Algorithms

Uriel Feige
Department of Computer Science and Applied Mathematics
The Weizman Institute
Rehovot 76100, Israel
feige@wisdom.weizmann.ac.il

November 17, 1999

1 Lecture 1, October 21, 1999

When designing an algorithm for a computational problem, we shall be concerned with correctness and with complexity. The measure that we shall often use for complexity is the number of basic operations as a function of the input size. This roughly corresponds to the running time of the algorithm, but not exactly. The actual running time also depends on the data structures used, on hierarchical structure of memory and locality of reference, on the instruction set of the particular processor and the relative costs of various operations, on the optimizing compiler, and so on. Giving a detailed analysis of running time is beyond the scope of this course. Instead, we shall illustrate the issues involved through the example of sorting.

The input to a sorting problem is a list of n elements $X = x_1, \ldots, x_n$. There is a total order relation on the elements (e.g., by size, or by lexicographic order). For simplicity and w.l.o.g. we may assume here that no two elements are equal. The *order* in X of an element x_i is equal to the number of elements in X not larger than x_i. The goal of the sorting algorithm is to come up with a permutation $\pi : \{1, n\} \longrightarrow \{1, n\}$ such that for every i, the order of x_i is $\pi(i)$.

We shall consider algorithms based on comparisons, and our measure of complexity will be the number of comparisons.

1.1 Insertion sort

Insertion sort: Consider the elements of X one by one. By the end of phase i, we hold a list X_i that is the sorted version of the prefix $x_1, \ldots, x_i$. In phase $i+1$, use binary search to insert x_{i+1} in its correct location in X_i, obtaining the list X_{i+1} which is the sorted version of the prefix $x_1, \ldots, x_{i+1}$. After phase n, output X_n.

Complexity: The number of comparison operations needed to insert element x_i is roughly $\log i$ (log denotes logarithms to the base 2). Hence the total number of comparisons is roughly $\sum_{i=2}^{n} \log i \leq n \log n$.

In terms of number of comparison operations, insertion sort is almost optimal w.r.t. comparison based algorithms. As computation branches only as the result of comparisons, every comparison based sorting algorithm needs to perform at least $\log n! \simeq n \log n$ comparisons so as to be able to produce each of the $n!$ possible outputs. (A similar lower bound holds also for comparison based randomized algorithms.)

However, in practice, insertion sort is not very efficient. The reason is that it needs to pay a lot in data movement operations. Assume for example that X_i is held in an array of length i. Then in order to insert x_{i+1} and create the array X_{i+1}, we may need to move $\Omega(i)$ elements, giving a total of $\Omega(n^2)$ data movements. For large values of n, the time needed to move data around may be a dominant factor in the running time for insertion sort, making the number of comparisons an inadequete measure of complexity.

Using more complicated data structures, it is possible to reduce the amount of data movement, perhaps at the expense of additional comparisons. This leads to algorithms such as *heapsort*. Hence the basic idea behind insertion sort does lead to algorithms that are of practical significance, though going from this basic idea to a practical algorithm involves additional design stages of the type that we will not be concerned with in this course.

1.2 Quicksort

We now consider another well known sorting algorithm.

Quicksort: Choose a *splitting* element x_i. Compare every element to x_i and split the list $X - \{x_i\}$ into X_l (the elements smaller than x_i) and X_h (the elements larger than x_i). Sort each of the lists X_l and X_h recursively.

The complexity of quicksort depends on the choice of splitting element. Ideally, for every sublist X' generated during a run of the algorithm, the splitting element for X' is the median of X', splitting X' into two equal size parts. This gives $\log n$ levels of recursion. As each element is compared at most once in each level, the number of comparisons then is roughly $n \log n$.

However, if the splitting element is always the largest element in X', we have n levels of recursion, requiring $\Omega(n^2)$ comparisons.

One reason why quicksort works well is that the average case behaves more like the best case than like the worst case, as we shall see shortly. But the average case does require more comparisons than insertion sort. So why is quicksort preferable? The reason is that quicksort pays very little overhead on top of comparisons. It is relatively easy to implement quicksort in a way that at most one data item is moved per comparison. (For example, use two pointers advancing from both sides on X', and think of the splitting element x as lying in between. Pointers stop on elements that are misplaced with respect to x, these elements are swapped, and then the pointers continue to advance.) Moreover, quicksort exhibits high locallity of reference, paying relatively little overhead for paging/caching.

Expected complexity: We consider here randomized quicksort, in which the splitting element for X' is chosen uniformly at random from the elements of X'. Intuitively, in this case X' is expected to split at a ratio of roughly $(1/4, 3/4)$, giving $O(\log n)$ levels of recursion, and $O(n \log n)$ comparisons. It is possible to use "brute force" and give a detailed analysis based on this intuition. However, we shall seek a simpler analysis.

Let us first prove again an upper bound on the number of comparisons. Consider an arbitrary pair of elements. Quicksort compares them at most once. Hence the total number of comparisons is at most $\binom{n}{2}$. (A similar upper bound holds for most comparison based sorting algorithms.)

Now let us use similar principles to bound the expected number of comparisons. For simplicity of notation, let $x_1, \ldots, x_n$ denote the sorted order of X (which is what the algorithm is trying to find). Let y_{ij} denote the indicator random variable that is 1 if x_i and x_j where directly compared during a run of quicksort, and 0 otherwise. (The random variable depends on the choices of random splitters.) Hence we are interested in the expectation $E[\sum_{i<j} y_{ij}]$. Working with expectations is very convenient, because of the *linearity of expectation*, which holds even for random variables that depend on each other. Hence we have $E[\sum_{i<j} y_{ij}] = \sum_{i<j} E[y_{ij}]$.

Observe that $E[y_{ij}] = \sum_k k\text{Prob}[y_{ij} = k]$ is exactly equal to the probability that x_i and x_j were directly compared. This probability is exactly $2/(j-i+1)$. (Consider the first time when one of $x_i, \ldots, x_j$ happens to be a splitting element. If either x_i or x_j are the splitting element, they are directly compared. Otherwise. they end up in different sublists and are never compared.) Hence

$$\sum_{i<j} E[y_{ij}] = \sum_{i<j} \frac{2}{j-i+1} \simeq \sum_{i=1}^{n-1} 2\ln(n-i) \simeq 2n\ln n$$

Hence the expected numer of comparisons in randomized quicksort is roughly $2n \ln n$.

1.3 Randomized algorithms

Randomized quicksort is a randomized algorithm. It is of the "Las Vegas" type, meaning that it always produces the correct answer, and randomization only effects the running time. It is possible to run a deterministic version of quicksort in which the splitting element is chosen by some deterministic rule (e.g., the first element in X', or the middle element of X'). In this case the running time would depend on the input. For most inputs, the running time would be $O(n \log n)$, but for some rare inputs, the running time may be as high as $\Omega(n^2)$. The purpose of choosing the splitting element at random is so as to have quicksort have expected $O(n \log n)$ running time for *every* input.

When analysing the running time of a randomized algorithm, one may be interested in the whole distribution (for every t, the probability that the algorithm runs for t steps). However, in most cases, the expectation is the major measure to consider. There are several reasons for this.

1. If the algorithm is run many times on many inputs, then by the laws of large numbers, the average run time will converge to the expectation. (The number of runs needed so as to observe such a convergence depends on other measures such as the variance. An implication of the items below is that the variance in the running time of randomized algorithms does not behave badly, implying rather quick convergence.)

2. Let T be the running time of a randomized algorithm. Because $T \geq 0$, we trivially have that for every $c > 1$, $\text{Prob}[T \geq cE[T]] \leq 1/c$. Hence we also have tail bounds on the probability of exceptionally long runs.

3. To get better control of the tail bounds, one may stop a Las Vegas algorithm once it runs for $2E[T]$ steps and restart it on the original input with fresh random coin tosses. In each period of $2E[T]$ steps it has probability at least 1/2 of finishing. Hence in this case, $\text{Prob}[T \geq cE[T]] \leq (1/2)^{c/2}$. The probability of exceptionally long runs decays exponentially.

4. For many Las Vegas algorithms (such as quicksort), restarting on the original input is not really necessary. We can restart on the concatenation of sublists that ended the previous phase (as this is a legal input), or better still, just continue the algorithm without interruption. Hence for randomized quicksort, the probability of exceptionally long runs decays exponentially.

(Homework in Handout 1)