COMP 3804 — Solutions Assignment 2

Question 1: Write your name and student number.

Solution: Cristiano Ronaldo, CR7

Question 2: You are given a sequence S of n numbers. An element x in S is called a *majority element* if it occurs more than $n/2$ times in S.

This question asks you to describe two algorithms that decide if the sequence S contains a majority element; if it does, the algorithm returns it; otherwise, the algorithm returns the message "there is no majority element".

You are highly encouraged to use any algorithm and any result that was discussed in class. In other words, try to make your algorithms as short as possible by using algorithms discussed in class as "black boxes".

(2.1) How many majority elements can there be? Justify your answer.

Solution: A majority element occurs more than $n/2$ times. If there were two majority elements, there would be more than $n/2 + n/2 = n$ elements, a contradiction. Thus, there can be at most one majority element.

(2.2) Show how the majority problem can be solved in $O(n \log n)$ time. Argue why your algorithm is correct and why the running time is $O(n \log n)$.

Solution: The main observation is the following: If the sequence is sorted, then equal elements are next to each other. This leads to the following algorithm:

- Sort the sequence, for example, using merge-sort.
- By scanning the sorted sequence once, count how many times each distinct element occurs.
- Take an element x that occurs most often. If x occurs more than $n/2$ times, return it; otherwise, return "there is no majority element".

The sorting step takes $O(n \log n)$ time. The scanning step takes $O(n)$ time. The scanning step can be done such that it gives x and the number of times it occurs. Thus, the final step can be done in $O(1)$ time. The total running time is

$$O(n \log n) + O(n) + O(1) = O(n \log n).$$

(2.3) Show how the majority problem can be solved in $O(n)$ time. Argue why your algorithm is correct and why the running time is $O(n)$.

Solution: The main observation is the following:

If there is a majority element, then it must be the median.

Why is this the case: Let m be the median. Assume there is a majority element x, and assume that $x \neq m$. Then $x < m$ or $x > m$. Assume that $x < m$ (the case when $x > m$ is symmetric). Since the number of elements less than m is at most $n/2$, the element x occurs at most $n/2$ times. Thus, x is not a majority element, which is a contradiction.

This leads to the following algorithm:

- Compute the median x.
- Scan the sequence and count how many times x occurs.
- If x occurs more than $n/2$ times, return it; otherwise, return "there is no majority element".

The median-finding step takes $O(n)$ time. The scanning step takes $O(n)$ time. The final step takes $O(1)$ time. The total running time is

$$O(n) + O(n) + O(1) = O(n).$$

Question 3: Some textbooks contain statements of the form "The running time of algorithm $\mathcal{A}$ is at least $O(n^2)$". Explain why such a statement does not make sense.

Solution: The statement "The running time of algorithm $\mathcal{A}$ is $O(n^2)$" means that the running time is *at most* cn^2 for some constant c. So the statement in this question says that the running time is *at least at most* cn^2 for some constant c. This is, of course, nonsense.

Question 4: You are given k lists, each one containing a sorted sequence of numbers. Let n denote the total number of elements in all these lists. Give an $O(n \log k)$–time algorithm that merges these k lists into one sorted list. Explain why the running time of your algorithm is $O(n \log k)$.

Hint: Use a min-heap. If $k = 2$, this problem should look familiar to you.

Solution: We will denote the k sorted lists by $L_1, \ldots, L_k$. The merged sorted numbers will be stored in an array.

If $k = 2$, then this problem is exactly the merge step in the merge-sort algorithm. The algorithm in this question generalizes this to an arbitrary number k of lists.

The idea is as follows. We walk, simultaneously, along all lists. For every i ($1 \leq i \leq k$), let y_i be the element of list L_i that we are currently visiting. (Initially, y_i is the first element of L_i, $1 \leq i \leq k$.) Then we find the smallest number, call it x, in $y_1, \ldots, y_k$. This element x is the next element in the final sorted order. If i is the index such that $x \in L_i$, then we set y_i to the successor of x.

Below, you find the algorithm in pseudocode. (It is fine if you explain the algorithm in English, as long as you make it very clear what you are doing.)

We will use an array $A[1 \ldots k]$, where $A[i]$ will store a pointer to the element y_i in the list L_i. We will use a min-heap $H[1 \ldots k]$ to store the elements $y_1, \ldots, y_k$. Finally, the final sorted sequence will be stored in the array $B[1 \ldots n]$. I assume that every element "knows" the list that contains this element.

> **for** $i = 1$ **to** k
> **do** $A[i] =$ pointer to the first element of L_i;
> $H[i] =$ first element of L_i
> **endfor**;
> $build\_min\_heap(H)$;
> $j = 1$;
> **while** $j \leq n$
> **do** $x = extract\_min(H)$;
> $B[j] = x$;
> $j = j + 1$;
> $i =$ index such that x was originally in L_i;
> $A[i] =$ pointer to the successor of $A[i]$ in L_i;
> $y =$ element pointed to by $A[i]$;
> $heap\_insert(H, y)$
> **endwhile**

The for-loop takes $O(k)$ time. It takes another $O(k)$ time to make a heap out of the array H. Since H stores k elements, one iteration of the while-loop takes $O(\log k)$ time. Hence, the entire while-loop takes $O(n \log k)$ time. The overall running time is

$$O(k + n \log k) = O(n \log k),$$

because $k \leq n$.

Remark: There is a technical detail. In the while-loop, we change $A[i]$ to the successor of $A[i]$. If we have reached the end of list L_i, then there is no successor. In order to avoid this, we can add the element ∞ at the end of each list L_i, $1 \leq i \leq k$.

Question 5: You are given a min-heap $A[1 \ldots n]$ on n elements and an integer k with $1 \leq k \leq n$. Give an algorithm that computes the k smallest elements in A in sorted order. The running time of your algorithm must be $O(k \log k)$. Explain why your algorithm is correct and why its running time is $O(k \log k)$.

Hint: It is easy to get an $O(k \log n)$–time algorithm. In a min-heap, the root of any subtree contains the minimum of all elements in that subtree. You may think of forming a min-heap consisting of $O(k)$ elements using the elements of the given heap A of n elements. As always, you may use any algorithm that was discussed in class.

Solution: We want to use the min-heap $A[1 \ldots n]$ to compute the k smallest elements in sorted order. The smallest element is equal to $A[1]$. What is the second smallest element? It is the smaller of $A[2]$ and $A[3]$, that is, the smaller of the two children of $A[1]$. Let us assume that $A[2] < A[3]$. Then $A[2]$ is the second smallest element. What is the third smallest element? It is the smaller of $A[3]$, $A[4]$ and $A[5]$, that is, the smaller of $A[3]$ and the two children of $A[2]$. Let us assume that $A[5] < A[3]$ and $A[5] < A[4]$. Then $A[5]$ is the third smallest element. What is the fourth smallest element? It is the smaller of $A[3]$, $A[4]$, $A[10]$ and $A[11]$, that is, the smaller of $A[3]$, $A[4]$, and the two children of $A[5]$. I hope the general idea is clear now.

Here is an outline of the algorithm. We use a min-heap H that contains the candidates for the next element in the sorted output. Initially, H stores only the element $A[1]$. Then we make a sequence of k iterations. In one iteration, we take the smallest element, call it x, in H, and delete it from H. This element x will be the next element in the output. Then we go back to the heap A, take the two children of x, and insert them into H.

More formally, the algorithm takes as input the min-heap $A[1 \ldots n]$ and the integer k with $1 \leq k \leq n$. The output is an array $B[1 \ldots k]$ containing the k smallest elements of A in sorted order. The algorithm will use a min-heap H.

Algorithm:
H = min-heap storing only the number $A[1]$;
$i = 1$;
while $i \leq k$
do $x = \mathit{extract\_min}(H)$;
    $B[i] = x$;
    $i = i + 1$;
    let j be the index such that $x = A[j]$;
    ($*$ comment: assume that j is stored with x
        at the moment when x is inserted into H $*$)
    if $2j \leq n$
    then $\mathit{heap\_insert}(H, A[2j])$
    endif;
    if $2j + 1 \leq n$
    then $\mathit{heap\_insert}(H, A[2j + 1])$
    endif
endwhile

To estimate the running time, we first find out how many elements the min-heap H will ever contain. Just before the first iteration, H contains only the element $A[1]$. During one iteration, the algorithm deletes one element from H and inserts at most two elements into H. Hence, during one iteration, the number of elements in H increases by at most one. Since there are k iterations, it follows that H never contains more than $k + 1$ elements.

We have seen in class that each of the operations *extract_min* and *heap_insert* takes time proportional to the logarithm of the number of elements stored in the heap. Hence, one iteration of the while-loop takes $O(\log k)$ time. Since there are k iterations, the total running time is $O(k \log k)$.

Question 6: Let $G = (V, E)$ be an undirected graph, which is given to you in the adjacency list format. Recall that $degree(u)$ denotes the degree of vertex u. Let $twodegree(u)$ be the sum of the degrees of u's neighbors, i.e.,

$$twodegree(u) = \sum_{v:\{u,v\} \in E} degree(v).$$

Describe an algorithm that computes $twodegree(u)$ for all vertices u, in $O(|V| + |E|)$ total time. Justify your answer.

Solution: We assume that the graph is stored using adjacency lists. Observe that the degree of a vertex is equal to the size of its adjacency list.

Step 1: For each vertex v, scan its adjacency list and count the number of vertices it contains. This number is equal to $degree(v)$. Store this number with vertex v.

Step 2: For each vertex u, scan its adjacency list and compute the sum of all values $degree(v)$, for all v in u's adjacency list. The resulting sum is equal to $twodegree(u)$.

Both Steps 1 and 2 take time proportional to $|V|$ plus the total size of all adjacency lists. Thus, the total running time is

$$O\left(|V| + \sum_{v \in V} degree(v)\right).$$

Since $\sum_{v \in V} degree(v) = 2|E|$, the total running time is $O(|V| + |E|)$.

Question 7: Let $G = (V, E)$ be a directed acyclic graph. In class, we have seen the following algorithm for topologically sorting the vertices of G:

- Set $k = 1$.
- While the graph is non-empty:
  - Find a vertex v of indegree zero.
  - Assign the number k to v.
  - Remove v from the graph.
  - Increase k by one.

Show how this algorithm can be implemented such that its running time is $O(|V| + |E|)$.

Solution: We assume that the graph is stored using adjacency lists. The algorithm is as follows:

Step 1:

- For each vertex u, initialize a variable $D(u) = 0$.

- For each vertex v, scan its adjacency list and, for each vertex u in this list, set $D(u) = D(u) + 1$.
- Comment: At this moment, for each vertex u, $D(u)$ is equal to $\mathit{indegree}(u)$.

Step 2: Initialize a list L containing all vertices v with $D(v) = 0$.

Step 3: Initialize $k = 1$ and $m = |V|$. (Comment: m is equal to the number of vertices that do not have a number yet.)

Step 4: While $m > 0$, do the following:

- Take the first vertex v in the list L.
- Delete v from L.
- Assign the number k to v.
- Increase k by one.
- Decrease m by one.
- For each edge (v, w) in E (thus, for each edge leaving v):
  - Decrease $D(w)$ by one.
  - If $D(w) = 0$: add w to the list L.

Step 1 takes $O(|V| + |E|)$ time. Step 2 takes $O(|V|)$ time. Step 3 takes $O(1)$ time. In the while-loop of Step 4, each vertex v is taken exactly once. The time spent when v is taken is $O(1 + \mathit{indegree}(v))$. Thus, the total running time is

$$O(|V| + |E|) + O(|V|) + O(1) + \sum_{v \in V} (1 + \mathit{indegree}(v)).$$

Since $\sum_{v \in V} \mathit{indegree}(v) = |E|$, the total running time is $O(|V| + |E|)$.