DSA - Seminar 5

---

1. Consider the following problem: Determine the sum of the largest *k* elements from a vector containing *n* distinct numbers. For example, if the array contains the following 10 elements [6, 12, 9, 91, 3, 5, 25, 81, 11, 23] and k = 3, the result should be: 91 + 81 + 25 = 197.

   I. Find the maximum k times (especially good if k is small)
   - If we just call the maximum function 3 times for our example, it will return 91 each time, so we need a solution where we also have an upper bound, and we are searching for the maximum which is less than that value.
   - First, maximum is 91. At the second call we want the maximum which is less than 91, we will get 81.
   - At the third call we want the maximum which is less than 81, we will get 25.
   - Complexity of the approach: $\Theta(k*n)$ – finding the maximum is $\Theta(n)$ and we do this k times.

   II. Similarly, we could do just k iterations of some sorting algorithms:
   - Selection sort (with descending sorting) finds the maximum and moves it to position 1. Then it finds the maximum of the remaining array and moves it to the second positions, etc. After k iterations the first k elements will be the ones we need to add together.
   - Bubble sort (with ascending sorting) – one iteration of bubble sort will move the maximum element to the end of the array. The next iteration will guarantee that the element on position n-1 is the second largest, etc. After k iterations, the last k elements will be the ones we need to sum up.
   - Complexity for both cases is $\Theta(k*n)$

   III. Sort the array in a descending order and pick the first k elements (especially good is k is large).
   - Sorting can be done in $\Theta(n*\log_2 n)$ time
   - Computing the sum of the first k elements: $\Theta(k)$
   - In total $\Theta(n*\log_2 n) + \Theta(k) \in \Theta(n*\log_2 n)$

   IV. Keep a sorted array of the k largest elements found so far. Initially this array will contain the first k elements (in sorted order). For the next elements of the input array, we compare the element with the minimum of the k selected numbers and if necessary remove the minimum and add the new element. For our example:
   - Initially the array contains [12, 9, 6] (I chose to keep them in descending order, but it would be the same with ascending ordering).
   - We process 91, it is greater than 6 (the minimum of the selected k elements), so we remove 6 and add 91. Our array of selected elements will be: [91, 12, 9]

- We process 3, it is less than 9 (the minimum of the selected k elements), so we do not modify the array
- We process 5, it is less than 9 (the minimum of the selected k elements), so we do not modify the array
- We process 25, it is greater than 9 (the minimum of the selected k elements) so we remove 9 and add 25. Our array of selected elements will be: [91, 25, 12].
- Etc.
- At the end, we need to compute the sum of the elements in k (or we can keep a variable that gets updated while we process the elements).
- Complexity:
    i. Processing one element can have a best case (element is less than the minimum, we just do a comparison: $\Theta(1)$) or a worst case (element is a new maximum, we need to move every element in the array one position to the right: $\Theta(k)$) => $O(k)$.
    ii. We need to process every element => $O(n*k)$

V. Use a binary max-heap. Add all the elements to the heap and remove the first k.

- Adding an element to a heap with n elements is $O(\log_2 n)$.
- Removing an element from a heap with n elements is $O(\log_2 n)$.
- In total we have $O(n*\log_2 n) + O(k*\log_2 n)$. Since n >= k, this is $O(n*\log_2 n)$

```
function sumOfK(elems, n, k) is:
//elems is an array of unique integer numbers
//n is the number of elements from elems
//k is the number of elements we want to sum up. Assume k <= n
        init(h, "≥") //assume we have the Heap data structure implemented. Weinitialize a heap with the relation "≥" (a max-heap)
        for i ← 1, n execute
                add(h, elems[i]) //add operation was discussed at Lecture 8
        end-for
        sum ← 0
        for i ← 1, k execute
                elem ← remove(h) //remove operation was discussed at Lecture 8
                sum ← sum + elem
        end-for
        sumOfK ← sum
end-function
```

VI. How can we reduce the complexity? Well, we do not need all the elements in the heap, we are always interested in the k largest ones (similar to solution IV, but instead of sorted array of k elements, we will have heap of k elements). If we consider the example from above, we can work in the following way, always keeping just the k maximum elements up until now:

- Initially we keep 6, 12, 9
- When we get to 91, we can drop 6, because we know for sure that it is not going to be part of the 3 maximum numbers (we already have 3 numbers greater than this). So we keep 12, 91, 9.

- When we get to 3, we know it is not going to be part of the 3 maximum elements (We already have 3 elements greater than that). Similar with 5.
- When we get to 25, we can drop 9, and go on with 12, 91, 25.
- Etc.

We can keep the k elements that we consider in a heap. Should it be a min-heap or max-heap?

When we have the k largest elements at a given point, we will be interested in the minimum of these elements (this is what we compare to the new element that is considered and this is what we remove if we find a larger one) so it should be a min-heap.

```
function sumOfK2(elems, n, k) is:
//elems is an array of unique integer numbers
//n is the number of elements from elems
//k is the number of elements we want to sum up. Assume k <= n
        init(h, "≤") //assume we have the Heap data structure implemented. We
        initialize a heap with the relation "≤" (a min-heap)
        for i ← 1, k execute //the first k elements are added "by default"
                add(h, elems[i])
        end-for
        for i ← k+1, n execute
                if elems[i] > getFirst(h) then //getFirst is an operation which returnsthe first element from the heap.
                        remove(h)//it returns the removed element, but we do not need itadd(h, elems[i])
                end-if
        sum ← 0
        for i ← 1, k execute
                elem ← remove(h) //remove operation was discussed at Lecture 8sum ← sum + elem
        end-for
        sumOfK2 ← sum
end-function
```

- Complexity? Our heap has maximum k elements, so operations have a complexity of $O(\log_2 k)$. We call add at most n times (worst case, when every element is greater than the root of the heap) and remove at most n times. So in total we have $O(n*\log_2 k)$
- If you do not use an already implemented heap, but have access to the representation, you can make the previous implementation slightly more efficient (complexity will not change though):
  - the last *for* will not have to remove elements, just simply to add up the sum of the elements from the heap array (but for this you need access to the array) – You can eliminate that for even if you don't have access to the representation, if you keep a variable with the sum so far: whenever you add to the heap, you add to the sum, whenever you remove from the heap you subtract from the sum.
  - the middle for loop will not have to do a remove and an add. You can just overwrite the element from position 1 (this is what would be removed anyway) with the newly added element and do a bubble-down on it.

VII. Can we improve complexity even more? We know that if we have an array we can transform it into a heap in a complexity O(n) – it was discussed at heapsort. So we can

do something similar to version V, but instead of adding the elements one by one to the heap we could transform the array into a max-heap and then remove k elements from it. This approach has a complexity of $O(n + k*\log_2 n)$