1. Argue that the worst-case time complexity of counting sort is $O(n+k)$, with $n$ the length of the input and $k$ indicating the range.

The algorithm of counting sort consists of four for-loops. The time complexity of the first for-loop is in $\Theta(k)$. The time complexity of the second for-loop is in $\Theta(n)$. The time complexity of the third for-loop is in $\Theta(k)$. The time complexity of the fourth for-loop is in $\Theta(n)$.

Hence the time complexity of counting sort is in $\Theta(n+k)$. If $k$ is in $O(n)$, then $\Theta(n+k)$ is $\Theta(n)$ so then counting sort is linear in the number of elements of the input-sequence.

We apply counting sort to the input sequence $A = [0, 3, 2, 0, 3, 5]$. First we allocate the counting array $C$ with indices $0, \ldots, 5$. The first loop yields the following:

$$C = [2, 0, 1, 2, 0, 1]$$

Then, in order to reserve space in the output array, we compute the number of occurrences of elements $\leq i$ for $i = 0, \ldots, 5$:

$$C = [2, 2, 3, 5, 5, 6]$$

Next we traverse the input array $A$ from right to left, and look for the value in $C$ at index $A[j]$ and put that value there in the output array $B$, and decrease the count in $C$. First step:

- $B = [?, ?, ?, ?, ?, 5]$
- $C = [2, 2, 3, 5, 5, 5]$
Further iterations:


$B = [?, ?, ?, ?, 3, 5]$

$C = [2, 2, 3, 4, 5, 5]$

consider $A[4] = 0$

$B = [?, 0, ?, ?, ?, 3, 5]$

$C = [1, 2, 3, 4, 5, 5]$


$B = [?, 0, 2, ?, 3, 5]$

$C = [1, 2, 2, 4, 5, 5]$


$B = [?, 0, 2, 3, 3, 5]$

$C = [1, 2, 2, 3, 5, 5]$

consider $A[1] = 0$

$B = [0, 0, 2, 3, 3, 5]$

$C = [0, 2, 2, 3, 5, 5]$

2. Apply bucket sort on the following input using an array of length 10:

$(7, a), (1, b), (1, c), (6, d), (3, e), (2, f), (8, g), (5, h), (7, i), (4, a)$

Step 1: we make a new bucket array $B$ of length 10,

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[(1, b), (1, c)]</td>
<td>(2, f)</td>
<td>(3, e)</td>
<td>(4, a)</td>
<td>(5, h)</td>
<td>(6, d)</td>
<td>[(7, a), (7, i)]</td>
<td>(8, g)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 2: We use insertion sort to sort every bucket. (Nothing to be done.)

Step 3: We construct the output by concatenating the lists together in order. Result:

$[(1, b), (1, c), (2, f), (3, e), (4, a), (5, h), (6, d), (7, a), (7, i), (8, g)]$

3. Apply radix sort to the following sequence:

22 57 7 66 100 36 44 15 43 10 3 9 13 4 29 25

Show step by step how the sequence is sorted.
4. Argue the correctness of radix sort.

Solution:

We proceed by induction on \( d \) which is the length of the sequences that are sorted. We assume that the sorting algorithm used in radix sort is correct and stable, and we denote that algorithm by \( C \).

Base case: \( d = 1 \). Then radix sort is correct using correctness of \( C \).

Induction step: we assume that radix sort is correct for \( d - 1 \). We show that radix sort is correct for \( d \). The iterations of the for-loop for \( i = 1 \) up to \( i = d - 1 \) yield that the \( d - 1 \) low-order digits are sorted, using (a consequence of) the induction hypothesis.

After these iterations we have a sequence say \( s \) that is the starting point for the for-loop for \( i = d \). We consider two elements \( a \) and \( b \) in \( s \), and we write \( a_d \) and \( b_d \) for their digit at dimension \( d \) (so the most significant position). If \( a_d \neq b_d \) then without loss of generality \( a_d < b_d \), and iteration with \( i = d \) of the for-loop puts \( a \) before \( b \) which is correct because according to the definition of the lexicographic order indeed \( a < b \). If \( a_d = b_d \) then the algorithm \( C \) will keep the order of \( a \) and \( b \) as in \( s \), because \( C \) is stable. This is correct, because by the induction hypothesis, \( a \) and \( b \) are in the right order for digits 1 up to \( d - 1 \), and because \( a_d = b_d \) they are also in the right order for digits 1 up to \( d \).

We use both correctness and stability of \( C \).

5. Show how to sort \( n \) integers in the range 0 to \( n^3 - 1 \) in \( O(n) \).

We can represent the numbers \( 0, \ldots, n^3 - 1 \) in base \( n \) with using three digits per numbers.
Example for \( n = 2 \): we represent \( 0, 1, 2, 3, 4, 5, 6, 7 \) in base 2:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>001</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
</tr>
<tr>
<td>10</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>111</td>
</tr>
</tbody>
</table>

Example for \( n = 3 \): we represent \( 0, 1, \ldots, 26 \) in base 3:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>001</td>
<td>002</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>012</td>
</tr>
<tr>
<td>02</td>
<td>021</td>
<td>022</td>
</tr>
<tr>
<td>10</td>
<td>101</td>
<td>102</td>
</tr>
<tr>
<td>11</td>
<td>111</td>
<td>112</td>
</tr>
<tr>
<td>12</td>
<td>121</td>
<td>122</td>
</tr>
<tr>
<td>20</td>
<td>201</td>
<td>202</td>
</tr>
<tr>
<td>21</td>
<td>211</td>
<td>212</td>
</tr>
<tr>
<td>22</td>
<td>221</td>
<td>222</td>
</tr>
</tbody>
</table>

Next we sort the numbers \( 0, \ldots, n^3 - 1 \) represented in base \( n \) using radix sort with dimension 3. If \( n \) is much larger than 3, the worst-case time complexity is in \( \mathcal{O}(n) \).

6. Explain why the worst-case running time for bucket sort is in \( \Theta(n^2) \).

How can the algorithm be adapted to one with worst-case running time in \( \mathcal{O}(n \log n) \), while keeping linear running time in the average case?

Solution:
We consider the pseudo-code for bucket sort as in the book (p.201). The first for-loop, at lines 3-4, performs \( n \) times an elementary operation, so is in \( \mathcal{O}(n) \). The same holds for the second for-loop, at lines 5-6 (assuming that insertion takes constant time). The last for-loop calls insertion sort at every index. In case that there is a index \( i \) such that \( B[i] \) contains approximately all \( n \) elements of the input-array \( A \), then the other indices are approximately empty, and the complete for-loop takes the worst-case time complexity of insertion sort for input size \( n \), so \( \mathcal{O}(n^2) \).

A simple improvement is obtained by using instead of insertion sort merge sort which is in \( \mathcal{O}(n \cdot \log n) \).

7. Perform the following sequence of operations on an initially empty stack. Give the contents of the stack after every operation.

\[
push(S, 5), push(S, 3), pop(S), push(S, 2), push(S, 8), pop(S), pop(S), push(S, 9), push(S, 1)
\]
Solution:
The stacks look as follows:

\[
\begin{array}{cccccccc}
5 & 3 & 8 \\
5 & 5 & 3 & 2 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
\end{array}
\]

8. Perform the following sequence of operations on an initially empty queue \( Q \). Give the contents of the queue after every operation.

\[
\text{enqueue}(Q, 5), \text{enqueue}(Q, 3), \text{dequeue}(Q), \text{enqueue}(Q, 2), \\
\text{enqueue}(Q, 8), \text{dequeue}(Q), \text{dequeue}(Q), \text{enqueue}(Q, 9), \\
\text{enqueue}(Q, 1)
\]

Solution:
The queues look as follows, with right the front (oldest elements), and left the back (youngest) in the queue. Then:

\[
\begin{array}{cccccc}
5 & 3 \\
3 & 5 \\
3 \\
2 & 3 \\
8 & 2 & 3 \\
8 \\
8 \\
9 & 8 \\
1 & 9 & 8 \\
\end{array}
\]

9. Provide pseudo-code descriptions for the operations \( \text{size}(Q) \), \( \text{isEmpty}(Q) \), and \( \text{head}(Q) \) of the abstract data type for queues. We assume \( n = Q.length \) to be the size of the array.

Solution:

\begin{algorithm}[H]
\textbf{Algorithm \text{size}(Q)}: \\
\hspace{1cm} \text{if } Q.head = Q.tail \text{ then} \\
\hspace{2cm} \text{return } 0 \\
\hspace{1cm} \text{if } Q.head < Q.tail \text{ then} \\
\hspace{2cm} \text{return } Q.tail - Q.head \\
\hspace{1cm} \text{if } Q.tail < Q.head \text{ then} \\
\hspace{2cm} \text{return } n - Q.head + Q.tail
\end{algorithm}
Algorithm isEmpty(Q):
    return Q.head = Q.tail

Algorithm head(Q):
    return Q[Q.head]

10. Assume we have two stacks at our disposal, with the operations as specified in the abstract data type for stacks. Use them to implement the operations enqueue and dequeue of the abstract data type for queues.
    Do it in such a way that the operation enqueue is in $O(1)$, and the operations dequeue is on average in $O(1)$.
    Solution:
    We assume two stacks $S_1$ and $S_2$ with operations push and pop in $O(1)$. We implement the operation enqueue of a queue $Q$ as follows:

    Algorithm enqueue(Q, e):
        push($S_1$, e)

    This operation is in $O(1)$.
    Idea for dequeue: If $S_2$ non-empty, then pop from $S_2$. If $S_2$ empty, then pop every element from $S_1$ and push on $S_2$; next pop from $S_2$. The following is pseudo-code for dequeue of a queue $Q$:

    Algorithm dequeue(Q):
        if isEmpty($S_2$) then
            while not isEmpty($S_1$) do
                push($S_2$, pop($S_1$))
            return pop($S_2$)

    This operation is in $O(1)$ on average. Every element which is added and removed enters and leaves (once) $S_1$, and enters and leaves (once) $S_2$.

11. Assume we have two queues at our disposal, with the operations as specified in the abstract data type for queues. Use them to implement the operations push and pop of the abstract data type for stacks.
Analyze the time complexity of push and pop in terms of $O$.

Solution:

We assume two queues $Q_1$ and $Q_2$ with operations enqueue and dequeue in $O(1)$. We implement the operation push as follows:

**Algorithm push(S, e):**

```
enqueue(Q_1, e)
```

This operation is in $O(1)$.

Idea for pop: If $Q_1$ non-empty, then dequeue every non-head element of $Q_1$, and enqueue them on $Q_2$; next dequeue from $Q_1$. If $Q_1$ empty, then dequeue every non-head element from $Q_2$, and enqueue them on $Q_1$; next dequeue $Q_2$.

Pseudo-code for pop:

**Algorithm pop(S):**

```
x := dequeue(Q_1)
while not isEmpty(Q_1) do
    enqueue(Q_2, x)
    x := dequeue(Q_1)
while not isEmpty(Q_2) do
    enqueue(Q_1, dequeue(Q_2))
return x
```

This operation is in $O(n)$.

12. Explain how to implement two stacks in one array $A[1 \ldots n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is $n$. The operations push and pop should run in $O(1)$ time.

Solution:

Assume an array of length $n$. Idea: one stack starts at array index 0, and continues on consecutive increasing indices. The other stack starts at array index $n$, and continues on consecutive decreasing indices.

For the first stack we use a variable $t_1$ which initially has value 0. For the second stack we use a variable $t_2$ which initially has value $n + 1$.

\[
\text{size}_1 = t_1 \\
\text{size}_2 = n - t_2 + 1 \\
\text{totalsize} = \text{size}_1 + \text{size}_2
\]
13. Adapt the procedures enqueue and dequeue from the book such that they also deal with overflow and underflow.

Solution:

(Indication.) For enqueue, add

\[
\text{if } (Q.\text{head} = Q.\text{tail} + 1) \text{ or } (Q.\text{head} = 1 \text{ and } Q.\text{tail} = Q.\text{length}) \text{ then}
\]
\[
\text{throw FullQueueException}
\]

For dequeue, add

\[
\text{if } (Q.\text{head} = Q.\text{tail}) \text{ then}
\]
\[
\text{throw EmptyQueueException}
\]

14. The Fibonacci numbers are given by the following recurrence: \( F_0 = 0, \)
\( F_1 = 1, \text{ en } F_i = F_{i-1} + F_{i-2} \text{ voor } i \geq 2. \)

Give pseudo-code for the algorithm that naively computes the Fibonacci numbers. Give an analysis of the running time of your algorithm using both a recursion tree and a recurrence equation.

No solution.

15. Read about the adaptation of quickSort to search for the kth smallest element in an array, for instance at the wiki-page for quickSelect.

Write your own version of the pseudo-code for quickSelect that takes as input an array \( A, \) indices \( l \) and \( r, \) and returns a smallest element of \( A[l \ldots r]. \)

No solution.