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 accesses 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

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

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).

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

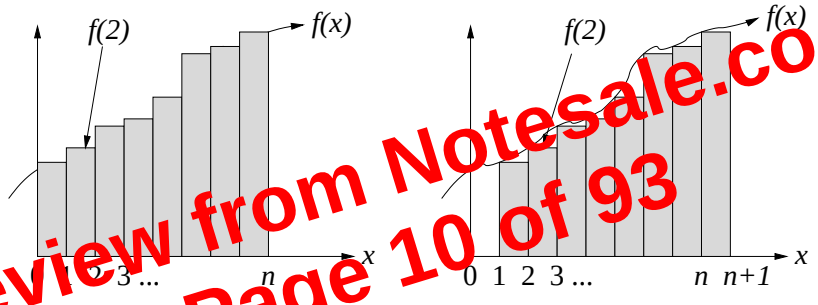

Figure 2: Approximating sums by integrals.

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, $f(x) = x^2$.

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

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.

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).

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 $n/2$ 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)

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.

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, $1 \leq k \leq n$, 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.

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 $n/4 \leq q \leq 3n/4$, 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 $n/4$. 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

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

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

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

Combining the constraints that $c \geq 1$, and $c \geq 20$, 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.

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

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

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.

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 $X = \langle YABBADABBADOO \rangle$ 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 $i' \le i$ and $j' \le j$ (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, $b[0..m, 0..n]$.

Longest Common Subsequence

```
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
```

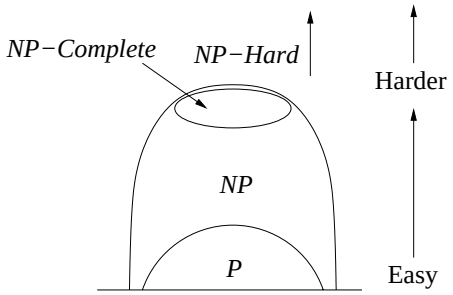

Figure 36: Relationship between P, NP, and NP-complete.

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.

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.

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.

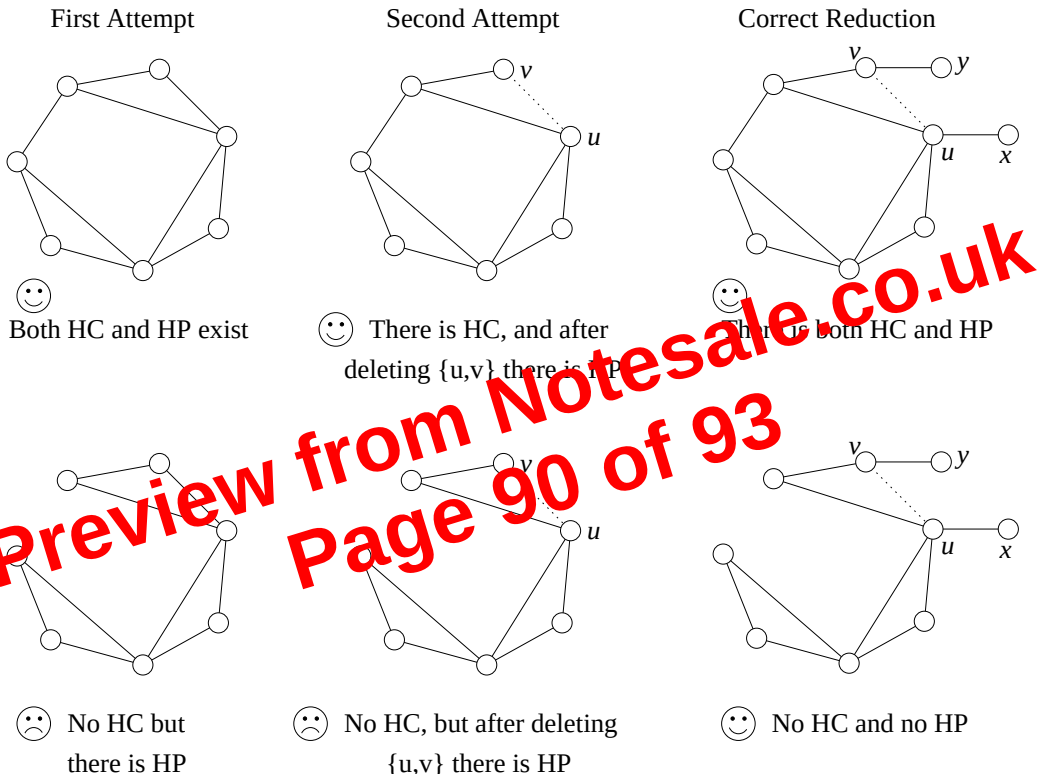

Figure 40: Hamiltonian cycle to Hamiltonian path attempts.

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.