Exercise Session 05

Exercise 1.
Consider the MAX-HEAPIFY procedure as defined below.

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
      largest = l
else
      largest = i
if r ≤ A.heap-size and A[r] > A[largest]
      largest = r
if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest)
```

Use induction to prove that MAX-HEAPIFY is correct.

Base case: A = 1 element, i = 1 (l, r and largest is undefined, but for practice it isn't undefined)

$$\begin{aligned}
l &= Left(i) = 2i = 2 \\
r &= Right(i) = 2i + 1 = 3 \\
if(l \leq A.heap - size) &= if(2 \leq 1) \\
largest &= 1 \\
if(r \leq A.heap - size) &= if(3 \leq 1) \\
if(largest \neq i) &= if(1 \neq 1) \\
A[1] &= A[i] = A[largest]
\end{aligned}$$

Thus fulfilling the max heapify property for a single element proving the base case correct, because the max-heapify property trivially fulfilled for a single element as it has no leaves.

Inductive Hypothesis: Assume that Max-Heafity correctly fulfils the Max-Heapify property for elements p

Inductive step Assuming that our Max-Heapify property is fulfilled for p elements we need to prove that introducing an additional element will still correctly fulfils the Max-Heapify property. Assume that we are at the parent-node p we need to show that the properties are still satisfied in the case that the child-node c is larger or less that the current p.
Case 1 - c is larger than p, and we know that A has $\geq$ that 2i elements

$$\begin{aligned}
c &= A[l] \\
p &= A[i] \\
l &= 2i \\
r &= undefined \\
if(l \leq A.heapsize \& A[l] > A[I]) &= if(2i \leq A.heapsize \& c > p) true \\
largest &= l \\
if(largest \neq i)& \\
Swap(A[i], A[largest] &= Swap(A[i], A[l]) = Swap(p, c)
\end{aligned}$$

Case 2 - p is larger than c, and we know that A has $\geq$ than 2i elements

$$\begin{aligned}
c &= A[l] \\
p &= A[i] \\
l &= 2i \\
r &= undefined \\
if(l \leq A.heapsize \& A[l] > A[I]) &= if(2i \leq A.heapsize \& c > p) false \\
largest &= i \\
if(largest &\neq i) false
\end{aligned}$$

Exercise 2.
Starting with the procedure MAX-HEAPIFY, 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?

```
MIN-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] < A[i]
    smallest = l
else
    smallest = i
if r ≤ A.heap-size and A[r] < A[smallest]
    smallest = r
if smallest ≠ i
    exchange A[i] with A[smallest]
    MIN-HEAPIFY(A, smallest)
```

Since it has the same amount of operations it has the same running time of $O(log(n))$

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?

Answer: PARTITION(A, p, r) will return $i = r$

How does QUICKSORT perform on arrays that have the same value compared with INSERTION-SORT and MERGESORT?

Quicksort has $\Theta(n^2)$ for arrays where all elements are the same. Mergesort does as always take $\Theta(n \cdot log(n))$. Insertion sort normally has the worst running time of the three algorithms but in the case that all elements are the same it has a time complexity of just $\Theta(n)$ as the while loop is instantly terminated when comparing each element.

Exercise 4.

Modify the pseudocode of the PARTITION procedure so that the QUICKSORT algorithm will sort in nonincreasing order. Argument about the correctness of your solution.

```
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
```

By modifying the if statment on line 4 of the Partition algorithm, each recursive call of partition splits the current array into to subarrays where $A[p, ..., i] \geq x \geq A[i+1, .., j]$.

Exercise 5.

Consider the pseudocode of Counting-Sort

Counting-Sort(A, B, k)

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

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 correct? Is it also stable? Justify your answers (not necessarily by providing a formal proof).

Lines 8-10 will produce the correct B (sorted) array because. However the algorithm is no longer stable as the position of the C array elements are no longer at the correct indexes, even though the B array is correctly sorted. position

★ **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.

Exercise 7.

Assume to use Quicksort as the sorting subroutine for Radix-Sort. Will the resulting procedure be correct? Justify your answer.

Radix-Sort requires a stable sorting sub-routine which has to be a stable sorting algorithm. Quicksort is not a stable sorting algorithm and Radix-Sort will therefor not produce the correct output.