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

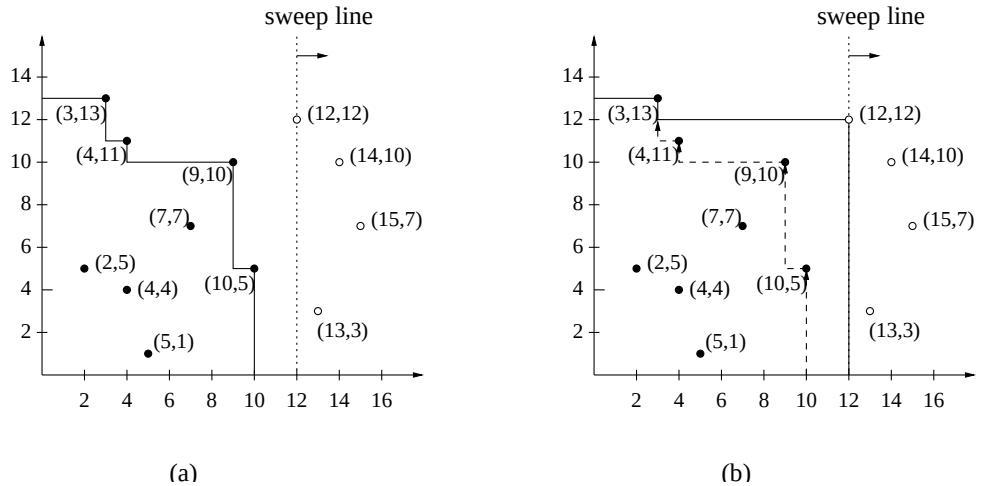

Figure 3: Plane sweep algorithm for 2-d maxima.

(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:

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

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.