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).

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

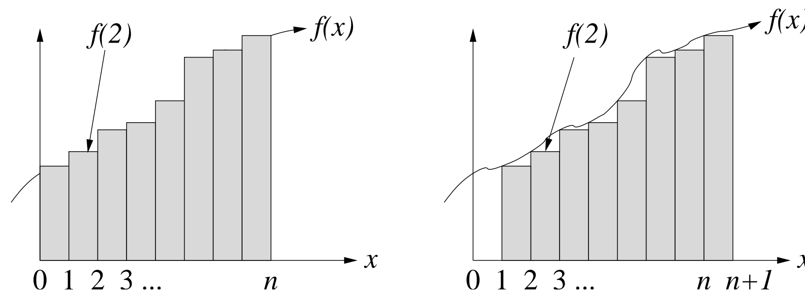

Figure 2: Approximating sums by integrals.

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

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

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.

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.

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

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 $p.x \leq q.x$ 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 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.

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