1. Consider the algorithm \textit{AddSomething}(A, n) :

\begin{algorithm}
\caption{AddSomething}(A, n)
\begin{algorithmic}
\Input Array A and integer n. A has size at least n
\If{n < 1} \Return 0
\Else
\State \text{temp} ← 3 \ast A[n - 1] + \text{AddSomething}(A, n - 2)
\EndIf
\Return temp
\end{algorithmic}
\end{algorithm}

(a) Write down recurrence equations that describe the running time of algorithm \textit{AddSomething}(A, n).

(b) Solve these recurrence equations and give the asymptotic (big-O) complexity of the algorithm.

Solution:

(a) \( T(0) = c \)
\( T(n) = k + T(n - 2) \)

(b) \( T(n) = k + T(n - 2) = k + k + T(n - 2 - 2) = k + k + k + T(n - 2 - 2 - 2) = \ldots = i \ast k + T(n - 2 \ast i) \).

Unwrapping will stop when \( n = 2 \ast i \), or, equivalently, when \( i = n/2 \). Thus
\[ T(n) = \frac{nk}{2} + T(0) = \frac{nk}{2} + c, \]

and the big-O complexity of the algorithm is \( O(n) \).

2. Prove that in a binary tree, the minimum number of internal nodes is \( h \), and the maximum number of internal nodes is \( 2^h - 1 \), where \( h \) is the height of the tree. That is prove \( h \leq i \leq 2^h - 1 \), where \( i \) is the number of internal nodes.

Solution:

The smallest number of internal nodes is 1 at each level of the tree, except the last level \( h \). Last level \( h \) has only external nodes. Therefore, the smallest number of internal nodes at levels \( j \), is 1, for \( 0 \leq j < h \), and the total smallest number of internal nodes is \( h \).

The largest number of internal nodes is when each level of the tree is full, i.e. the tree has exactly \( 2^l \) nodes at each level \( l \). Nodes at levels 0, \( h - 1 \) are internal and nodes at level \( h \) has only external nodes. Therefore the largest number of internal nodes is \( 2^0 + 2^1 + \ldots + 2^{h-1} = 2^h - 1 \).

3. Draw a binary tree where each node stores a character key, and preorder traversal visits nodes in the order LXZFGED, and postorder traversal visits nodes in the order ZFXEDGL.

Solution:
4. Suppose that a heap stores 210 elements. What is its height?

Solution: The maximum number of nodes at level \( l \) of the heap is \( 2^l \). Therefore a heap of height \( h \) stores at most \( 2^h + 2^{h-1} + \ldots + 2^1 + 1 \) nodes. If \( h = 6 \), then the heap stores at most \( 2^7 - 1 = 127 \) elements, which is not enough. So the heap must be of height 7, since for height 7 a heap can store \( 2^8 - 1 = 255 \). The last level of the heap is not full, in this particular case, since it only has 210 elements.

5. Suppose that a heap has height 10 and stores only unique keys. At which levels of the heap can the 4th largest key be stored?

Solution: Let node \( n \) be the node storing 4th largest key. All the descendants of node \( n \) must have larger keys. Therefore the maximum number of the descendants for node \( n \) is 3. Nodes at level \( h \) (the last level of the heap) have no descendants. Nodes at level \( h - 1 \) have either 0, 1, or 2 descendants. Nodes at level \( h - 2 \) have anywhere from 2 to 6 descendants. Nodes at level \( h - 3 \) have at least 6 descendants. Therefore the forth largest node can only be at levels \( h, h - 1, h - 2 \), which is 10,9, and 8 in our problem.
6. (a) Suppose $T$ is a binary tree that at each node stores an integer key. Let us call a node left-unbalanced if the sum of keys in its left subtree is less than the sum of keys in its right subtree. For example, in the tree above, the root node is left-unbalanced, since the sum of keys in its left subtree is 4 and in the right subtree is 5. Also the left child of the root node is left-unbalanced. Write an algorithm that takes as an input the root of the tree and returns the number of left-unbalanced nodes. For the tree above, your algorithm should return 2.

Solution: Let each node $v$ store a variable $v.s$ that we can use for storage.

Algorithm \textit{SumUnbalanced}(v)
\begin{itemize}
  \item if $v$.external()
    \begin{itemize}
      \item $v.s \leftarrow v.key$
      \item return 0
    \end{itemize}
  \item else
    \begin{itemize}
      \item $sumLeft \leftarrow 0, sumRight \leftarrow 0$
      \item $resultLeft \leftarrow 0, resultRight \leftarrow 0$
      \item if $v$.hasLeft()
        \begin{itemize}
          \item $resultLeft \leftarrow \text{SumUnbalanced}(v.left)$
          \item $sumLeft \leftarrow v.left.s$
        \end{itemize}
      \item if $v$.hasRight()
        \begin{itemize}
          \item $resultRight \leftarrow \text{SumUnbalanced}(v.right)$
          \item $sumRight \leftarrow v.right.s$
        \end{itemize}
      \item $v.s = sumLeft + sumRight + v.key$
      \item if $sumLeft < sumRight$
        \begin{itemize}
          \item return $1 + resultLeft + resultRight$
        \end{itemize}
      \item else return $resultLeft + resultRight$
    \end{itemize}
\end{itemize}

(b) Analyze the running time of your algorithm in part (a) as a function of the number of tree nodes $n$.

Solution: The algorithm above performs post-order tree traversal and therefore it visits each node exactly once. At each visitation it does a constant amount of operations, and therefore the running time is linear in the number of nodes, i.e. $O(n)$
7. Prove that if every internal node in a tree has exactly 3 children, then \( e = 2i + 1 \), where \( e \) is the number of external nodes and \( i \) is the number of internal nodes.

Solution: Pace 2 stones (markers) on each external node. Total number of stones is \( 2 \times \) (number of external nodes). Then leave one stone on each leaf and pass another stone to its parent. Any internal node will get exactly 3 stones, since it has exactly 3 children. Let every internal node pass 1 stone to its parent. Continue doing this in the bottom-up fashion. At the end, when no more stones can be passed from a node to its parents, the root has exactly 3 stones, every internal node has exactly 2 stones, and every external node has 1 stone. Since the total number of stones is \( 2 \times \) (number of external nodes), we have the desired result.