1. (recap)

Give an algorithm in $O(n \log n)$ that takes as input a sequence $A$ of $n$ integers, and gives back as output a sequence in which every integer from $A$ occurs exactly once.

Solution:

Intuition: first sort the sequence, which can be done in $O(n \log n)$, then walk through the sequence to remove duplicates.

Pseudo-code for the algorithm:

```plaintext
Algorithm removeDuplicates(A):
    A := mergeSort(A)
    n := A.length()
    duplicates := 0
    for i := 2 to n:
        if A[i] = A[i - 1] then
            duplicates := duplicates + 1
    new Array B[n - duplicates]
    j := 2
    for i := 2 to n:
        if A[i] != A[i - 1] then
            j := j + 1
    return B
```

2. (recap)

The recurrence for merge sort is:

$$T(n) = \begin{cases} 
1 & \text{als } n = 1 \\
2T(\frac{n}{2}) + n & \text{als } n > 1 
\end{cases}$$

Solve the recurrence for $n$ a power of 2, and argue that the worst-case time complexity of merge sort is hence in $O(n \log n)$.

Solution:

We use the substitution method to solve this recurrence. For large $n$: 

$$T(n) = \begin{cases} 
1 & \text{als } n = 2^0 \\
2T(2^{k-1}) + n & \text{als } n > 2^0 
\end{cases}$$

Using the substitution method, we can show that $T(n) = O(n \log n)$.

...
\[
T(n) = 2T\left(\frac{n}{2}\right) + n \\
= 2(2T\left(\frac{n}{4}\right) + \frac{n}{2}) + n \\
= 4T\left(\frac{n}{4}\right) + 2n \\
= 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n \\
= 8T\left(\frac{n}{8}\right) + 3n \\
= \ldots \\
= 2^i T\left(\frac{n}{2^i}\right) + in
\]

We arrive at the start value for \(n = 1\) if \(\frac{n}{2} = 1\). Then we havet: \(n = 2^i\) en dus \(i = \log n\). Substitute \(i = \log(n)\). This yields:

\[
T(n) = n + n \log(n)
\]

3. What is the shape of a tree with the highest possible level numbering using the minimum number of nodes?
(We have seen this already.)
Solution: a comb to the right

```
  * \
  / \
  * * \
  / \
  * * \
  / \
  * *
```

4. Give pseudo-code for the algorithm of *level order traversal* of a binary tree. (We visit the nodes layer by layer from top to bottom and per layer from left to right.)
Solution:

```
Algorithm levelOrder(T):
    Q := new Queue
    Q.enqueue(T.root())
    while not Q.isEmpty() do
        v := dequeue(Q)
        visit v
        if v.left \(\neq\) nil then
            enqueue(Q, v.left)
        if v.right \(\neq\) nil then
            enqueue(Q, v.right)
```
5. Which node is visited last in a 
• preorder traversal
• inorder traversal
• postorder traversal

Solution:
• In een preorder traversal: the leaf most to the right is visited last.
• In een inorder traversal: the leaf most to the right is visited last.
• In een postorder traversal: the root is visited last.

6. Give pseudo-code of a non-recursive algorithm for inorder traversal. Use a stack. Also describe step by step the result of applying your algorithm to the following tree:

Solution:
First, we assume that all nodes have either 0 or 2 successors. In the following, $T$ is a binary tree and $S$ is a stack. This illustrates using a stack can be an alternative to recursion.

**Algorithm inorder**(T):

- $S := \text{new Stack}$
- $S := S.push(T.root)$
- $v := T.root$
- $\text{while } S \neq \emptyset \text{ do}$
  - $\text{while } v.left \neq \text{nil do}$
    - $v := v.left$
    - $S.push(v)$
    - $v := S.pop()$
    - $\text{visit}(v)$
  - $\text{if } v.right \neq \text{nil then}$
    - $v := v.right$
    - $S.push(v)$
- $v = a$
- $S = a$
• $v = b$
  $S = b, a$

• $c = c$
  $S = c, b, a$

• $v = c$
  $S = b, a$
  visit $c$

• $v = b$
  $S = a$
  visit $b$

• $v = d$
  $S = d, a$

• $v = d$
  $S = a$
  visit $d$

• $v = a$
  $S = []$
  visit $a$

• $v = e$
  $S = e$

• $v = e$
  $S = e$
  visit $e$

7. Give pseudo-code for an algorithm that takes as input a node $v$ in a binary tree, and gives as output the node that is visited directly after $v$ in case of
  • preorder traversal
  • inorder traversal
  • postorder traversal

Solution:

• Picture with on the left $v$ an internal node, and on the right $v$ a leaf:
The following algorithm takes as input a node $v$ in a binary tree and gives back as output the node that is visited after $v$ in a preorder traversal:

**Algorithm** preorderNext($T, v$):

if $T$.isInternal($v$) then
    return $T$.left($v$)
else
    $p := T$.parent($v$)
    while $T$.left($p$) $\neq v$ do
        $v := p$
        $p := T$.parent($p$)
    return $T$.right($p$)

- Picture with left $v$ an internal node, and right $v$ a leaf:

  ![Diagram](image1)

  We can use here the algorithm successor as in the book and on the slides.

- Picture with left $v$ a left-child, and right $v$ a right-child:

  ![Diagram](image2)

(To be updated.) This algorithm takes as input a node $v$ in a binary tree and gives back as output the node that is visited after $v$ in a postorder traversal:
Algorithm `postorderNext(T, v)`:
   if `T.left(T.parent(v)) = v` then
      \( w := T.sibling(v) \)
      while `T.isInternal(w)` do
         \( w := T.left(w) \)
      return \( w \)
   else
      return `T.parent(v)`

8. Give pseudo-code of the algorithm `lca` that takes as input two nodes \( v \) and \( w \) in a tree \( T \), and that gives as output the least common ancestor of \( v \) and \( w \) in \( T \).

   What is the time complexity of your algorithm in terms of \( O \)?

   Solution, very informally and not complete:

   We look for the lowest common ancestor of two nodes \( v \) and \( w \) in a binary tree \( T \). (Such a common ancestor exists, and the lowest common ancestor is unique.) Idea: replace the lowest of the two nodes by its direct predecessor, and continue doing so until both nodes have equal depth. Then replace ‘in parallel’ both nodes by their direct predecessors, until those two are equal.

   **Algorithm `LCA(T, v, w)`:**
   
   while `depth(T, v) > depth(T, w)` do
      \( v := T.parent(v) \)
   while `depth(T, w) > depth(T, v)` do
      \( w := T.parent(w) \)
   while \( v \neq w \) do
      \( v := T.parent(v) \)
      \( w := T.parent(w) \)
   return \( v \)

   This algorithm is in \( O(n) \).

9. Consider the array representation of a heap with 15 elements. What is the sequence of indices for a preorder traversal of the heap? And for the inorder and postorder traversal?

   Solution:

   We only use the property that a heap is an almost complete binary tree. The (max- or min-) heap property is not relevant here.

   Using preorder: 1, 2, 4, 8, 9, 5, 10, 11, 3, 6, 12, 13, 7, 14, 15.
Using inorder: 8, 4, 9, 2, 10, 5, 11, 1, 12, 6, 13, 3, 14, 7, 15.
Using postorder: 8, 9, 4, 10, 11, 5, 2, 12, 13, 6, 14, 15, 7, 3, 1.

10. Give an example showing that a preorder traversal of a min-heap does not necessarily yield a non-decreasing sequence.
Solution:
A small example showing that a preorder traversal of a min-heap does not necessarily give the keys in non-decreasing order:

```
    1
   / \  
  3   2
```

11. Give an example showing that a preorder traversal of a min-heap does not necessarily yield a non-increasing sequence.
Solution:
A small example showing that a postorder traversal of a min-heap does not necessarily give the keys in non-increasing order:

```
    1
   / \  
  2   3
```

12. Depict binary trees with heights 2, 3, 4, 5, and 6, all with the seven keys 1, 4, 5, 10, 16, 17, 21.
Solution:
With height 2:
```
  10
 / \  
 4 17
 / \  / \ 
 1 5 16 21
```

With height 3:
```
  10
 / \  
 5 17
 /  /  
4 16 21
 / 
1
```
With height 4:

```
   17
    / \
   10  21
    / \
   5   16
    / \
   4   \\
    / \
   1   \
```
13. Build a binary search tree by adding one by one, and in this order, the following numbers: 5, 6, 3, 8, 7, 4, 1, 2.

Solution:

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

14. How many binary search trees with labels 1, 2, 3, 4 exist? Is it relevant what the labels are? And that it is a binary search tree?

Solution:

```
    5
   / \
  3   6
 / \ / \ 
1  4 7  8
```
There are 14 binary search trees with keys 1, 2, 3, 4. With key 1 on the root: 5. With key 4 on the root: also 5. With 2 on the root: 2. With 3 on the root: also 2.

We can schematically render the binary search tree with key 1 on the root as follows:

```
  /  \
 /    \ 
<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
</tr>
</thead>
</table>
```

The binary trees consisting of three nodes correspond to the different ways in which we can put parentheses in an expression \(abc\) where between the letter is a binary operator. The \(a, b, c\) then correspond to the leaves of the binary tree.

In general, the number of binary trees with \(n\) nodes (the different ways of putting parentheses in an expression of \(n + 1\) say letters in a binary operator) is the \(n\)th Catalan number:

\[
C_n = \frac{1}{n+1} \binom{2n}{n}
\]

The first Catalan numbers (the first one for \(n = 0\)):

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, . . .

15. For every of the following statements: explain shortly why it is true or give a counterexample showing it is false.

(a) There exists a binary search tree which is also a min-heap.
   Solution:
   Yes: a binary search tree with only one node.

(b) There exists a binary search tree which is also a max-heap.
   Solution:
   Yes: a binary search tree with only one node, and also for instance
   
   `2
   / \
  1`

(c) Every binary search tree is a min-heap.
   Solution:
   No. For instance the following binary search trees are no min-heaps:
   
   `2
   / \
  1`
   `2
   / \`
  `1 3`
The first one does not satisfy the min-heap property (but is an almost complete binary tree).
The second one is not an almost complete binary treee (but does satisfy the min-heap property).

(d) Every min-heap is a binary search tree.
Solution:
No: for instance the following min-heap is not a binary search tree:

```
1
/ 
2
```

16. Assume a binary search tree with labels the numbers 1 till 1000. We search a node labeled 363. Which of the following sequences cannot be the labels of nodes visited in searching 363?

(a) 2, 252, 401, 398, 330, 344, 397, 363.
(b) 924, 220, 911, 244, 898, 258, 362, 363.
(c) 925, 202, 911, 240, 912, 245, 363.
(d) 2, 399, 387, 219, 266, 382, 381, 278, 363.
(e) 935, 278, 347, 621, 299, 392, 358, 363.

Solution:
(To be checked; there was something.)
The sequence in (c) cannot be the labels of nodes visited in a binary tree search because 912 cannot be to left of 911. The sequence in(e) cannot be the labels of nodes visited in a binary tree search because 299 cannot be to the right of 347.

17. Give pseudo-code for a recursive algorithm that finds the node with the smallest label in a sub-tree starting with node $v$.
Same for finding the node with the largest key, starting with node $n$.
Solution:

```
Algorithm treeMinRec(v):
    if v.left = null then
        return v
    else
        return treeMinRec(v.left)
```

Second algorithm still missing.
18. Add 4 to the following binary search tree, and next add 14:

```
                     11
                    /   \
                   8     14
                  /   /  \
                 5   9    13
                /   /  \
               3   6    10
              /   /    /  \
             1   4    7    17
```

Solution:
We add 4 and next add 14; this yields the following tree:

```
                     11
                    /   \
                   8     14
                  /   /  \
                 5   9    13
                /   /  \
               3   6    10
              /    /  \
             1    4    7
```

19. Remove 6 from the following binary search tree, and next remove 14.

```
                     11
                    /   \
                   8     14
                  /   /  \
                 5   9    13
                /   /  \
               3   6    10
              /   /    /  \
             1   4    7    17
```

Solution:
First we remove 6. This yields:
Then we remove 14. Here in applying treeDelete we use the last case, where the node to be deleted has both a left and a right child. The result is as follows:

20. We can sort a sequence as follows: add the numbers one by one to a binary search tree, then visit the nodes using inorder. What is the best-case and the worst-case time complexity of this algorithm?

Solution:

Algorithm treeSort\((A, n)\):

1. new binary search tree \(T\)

2. for \(i := 1\) to \(n\) do
   1. treeInsert\((T, A[i])\)
   2. inOrder\((T.root)\)

In the inorder traversal of \(T\), we must use visit to print the key of the node to the output.
The best case if we happen to get an almost complete binary tree. Then the time complexity is in $\Theta(n \log n)$.

The worst case if we happen to get a ‘linear’ binary tree of height $n$. Then the time complexity is in $\Theta(n^2)$. 