Supervision work
Supervision 10
26 Jan 2016

Tudor Mihai Avram
Homerton College – tma33@cam.ac.uk

January 25, 2016

Algorithms
— Worksheet 2 —

1 Exercise 1

If we have the permutation a = (7, 6, 5, 4, 3, 2, 1), the number of times the comparation fails is equal to $\frac{n*(n-1)}{2} = O(n^2)$.

2 Exercise 2

Both bubble sort and selection sort will make the same number of operations on any random input. Therefore, in the worst-case scenario, when the array is already sorted, their complexity will still be $O(n^2)$.
On the other hand, the number of computations made by insertion sort depend on the format of the input. Taking the same worst-case scenario, insertion sort's complexity will tend to $O(n)$.
In conclusion, even though in theory all three sorting algorithms have a complexity of $O(n^2)$, in practice **insertion sort** behaves better than the other two.

3 Exercise 4

Algorithm 1 Bottom-up merge sort

```
procedure MERGE(a,l,h,r) :
    i ← l
    j ← h + 1
    k ← l

    while i ≤ h and j ≤ r  do                                  ▷ merging
        if a[i] < a[j] then :
            aux[k] ← a[i]
            i ← i + 1
        else:
            aux[k] ← a[j]
            j ← j + 1
        k ← k + 1
    end while

    while i ≤ h do :                                           ▷ leftovers in the left half
        aux[k] ← a[i]
        i ← i + 1
        k ← k + 1
    end while

    while j ≤ r do :                                           ▷ leftovers in the right half
        aux[k] ← a[j]
        j ← j + 1
        k ← k + 1
    end while

    i ← l
    while i ≤ r do :                                           ▷ pasting the auxiliary array in the main one
        a[i] ← aux[i]
        i ← i + 1
    end while

procedure MERGESORT(a,l, r) :
    if l >r then return
    h ← int((l+r)/2)

    mergeSort(a,l,h)                                           ▷ left half
    mergeSort(a,h+1,r)                                         ▷ right half

    merge(a,l,h,r)                                             ▷ merging the 2 halfs
```

4 Exercise 3

In order to reduce the amount of extra space used by merge sort to $\frac{n}{2}$, we need to change the recursive calls inside the function. We will keep the sorted first half in a new array, while the sorted second half will be returned on the correspondent pointer in the original array. The merging will be done directly over the original array, instead of using an auxiliary one. Therefore, we only need to use one auxiliary array, of length $\frac{n}{2}$.

5 Exercise 5

(i)

Algorithm 2 InsertionSort (with 1st half sorted)

```
procedure InsSort(a) :
    n ← len(a)
    for i from int(n/2) (included) to n(excluded) do :
        j ← i − 1
        while j ≥ 0 and a[j]>a[j + 1] do :
            swap(a[j], a[j + 1])
            j ← j + 1
        end while
    end for
```

The only change I can see in this case is to start the insertion sort from $\frac{n}{2}$, instead of 0. The insertion sort itself always keeps the first $i - 1$ elements of the array sorted, so the function is not very different from the classic insertion sort.

6 Exercise 6

(i) **Merge sort** :

1. $(4, 6, 3, 2, 7, 3, 9)$ — Initial state
2. $(4, 6, 3, 2, 7, 3, 9)$ — After the first merge
3. $(4, 6, 3, 2, 7, 3, 9)$ — After the 2nd merge
4. $(3, 4, 6, 2, 7, 3, 9)$ — After the 3rd merge
5. $(3, 4, 6, 2, 7, 3, 9)$ — After the 4th merge
6. $(3, 4, 6, 2, 7, 3, 9)$ — After the 5th merge
7. $(3, 4, 6, 2, 3, 7, 9)$ — After the 6th merge
8. $(2, 3, 3, 4, 6, 7, 9)$ — After the 7th merge

(ii) **Quicksort** :
Note : I always choose first element of the (sub)array to be the pivot

1. $(4, 6, 3, 2, 7, 3, 9)$ — Initial state
2. $(3, 3, 2, 4, 7, 6, 9)$ — Step 1 (with 4 as a pivot)
3. $(2, 3, 3, 4, 7, 6, 9)$ — Step 2 (with 3 as a pivot for the 1st half)
4. $(3, 2, 3, 4, 6, 7, 9)$ — Step 3 (with 7 as a pivot for the 2nd half)

(iii) **Heapsort** :

1. $(2, 3, 3, 4, 7, 9, 6)$ — Original heap
2. $(3, 3, 4, 7, 9, 6)$ — the reconstructed heap after 2 was popped out
3. $(3, 4, 6, 7, 9)$ — the reconstructed heap after 3 was popped out
4. $(4, 6, 7, 9)$ — the reconstructed heap after 3 was popped out
5. $(6, 7, 9)$ — the reconstructed heap after 4 was popped out
6. $(7, 9)$ — the reconstructed heap after 6 was popped out
7. (9) — the reconstructed heap after 7 was popped out

7 Exercise 7

Picking the pivot at a fixed position k in an array, is the same as picking it at random. This happens because it is the pivot's value, not its position that determines the performance of the quicksort algorithm. So, even if we choose the pivot to be on a fixed position or at a random one, the algorithm's general performance will not be affected.

8 Exercise 8

(i) In order to find the minimum of n elements, we initialise a variable $min = a[0]$ and we compare min to $a[i], \forall n$ integer,$1 \leq i < n$ and we update it if we find a smaller number. Therefore, we need $n-1$ comparations to get the minimum of n integers.
(ii) If we want to find the second minimum of an n-integers array, we also need to keep track of the minimum. Let the 2 variables be $min1$ for the first minimum and $min2$ for the second one. We initialise the values as it follows : $min1 = min(a[0], a[1])$ and $min1 = max(a[0], a[1])$, so we need 1 comparation for this operation. The rest of the algorithm goes through every the number $a[i], \forall n$ integer, $2 \leq i < n$:

```
for (int i = 2; i < n; i++) {
        if (a[i] < min1) {
                // we update both minimums
                min2 = min1;
                min1 = a[i];
        }
        else if (a[i] < min2) {
                // we update only min2
                min2 = a[i];
        }
}
```

From the above algorithm, we can see that, in worst case, we have to make 2 comparations for every i, $2 \leq i < n$. In conclusion, we need $2*(n-2)+1 = 2*n-3$ comparations.

9 Exercise 9

For both implementations, I used the following 2 auxiliary methods :

```
private static void swap(int[] a, int i, int j) {
        // PRECONDITIONS :
        if (i < 0 || j < 0) return;
        if (i > a.length || j > a.length) return;
        // actual implementation :
        int aux = a[i];
        a[i] = a[j];
        a[j] = aux;
}

private static int partition(int[] a, int left, int right,
                int pivot) {
        int i = left - 1;
```

```
        int p = a[pivot];
        swap(a,pivot,right);
        for(int j=left; j < right; j++) {
                if(a[j] <= p) {
                        i++;
                        swap(a,i,j);
                }
        }
        swap(a,i+1,right);
        return i+1;
}
```

(i) pivot on the first element of the partition :

```
private static void quickSort1(int[] a, int left,
                        int right) {
        // pivot on the first element
        if (right <= left) return;
        int mid = partition(a, left, right, left);
        quickSort1(a, left, mid - 1);
        quickSort1(a, mid + 1,right);
}
```

(ii) pivot chosen at random :

```
private static void quickSort2(int[] a, int left,
                        int right) {
        // pivot chosen at random
        if (right <= left) return;

        // getting the random value for the pivot
        int dif = right - left;
        int pivot = left + randomNumber.nextInt(dif);

        //actual implementation
        int mid = partition(a, left, right, pivot);
        quickSort2(a, left, mid - 1);
        quickSort2(a, mid + 1,right);
        }
```

Note : *randomNumber* is a global variable for the class.

Both methods (*quickSort*1 and *quickSort*2) where tested using the following main function :

```
public static void main(String[] args) {
        int[] a = new int[7];
        a[0] = 4; a[1] = 6; a[2] = 3;
        a[3] = 2; a[4] = 7; a[5] = 3;
        a[6] = 9;
        // fixed pivot
        quickSort1(a,0,6);
        for (int i  = 0; i < 7; i++) {
                System.out.println(a[i]);
        }

        a[0] = 4; a[1] = 6; a[2] = 3;
        a[3] = 2; a[4] = 7; a[5] = 3;
        a[6] = 9;
        // random pivot
        quickSort2(a,0,6);
        for (int i  = 0; i < 7; i++) {
                System.out.println(a[i]);
        }
}
```

10 Exercise 10

STABLE : insertion sort, merge sort, bubble sort, selection sort;
NOT STABLE : quicksort, heap sort, counting sort,