Advanced Algorithms

Class Notes for Thursday, September 20, 2012

Bernard Moret

This set of notes is rather short, as much of the first class meeting was devoted to administrative issues.

1 Topics To Be Covered

The goal of the course is to learn how to design algorithms for new problems, so the emphasis will be on algorithm design. However, in order to evaluate a design, we first need to be able to analyze an algorithm (not to mention prove its correctness), so we will begin with analysis of algorithms.

1.1 Analysis of Algorithms

I assume that all of you have taken an undergraduate course in algorithms in which you have seen how to analyze the worst-case running time of an algorithm. Worst-case analysis (of, usually, running time) is the principal mode of analysis for algorithms, but there are other important ones that you may not have seen. Thus I will start with a very brief review of worst-case analysis and then move on to cover these other styles in a bit more detail. These other styles include amortized analysis, randomized analysis, and competitive analysis.

1.1.1 Amortized analysis

Amortized analysis is used mostly with data structures; it is a type of worst-case analysis, but, instead of focusing on the worst-case performance of a single data structure operation, it analyzes the worst-case performance of a long series of operations. Think of these two strategies for washing dishes at home. Strategy I, the "neat" strategy, consists of washing, drying, and putting up the dishes after every single meal, no matter how small. Strategy II, the "messy" strategy (often favored by males), is to pile up dirty dishes in the sink until it becomes difficult to add another (or until scary creatures start emerging from the sink. . .), at which point one gets down to a serious session of dishwashing, cleaning everything and putting it all up. In the first case, every meal has a dishwashing cost, but the worst-case cost is not that high (unless you are in the habit of inviting 50 friends over to dinner); in the second case, most meals have no dishwashing cost at all, but every now and then one meal has a huge dishwashing cost. If we *amortize* the costs of each strategy over many weeks or months, we may find that the total time spend cleaning, drying, and putting up dishes is much the same, or perhaps (due to large-scale efficiencies) slightly smaller for the messy strategy. (Obviously, there are other considerations at play here!) We can do the same with data structures. Many data structures have been designed to control the worst-case cost per operation (think of binary heaps or balanced search trees, for instance), but an algorithm typically builds up a data structure only to demolish it and so the cost per operation is not as important as the sum of the costs of all operations during the execution of the algorithm. We shall see that we can usually beat the worst-case data structures in such settings by designing with amortized costs in mind.

1.1.2 Randomized analysis

Randomized analysis is used for so-called randomized algorithms, algorithms that deliberately use random choices as part of their strategy. The classic example you may have seen is quicksort: we know that, if we always pick the first element (or the last element, or any element at a fixed index) in the array as the pivot for partitioning, quicksort may take quadratic time in the worst case. However, if we pick a random element, then the expected running time of quicksort is provably $Ø(n \log n)$. By making random choices, the algorithm protects itself against the worst case behavior in the following sense: it can still run, in the worst case, in quadratic time, but the probability that this will happen is very low for any given input. In particular, it is no longer possible to use the same input data to force quadratic time: internal random choices make the behavior of the algorithm largely data-independent. We shall see during the semester that randomization allows us to design algorithms that are much simpler (and thus often much faster) than their worst-case cousins.

1.1.3 Competitive analysis

Finally, competitive analysis is used when we have no way of gauging the running time of our algorithm, nor, indeed, of any algorithm, but may still be able to compare the running time of our algorithm against that of an ideal, optimal (but of course completely unknown) algorithm. Here the classical example is list access, a simplified version of the paging problem in operating systems. You are given a linked list of items, each with a unique key; a user of the system will produce a series of search requests, each time providing the key of an item in the list. The list can only be accessed from the front, so that accessing the ith element in the list takes $\Theta(i)$ time. How should we maintain the list? Obviously, an optimal strategy would have to be tailored to the particular series of requests that the user will make: at the end of the day, with knowledge of the requests we had to fulfill, we could compute such an optimal strategy (nothing like hindsight!). We then could also compare the cost of the strategy we actually used to the cost of the optimal strategy. This is the essence of competitive analysis: ideally, we would end up proving some bound on the ratio between the cost of our strategy and that of the optimal strategy. In the case of the list access problem, a simple strategy is to move the requested item to the front of the list every time it is requested—on the grounds that a frequently requested item has a good chance of getting requested again soon, in which case moving it to the front will make the next such request easy to service. We will show in class that this “move-to-front” strategy has a worst-case competitive ratio of 2, that is, that it cannot be cost more than twice the cost of the optimal strategy. (The paging strategy known as LRU—for least recently used—is a version of this “move-to-front” strategy.)

1.2 Design of Algorithms

Algorithm design dates back at least to Euclid (his famous algorithm to compute the greatest common divisor of two numbers), and has been an area of intense research for at least the last 50 years. Yet in all that time, only a few broad algorithmic strategies have been devised. Focusing only on those devised to produce efficient algorithms that return exact solutions, these strategies can be roughly classified as greediness, iterative improvement, divide-and-conquer, and dynamic programming.

1.2.1 Greedy algorithms

In computation, greed is often good; and greed is always simple. A greedy algorithm optimizes its decision on a purely local basis, not looking ahead to remaining decisions. It builds a solution piece by piece, at each step choosing that piece that optimizes the current partial solution. Typical examples include both Prim’s and Kruskal’s algorithms for minimum spanning trees, Dijkstra’s algorithm for shortest paths, etc. Greedy algorithms are also widely used as heuristics for hard problems, such as scheduling tasks, the famous travelling salesperson problem, and many others. Greedy algorithms are efficient, easy to design and implement, and easy to analyze in terms of running time; it is often difficult, however, to prove anything about the quality of the solutions they return.

1.2.2 Iterative improvement algorithms

Iterative improvement algorithms, as their name indicates, start with some solution and attempt to improve it, repeating the work after each iteration, until no more improvement is possible. The iteration uses one specific way to look for possible improvements, so even after the algorithm has concluded that no more improvement is possible, the solution may not be optimal: no more improvement is possible using the algorithm chosen for the iterative step, but another algorithm might still find improvements. A typical example is the 2-opt strategy for the travelling salesperson problem: given a particular tour, test each pair of non-adjacent edges in the tour, say (i, j) and (k, l), to see whether replacing them by cross-edges, in this case (i, l) and (j, k), would shorten the tour; if any such exchange does improve the tour, apply it and repeat the step. The main application of iterative improvement algorithms is in matching and flow problems—crucial problems underlying most tasks in job scheduling, transportation, and communication.

1.2.3 Divide-and-conquer algorithms

You should be familiar with mergesort, which is perhaps one of the simplest and most effective uses of divide-and-conquer, D&C for short. A D&C algorithm evaluates the size of the instance; if the instance is small enough, it applies some simple method to solve it; otherwise, it divides the instance into subinstances, often (as in mergesort) non-overlapping ones, solves the subinstances by recursion, then uses the solutions to the subinstances to construct the solution to the original instance. The correctness of a D&C algorithm does not depend on the manner in which the instances are divided (mergesort will correctly sort an array even if the array is divided into very uneven parts), but its running time often does. D&C algorithms tend to be simple (one glaring exception is matrix multiplication) and efficient; they are also easy to analyze. D&C algorithms are common in computational geometry, but relatively few optimization problems can be solved optimally with D&C.

1.2.4 Dynamic programming algorithms

Dynamic programming algorithms, DP for short, take over where D&C algorithms leave off. Unlike D&C algorithms, they do not require that any division of the instance into subinstances yield correct answers; instead, they explore every possible way of breaking up the instance. DP algorithms are therefore much slower than D&C (an efficient DP algorithm typically runs in cubic time, although some run faster, even in linear time), but more broadly applicable. Most problems dealing with genomic data are solved through DP techniques. DP algorithms can be difficult to devise, but they are trivial to implement and analyze. Since their basic approach is to store every intermediate result, they are space-intensive.

1.3 Optimization and search algorithms for hard problems

We will not address this topic in the course. One can of course formulate heuristics using any of the approaches listed above, but to derive optimal solutions one needs to resort to algorithms that perform some form of state space search, such as back-tracking, branch-and-bound, etc., or rephrase the problem in terms of constraints and use constraint solvers such as integer-linear programming methods. One can also use popular generic heuristic approaches such as simulated annealing, genetic algorithms, classifier systems, ant-colony optimization, and many others of the same ilk, but these general-purpose approaches cannot be analyzed, are usually very compute-intensive, require a lot of hand-tuning, and typically underperform purpose-built methods,

2 Mathematical Background

2.1 Notation

In algorithm analysis, the idea is to produce analyses valid under any reasonable implementation, that is, independent of languages, coders, compilers, and machines. Moreover, most algorithms have startup costs (initializing various data structures, for instance) that tend to figure prominently in the running time for small instances, but make no difference for large instances. For these various reasons, algorithms are analyzed in asymptotic terms: that is, we characterize their behavior on unboundedly large instances. This approach makes sense, but raises a number of questions that have proved very hard to answer: what do we mean by a "large" instance? when is "asymptopia" (the region where asymptotic behavior dominates) too far from real-world applications? should we take into account hardware limitation (word size, both for addressing very large arrays and for questions of arithmetic precision)? These questions do not have good global answers, but much of their difficulty vanishes when focusing on a particular class of problems.

So how do we handle asymptotic behavior? Mathematicians have solved this issue for us already—limits of functions and series provide every tool we need. However, mathematicians have more than one version of "asymptotic behavior" and even have ways to convert between versions. The main two version are *almost everywhere (a.e.)* and *infinitely often (i.o.)* asymptotics. If we want to say that a function f does not grow any faster than another function g as their argument n grows large, we are looking at an inequality of the form $f(n) \leq \alpha \cdot g(n)$, where $\alpha > 0$ is some suitable proportionality constant, and want this inequality to hold for unboundedly large values of n. The question, though, is: for all sufficiently large values of n? or for infinitely many values of n? The first defines a.e. asymptotics and we would then require

$$\exists N,\ \forall n > N,\ f(n) \leq \alpha \cdot g(n)$$

which defines "asymptopia" as beginning (for this particular problem) at the value N. The second defines i.o. asymptotics and we would simply require

$$\forall N,\ \exists n > N,\ f(n) \leq \alpha \cdot g(n)$$

That is, there are infinitely many values of n for which the inequality holds, but there need not be any point beyond which it always holds. The two are of course related: all we did is to exchange the two quantifiers: universal for existential and vice versa. In terms of bounds, a.e. bounds are stronger than i.o. bounds, so we may prefer them; but tight i.o. bounds may exist in case where only loose a.e. bounds can be derived (as we shall see), so we have a tradeoff.

In real-time computing, we may need absolute guarantees on the running time of certain routines (imagine an interrupt handler for certain dangerous conditions in a chemical plant, or automated control of planes on landing at airports), so a.e. upper bounds make sense. On the other hand, when it comes to bad news (lower bounds), it is enough to know that we cannot escape the bad behavior, and so i.o. lower bounds suffice. Thus our upper bounds (big Oh notation) should be defined in a.e. terms and our lower bounds (big Omega notation) in i.o. terms. Indeed, this is precisely how they were first defined, by Don Knuth and others. However, this proved somewhat confusing and the number of cases where it made a useful difference proved rather limited, so for over 30 years now bounds are all given in a.e. terms.

Before we move on, let us look at one example where the original definition would be more meaningful. Consider the trivial approach to testing a number n for primality: we simply run a loop from 2 to $\lfloor\sqrt{n}\rfloor$, checking for each value whether it is a divisor of n, stopping immediately (and answering no) if it is; if we run through all iterations without stopping, we answer yes and halt. The worst-case for this algorithm is a prime number; the best case is an even number. The behavior is linear in n (and thus exponential in the input size, which is $\log_2 n$), but we cannot use a precise a.e. lower bound: in a.e. terms, all we can state is that this algorithm takes $\Omega(\log n)$ time. In contrast, however, if we use an i.o. definition, then we can write that this algorithm takes $\Omega(n)$ time, since there is an infinity of primes. The exponentially tighter lower bound here justifies the use of i.o. lower bounds.

2.2 Recurrences

You have no doubt seen that setting up recurrences is a useful way to capture the time complexity of an algorithm. And many of you no doubt went on to study how to solve recurrences in some detail. Formal methods for solving recurrences are the same as those for solving differential equations: the simplest is the method of characteristic roots, while the more powerful is the method of transforms (also called generating functions). In addition, various informal methods abound: get the first few values and guess the series (not very likely to succeed for an exact answer), repeatedly substitute terms in the recurrence (unfold the recurrence) and attempt to guess the form of the answer from the terms generated (much more fruitful), and so forth. This all makes for a fun mathematics, but it is by and large not all that useful in algorithmic analysis, for the simple reason that solvable recurrences arising from algorithmic analysis tend to one of a very few forms and, sadly, most nontrivial algorithms give rise to recurrences that we cannot solve through formal methods.

Let us take a quick look at an example of the latter. Below is a program intended to sort an array; the programmer was well intentioned and had learned about mergesort and selection sort, but got a bit confused along the way…

```
procedure SillySort(var A: arraytype; i, j: integer);
  (* sorts subarray A[i..j] *)
  var temp: integer;
  begin
    if i < j  (* If i = j there is nothing to do. *)
      then begin
             m := (i+j) div 2;
             SillySort(A,i,m);
             SillySort(A,m+1,j);
             (* Largest element in the entire array is the larger
                of A[m] and A[j]. *)
             if A[m] > A[j]
               then begin (* swap *)
                      temp := A[m]; A[m] := A[j]; A[j] := temp
                    end;
             (* Largest element in place--sort the rest. *)
             SillySort(A,i,j-1)
           end
  end; (* SillySort *)
```

Due to the recursive nature of the algorithm, setting up a recurrence to characterize its running time is straightforward:

$$t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + t(n-1) + c_1, \qquad \text{with } t(1) = c_2$$

Solving it, however, turns out to be much harder: none of the methods in our bag of tricks works—generating functions and characteristic roots, with or without transformations, all fail. This failure suggests bounding $t(n)$ rather than deriving an exact expression for it. Bounding usually proceeds in stages, from loose initial bounds to tighter bounds. First note that $t(n)$ grows no faster than 3^n, since 3^n characterizes the function, $t'(n)$, obtained from $t(n)$ by replacing each $t(n/2)$ term with the larger term, $t(n-1)$. Then note that $t(n)$ grows faster than any polynomial: if we had $t(n) = \sum_{i=0}^{k} a_i n^i$, then, substituting into the recurrence, looking only at the terms of highest degree, and ignoring the floors and ceilings, we should get

$$a_k n^k + \cdots = a_k(n-1)^k + \cdots + 2a_k\left(\frac{n}{2}\right)^k + \cdots ,$$

which simplifies to

$$a_k n^k + \cdots = a_k n^k + \frac{a_k}{2^{k-1}} n^k + \cdots ,$$

which is impossible for large n. Trying to show that $t(n)$ is exponential, i.e., $\Omega(2^{n^\varepsilon})$ for some ε, we get a similar contradiction: $t(n)$ does not grow that fast, because we have $\lim_{n\to\infty} 2^{(n-1)^\varepsilon}/2^{n^\varepsilon} = 0$. Thus we must conclude that $t(n)$ is a subexponential function. Another approximation gives us an explicit subexponential bound for the function: since the tail recursion results in n iterations on a steadily decreasing range, an upper bound can be derived by treating each tail recursive call as if it were working on the entire original array, giving us (for n a power of 2):

$$t(n) \le n\cdot\bigl(2t(n/2) + c\bigr).$$

Substituting $n = 2^k$, we get the new recurrence

$$g(k) \le 2^{k+1} g(k-1) + c\cdot 2^k.$$

If we ignore the driving function and use repeated substitution, we see that $g(k)$ grows roughly as 2^{k^2}. In fact, if we assume that $g(k)$ is of the form $2^{\alpha k^2 + \beta k + \gamma}$, we can derive the bound

$$g(k) = O(2^{\frac{k}{2}(k+4)}),$$

so that $t(n)$ is $O(n^{a \log n})$ for some suitable constant, a. Thus $t(n)$ grows no faster than a subexponential. In fact, $t(n)$ is $\Omega(n^{b \log n})$, as can be shown by further arguments in the same style.

The morale of this tale is that even very simple programs, for which setting up a recurrence is trivial, can yield recurrences that are extremely difficult to solve. We did solve it, eventually, but not through powerful formal solution methods such as generating functions: instead, we used clever observations to get increasingly tighter bounds until our upper and lower bounds coincided. Thus there is little point in perfecting our mastery of formal solution methods; instead, we shall see throughout the course various *ad hoc* approaches to the solution of recurrences, tailored to the recurrence and to the level of precision we want in the analysis.