Exercise 2.2-2

Jack Maney

August 15, 2013

Throughout, we'll assume that there are n elements in the list that we wish to sort ($n > 1$). As per the book, we'll use c_i to denote the (constant) cost for running the i^{th} line of code.

Before running through the code for the selection sort algorithms (one recursive, one not recursive) below, let's first answer some of the other questions within this problem. There need only be $n - 1$ swaps within selection sort because once the smallest $n - 1$ elements are sorted, the remaining element must be the n^{th} smallest–ie the maximum.

We first consider the recursive sort method for Selection sort, as per SelectionSort.java.

```
1  public static <T extends Comparable<T>> void sort(AbstractList<T> list){
2  
3          if(list.size() > 1){
4  
5                  T min = list.get(0);
6                  int minIndex = 0;
7  
8                  for(int i = 1; i < list.size(); i++){
9  
10                         if(list.get(i).compareTo(min) < 0){
11                                 min = list.get(i);
12                                 minIndex = i;
13                         }
14                 }
15 
16                 if(minIndex > 0){
17                         T temp = list.get(0);
18                         list.set(0,min);
19                         list.set(minIndex,temp);
20                 }
21 
22                 ArrayList<T> restOfList = new ArrayList<>();
23 
24                 for(int i = 1; i < list.size(); i++){
```

```
25                              restOfList.add(list.get(i));
26                      }
27
28                      sort(restOfList);
29
30                      for(int i = 1; i < list.size(); i++){
31                              list.set(i,restOfList.get(i - 1));
32                      }
33
34              }
35      }
```

The thing to worry about is on line 28, where we call our sort method on the unsorted elements of our list. Since the cost of this operation is obviously not constant, let's denote the cost by s_n.

We let t denote the number of times that the condition tested in line 10 is true.

cost	times
c_3	1
c_5	$n-1$
c_6	$n-1$
c_8	n
c_{10}	$n-1$
c_{11}	t
c_{12}	t
c_{16}	1
c_{17}	1 if $t > 0$, else 0
c_{18}	1 if $t > 0$, else 0
c_{19}	1 if $t > 0$, else 0
c_{22}	1
c_{24}	n
c_{25}	$n-1$
s_n	1
c_{30}	n
c_{31}	$n-1$

Now, since $0 \le t \le n$, there exist constants a and b such that the total running time is

$$T(n) = an + b + s_n.$$

Applying induction, there exist positive constants $a_1, a_2, \cdots, a_n$ such that

$$T(n) = na_n + (n-1)a_{n-1} + \cdots + 2a_2 + a_1.$$

Letting $m = \min(a_1, a_2, \cdots, a_n)$ and $M = \max(a_1, a_2, \cdots, a_n)$, we have

$$m \sum_{i=1}^{n} i <= T(n) <= M \sum_{i=1}^{n} i,$$

and therefore this algorithm has a running time of $\Theta\left(\sum_{i=1}^{n} i\right) = \Theta(n^2)$.

Note that for any array, there are several lines that are still evaluated either n or $n-1$ times for each iteration of the recursion (e.g. c_5, c_6, c_8, et al). Thus, the best case, worst case, and (therefore) average running times are also $\Theta(n^2)$.

We now move on to the non-recursive version. We first start with computing the running time of the *smallestIndex* method (with comments omitted). This method returns the index of the k^{th} smallest element of a list of n elements $1 \leq k \leq n$ (or the first such index if multiple such minima exist).

```java
public static <T extends Comparable<T>> int smallestIndex(AbstractList<T> list,int k){

        if(k < 0 || k >= list.size()){
                throw new IllegalArgumentException();
        }

        ArrayList<Integer> allowedIndices = new ArrayList<>();

        for(int i = 0; i < list.size(); i++){
                allowedIndices.add(i);
        }

        int result = -1;

        for(int i = 0; i < k; i++){

                T min = null;

                for (Integer j : allowedIndices) {

                        if(min == null){
                                min = list.get(j);
                        }
                        else if(min.compareTo(list.get(j)) > 0){
                                min = list.get(j);
                        }

                }

                result = list.indexOf(min);

32                      allowedIndices.remove(new Integer(result));
33              }
34
35              return result;
36  }
```

As before, we'll use c_i to denote constant running time on line i. The main thing to notice is the nested loop in lines 15–33. The outer loop runs k times, and for a given i with $0 \leq i < k$, the inner loop runs $n - i$ times. So, the condition in line 21 is evaluated a total of

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

times. Finally, we let t_{ij} denote the number of times that line 25 is run (ie the number of times that we find a new minimum in our list within our allowed indices). Note that $1 \leq t_{ij} \leq n$.

c_3	1
c_4	1
c_7	1
c_9	$n+1$
c_{10}	n
c_{13}	1
c_{15}	$k+1$
c_{17}	k
c_{19}	$\frac{n(n+1)-k(k+1)}{2}$
c_{21}	$\frac{n(n+1)-k(k+1)}{2}$
c_{22}	k
c_{24}	$\frac{n(n+1)-k(k+1)}{2}$
c_{25}	t_{ij}
c_{30}	k
c_{32}	k
c_{35}	1

So, there exist constants a, b, c, d, e such that the total running time is

$$T(n,k) = an^2 + bn + ck^2 + dk + e$$

Therefore this method has a running time of $\Theta(n^2 + k^2)$.
Now, let's look at the non-recursive selection sort method itself:

```java
1  public static <T extends Comparable<T>> void sort(AbstractList<T> list){
2
3          for(int i = 0; i < list.size() - 1; i++){
4
5                  T temp = list.get(i);
6                  int index = smallestIndex(list,i+1);
7                  T smallest = list.get(index);
8                  list.set(i, smallest);
9                  list.set(index, temp);
10         }
11
12 }
```

Letting c_i again denote constant running times, we note that for each $1 \leq i < n-1$, line 6 has a running time of $\Theta\left(n^2 + (i+1)^2\right)$.

c_3	$n+1$
c_5	n
$\Theta\left(n^2 + (i+1)^2\right)$	$n-1$
c_7	n
c_8	n
c_9	n

Therefore, there exist constants a, b, c, d such that the total running time of this algorithm is

$$T(n) = an^3 + bn^2 + cn + d,$$

and we have the running time as $\Theta(n^3)$.

Since the runtime of *smallestIndex* method is at least $\Theta(n^2)$ and *smallestIndex* is evaluated $n-1$ times within the loop in the *sort* method, we have a runtime of $\Theta(n^3)$ for the best, average, and worst case scenarios.