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.)

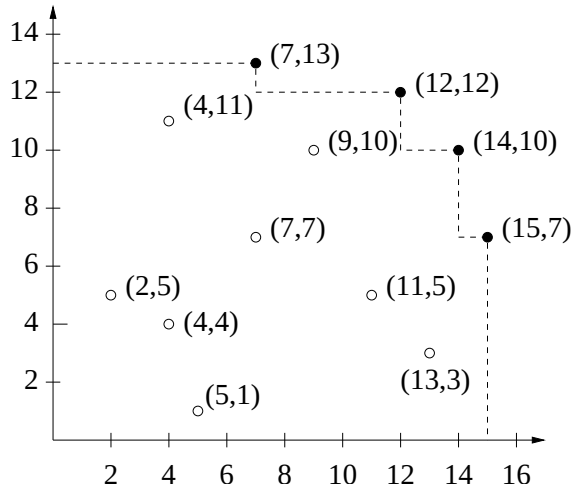

Figure 1: Maximal Points.

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

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