

## CS303 (Spring 2008)— Solutions to Assignment 2

### Problem 1

(a) Correctness Proof for Selection Sort

We want to prove that the algorithm outputs a sorted array with the same elements as before. The main step in proving the correctness of this algorithm is coming up with the right invariant (induction hypothesis), one that will let us measure the stepwise progress of the algorithm, and, when applied for the  $n^{\text{th}}$  iteration, imply that the algorithm has sorted the array. In coming up with the right invariant, it is usually right to use one's intuition about what the algorithm does (or is supposed to do). Here, we arrive at the following invariant:

INV: After  $i$  iterations, (1) the array still contains all the same elements as before, and (2) the array positions  $a[j]$  for  $1 \leq j \leq i$  contain the  $j^{\text{th}}$  smallest elements, sorted.

Before we proceed to proving this invariant by induction, it is useful to contrast it with the invariant we used for Insertion Sort. The first part is identical, but for the second part, we used to say that the first  $i$  positions still contain *the same* elements as before, except now in sorted order. In Selection Sort, the first  $i$  positions contain the elements they should contain in the end.

In both cases, when we apply the invariant for  $i = n$ , we obtain that the array still contains the same elements and is sorted, which means that the algorithm did the right thing. So we want to establish INV by induction now.

- The base case: With  $i = 0$ , the algorithm has not done anything yet, so the array of course is still unchanged, and the first 0 elements contain what they should (since there are no elements).
- The induction step from  $i$  to  $i + 1$ : We get to assume that after  $i$  iterations, the invariant held for  $i$ , and need to prove that after one more iteration, it holds for  $i + 1$ . We prove the two parts of INV separately. For part (1), we get to assume by induction hypothesis that the array still contained the same numbers after  $i$  iterations. The only way in which the array is changed is by the **swap** operation, which only changes the order of elements in an array, but does not overwrite anything. So the array still contains the same elements as before after  $i + 1$  iterations.

For part (2), we need to show that the first  $i + 1$  array positions contain the  $i + 1$  smallest elements, in the correct order. For the first  $i$  elements, we can argue that the array already contained the correct elements after  $i$  iterations (by Induction Hypothesis for  $i$ ), and iteration number  $i + 1$  does not access or overwrite or swap those positions, so they still contain the right elements. To prove that the element in position  $i + 1$  is the right one, we notice that the inner loop finds the smallest element between positions  $i + 1, \dots, n$  (this could be proved by another induction, on  $j$ , very similar to the example of computing the maximum from class). Because the  $i$  smallest elements are in positions  $1, \dots, i$ , the smallest among the remaining is the  $(i + 1)$ -smallest element overall, which is then swapped into position  $i + 1$ , so that that position contains the right element. This completes the inductive proof.

- (b) For each iteration  $i$ , the inner loop runs from  $i$  to  $n$ , and does constant amount of work for each  $j$  value. Hence, the inner loop takes time  $O(n - i)$ , which is  $O(n)$ . So the total amount of work is

$$\sum_{i=1}^{n-1} O(n) = O(\sum_{i=1}^{n-1} n) = O((n-1)n) = O(n^2).$$

Can the algorithm really take time  $n^2$ ? Let us look for a lower bound on its performance. In fact, notice that for *any* array, the algorithm will run the inner loop from  $i$  to  $n$  (even if the array were already sorted), so the running time of the  $i^{\text{th}}$  iterations is  $\Omega(n - i)$ . So the total running time is

$$\sum_{i=1}^{n-1} \Omega(n - i) = \Omega(\sum_{i=1}^{n-1} n - i) = \Omega(\sum_{i=1}^{n-1} i) = \Omega(n(n - 1)/2) = \Omega(n^2)$$

(In the middle step here, we noticed that the sum  $\sum_{i=1}^{n-1} n - i$  is equal to  $(n - 1) + (n - 2) + \dots + 3 + 2 + 1$ . That is the same as  $1 + 2 + 3 + \dots + (n - 2) + (n - 1)$ , just in opposite order. So we were allowed to replace the sum by  $\sum_{i=1}^{n-1} i$ .) So our analysis is actually tight up to constants: the algorithm in the worst case takes exactly  $\Theta(n^2)$  steps (and in fact, we just saw that it takes the same amount for the best case, too). Notice that our analysis also showed that in this case, our crude bound of replacing  $n - i$  by  $n$  did not cost us more than a constant factor.

- (c) In each iteration, we only perform at most one **swap** operation, and there are  $n$  iterations, for a total of  $O(n)$  swaps. On the other hand, if the array is the worst case array  $[n, 1, 2, 3, \dots, n - 1]$ , then the algorithm swaps on each of the  $n - 1$  iterations, for a total of  $\Omega(n)$ . Thus, the total number of swaps is  $\Theta(n)$ . Notice that the array  $[n, n - 1, \dots, 3, 2, 1]$  causes  $n/2$  swaps, so it is also good enough to prove  $\Omega(n)$ , though it is off by a factor of 2 from the worst case.

## Problem 2

Here is our Java implementation of the two algorithms.

### Insertion Sort

```
import java.util.Date;

public final class InsertionSort
{
    public static int [] a;
    public static void main (String [] args) throws Exception
    {
        for (int z = 0; z < args.length; z ++)
        {
            int n = Integer.parseInt(args[z]);
            a = new int[n];
            for (int i = 0; i < n; i ++) a[i] = n-i;
            Date startTime = new Date ();
            for (int i = 1; i < n; i ++)
            {
                int j = i, temp = a[i];
                while (j > 0 && (temp < a[j-1]))
                {
                    a[j] = a[j-1];
                    j --;
                }
                a[j] = temp;
            }
            System.out.println (n + ": " + (new Date().getTime() - startTime.getTime()));
        }
    }
}
```

(Notice the slight optimization of avoiding a number of unnecessary swaps.)

## Selection Sort

```
import java.util.Date;

public final class SelectionSort
{
    public static int [] a;
    public static void main (String [] args) throws Exception
    {
        for (int z = 0; z < args.length; z ++)
        {
            int n = Integer.parseInt(args[z]);
            a = new int[n];
            for (int i = 1; i < n; i ++) a[i] = i;
            a[0] = n;
            Date startTime = new Date ();
            for (int i = 0; i < n-1; i ++)
            {
                int k = i;
                for (int j = i+1; j < n; j ++)
                    if (a[j] < a[k]) k = j;
                int temp = a[i];
                a[i] = a[k];
                a[k] = temp;
            }
            System.out.println (n + ": " + (new Date().getTime() - startTime.getTime()));
        }
    }
}
```

(Notice that the worst case generated in the code here is  $[n, 1, 2, 3, \dots, n - 1]$ , which might make a slight difference in the running time.)

## Running Times

Below is a plot of the running times for different array sizes. These were measured on a fairly old Pentium machine, running Linux and Sun JDK. I didn't control for background processes or such. The running time is given in milliseconds. The plot also shows the curves of quadratic functions with appropriately fitted constants. The constants (for running time in milliseconds) were  $c_I = 0.0000019$  and  $c_S = 0.0000018$ . (If you want microseconds, get rid of three zeroes.) Thus, in the worst case, Selection Sort is a little faster than Insertion Sort (probably mostly due to fewer swaps). However, notice that Selection Sort is not much faster even in the *best* case, whereas Insertion Sort is extremely fast if the array is already mostly sorted. In practice, Insertion Sort is almost certainly preferable.

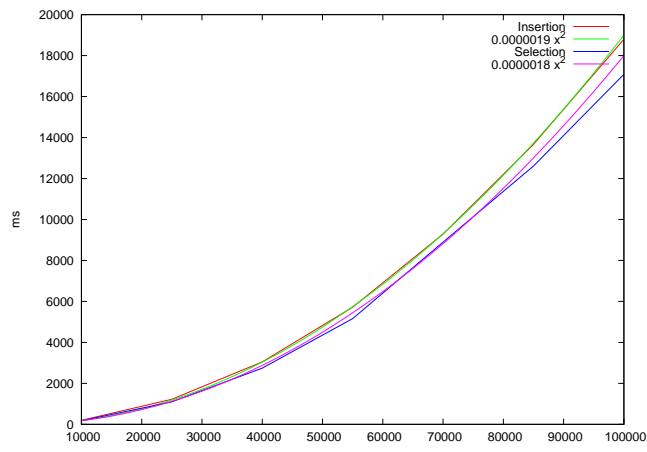


Figure 1: Running Times of Sorting Algorithms and Fits