1. Show the following in some detail:

(a) \(5n^2 + 3n \log n + 2n + 5 \in \mathcal{O}(n^2)\)
    for \(n \geq 2\) we have \(5n^2 + 3n \log n + 2n + 5 \leq (5 + 3 + 2 + 5)n^2\), so take \(c = 15\).

(b) \(20n^3 + 10n \log n + 5 \in \mathcal{O}(n^3)\)
    for \(n \geq 1\) we have \(20n^3 + 10n \log n + 5 \leq 35n^2\), so take \(c = 35\).

(c) \(3 \log n + 2 \in \mathcal{O}(\log n)\)
    for \(n \geq 2\) we have \(3 \log n + 2 \leq 5 \log n\) (nb: \(\log 1 = 0\)).

(d) \(2^{n+2} \in \mathcal{O}(2^n)\).
    for \(n \geq 1\) we have \(2^{n+2} = 2^n \cdot 2^2 = 4 \cdot 2^n\), so take \(c = 4\).

(e) \(2n + 100 \log n \in \mathcal{O}(n)\).
    for \(n \geq 2\) we have \(2n + 100 \log n \leq 102n\), so take \(c = 102\).

2. Give a concrete example of \(f_1\) and \(f_2\) such that \(f_1(n) \in \mathcal{O}(g_1(n))\) and \(f_2(n) \in \mathcal{O}(g_2(n))\) but \(f_1(n) - f_2(n) \notin \mathcal{O}(g_1(n) - g_2(n))\).

A possible solution: \(n^2 + n \in \mathcal{O}(n^2)\) and \(n^2 \in \mathcal{O}(n^2)\). However, \(n^2 + n - n^2 = n \notin \mathcal{O}(n^2 - n^2) = \mathcal{O}(0)\).

3. Describe a recursive algorithm for finding the maximum element in an array of \(n\) elements. Analyse the worst-case time complexity of your algorithm.

Solution:

\begin{verbatim}
Algorithm arrayMax(A, n):
    if n = 0 then
        return error
    if n = 1 then
        return A[1]
    if n > 1 then
        m := arrayMax(A, n - 1)
        if m > A[n] then
            return m
        else
            return A[n]
\end{verbatim}
A function for the worst-case time complexity of this algorithm can be given via the following recurrence equation:

\[
T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T(n - 1) + 1 & \text{if } n > 1 
\end{cases}
\]

We use the substitution method to solve the recurrence equation. For large \(n\):

\[
T(n) = T(n - 1) + 1 = T(n - 2) + 1 + 1 = T(n - 3) + 3 = \ldots = T(n - i) + i
\]

We substitute \(i = n - 1\). Then we have:

\[
T(n) = 1 + n - 1 = n
\]

So the worst-case time complexity of \texttt{arrayMax} is in \(\mathcal{O}(n)\).

4. Do we have \(2^{n+1} \in \mathcal{O}(2^n)\)? Do we have \(2^n \in \mathcal{O}(2^n)\)?

Solution:

We have \(2^{n+1} \in \mathcal{O}(2^n)\). We show: there exists \(c\) and there exists \(n_0\) such that \(2^{n+1} \leq c \cdot 2^n\) for all \(n \geq n_0\). Take \(c = 2\) and \(n_0 = 1\); indeed we have \(2^{n+1} \leq 2 \cdot 2^n\) for all \(n \geq 1\).

We do not have \(2^n \in \mathcal{O}(2^n)\). Suppose \(2^{2n} \in \mathcal{O}(2^n)\). There exists \(c\), and there exists \(n_0\) such that \(2^{2n} \leq c \cdot 2^n\) for all \(n \geq n_0\). That means \(2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n\) so \(2^n \leq c\). Such a constant \(c\) does not exist.

5. Show the following: if \(f_1(n) \in \mathcal{O}(g_1(n))\) and \(f_2(n) \in \mathcal{O}(g_2(n))\) then \(f_1(n) + f_2(n) \in \mathcal{O}(g_1(n) + g_2(n))\).

Solution:

Assume \(f_1(n) \in \mathcal{O}(g_1(n))\), then \(\exists n_1 \in \mathbb{N} \exists c_1 > 0 : n \geq n_1 \Rightarrow f_1(n) \leq c_1 \cdot g_1(n)\). Analogously, assume \(f_2(n) \in \mathcal{O}(g_2(n))\) then \(\exists n_2 \in \mathbb{N} \exists c_2 > 0 : n \geq n_2 \Rightarrow f_2(n) \leq c_2 \cdot g_2(n)\). Let \(n^* := \max\{n_1, n_2\}\) and \(c^* := \max\{c_1, c_2\}\). Then we have: if \(n \geq n^*\) then \(f_1(n) \leq c_1 \cdot g_1\) and hence \(f_1(n) \leq c^* \cdot g_1\), and \(f_2(n) \leq c_2 \cdot g_2\) and hence \(f_2(n) \leq c^* \cdot g_2\). So we have \((f_1 + f_2)(n) \leq c^*((g_1 + g_2)(n))\) voor \(n \geq n^*\).

6. Depict the max-heap [16, 14, 10, 8, 7, 9, 3, 2, 4, 1] as a tree.

Solution:
7. Give the array representation of the max heap of the following picture:

```
24
23 22
12 21 20 8
11 10 18 16 5
```

Solution:

`[24, 23, 22, 12, 21, 20, 8, 11, 10, 18, 16, 5]`

8. Is the array

`[23, 17, 14, 6, 13, 10, 1, 5, 7, 12]`

a max-heap?

Solution:

No, because of the branch from 6 to 7.

9. Is an array of decreasing numbers always a max-heap?

Solution:

Yes, because for every $i$ such that also $2i$ is an index in the array, we have $A[i] > A[2i]$, and similarly for every $i$ such that also $2i + 1$ is an index in the array, we have $A[i] > A[2i + 1]$.

10. Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

Solution:
Also for the subtree the max-heap property holds. That is, walking from a given node downwards in the tree visualisation of the heap, the keys do not increase. So the keys on the nodes below the root of the subtree are all smaller than or equal to the key on the root of the subtree.

11. Where in a max-heap is the smallest element, assuming that all elements are distinct?
Solution:
Because all elements are different, there is a unique smallest key in the heap, which must be the key of one of the leaves.

12. What are the minimum and maximum numbers of elements in a heap of height $h$?
Solution:
Recall that a heap is an almost complete binary tree.
The minimum number of nodes in a heap of height $h$ is when the last level (nodes at depth $h$) has just one node. Then it has $2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} + 1 = 2^h - 1 + 1 = 2^h$ nodes. For instance, a smallest heap of height 3 has $2^3 = 8$ nodes.
The maximum number of nodes in a heap of height $h$ is when the last level (nodes at depth $h$) is completely filled. (The heap then actually has the shape of a complete binary tree.) Then it has $2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$ nodes. For instance, a largest heap of height 3 has $2^4 - 1 = 15$ nodes.

13. Show that an $n$-element heap has height $\lceil \log(n) \rceil$.
Given an $n$-element heap, we know from the previous exercise that its height $h$ should satisfy $2^h \leq n \leq 2^{h+1} - 1$. The inequality $2^h \leq n$ yields $h \leq \log n$. From the inequality $n \leq 2^{h+1} - 1$ follows $n < 2^{h+1}$ and hence $\log n < h + 1$. Hence $h \leq \log n < h + 1$, so $h = \lceil \log n \rceil$.

14. Turn the following sequences into a heap, using the procedure `downMaxHeap` from the slides which is the same as the procedure `MaxHeapify` from the book. First determine (by hand) the index of the node that should be bubbled. Use the Figure in the book as model.

$1 8 6 5 3 7 4$
$27 17 3 16 13 10 1 5 7 12 4 8 9 0$

The first array contains $2^3 - 1 = 7$ elements. The for-loop from the procedure `buildMaxHeap` considers indices 3, 2, 1.
Applying `downMaxHeap` to index 3 yields bubbling down the 6. In a picture:

```
1 8 6 5 3 7 4
27 17 3 16 13 10 1 5 7 12 4 8 9 0
```

1
Then considering index $i = 2$ does not result in a swapping, because 8 is larger than 5 and larger than 3.

Finally index $i = 1$ is considered. This results in two calls of \texttt{downMaxHeap} that bubbles down the key 1. The first step gives

```
  1
 / \ 
 8 7 
/ \ / \
5 3 6 4
```

and the second step gives

```
  8
 / \ 
 1 7 
/ \ / \
5 3 6 4
```

So the final result in array representation is [8, 5, 7, 1, 3, 6, 4].

The second sequence is turned into a max-heap by applying \texttt{downMaxHeap} to index 3.

```
                27
               /  \  
              /    \ 
             17    3
            /    /  
           /    / \
          16 13 10 1
         /  /  /  \
        / / / / \
      5 7 12 4 8 9 0
```
15. Provide pseudo-code for a procedure `downMinHeap` or in book-terminology `MinHeapify`, similar to `downMaxHeap` or `MaxHeapify` that takes as input an array `A` and an index `i` in `A`, and that let the key at `i` bubble down to restore the min-heap property.

Solution:

Algorithm `downMinHeap(A, i)`:

1. `l := left(i)`
2. `r := right(i)`
4.   `smallest := l`
5. else
6.   `smallest := i`
8.   `smallest := r`
9. if `smallest ≠ i` then
10.   `swap(A[i], A[smallest])`
11. `downMinHeap(A, smallest)`
16. Illustrate the operation of buildMaxHeap on the sequence

\[5\ 3\ 17\ 10\ 84\ 19\ 6\ 22\ 9\]

using Figure 6.3 from the book as model.

Solution:
Here using the array-representation:

\[
\begin{array}{c}
[5, 3, 17, 10, 84, 19, 6, 22, 9] \\
\rightarrow (\text{swap 10 and 22}) \\
[5, 3, 17, 22, 84, 19, 6, 10, 9] \\
\rightarrow (\text{swap 17 and 19}) \\
[5, 3, 19, 22, 84, 17, 6, 10, 9] \\
\rightarrow (\text{swap 84 and 3}) \\
[5, 84, 19, 22, 3, 17, 6, 10, 9] \\
\rightarrow (\text{swap 5 and 84}) \\
[84, 5, 19, 22, 3, 17, 6, 10, 9] \\
\rightarrow (\text{swap 5 and 22}) \\
[84, 22, 19, 5, 3, 17, 6, 10, 9] \\
\rightarrow (\text{swap 5 and 10}) \\
[84, 22, 19, 10, 3, 17, 6, 5, 9] \\
\end{array}
\]
done

17. Use heapsort to sort the following sequences; use Figure 6.4 in the book as model.

\[
\begin{align*}
1 &\ 2 &\ 3 &\ 4 &\ 5 \\
5 &\ 4 &\ 3 &\ 2 &\ 1 \\
5 &\ 13 &\ 2 &\ 25 &\ 7 &\ 17 &\ 20 &\ 8 &\ 4 \\
\end{align*}
\]

Solution:
Sorting 1 2 3 4 5, here using the array-representation:
First build a max-heap:

\[
[1, 2, 3, 4, 5] \rightarrow [1, 5, 3, 4, 2] \rightarrow [5, 1, 3, 4, 2] \rightarrow [5, 4, 3, 1, 2]
\]

Then we continue with the swap, remove, reconstruct steps.
We swap 1 and 5, disconnect 5 from the heap, and apply down-heap bubble at position 1. This yields:

\[
[4, 2, 3, 1 | 5]
\]

We swap 4 and 1, disconnect 4 from the heap, and apply down-heap bubble at position 1. This yields:

\[
[3, 2, 1 | 4, 5]
\]

We swap 3 and 1, disconnect 3 from the heap, and apply down-heap bubble at position 1. This yields:

\[
[2, 1 | 3, 4, 5]
\]
We swap 2 and 1, disconnect 2 from the heap, and apply down-heap bubble at position 1. This yields:

\[ [1 \mid 2, 3, 4, 5] \]

Done, result \([1, 2, 3, 4, 5]\).

Sorting 5 4 3 2 1, here using the array-representation:
First build a max-heap: nothing to be done. We swap 5 and 1, disconnect 5 from the heap, and apply down-heap bubble:

\[ [4, 2, 3, 1 \mid 5] \]

We swap 4 and 1, disconnect 4 from the heap, and apply down-heap bubble:

\[ [3, 2, 1 \mid 4, 5] \]

We swap 3 and 1, disconnect 3 from the heap, and apply down-heap bubble:

\[ [2, 1 \mid 3, 4, 5] \]

We swap 2 and 1, disconnect 2 from the heap, apply down-heap bubble:

\[ [1 \mid 2, 3, 4, 5] \]

Done.

Sorting 5 13 2 25 7 17 20 8 4, here using the array-representation:
(to be done)

18. Describe an algorithm for checking whether an array of numbers is a max-heap, and determine its worst-case time complexity.

Solution:

\[
\textbf{Algorithm} \ \text{isMaxHeap}(A): \\
\text{\hspace{1em} } n := A.length \\
\text{\hspace{1em} } i := \lfloor n/2 \rfloor \\
\text{\hspace{1em} } \text{if \ } n \text{ \ even \ then} \\
\text{\hspace{2em} } \text{if \ not \ } A[i] \geq A[2i] \text{ \ then} \\
\text{\hspace{3em} } \text{return \ false} \\
\text{\hspace{2em} } i := i - 1 \\
\text{\hspace{1em} } \text{while \ } i > 0 \text{ \ do} \\
\text{\hspace{2em} } \text{if \ (not \ } A[i] \geq A[2i]) \text{ \ or \ (not \ } A[i] \geq A[2i+1]) \text{ \ then} \\
\text{\hspace{3em} } \text{return \ false} \\
\text{\hspace{2em} } i := i - 1 \\
\text{\hspace{1em} } \text{return \ true}
\]
This algorithm is in $O(n)$.

19. Give a definition of a ternary max-heap.

Solution:

Intuition: a ternary max-heap is a almost complete ternary tree, where on a path downwards in the tree the keys do not increase.

A ternary tree is a tree in which every node has at most 3 successors. A complete ternary tree is a ternary tree is a tree in which all leaves occur at the same depth, and the last depth is completely filled. As a consequence, every node has either 0 or 3 successors. An almost complete ternary tree is a complete ternary tree where the last depth is filled from left to right, but not necessarily completely.

How to view an almost complete ternary tree as an array? The root gets index 0. So we will also use an array starting with index 0. The children of a node with label $i$ get labels $3i + 1$, $3i + 2$, $3i + 3$ for the left, middle, right child. We can give the algorithms for $\text{parent}$, $\text{left}$, $\text{middle}$, and $\text{right}$.

Algorithm $\text{left}(i)$:
\[
\text{return } 3i + 1
\]

Algorithm $\text{middle}(i)$:
\[
\text{return } 3i + 2
\]

Algorithm $\text{right}(i)$:
\[
\text{return } 3i + 3
\]

Algorithm $\text{parent}(i)$:
\[
\text{return } \lfloor (i - 1)/3 \rfloor
\]

We will use ternary heaps in the array-representation. A ternary heap is an array $T$ with first index 0, that satisfies the condition
\[
T[\text{parent}(i)] \geq i
\]