CS 218 : Design and Analysis of Algorithms

Lecture 4: Basic Principles of Algorithm Design

Lecturer: *Sundar Vishwanathan*
Computer Science & Engineering — Indian Institute of Technology, Bombay

We begin with the problem of finding a minimum element in an array. The standard algorithm is to scan the array from left to right maintaining the current minimum. Here is "code" that implements the above algorithm.

1. min is assigned the first element in the array
2. For the other elements in the array do
3. If min is greater than the current element
   Replace min by the current element;

The array above is scanned from left to right. For this course we discourage writing of code. It is enough and in fact preferable to describe your algorithm in English. However there should be enough detail in your description that if needed one can convert this description to code. Even if you do write code, it must be heavily commented and must be preceded by a description of the working of your algorithm in English.

Here is another way to view the design of this algorithm. Assuming that there is an algorithm that finds the minimum on arrays of size at most $n-1$, can you design an algorithm to find the minimum for arrays of size n?. The above question is typical of inductive reasoning and is *the most important approach to algorithm design.* The answer: Use the algorithm for array size $n-1$ to compute the minimum of the first $n-1$ elements. Now compare the result of this with the nth element and output the minimum.

While this is a simple problem, let us highlight the main design principles.

Principle 1. Induction. Assuming there is an algorithm for inputs of smaller size, construct one for the current input. Another way to think about this is how do you extend the solution on smaller sized input to the current input?

Once we answer this question, the actual algorithm is written using either recursion or iteration. The algorithm for smaller sized input is a recursive call. Initially, for this course, stick to recursion.

Principle 2. Order. This relates to the order in which we consider the input. In this case we looked at the array elements in order of increasing index. In this simple case order does not really play a role. We could have looked at the elements in any order. But soon we shall see examples where the order plays a crucial role.

Principle 3. Store reusable old values. Here we stored the previous minimum which we used for the next step. The general principle is store values which you may reuse.

To be effective, you should be able to say which values are being reused. At this point this may not seem like a big deal, but when combined with the other two design principles, it often is non-trivial to spot. The first principle is problem independent. It applies to every problem. The other two are problem dependent.

With any algorithm we need to make two claims. One that it is correct and the other on its efficiency. For most problems in this course it will be obvious that the algorithms we construct are correct. If you are designing algorithms that are not obviously correct, you should revisit your design. In fact, when you construct algorithms, correctness should be your primary goal.

Aim first for simple algorithms that are obviously correct. Later you can work to improve it. Proving bounds on efficiency will be the difficult part.

Let us analyze the algorithm for finding the minimum. The operations performed are : incrementing an index, comparing two elements and assignment. In any case, the total time taken is a constant times the number of comparisons. We focus on the number of comparisons. The number of comparisons is $n-1$. Each element of the array $A[i], 2 \leq i \leq n$, is compared with the minimum. Can we do better? It seems like we cannot! Here is one argument which is incorrect. Every element of the array has to be compared with some other element in the array. Hence we need to make at least n comparisions. Can you spot the error in the above argument? The reason is that $\lceil n/2 \rceil$ comparisons suffice to make sure that every element is compared with some other element.

Exercise 1 Show this.

Exercise 2 : Show that $\lceil n/2 \rceil$ comparisons are also necessary to ensure that every element is compared at least once. Note that Exercise 2 proves that we need to make at least $\lceil n/2 \rceil$ comparisons. Here is a correct argument. Initially there are n candidates for the minimum. By candidates we mean that one can assign values to the elements, which are consistent with the answers of the comparisons made so far, so that each of the candidates can be made the minimum. The crucial fact is that with each comparison one can decrease the number of candidates by at most one. Can you write a sentence explaining why this is true? Also note that this is sufficient to prove that we require $n-1$ comparisons to find the minimum.

Another way to see this is the following. Consider the following graph on n vertices. There is one vertex per element in the array. When an element $A[i]$ is compared with an element $A[j]$, draw an edge between vertices i and j. The number of candidates at any stage is at least the number of connected components. We can now invoke the well known fact that one needs $n-1$ edges to make an n-vertex graph connected. This shows that the obvious algorithm we had, to find the minimum, is optimal.

We strongly suggest that you review the three principles over and over again before you begin designing any algorithm. Especially so, if you are having trouble designing an algorithm.

We mention another classic from your data structures course. Binary search in a sorted array. You first compare with the middle element of the array–a clear case where the right order is important. Then you recurse on the appropriate portion of the array.