but not both equal). We begin with some observations.

Basis: $c[i,0] = c[j,0] = 0$. If either sequence is empty, then the longest common subsequence is empty.

Last characters match: Suppose $x_i = y_j$. Example: Let $X_i = \langle ABCA \rangle$ and let $Y_j = \langle DACA \rangle$. Since both end in A, we claim that the LCS must also end in A. (We will explain why later.) Since the A is part of the LCS we may find the overall LCS by removing A from both sequences and taking the LCS of $X_{i-1} = \langle ABC \rangle$ and $Y_{j-1} = \langle DAC \rangle$ which is $\langle AC \rangle$ and then adding A to the end, giving $\langle ACA \rangle$ as the answer. (At first you might object: But how did you know that these two A's matched with each other. The answer is that we don't, but it will not make the LCS any smaller if we do.)

Thus, if $x_i = y_j$ then $c[i,j] = c[i-1, j-1] + 1$.

Last characters do not match: Suppose that $x_i \neq y_j$. In this case x_i and y_j cannot both be in the LCS (since they would have to be the last character of the LCS). Thus either x_i is *not* part of the LCS, or y_j is *not* part of the LCS (and possibly *both* are not part of the LCS).

In the first case the LCS of X_i and Y_j is the LCS of X_{i-1} and Y_j, which is $c[i-1,j]$. In the second case the LCS is the LCS of X_i and Y_{j-1} which is $c[i,j-1]$. We do not know which is the case, so we try both and take the one that gives us the longer LCS.

Thus, if $x_i \neq y_j$ then $c[i,j] = \max(c[i-1,j], c[i,j-1])$.

We left undone the business of showing that if both strings end in the same character, then the LCS must also end in this same character. To see this, suppose by contradiction that both characters end in A, and further suppose that the LCS ended in a different character B. Because A is the last character of both strings, it follows that this particular instance of the character A cannot be used anywhere else in the LCS. Thus, we can add it to the end of the LCS, creating a longer common subsequence. But this would contradict the definition of the LCS as being longest.

Combining these observations we have the following rule:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Implementing the Rule: The task now is to simply implement this rule. As with other DP solutions, we concentrate on computing the maximum length. We will store some helpful pointers in a parallel array, $b[0..m, 0..n]$.

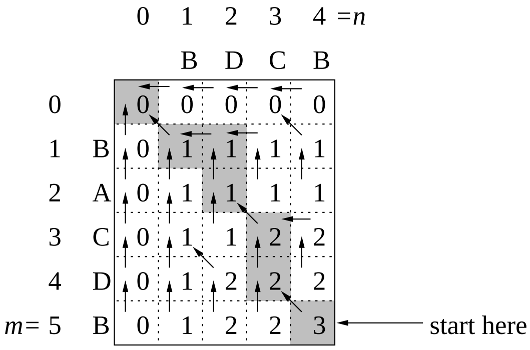
```
LCS(char x[1..m], char y[1..n]) {
    int c[0..m, 0..n]
    for i = 0 to m do {
        c[i,0] = 0    b[i,0] = SKIPX              // initialize column 0
    }
    for j = 0 to n do {
        c[0,j] = 0    b[0,j] = SKIPY              // initialize row 0
    }
    for i = 1 to m do {
        for j = 1 to n do {
            if (x[i] == y[j]) {
                c[i,j] = c[i-1,j-1]+1             // take X[i] and Y[j] for LCS
                b[i,j] = ADDXY
            }
            else if (c[i-1,j] >= c[i,j-1]) {      // X[i] not in LCS
                c[i,j] = c[i-1,j]
                b[i,j] = SKIPX
            }
            else {                                // Y[j] not in LCS
                c[i,j] = c[i,j-1]
                b[i,j] = SKIPY
            }
        }
    }
    return c[m,n];
}
```

		0	1	2	3	4 =*n*
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
m= 5	B	0	1	2	2	3

LCS Length Table

X = BACDB
Y = BDCB

LCS = BCB



with back pointers included

Figure 33: Longest common subsequence example.

The running time of the algorithm is clearly $O(mn)$ since there are two nested loops with m and n iterations, respectively. The algorithm also uses $O(mn)$ space.

Extracting the Actual Sequence: Extracting the final LCS is done by using the back pointers stored in $b[0..m, 0..n]$. Intuitively $b[i,j] = ADDXY$ means that $X[i]$ and $Y[j]$ together form the last character of the LCS. So we take this common character, and continue with entry $b[i-1, j-1]$ to the northwest ($\nwarrow$). If $b[i,j] = SKIPX$, then we know that $X[i]$ is not in the

LCS, and so we skip it and go to $b[i-1,j]$ above us ($\uparrow$). Similarly, if $b[i,j] = SKIPY$, then we know that $Y[j]$ is not in the LCS, and so we skip it and go to $b[i,j-1]$ to the left ($\leftarrow$). Following these back pointers, and outputting a character with each diagonal move gives the final subsequence.

Print Subsequence

```
getLCS(char x[1..m], char y[1..n], int b[0..m,0..n]) {
    LCS = empty string
    i = m
    j = n
    while(i != 0 && j != 0) {
        switch b[i,j] {
        case ADDXY:
            add x[i] (or equivalently y[j]) to front of LCS
            i--;   j--;   break
        case SKIPX:
            i--;  break
        case SKIPY:
            j--;  break
        }
    }
    return LCS
}
```

Lecture 26: Chain Matrix Multiplication

(Thursday, April 30, 1998)
Read: Section 16.1 of CLR.

Chain Matrix Multiplication: This problem involves the question of determining the optimal sequence for performing a series of operations. This general class of problem is important in compiler design for code optimization and in databases for query optimization. We will study the problem in a very restricted instance, where the dynamic programming issues are easiest to see.

Suppose that we wish to multiply a series of matrices

$$A_1 A_2 \dots A_n$$

Matrix multiplication is an associative but not a commutative operation. This means that we are free to parenthesize the above multiplication however we like, but we are not free to rearrange the order of the matrices. Also recall that when two (nonsquare) matrices are being multiplied, there are restrictions on the dimensions. A $p \times q$ matrix has p rows and q columns. You can multiply a $p \times q$ matrix A times a $q \times r$ matrix B, and the result will be a $p \times r$ matrix C. (The number of columns of A must equal the number of rows of B.) In particular for $1 \le i \le p$ and $1 \le j \le r$,

$$C[i,j] = \sum_{k=1}^{q} A[i,k]B[k,j].$$

Observe that there are pr total entries in C and each takes $O(q)$ time to compute, thus the total time (e.g. number of multiplications) to multiply these two matrices is $p \cdot q \cdot r$.

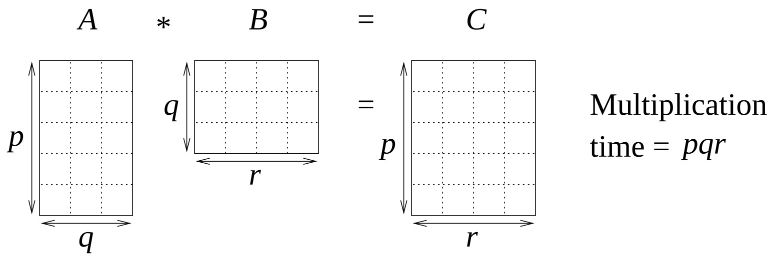


Figure 34: Matrix Multiplication.

Note that although any legal parenthesization will lead to a valid result, not all involve the same number of operations. Consider the case of 3 matrices: A_1 be 5×4, A_2 be 4×6 and A_3 be 6×2.

$$\begin{aligned} mult[((A_1 A_2) A_3)] &= (5 \cdot 4 \cdot 6) + (5 \cdot 6 \cdot 2) = 180, \\ mult[(A_1 (A_2 A_3))] &= (4 \cdot 6 \cdot 2) + (5 \cdot 4 \cdot 2) = 88. \end{aligned}$$

Even for this small example, considerable savings can be achieved by reordering the evaluation sequence. The Chain Matrix Multiplication problem is: Given a sequence of matrices $A_1, A_2, \ldots, A_n$ and dimensions $p_0, p_1, \ldots, p_n$ where A_i is of dimension $p_{i-1} \times p_i$, determine the multiplication sequence that minimizes the number of operations.

Important Note: This algorithm does not perform the multiplications, it just figures out the best order in which to perform the multiplications.

Naive Algorithm: We could write a procedure which tries all possible parenthesizations. Unfortunately, the number of ways of parenthesizing an expression is very large. If you have just one item, then there is only one way to parenthesize. If you have n items, then there are $n-1$ places where you could break the list with the outermost pair of parentheses, namely just after the 1st item, just after the 2nd item, etc., and just after the $(n-1)$st item. When we split just after the kth item, we create two sublists to be parenthesized, one with k items, and the other with $n-k$ items. Then we could consider all the ways of parenthesizing these. Since these are independent choices, if there are L ways to parenthesize the left sublist and R ways to parenthesize the right sublist, then the total is $L \cdot R$. This suggests the following recurrence for $P(n)$, the number of different ways of parenthesizing n items:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

This is related to a famous function in combinatorics called the *Catalan numbers* (which in turn is related to the number of different binary trees on n nodes). In particular $P(n) = C(n-1)$ and

$$C(n) = \frac{1}{n+1}\binom{2n}{n}.$$

Applying Stirling's formula, we find that $C(n) \in \Omega(4^n/n^{3/2})$. Since 4^n is exponential and $n^{3/2}$ is just polynomial, the exponential will dominate, and this grows very fast. Thus, this will not be practical except for very small n.

Dynamic Programming Solution: This problem, like other dynamic programming problems involves determining a structure (in this case, a parenthesization). We want to break the problem into subproblems, whose solutions can be combined to solve the global problem.

For convenience we can write $A_{i..j}$ to be the product of matrices i through j. It is easy to see that $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix. In parenthesizing the expression, we can consider the highest level of parenthesization. At this level we are simply multiplying two matrices together. That is, for any k, $1 \leq k \leq n-1$,

$$A_{1..n} = A_{1..k} A_{k+1..n}.$$

Thus the problem of determining the optimal sequence of multiplications is broken up into 2 questions: how do we decide where to split the chain (what is k?) and how do we parenthesize the subchains $A_{1..k}$ and $A_{k+1..n}$? The subchain problems can be solved by recursively applying the same scheme. The former problem can be solved by just considering all possible values of k. Notice that this problem satisfies the principle of optimality, because if we want to find the optimal sequence for multiplying $A_{1..n}$ we must use the optimal sequences for $A_{1..k}$ and $A_{k+1..n}$. In other words, the subproblems must be solved optimally for the global problem to be solved optimally.

We will store the solutions to the subproblems in a table, and build the table in a bottom-up manner. For $1 \leq i \leq j \leq n$, let $m[i,j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$. The optimum cost can be described by the following recursive definition. As a basis observe that if $i = j$ then the sequence contains only one matrix, and so the cost is 0. (There is nothing to multiply.) Thus, $m[i,i] = 0$. If $i < j$, then we are asking about the product $A_{i..j}$. This can be split by considering each k, $i \leq k < j$, as $A_{i..k}$ times $A_{k+1..j}$.

The optimum time to compute $A_{i..k}$ is $m[i,k]$, and the optimum time to compute $A_{k+1..j}$ is $m[k+1,j]$. We may assume that these values have been computed previously and stored in our array. Since $A_{i..k}$ is a $p_{i-1} \times p_k$ matrix, and $A_{k+1..j}$ is a $p_k \times p_j$ matrix, the time to multiply them is $p_{i-1} \cdot p_k \cdot p_j$. This suggests the following recursive rule for computing $m[i,j]$.

$$\begin{aligned} m[i,i] &= 0 \\ m[i,j] &= \min_{i \leq k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_k p_j) \qquad \text{for } i < j. \end{aligned}$$

It is not hard to convert this rule into a procedure, which is given below. The only tricky part is arranging the order in which to compute the values. In the process of computing $m[i,j]$ we will need to access values $m[i,k]$ and $m[k+1,j]$ for k lying between i and j. This suggests that we should organize things our computation according to the number of matrices in the subchain. Let $L = j-i+1$ denote the length of the subchain being multiplied. The subchains of length 1 ($m[i,i]$) are trivial. Then we build up by computing the subchains of lengths $2, 3, \ldots, n$. The final answer is $m[1,n]$. We need to be a little careful in setting up the loops. If a subchain of length L starts at position i, then $j = i + L - 1$. Since we want $j \leq n$, this means that $i + L - 1 \leq n$, or in other words, $i \leq n - L + 1$. So our loop for i runs from 1 to $n - L + 1$ (to keep j in bounds).

Chain Matrix Multiplication

```
Matrix-Chain(array p[1..n], int n) {
    array s[1..n-1,2..n]
    for i = 1 to n do m[i,i] = 0                    // initialize
    for L = 2 to n do {                             // L = length of subchain
        for i = 1 to n-L+1 do {
            j = i + L - 1
            m[i,j] = INFINITY
            for k = i to j-1 do {
                q = m[i, k] + m[k+1, j] + p[i-1]*p[k]*p[j]
                if (q < m[i, j]) { m[i,j] = q;  s[i,j] = k }
            }
```

```
        }
    }
    return m[1,n] and s
}
```

The array $s[i,j]$ will be explained later. It is used to extract the actual sequence. The running time of the procedure is $\Theta(n^3)$. We'll leave this as an exercise in solving sums, but the key is that there are three nested loops, and each can iterate at most n times.

Extracting the final Sequence: To extract the actual sequence is a fairly easy extension. The basic idea is to leave a *split marker* indicating what the best split is, that is, what value of k lead to the minimum value of $m[i,j]$. We can maintain a parallel array $s[i,j]$ in which we will store the value of k providing the optimal split. For example, suppose that $s[i,j] = k$. This tells us that the best way to multiply the subchain $A_{i..j}$ is to first multiply the subchain $A_{i..k}$ and then multiply the subchain $A_{k+1..j}$, and finally multiply these together. Intuitively, $s[i,j]$ tells us what multiplication to perform *last*. Note that we only need to store $s[i,j]$ when we have at least two matrices, that is, if $j > i$.

The actual multiplication algorithm uses the $s[i,j]$ value to determine how to split the current sequence. Assume that the matrices are stored in an array of matrices $A[1..n]$, and that $s[i,j]$ is global to this recursive procedure. The procedure returns a matrix.

Extracting Optimum Sequence

```
Mult(i, j) {
    if (i > j) {
        k = s[i,j]
        X = Mult(i, k)                      // X = A[i]...A[k]
        Y = Mult(k+1, j)                    // Y = A[k+1]...A[j]
        return X*Y;                         // multiply matrices X and Y
    }
    else
        return A[i];
}
```

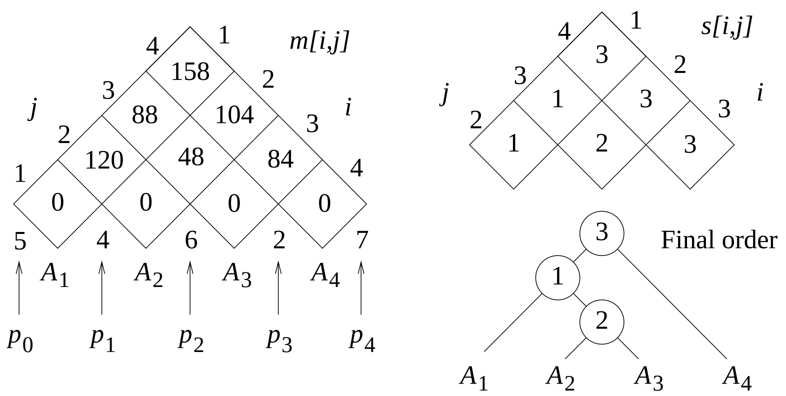


Figure 35: Chain Matrix Multiplication.

In the figure below we show an example. This algorithm is tricky, so it would be a good idea to trace through this example (and the one given in the text). The initial set of dimensions are $\langle 5,4,6,2,7\rangle$ meaning that we are multiplying A_1 (5×4) times A_2 (4×6) times A_3 (6×2) times A_4 (2×7). The optimal sequence is $((A_1(A_2A_3))A_4)$.

Lecture 27: NP-Completeness: General Introduction

(Tuesday, May 5, 1998)
Read: Chapt 36, up through section 36.4.

Easy and Hard Problems: At this point of the semester hopefully you have learned a few things of what it means for an algorithm to be efficient, and how to design algorithms and determine their efficiency asymptotically. All of this is fine if it helps you discover an acceptably efficient algorithm to solve your problem. The question that often arises in practice is that you have tried every trick in the book, and still your best algorithm is not fast enough. Although your algorithm can solve small problems reasonably efficiently (e.g. $n \leq 20$) the really large applications that you want to solve (e.g. $n \geq 100$) your algorithm does not terminate quickly enough. When you analyze its running time, you realize that it is running in *exponential time*, perhaps $n^{\sqrt{n}}$, or 2^n, or $2^{(2^n)}$, or $n!$, or worse.

Towards the end of the 60's and in the eary 70's there were great strides made in finding efficient solutions to many combinatorial problems. But at the same time there was also a growing list of problems for which there seemed to be no known efficient algorithmic solutions. The best solutions known for these problems required exponential time. People began to wonder whether there was some unknown paradigm that would lead to a solution to these problems, or perhaps some proof that these problems are inherently hard to solve and no algorithmic solutions exist that run under exponential time.

Around this time a remarkable discovery was made. It turns out that many of these "hard" problems were interrelated in the sense that if you could solve any one of them in polynomial time, then you could solve all of them in polynomial time. The next couple of lectures we will discuss some of these problems and introduce the notion of P, NP, and NP-completeness.

Polynomial Time: We need some way to separate the class of efficiently solvable problems from inefficiently solvable problems. We will do this by considering problems that can be solved in polynomial time.

We have measured the running time of algorithms using worst-case complexity, as a function of n, the size of the input. We have defined input size variously for different problems, but the bottom line is the number of bits (or bytes) that it takes to represent the input using any *reasonably efficient encoding.* (By a reasonably efficient encoding, we assume that there is not some significantly shorter way of providing the same information. For example, you could write numbers in unary notation $11111111_1 = 100_2 = 8$ rather than binary, but that would be unacceptably inefficient.)

We have also assumed that operations on numbers can be performed in constant time. From now on, we should be more careful and assume that arithmetic operations require at least as much time as there are bits of precision in the numbers being stored.

Up until now all the algorithms we have seen have had the property that their worst-case running times are bounded above by some *polynomial* in the input size, n. A *polynomial time algorithm* is any algorithm that runs in time $O(n^k)$ where k is some constant that is independent of n. A problem is said to be *solvable in polynomial time* if there is a polynomial time algorithm that solves it.

Some functions that do not "look" like polynomials but are. For example, a running time of $O(n \log n)$ does not look like a polynomial, but it is bounded above by a the polynomial $O(n^2)$, so it is considered to be in polynomial time.

On the other hand, some functions that do "look" like polynomials are not. For example, a running time of $O(n^k)$ is *not* considered in polynomial time if k is an input parameter that could vary as a function of n. The important constraint is that the exponent in a polynomial function must be a *constant* that is independent of n.

Decision Problems: Many of the problems that we have discussed involve *optimization* of one form or another: find the shortest path, find the minimum cost spanning tree, find the knapsack packing of greatest value. For rather technical reasons, most NP-complete problems that we will discuss will be phrased as decision problems. A problem is called a *decision problem* if its output is a simple "yes" or "no" (or you may think of this as True/False, 0/1, accept/reject).

We will phrase many optimization problems in terms of decision problems. For example, rather than asking, what is the minimum number of colors needed to color a graph, instead we would phrase this as a decision problem: Given a graph G and an integer k, is it possible to color G with k colors. Of course, if you could answer this decision problem, then you could determine the minimum number of colors by trying all possible values of k (or if you were more clever, you would do a binary search on k).

One historical artifact of NP-completeness is that problems are stated in terms of *language-recognition problems*. This is because the theory of NP-completeness grew out of automata and formal language theory. We will not be taking this approach, but you should be aware that if you look in the book, it will often describe NP-complete problems as languages.

Definition: Define P to be the set of all decision problems that can be solved in polynomial time.

NP and Polynomial Time Verification: Before talking about the class of NP-complete problems, it is important to introduce the notion of a verification algorithm. Many language recognition problems that may be very hard to solve, but they have the property that it is easy to *verify* whether its answer is correct.

Consider the following problem, called the *undirected Hamiltonian cycle problem* (UHC). Given an undirected graph G, does G have a cycle that visits every vertex exactly once.

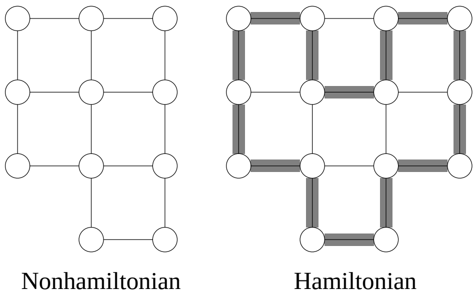


Figure 36: Undirected Hamiltonian cycle.

An interesting aspect of this problems is that *if* the graph did contain a Hamiltonian cycle, then it would be easy for someone to *convince* you that it did. They would simply say "the cycle is $\langle v_3, v_7, v_1, \ldots, v_{13}\rangle$". We could then inspect the graph, and check that this is indeed a legal cycle and that it visits all the vertices of the graph exactly once. Thus, even though we know of no efficient way to *solve* the Hamiltonian cycle problem, there is a very efficient way to *verify* that a given graph is Hamiltonian. The given cycle is called a *certificate*. This is some piece of information which allows us to verify that a given string is in a language. If it is possible to verify the accuracy of a certificate for a problem in polynomial time , we say that the problem is *polynomial time verifiable.*

Note that not all languages have the property that they are easy to verify. For example, consider the problem of determining whether a graph has *exactly one* Hamiltonian cycle. It would be easy for someone to convince that it has at least one, but it is not clear what someone (no matter how smart) would say to you to convince you that there is not another one.

Definition: Define *NP* to be the set of all decision problems that can be verified by a polynomial time algorithm.

Beware that polynomial time verification and polynomial time solvable are two very different concepts. The Hamiltonian cycle problem is NP-complete, and so it is widely believed that there is no polynomial time solution to the problem.

Why is the set called "NP" rather than "VP"? The original term NP stood for "nondeterministic polynomial time". This referred to a program running on a *nondeterministic computer* that can make guesses. Basically, such a computer could nondeterministically guess the value of certificate, and then verify that the string is in the language in polynomial time. We have avoided introducing nondeterminism here. It would be covered in a course on complexity theory or formal language theory.

Like P, NP is a set of languages based on some complexity measure (the complexity of verification). Observe that P $\subseteq$ NP. In other words, if we can solve a problem in polynomial time, then we can certainly verify that an answer is correct in polynomial time. (More formally, we do not even need to see a certificate to solve the problem, we can solve it in polynomial time anyway).

However it is not known whether P = NP. It seems unreasonable to think that this should be so. In other words, just being able to verify that you have a correct solution does not help you in finding the actual solution very much. Most experts believe that P $\neq$ NP, but no one has a proof of this.

NP-Completeness: We will not give a formal definition of NP-completeness. (This is covered in the text, and higher level courses such as 451). For now, think of the set of *NP-complete* problems as the "hardest" problems to solve in the entire class NP. There may be even harder problems to solve that are not in the class NP. These are called *NP-hard* problems.

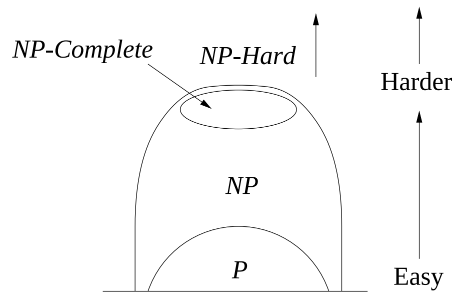


Figure 37: Relationship between P, NP, and NP-complete.

One question is how can we the notion of "hardness" mathematically formal. This is where the concept of a reduction comes in. We will describe this next.

Reductions: Before discussing reductions, let us first consider the following example. Suppose that there are two problems, A and B. You know (or you strongly believe at least) that it is impossible to solve problem A in polynomial time. You want to prove that B cannot be solved in polynomial time. How would you do this?

We want to show that

$$(A \notin \mathrm{P}) \Rightarrow (B \notin \mathrm{P}).$$

To do this, we could prove the contrapositive,

$$(B \in \mathrm{P}) \Rightarrow (A \in \mathrm{P}).$$

In other words, to show that B is not solvable in polynomial time, we will suppose that there is an algorithm that solves B in polynomial time, and then derive a contradiction by showing that A can be solved in polynomial time.

How do we do this? Suppose that we have a subroutine that can solve any instance of problem B in polynomial time. Then all we need to do is to show that we can use this subroutine to solve problem A in polynomial time. Thus we have "reduced" problem A to problem B.

It is important to note here that this supposed subroutine is really a *fantasy*. We know (or strongly believe) that A cannot be solved in polynomial time, thus we are essentially proving that the subroutine cannot exist, implying that B cannot be solved in polynomial time.

Let us consider an example to make this clearer. It is a fact that the problem of determining whether an undirected graph has a Hamiltonian cycle (UHC) is an NP-complete problem. Thus, there is no known polynomial time algorithm, and in fact experts widely believe that no such polynomial time algorithm exists.

Suppose your boss of yours tells you that he wants you to find a polynomial solution to a different problem, namely the problem of finding a Hamiltonian cycle in a *directed graph* (DHC). You think about this for a few minutes, and you convince yourself that this is not a reasonable request. After all, would allowing directions on the edges make this problem any easier? Suppose you and your boss both agree that the UHC problem (for undirected graphs) is NP-complete, and so it would be unreasonable for him to expect you to solve this problem. But he tells you that the directed version is easier. After all, by adding directions to the edges you eliminate the ambiguity of which direction to travel along each edge. Shouldn't that make the problem easier? The problem is, how do you convince your boss that he is making an unreasonable request (assuming your boss is willing to listen to logic).

You explain to your boss: "Suppose I could find an efficient (i.e., polynomial time) solution to the DHC problem, then I'll show you that it would then be possible to solve UHC in polynomial time." In particular, you will use the efficient algorithm for DHC (which you still haven't written) as a subroutine to solve UHC. Since you both agree that UHC is not efficiently solvable, this means that this efficient subroutine for DHC must not exist. Therefore your boss agrees that he has given you an unreasonable task.

Here is how you might do this. Given an undirected graph G, create a directed graph G' by just replacing each undirected edge $\{u, v\}$ with two directed edges, (u, v) and (v, u). Now, every simple path in the G is a simple path in G', and vice versa. Therefore, G has a Hamiltonian cycle if and only if G' does. Now, if you could develop an efficient solution to the DHC problem, you could use this algorithm and this little transformation solve the UHC problem. Here is your algorithm for solving the undirected Hamiltonian cycle problem. You take the undirected graph G, convert it to an equivalent directed graph G' (by edge-doubling), and then call your (supposed) algorithm for directed Hamiltonian cycles. Whatever answer this algorithm gives, you return as the answer for the Hamiltonian cycle.

UHC to DHC Reduction

```
bool Undir_Ham_Cycle(graph G) {
    create digraph G' with the same number of vertices as G
    for each edge {u,v} in G {
```

```
            add edges (u,v) and (v,u) to G'
        }
        return Dir_Ham_Cycle(G')
    }
```

You would now have a polynomial time algorithm for UHC. Since you and your boss both agree that this is not going to happen soon, he agrees to let you off.

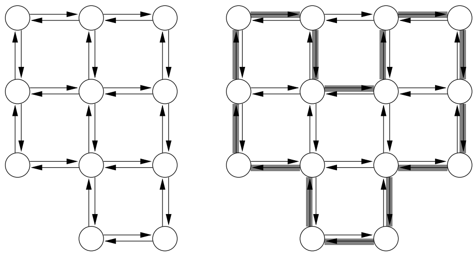


Figure 38: Directed Hamiltonian cycle reduction.

Notice that neither problem UHC or DHC has been solved. You have just shown how to convert a solution to DHC into a solution for UHC. This is called a *reduction* and is central to NP-completeness.

Lecture 28: NP-Completeness and Reductions

(Thursday, May 7, 1998)
Read: Chapt 36, through Section 36.4.

Summary: Last time we introduced a number of concepts, on the way to defining NP-completeness. In particular, the following concepts are important.

Decision Problems: are problems for which the answer is either "yes" or "no." The classes P and NP problems are defined as classes of decision problems.

P: is the class of all decisions problems that can be solved in polynomial time (that is, $O(n^k)$ for some constant k).

NP: is defined to be the class of all decision problems that can be *verified* in polynomial time. This means that if the answer to the problem is "yes" then it is possible give some piece of information that would allow someone to verify that this is the correct answer in polynomial time. (If the answer is "no" then no such evidence need be given.)

Reductions: Last time we introduced the notion of a reduction. Given two problems A and B, we say that A is *polynomially reducible* to B, if, given a polynomial time subroutine for B, we can use it to solve A in polynomial time. (Note: This definition differs somewhat from the definition in the text, but it is good enough for our purposes.) When this is so we will express this as

$$A \leq_P B.$$

The operative word in the definition is "if". We will usually apply the concept of reductions to problems for which we strongly believe that there is no polynomial time solution.

Some important facts about reductions are:

Lemma: If $A \leq_P B$ and $B \in$ P then $A \in$ P.

Lemma: If $A \leq_P B$ and $A \notin$ P then $B \notin$ P.

Lemma: (Transitivity) If $A \leq_P B$ and $B \leq_P C$ then $A \leq_P C$.

The first lemma is obvious from the definition. To see the second lemma, observe that B cannot be in P, since otherwise A would be in P by the first lemma, giving a contradiction. The third lemma takes a bit of thought. It says that if you can use a subroutine for B to solve A in polynomial time, and you can use a subroutine for C to solve B in polynomial time, then you can use the subroutine for C to solve A in polynomial time. (This is done by replacing each call to B with its appropriate subroutine calls to C).

NP-completeness: Last time we gave the informal definition that the NP-complete problems are the "hardest" problems in NP. Here is a more formal definition in terms of reducibility.

Definition: A decision problem $B \in$ NP is *NP-complete* if

$$A \leq_P B \text{ for all } A \in \text{NP}.$$

In other words, if you could solve B in polynomial time, then every other problem A in NP would be solvable in polynomial time.

We can use transitivity to simplify this.

Lemma: B is NP-complete if

(1) $B \in NP$ and

(2) $A \leq_P B$ for some NP-complete problem A.

Thus, if you can solve B in polynomial time, then you could solve A in polynomial time. Since A is NP-complete, you could solve every problem in NP in polynomial time.

Example: 3-Coloring and Clique Cover: Let us consider an example to make this clearer. Consider the following two graph problems.

3-coloring (3COL): Given a graph G, can each of its vertices be labeled with one of 3 different "colors", such that no two adjacent vertices have the same label.

Clique Cover (CC): Given a graph G and an integer k, can the vertices of G be partitioned into k subsets, $V_1, V_2, \ldots, V_k$, such that $\bigcup_i V_i = V$, and that each V_i is a clique of G.

Recall that the *clique* is a subset of vertices, such that every pair of vertices in the subset are adjacent to each other.

3COL is a known NP-complete problem. Your boss tells you that he wants you to solve the CC problem. You suspect that the CC problem is also NP-complete. How do you prove this to your boss?

There are two things to be shown, first that the CC problem is in NP. We won't worry about this, but it is pretty easy. (In particular, to convince someone that a graph has a clique cover of size k, just specify what the k subsets are. Then it is an easy matter to verify that they form a clique cover.)

The second item is to show that a known NP-complete problem (we will choose 3COL) is polynomially reducible to CC. To do this, you assume that you have access to a subroutine `CliqueCover(G, k)`. Given a graph G and an integer k, this subroutine returns true if G has a clique cover of size k and false otherwise, and furthermore, this subroutine runs in polynomial time. How can we use this "alleged" subroutine to solve the well-known hard 3COL problem?

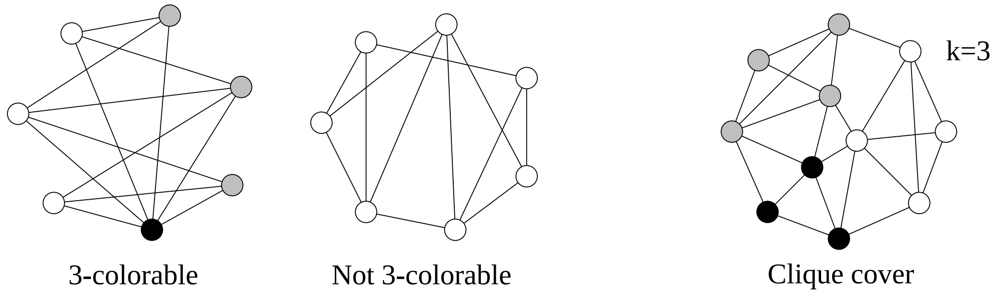


Figure 39: 3-coloring and Clique Cover.

We want to write a polynomial time subroutine for 3COL, and this subroutine is allowed to call the subroutine `CliqueCover(G,k)` for any graph G and any integer k.

Let's see in what respect the two problems are similar. Both problems are dividing the vertices up into groups. In the clique cover problem, for two vertices to be in the same group they must be adjacent to each other. In the 3-coloring problem, for two vertices to be in the same color group, they must not be adjacent. In some sense, the problems are almost the same, but the requirement adjacent/non-adjacent is exactly reversed.

Recall that if G is a graph, then $\overline{G}$ is the *complement* graph, that is, a graph with the same vertex set, but in which edges and nonedge have been swapped. The main observation is that a graph G is 3-colorable, if and only if its complement $\overline{G}$, has a clique-cover of size $k = 3$. We'll leave the proof as an exercise.

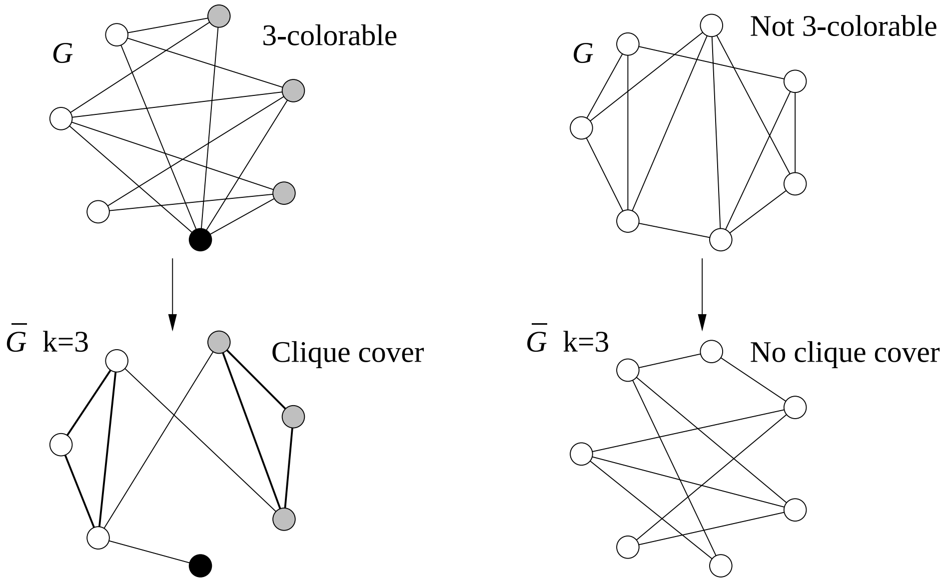


Figure 40: Clique covers in the complement.

Using this fact, we can reduce the the 3-coloring problem to the clique cover problem as follows. Remember that this means that, if we had a polynomial time procedure for the clique cover problem then we could use it as a subroutine to solve the 3-coloring problem Given the graph we want to compute the 3-coloring for, we take its complement and then invoke the clique cover, setting $k = 3$.

3COL to CC Reduction

```
bool 3Colorable(graph G) {
    let G' = complement(G)
    return CliqueCover(G',3)
}
```

There are a few important things to observe here. First, we never needed to implement the `CliqueCover()` procedure. Remember, these are all "what if" games. If we could solve CC in polynomial time, then we could solve 3COL. But since we know that 3COL is hard to solve, this means that CC is also hard to solve.

A second thing to observe that is the direction of the reduction. In normal reductions, you reduce the problem that you do not know how to solve to one that you do know how to solve. But in NP-complete, we do not know how to solve either problem (and indeed we are trying to show that an efficient solution does not exist). Thus the direction of the reduction is naturally backwards. You reduce the known problem to the problem you want to show is NP-complete. Remember this! It is quite counterintuitive.

Remember: Always reduce the known NP-complete problem to the problem you want to prove is NP-complete.

The final thing to observe is that the reduction really didn't attempt to solve the problem at all. It just tried to make one problem look more like the other problem. A reductionist might go so far as to say that there really is *only one* NP-complete problem. It has just been dressed up to look differently.

Example: Hamiltonian Path and Hamiltonian Cycle: Let's consider another example. We have seen the Hamiltonian Cycle (HC) problem (Given a graph, does it have a cycle that visits every vertex exactly once?). Another variant is the Hamiltonian Path (HP) problem (Given a graph, does it have a simple path that visits every vertex exactly once?)

Suppose that we know that the HC problem is NP-complete, and we want to show that the HP problem is NP-complete. How would we do this. First, remember what we have to show, that a known NP-complete problem is reducible to our problem. That is, HC $\leq_P HP$. In other words, suppose that we had a subroutine that could solve HP in polynomial time. How could we use it to solve HC in polynomial time?

Here is a first attempt (that doesn't work). First, if a graph has't a Hamiltonian cycle, then it certainly must have a Hamiltonian path (by simply deleting any edge on the cycle). So if we just invoke the HamPath subroutine on the graph and it returns "no" then we can safely answer "no" for HamCycle. However, if it answers "yes" then what can we say? Notice, that there are graphs that have Hamiltonian path but no Hamiltonian cycle (as shown in the figure below). Thus this will not do the job.

Here is our second attempt (but this will also have a bug). The problem is that cycles and paths are different things. We can convert a cycle to a path by deleting any edge on the cycle. Suppose that the graph G has a Hamiltonian cycle. Then this cycle starts at some first vertex u then visits all the other vertices until coming to some final vertex v, and then comes back to u. There must be an edge $\{u, v\}$ in the graph. Let's delete this edge so that the Hamiltonian cycle is now a Hamiltonian path, and then invoke the HP subroutine on the resulting graph. How do we know which edge to delete? We don't so we could try them all. Then if the HP algorithm says "yes" for any deleted edge we would say "yes" as well.

However, there is a problem here as well. It was our intention that the Hamiltonian path start at u and end at v. But when we call the HP subroutine, we have no way to enforce this condition. If HP says "yes", we do not know that the HP started with u and ended with v. We cannot look inside the subroutine or modify the subroutine. (Remember, it doesn't really exist.) We can only call it and check its answer.

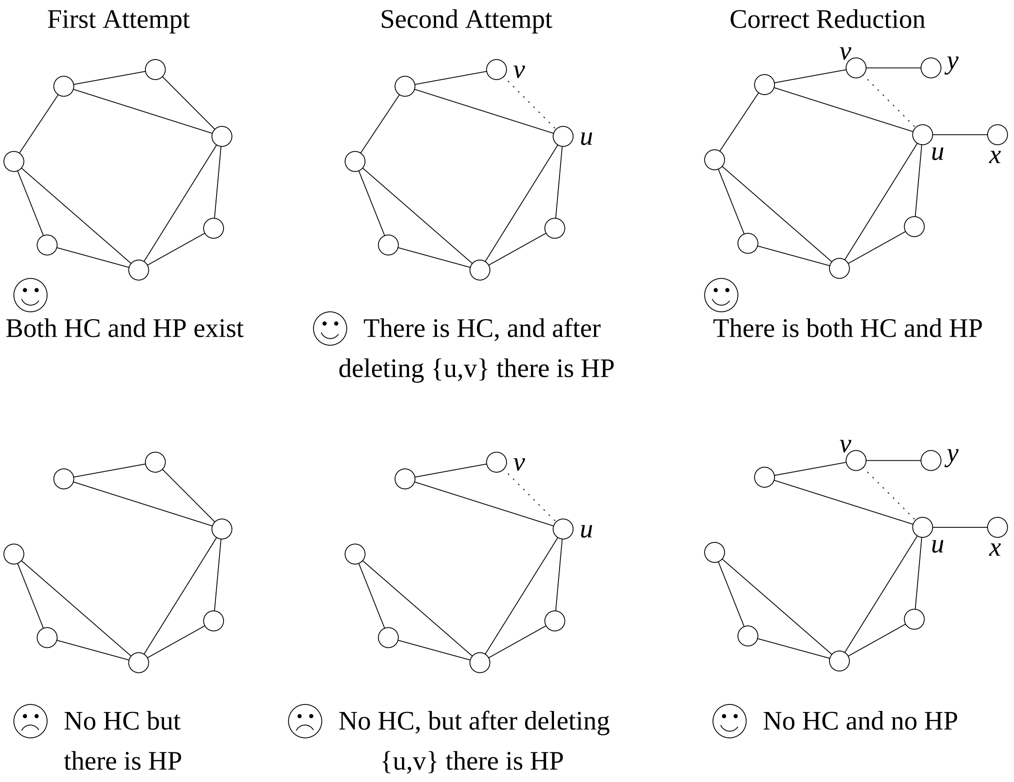


Figure 41: Hamiltonian cycle to Hamiltonian path attempts.

So is there a way to force the HP subroutine to start the path at u and end it at v? The answer is yes, but we will need to modify the graph to make this happen. In addition to deleting the edge from u to v, we will add an extra vertex x attached only to u and an extra vertex y attached only to v. Because these vertices have degree one, if a Hamiltonian path exists, it must start at x and end at y.

This last reduction is the one that works. Here is how it works. Given a graph G for which we want to determine whether it has a Hamiltonian cycle, we go through all the edges one by one. For each edge $\{u, v\}$ (hoping that it will be the last edge on a Hamiltonian cycle) we create a new graph by deleting this edge and adding vertex x onto u and vertex y onto v. Let the resulting graph be called G'. Then we invoke our Hamiltonian Path subroutine to see whether G' has a Hamiltonian path. If it does, then it must start at x to u, and end with v to y (or vice versa). Then we know that the original graph had a Hamiltonian cycle (starting at u and ending at y). If this fails for all edges, then we report that the original graph has no Hamiltonian cycle.

HC to HP Reduction

```
bool HamCycle(graph G) {
    for each edge {u,v} in G {
        copy G to a new graph G'
        delete edge {u,v} from G'
```

```
        add new vertices x and y to G'
        add new edges {x,u} and {y,v} to G'
        if (HamPath(G')) return true
    }
    return false                        // failed for every edge
}
```

This is a rather inefficient reduction, but it does work. In particular it makes $O(e)$ calls to the `HamPath()` procedure. Can you see how to do it with fewer calls? (Hint: Consider applying this to the edges coming out of just one vertex.) Can you see how to do it with only one call? (Hint: This is trickier.)

As before, notice that we didn't really attempt to solve either problem. We just tried to figure out how to make a procedure for one problem (Hamiltonian path) work to solve another problem (Hamiltonian cycle). Since HC is NP-complete, this means that there is not likely to be an efficient solution to HP either.

Lecture 29: Final Review

(Tuesday, May 12, 1998)

Final exam: As mentioned before, the exam will be comprehensive, but it will stress material since the second midterm exam. I would estimate that about 50–70% of the exam will cover material since the last midterm, and the remainder will be comprehensive. The exam will be closed book/closed notes with three sheets of notes (front and back).

Overview: This semester we have discussed general approaches to algorithm design. The goal of this course is to improve your skills in designing good programs, especially on complex problems, where it is not obvious how to design a good solution. Finding good computational solutions to problems involves many skills. Here we have focused on the higher level aspects of the problem: what approaches to use in designing good algorithms, how generate a rough sketch the efficiency of your algorithm (through asymptotic analysis), how to focus on the essential mathematical aspects of the problem, and strip away the complicating elements (such as data representations, I/O, etc.)

Of course, to be a complete programmer, you need to be able to orchestrate all of these elements. The main thrust of this course has only been on the initial stages of this design process. However, these are important stages, because a poor initial design is much harder to fix later. Still, don't stop with your first solution to any problem. As we saw with sorting, there may be many ways of solving a problem. Even algorithms that are asymptotically equivalent (such as MergeSort, HeapSort, and QuickSort) have advantages over one another.

The intent of the course has been to investigate basic techniques for algorithm analysis, various algorithm design paradigms: divide-and-conquer graph traversals, dynamic programming, etc. Finally we have discussed a class of very hard problems to solve, called NP-complete problems, and how to show that problems are in this class. Here is an overview of the topics that we covered this semester.

Tools of Algorithm Analysis:

Asymptotics: O, Ω, Θ. General facts about growth rates of functions.

Summations: Analysis of looping programs, common summations, solving complex summations, integral approximation, constructive induction.

Recurrences: Analysis of recursive programs, strong induction, expansion, Master Theorem.

Sorting:

Mergesort: Stable, $\Theta(n \log n)$ sorting algorithm.

Heapsort: Nonstable, $\Theta(n \log n)$, in-place algorithm. A heap is an important data structure for implementation of priority queues (a queue in which the highest priority item is dequeued first).

Quicksort: Nonstable, $\Theta(n \log n)$ expected case, (almost) in-place sorting algorithm. This is regarded as the fastest of these sorting algorithms, primarily because of its pattern of locality of reference.

Sorting lower bounds: Any sorting algorithm that is based on comparisons requires $\Omega(n \log n)$ steps in the worst-case. The argument is based on a decision tree. Considering the number of possible outcomes, and observe that they form the leaves of the decision tree. The height of the decision tree is $\Omega(\lg N)$, where N is the number of leaves. In this case, $N = n!$, the number of different permutations of n keys.

Linear time sorting: If you are sorting small integers in the range from 1 to k, then you can applying counting sort in $\Theta(n + k)$ time. If k is too large, then you can try breaking the numbers up into smaller digits, and apply radix sort instead. Radix sort just applies counting sort to each digit individually. If there are d digits, then its running time is $\Theta(d(n + k))$, where k is the number of different values in each digit.

Graphs: We presented basic definitions of graphs and digraphs. A graph (digraph) consists of a set of vertices and a set of undirected (directed) edges. Recall that the number of edges in a graph can generally be as large as $O(n^2)$, but is often smaller (closer to $O(n)$). A graph is *sparse* if the number of edges is $o(n^2)$, and dense otherwise.

We discussed two representations:

Adjacency matrix: $A[u, v] = 1$ if $(u, v) \in E$. These are simple, but require $\Theta(n^2)$ storage. Good for dense graphs.

Adjacency list: $Adj[u]$ is a pointer to a linked list containing the neighbors of u. These are better for sparse graphs, since they only require $\Theta(n + e)$ storage.

Breadth-first search: We discussed one graph algorithm: *breadth first search*. This is a way of traversing the vertices of a graph in increasing order of distance from a source vertex. Recall that it colors vertices (white, gray, black) to indicate their status in the search, and it also uses a FIFO queue to determine which order it will visit the vertices. When we process the next vertex, we simply visit (that is, enqueue) all of its unvisited neighbors. This runs in $\Theta(n + e)$ time. (If the queue is replaced by a stack, then we get a different type of search algorithm, called depth-first search.) We showed that breadth-first search could be used to compute shortest paths from a single source vertex in an (unweighted) graph or digraph.

Dynamic Programming: Dynamic programming is an important design technique used in many optimization problems. Its basic elements are those of subdividing large complex problems into smaller subproblems, solving subproblems in a bottom-up manner (going from smaller to larger). An important idea in dynamic programming is that of the principal of optimality: For the global problem to be solved optimally, the subproblems should be solved optimally. This is not always the case (e.g., when there is dependence between the subproblems, it might be better to do worse and one to get a big savings on the other).

Floyd-Warshall Algorithm: (Section 26.2) Shortest paths in a weighted digraph between all pairs of vertices. This algorithm allows negative cost edges, provided that there are no negative cost cycles. We gave two algorithms. The first was based on a DP formulation of building up paths based on the number of edges allowed (taking $\Theta(n^4)$ time). The second (the Floyd-Warshall algorithm) uses a DP formulation based on considering which vertices you are allowed to pass through. It takes $O(n^3)$ time.

Longest Common Subsequence: (Section 16.3) Find the longest subsequence of common characters between two character strings. We showed that the LCS of two sequences of lengths n and m could be computed in $\Theta(nm)$.

Chain-Matrix Multiplication: (Section 16.1) Given a chain of matrices, determine the optimum order in which to multiply them. This is an important problem, because many DP formulations are based on deriving an optimum binary tree given a set of leaves.

NP-completeness: (Chapt 36.)

Basic concepts: Decision problems, polynomial time, the class P, certificates and the class NP, polynomial time reductions, NP-completeness.

NP-completeness reductions: We showed that to prove that a problem is NP-complete you need to show (1) that it is in NP (by showing that it is possible to verify correctness if the answer is "yes") and (2) show that some known NP-complete problem can be reduced to your problem. Thus, if there was a polynomial time algorithm for your problem, then you could use it to solve a known NP-complete problem in polynomial time.

We showed how to reduce 3-coloring to clique cover, and how to reduce Hamiltonian cycle to Hamiltonian path.