Exercise Session 05

Exercise 1.
Consider the MAX-HEAPIFY procedure (CLRS, pp. 154) as defined below.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4        largest = l
 5  else
 6        largest = i
 7  if r ≤ A.heap-size and A[r] > A[largest]
 8        largest = r
 9  if largest ≠ i
10        exchange A[i] with A[largest]
11        MAX-HEAPIFY(A, largest)
```

Use induction to prove that MAX-HEAPIFY is correct.

Solution 1.
We start by formalising what we have to prove.

> Let $A[1\,..\,n]$ be an array and $i \in \{1, \ldots, n\}$ be and index. If the subtrees rooted respectively at $\textsc{Left}(i)$ and $\textsc{Right}(i)$ are max-heaps (i.e., they satisfy the max-heap property), after executing $\textsc{Max-Heapify}(A, i)$ the subtree rooted at i satisfies the max-heap property.

We prove the above statement by induction on the structure[1] of the binary tree rooted at i.

Base Case. In this case the tree rooted at i is a leaf node, hence it has no subtrees. This means that $\textsc{Left}(i) > \textsc{Right}(i) > A.\textit{heap-size}$. In this case after executing lines 1–8 we have that $\textit{largest} = i$. Therefore $\textsc{Max-Heapify}(A, i)$ leaves the array A unchanged. Since i is a leaf node, the subtree rooted at i trivially satisfies the max-heap property.

Inductive step. In this case we have that i has at least the left subtree. Two cases the possible, either only the left sub-tree exists (i.e., $\textsc{Left}(i) \leq A.\textit{heap-size} < \textsc{Right}(i)$), or both sub-tree exist (i.e., $\textsc{Left}(i) < \textsc{Right}(i) \leq A.\textit{heap-size}$). We will cover only the latter case because the former case can be proven similarly.

In this case after executing lines 1–8, $\textit{largest} \in \{i, \textsc{Left}(i), \textsc{Right}(i)\}$ and

$$A[\textit{largest}] = \max\{A[i], A[\textsc{Left}(i)], A[\textsc{Right}(i)]\}\,. \tag{1}$$

We consider three distinct cases separately.

If $\textit{largest} = i$, then $\textsc{Max-Heapify}(A, i)$ leaves the array A unchanged. We have to check that for all index $j \neq i$ in the subtree rooted at i it holds $A[\textsc{Parent}(j)] \geq A[j]$. By (1), $A[i] \geq A[\textsc{Left}(i)]$ and $A[i] \geq A[\textsc{Right}(i)]$. This is equivalent to write $A[\textsc{Parent}(\textsc{Left}(i))] \geq A[\textsc{Left}(i)]$ and $A[\textsc{Parent}(\textsc{Right}(i))] \geq A[\textsc{Right}(i)]$. By hypothesis, the subtrees rooted respectively at $\textsc{Left}(i)$ and $\textsc{Right}(i)$ satisfy the max-heap property. This ensures that the max-heap condition holds also for all the remaining nodes in the subtree rooted at i.

If $\textit{largest} = \textsc{Left}(i)$, then in line 10 $A[i]$ and $A[\textsc{Left}(i)]$ are exchanged. After the exchange, by (1), we have that $A[i] \geq A[\textsc{Left}(i)]$ and $A[i] \geq A[\textsc{Right}(i)]$. Again, this is equivalent to $A[\textsc{Parent}(\textsc{Left}(i))] \geq A[\textsc{Left}(i)]$ and $A[\textsc{Parent}(\textsc{Right}(i))] \geq A[\textsc{Right}(i)]$. The subtree rooted at $\textsc{Right}(i)$ is left untouched and, by hypothesis it satisfies the max-heap property. As

[1] See these slides to recall the concept of structural induction.

for the left subtree, we have the value of the root has changed (after execution of line 10), but all other nodes behind it have not changed. Thus, the subtrees rooted at LEFT(LEFT(i)) and RIGHT(LEFT(i)), if they exist, they satisfy the max-heap property. Therefore, by inductive hypothesis after executing line 11 the max-heap property is restored on the subtree rooted at LEFT(i). As argued before, this ensures that the max-heap condition holds also for all the remaining nodes in the subtree rooted at i.

If *largest* = RIGHT(i) we can proceed similarly to the previous case.

This concludes the proof of correctness of the MAX-HEAPIFY procedure.

Exercise 2.
Starting with the procedure MAX-HEAPIFY (CLRS, pp. 154), write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

Solution 2.
The procedure for MIN-HEAPIFY(A, i) assumes that the children of i are min-heaps and works similarly to MAX-HEAPIFY by pushing down the value $A[i]$ exchanging it with the smallest element among $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$. The pseudocode for MIN-HEAPIFY is described below.

```
MIN-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] < A[i]
 4        smallest = l
 5  else
 6        smallest = i
 7  if r ≤ A.heap-size and A[r] < A[smallest]
 8        smallest = r
 9  if smallest ≠ i
10        exchange A[i] with A[smallest]
11        MIN-HEAPIFY(A, smallest)
```

The worst-case running time of MIN-HEAPIFY work can be proved to be $O(\lg n)$ by following the same arguments used for MAX-HEAPIFY.

Exercise 3.
Consider the pseudocode of the PARTITION procedure.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Assume that all elements in the array $A[p\,..\,r]$ are equal, that is, $A[p] = A[p+1] = \cdots = A[r]$. What value will PARTITION(A, p, r) return? How does QUICKSORT perform on arrays that have the same value compared with INSERTION-SORT and MERGESORT?

Solution 3.
Note that if $A[p\,..\,r]$ have the same value, the condition in line 4 is always true and lines 5–6 are executed. Therefore at the end of the for loop the value of i will be $p-1+r-p=r-1$, and the value returned by PARTITION(A,p,r) will be r.
As argued above, if the array A has the same value, procedure PARTITION produces unbalanced partitionings at each recursive call of QUICKSORT causing the algorithm to perform in time $\Theta(n^2)$ (i.e., the worst-case running time). In contrast, INSERTION-SORT performs in time $\Theta(n)$ when the input array is already sorted (i.e., the best-case running time). Hence, for arrays that have the same value insertion sort performs much better than quicksort. The same argument can be trivially generalised for arrays that are already in increasing order.
As for MERGESORT its complexity is not affected the relative order of the elements in A. Therefore it will still run on $\Theta(n \lg n)$.

Exercise 4.
Modify the pseudocode of the PARTITION procedure so that the QUICKSORT algorithm (CLRS, pp. 171) will sort in nonincreasing order. Argument about the correctness of your solution.

Solution 4.
It suffices to modify the partition procedure by replacing in line 4 the condition $A[j] \leq x$ with $A[j] \geq x$. This modification makes sure that all elements in the left partition are greater than or equal to the pivot element x and (dually) all elements that are smaller than x will fall in the right partition. The modified pseudocode for PARTITION is

PARTITION(A,p,r)
1 $x = A[r]$
2 $i = p-1$
3 **for** $j = p$ **to** $r-1$
4 **if** $A[j] \geq x$
5 $i = i+1$
6 exchange $A[i]$ with $A[j]$
7 exchange $A[i+1]$ with $A[r]$
8 **return** $i+1$

Exercise 5.
Consider the pseudocode of COUNTING-SORT (CLRS, pp. 195)

COUNTING-SORT(A,B,k)
1 let $C[0\,..\,k]$ be a new array
2 **for** $j-1$ **to** k
3 $C[i] = 0$
4 **for** $j = 1$ **to** $A.length$
5 $C[A[j]] = C[A[j]] + 1$
6 **for** $i = 1$ **to** k
7 $C[i] = C[i] + C[i-1]$
8 **for** $j = A.length$ **downto** 1
9 $B[C[A[j]]] = A[j]$
10 $C[A[j]] = C[A[j]] - 1$

Modify the above pseudocode by replacing the **for**-loop header in line 8 as

8 **for** $j = 1$ **to** $A.length$

Is the modified algorithm stable?

Solution 5.
The modified algorithm is not stable. As for the original COUNTING-SORT algorithm, in the final **for** loop an element equal to one taken from A earlier is placed before the earlier one (i.e., at a lower index position) in the output array B. The original algorithm was stable because an element taken from A later started out with a lower index than one taken earlier. But in the modified algorithm, an element taken from A later started out with a higher index than one taken earlier.
In particular, the algorithm still places the elements with value k in positions $C[k-1]+1$ through $C[k]$, but in the reverse order of their appearance in A.

★ **Exercise 6.**
Use induction to prove that RADIX-SORT is correct. Where does your proof need the assumption that the intermediate sorting procedure is stable? Justify your answer.

Solution 6.
We proceed by induction on the number of digits d.

Base Case ($d = 1$). there is only one digit, so sorting on that digit sorts the array.

Inductive step ($d > 1$). Assuming that radix sort works for $d-1$ digits, we'll show that it works for d digits. Radix sort sorts separately on each digit, starting from digit 1. Thus, radix sort of d digits, which sorts on digits $1, \ldots, d$ is equivalent to radix sort of the low-order $d-1$ digits followed by a sort on digit d. By our induction hypothesis, the sort of the low-order $d-1$ digits works, so just before the sort on digit d, the elements are in order according to their low-order $d-1$ digits. The sort on digit d will order the elements by their d-th digit. Consider two elements, a and b, with d-th digits a_d and b_d respectively.

- If $a_d < b_d$, the sort will put a before b, which is correct, since $a < b$ regardless of the low-order digits.
- If $a_d > b_d$, the sort will put a after b, which is correct, since $a > b$ regardless of the low-order digits.
- If $a_d = b_d$, the sort will leave a and b in the same order they were in, because it is stable. But that order is already correct, since the correct order of a and b is determined by the low-order $d-1$ digits when their d-th digits are equal, and the elements are already sorted by their low-order $d-1$ digits.

If the intermediate sort were not stable, it might rearrange elements whose d-th digits were equal –elements that were in the right order after the sort on their lower-order digits.

Exercise 7.
Assume to use QUICKSORT as the sorting subroutine for RADIX-SORT. Will the resulting procedure be correct? Justify your answer.

Solution 7.
To be correct RADIX-SORT requires to use a *stable* sorting subroutine. Unfortunately, QUICKSORT is not a stable sorting algorithm. This is because the exchange of values performed in line 6 of the PARTITION procedure may change the relative order of two occurrences of the same value within the array A. Specifically, consider the array $A = [3', 3'', 3''', 0, 1]$ where the the three occurrences of the value 3 are marked as $3'$ and $3''$, and $3'''$ respectively. The execution of QUICKSORT$(A, 1, A.length)$ will reorder the elements as $A = [0, 1, 3''', 3', 3'']$.