Lecture 11: Sorting. Proofs of correctness and lower bounds. Selection sort.

Doina Precup
With many thanks to Prakash Panagaden and Mathieu Blanchette

January 29, 2014

1 Sorting

Suppose that you are given an array of elements and you want to put the elements in increasing or decreasing order. This can be useful for many different reasons, but here I will note just two. First, as we saw, we can search much faster in a sorted array than in an unsorted array (by doing binary search, instead of sequential search). Second, when we present data to customers, it can be very useful to show it in order (e.g. the names of students in a class, the bank transactions ordered by the date etc). Sorting is one of the most studied problems in computer science, and many different algorithms have been developed. Today we will discuss our first sorting algorithm: selection sort.

2 Selection sort

The idea of selection sort is very simple: we can repeatedly select the maximum element in an array, move it in the last position, then select the maximum in the remaining portion, move it into the second to last position etc., until no elements are left. For examples, supposed we have the following array:

1 4 2 3

We find the max element (4) and its index, and we swap it in the last position. This leads to the array:

1 3 2 4

Now we will look at the first 3 elements in the array only, find the max, and swap it in the last position in this array portion:

1 2 3 4

Further iterations of the algorithm will look at the first 2 elements, then the first element, but no further modifications are necessary.

In order to write this algorithm, we will need a "helper" function (or method, in Java), which finds the index of the maximum element within the first n elements of an array and returns it. We now turn our attention to this algorithm.

3 Finding the maximum in an array revisited

Consider the simple algorithm of finding the index to the maximum element in an array of numbers:

Algorithm findMaxIndex(a,n)
Input: An array a of n elements, which can be compared to each other
Output: The index of the largest element in the array
int $max \leftarrow a[0]$
int $indmax \leftarrow 0$
for $i \leftarrow 1$ **to** $n-1$
  if ($a[i] > max$) **then**
    $max \leftarrow a[i]$
    $indmax \leftarrow i$
  endif
return $indmax$

How many steps does this take? Let us count the comparisons that take place inside the loop. The loop gets executed exactly $n-1$ times, regardless of how the array looks. So the best, worst and average number of comparisons is $n-1$. Indeed, this $n-1$ is a **lower bound** on the number of operations needed. In other words, without doing at least this many comparisons, you cannot generate a proof that the claimed maximum is indeed the maximum. Indeed, suppose that we could come up with an algorithm that does only $n-2$ comparisons. In this case, there must be some element in the array that never gets compared with the max. If you imagine an **adversary** arranging the input data, he or she can hide the true max in the element that never gets compared with the candidate max. So the solution you return will be wrong.

In general, **lower bounds** on the running time of an algorithm tell you how many operations are needed, in the worst case, **by any algorithm** solving this problem. If you prove a lower bound for a problem, then you know that no algorithm, no matter how smart or fancy, can beat this, in the worst case. The typical way to prove lower bounds is to use **adversarial arguments**, like the one we saw above. This assumes that an adversary has knowledge of what you are doing, and tries to do what's worst for you. Providing these kinds of proofs is often quite tricky.

How do we prove that this algorithm is correct? We need to note that, after every execution of the loop, max will contain the largest element in the part of the array seen so far, and indmax will contain the corresponding index. We can write this as follows:

$$\text{after } i \text{ iterations } \forall j \leq i, a[indmax] \geq a[j]$$

We will say that this condition is a **loop invariant**. Since it holds after every execution of the loop, it will also hold at the end of the algorithm, so at the end $a[indmax] \geq a[i], \forall i < n$. In

general, proving the correctness of iterative algorithms (algorithms involving loops) requires us to use loop invariants. These will be different depending on the algorithm. So there is no "blueprint" for how these proof should go. We will see next time that it is actually quite a bit easier to prove the correctness of recursive algorithms, using a technique called proof by induction.

4 Selection Sort pseudocode and running time

The pseudocode of the algorithm is as follows:

Algorithm selectionSort(a, n)
Input: An array a of n elements
Output: The array will be sorted in place (i.e. after the algorithm finishes, the elements of a will be in non-decreasing order)
int $i \leftarrow n-1$
while $i > 0$ **do**
  int $indmax \leftarrow$findMaxIndex(a,i)
  swap(a, i, $indmax$)
  $i \leftarrow i-1$
return

Here, findMaxIndex is the algorithm fro the previous section. The swap algorithm simple swaps the values of the elements whose indices are sent as arguments:

Algorithm swap(a,i,j)
Input: Array a and indices i and j
Output: The content of elements $a[i]$ and $a[j]$ will be swapped
$tmp \leftarrow a[i]$
$a[i] \leftarrow a[j]$
$a[j] \leftarrow tmp$
return

Now let us prove that the selection sort algorithm works correctly. This means that at the end of the execution, we need to have $a[j] \leq a[i], \forall j < i$. To prove that the algorithm works correctly, we need to find a **loop invariant** which helps us show the condition above. Let the loop invariant be the following:

$$\text{after iteration with index} i, a[i] \geq a[j], \forall j < i$$

To show this, recall that findMaxIndex works correctly, as we showed before. So it will find the largest element among indices 1 to i. Then the swap will put this value at position i. Since we are working backwards, after the first iteration, we have $a[n] \geq a[j], \forall j < n$. After the second iteration, $a[n-1] \geq a[j], \forall j < n-1$ etc. Moreover, after the iteration with index i, element $a[i]$ is not visited anymore. This proves the loop invariant. Note that by putting together the statements that are true after each i, the loop invariant implies the ordering condition that we want to show.

To find the complexity of selection sort, consider the number of comparisons that will have to

be executed. We know that the algorithm for finding the max of n elements takes $n-1$ comparisons. So here we will have $(n-1)$ comparisons in the first call to findMaxIndex, then $n-2$ in the second call etc. The comparison in the while loop is executed n times. So the total number of comparisons will be:

$$n + (n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n} i = \frac{(n+1)n}{2}$$

To see this equality, you can simply group the first and last term, second and second-to-last etc. Each such sum is $n+1$, and there should be $n/2$ such terms. So (based on the calculation above) the algorithm is $O(n^2)$. We will use induction to prove this formally next lecture.