COMP 3804 — Solutions Tutorial February 2

Problem 1: Some algorithms textbooks have statements of the type

Every comparison-based sorting algorithm takes at least $O(n \log n)$ time.

Does such a statement make sense?

Solution: Even Professor Bieber knows that this does not make any sense: The statement says: For some constant c, every comparison-based sorting algorithm takes at least at most $cn \log n$ time. It is like saying that every beer bottle costs at last at most \$100.

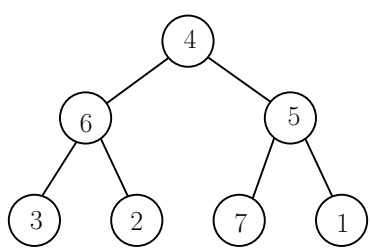
Problem 2: Let $A[1 \ldots n]$ be an array storing n numbers. In the January 25 lecture, we have seen algorithm BuildHeap(A) that rearranges the numbers in the input array A such that the resulting array is a max-heap; see page 56 of my handwritten notes. This algorithm uses the Heapify-procedure as a subrouting; see page 53 of my handwritten notes. Consider the following variant of this algorithm:

Algorithm BuildHeap$'(A)$:
for $i = 1$ **to** $\lfloor n/2 \rfloor$
do Heapify(A, i)
endfor

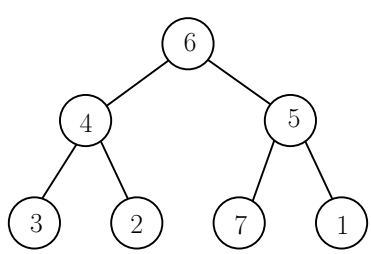
Give an example of an array $A[1 \ldots n]$, where n is a small integer (such as $n = 7$), which shows that algorithm BuildHeap$'$ may not result in a max-heap.

Solution: We take the input array $A[1 \ldots 7] = [4, 6, 5, 3, 2, 7, 1]$. For this case, algorithm BuildHeap$'(A)$ runs, in this order, Heapify$(A, 1)$, Heapify$(A, 2)$, and Heapify$(A, 3)$.

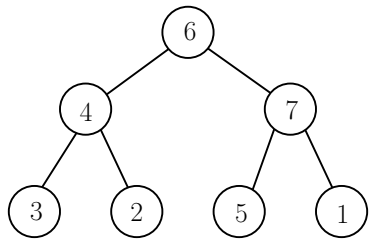
The tree representation of the input array is the following:



The call Heapify$(A, 1)$ results in the following tree:



The call $\textsc{Heapify}(A, 2)$ does not change the tree. The call $\textsc{Heapify}(A, 3)$ results in the following tree:



This is not a max-heap, because element 7 is not at the root.

Problem 3: Let $A[1 \ldots n]$ be an array storing n pairwise distinct numbers, and let k be an integer with $0 \leq k < n$. We say that this array is *k-sorted*, if for each i with $1 \leq i \leq n$, the entry $A[i]$ is at most k positions away from its position in the sorted order.

For example, a sorted array is 0-sorted. As another example, the array

$$A[1 \ldots 10] = [1, 4, 5, 2, 3, 7, 8, 6, 10, 9]$$

is 2-sorted, because each entry $A[i]$ is at most 2 positions away from its position in the sorted order. For $i = 3$, $A[3]$ is 2 positions away from its position, 5, in the sorted array. For $i = 9$, $A[9]$ is 1 position away from its position, 10, in the sorted array.

Describe an algorithm $\textsc{Sort}$ that has the following specification:

Algorithm $\textsc{Sort}(A, k)$**:**
Input: An array $A[1 \ldots n]$ of n pairwise distinct numbers and an integer k with $2 \leq k < n$. This array is k-sorted.
Output: An array $B[1 \ldots n]$ containing the same numbers as the input array. The array B is sorted.
Running time: Must be $O(n \log k)$.

Explain why your algorithm is correct and why the running time is $O(n \log k)$.
Hint: Use a min-heap of a certain size.

Solution: The approach is as follows:

- Let H be the set consisting of the first $k + 1$ elements in the input array $A[1 \ldots n]$.
- Since the input array is k-sorted, the smallest element in the entire array $A[1 \ldots n]$ is the smallest element in the set H. We find the smallest element in H, delete it from H, and store it at $B[1]$.
- We add $A[k + 2]$ to the set H. Since the input array is k-sorted, the second smallest element in the entire array $A[1 \ldots n]$ is the second smallest element in the subarray $A[1 \ldots k+2]$, which is the smallest element in the set H. We find the smallest element in H, delete it from H, and store it at $B[2]$.

- We add $A[k+3]$ to the set H. Since the input array is k-sorted, the third smallest element in the entire array $A[1 \ldots n]$ is the third smallest element in the subarray $A[1 \ldots k+3]$, which is the smallest element in the set H. We find the smallest element in H, delete it from H, and store it at $B[3]$.

- We continue this process until $B[1 \ldots n-k-1]$ stores, in sorted order, the $n-k-1$ smallest element in the input array $A[1 \ldots n]$. At this moment, the set H consists of the $k+1$ largest elements in the input array $A[1 \ldots n]$. We add the elements of H to the subarray $B[n-k \ldots n]$, one by one, from smallest to largest.

- How do we store the set H? We need the operations INSERT and EXTRACTMIN. This suggests that we store H in a min-heap.

```
Algorithm SORT(A, k):
Comment: Array A[1...n] is k-sorted.
Comment: The sorted numbers will be stored in array B[1...n].
initialize an array H[1...k+1];
for i = 1 to k+1
do H[i] = A[i]
endfor;
BUILDHEAP(H);
for i = 1 to n-k-1
do x = EXTRACTMIN(H);
    B[i] = x;
    INSERT(H, A[k+1+i])
endfor;
for i = 1 to k+1
do x = EXTRACTMIN(H);
    B[n-k-1+i] = x
endfor
```

Regarding the running time:

- Initializing the array H takes $O(k)$ time, which is $O(n)$.

- The first for-loop takes $O(k)$ time, which is $O(n)$.

- The call to BUILDHEAP(H) takes $O(k)$ time, which is $O(n)$.

- During the second for-loop, at any moment, the min-heap has size k or $k+1$, because we always delete the smallest element and then insert a new element. Each call to EXTRACTMIN and INSERT takes $O(\log k)$ time. The number of iterations of the second for-loop is $n-k-1$, which is at most n. Thus, the total time for the second for-loop is $O(n \log k)$.

- During the third for-loop, at any moment, the min-heap has size at most $k+1$, because we only delete elements. Each call to ExtractMin takes $O(\log k)$ time. The number of iterations of the third for-loop is $k+1$, which is at most n. Thus, the total time for the third for-loop is $O(n \log k)$.

- We conclude that the total running time is

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