DAA Assignment III

1.

A. What is the disadvantage of merge sort?

- In many implementations, if the list is N long, then it needs 2 x N memory space to handle the sort.
- If recursion is used to code the algorithm then it uses twice as much stack memory as quick-sort – on the other hand it is not difficult to code using iteration rather than recursion and so avoid the memory penalty.

B. Write a recurrence relation for the average case analysis of Quick sort and solve it by showing all the steps clearly.

Quicksort is a divide-and-conquer algorithm for sorting a list S of n comparable elements (e.g. an array of integers).

Given its recursive design, the analysis of quick sort involves solving the recurrence relation $t(n)$ that describes its run time. Its run time $t(n)$ is equal to the sum of run times of the two recursive calls and of the run time $f(n)$ required for selecting the pivot and partitioning S into S_L and S_R. We have $f(n) = \Theta(n)$, because this work can be done in one pass through S. Hence, $t(n)$ is given by:

$$t(n) = t(i) + t(n-i-1) + n, \quad \text{for } n > 1, \text{ and } t(0) = t(1) = 1, \tag{1}$$

where $i = |S_L|$ is the size of the left sublist.

Average-Case Analysis

For the pivoting and partitioning strategy defined above, we can assume that each possible size of S_L (and consequently S_R) is equally likely. Possible sizes are 0, 1 ..., $n-1$, each having a probability $1/n$. Hence, the average value of the run time $t(n)$ in Eq. 1 is given by,

$$t(n) = \frac{1}{N}[\sum_{i=0}^{n-1} t(i) + t(n-i-1)] + n \tag{2}$$

Since $\sum_{i=0}^{n-1} t(i) = \sum_{i=0}^{n-1} t(n-i-1)$, $t(n)$ can be written as,

$$t(n) = \frac{2}{N}[\sum_{i=0}^{n-1} t(i)] + n \tag{3}$$

The steps below apply some algebraic manipulations in order to put the recurrence relation in a solvable form. First, multiplying by n, we get

$$nt(n) = 2[\sum_{i=0}^{n-1} t(i)] + n^2 \tag{4}$$

Substituting n by $n-1$, we get,

$$(n-1)t(n-1) = 2[\sum_{i=0}^{n-2} t(i)] + (n-1)^2 \tag{5}$$

By subtracting Eq. 5 from Eq. 4, we obtain,

$$nt(n) - (n-1)t(n-1) = 2t(n-1) + 2n - 1 \tag{6}$$

Rearranging and simplifying, we get,

$$nt(n) = (n+1)t(n-1) + 2n \tag{7}$$

Dividing both sides by $n(n+1)$, we get,

$$\frac{t(n)}{n+1} = \frac{t(n-1)}{n} + \frac{2}{n+1} \tag{8}$$

Applying back substitution we get,

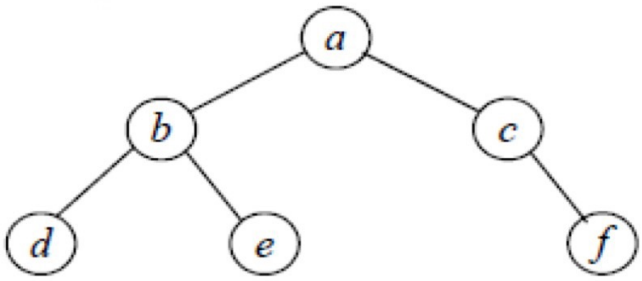
$$\begin{aligned}
\tfrac{t(n)}{n+1} &= \tfrac{t(n-2)}{n-1} + \tfrac{2}{n} + \tfrac{2}{n+1} \\
&= \tfrac{t(n-3)}{n-2} + \tfrac{2}{n-1} + \tfrac{2}{n} + \tfrac{2}{n+1} \\
&= \cdots \\
&= \tfrac{t(1)}{2} + 2\sum_{i=3}^{n+1} \frac{1}{i} \\
&\approx \tfrac{1}{2} + 2[\ln(n+1) + \gamma - \tfrac{3}{2}]
\end{aligned}$$

where $\gamma \approx 0.577$ is Euler's constant. Therefore,

$$\begin{aligned}
\frac{t(n)}{n+1} &= O(\log n) \quad \text{and hence,} \\
t(n) &= O(n \log n).
\end{aligned}$$

2.

A. Write a non-recursive algorithm for inorder traversal of binary trees. Write the preorder and postorder traversal of the following tree.




```
// Algorithm In-order traversal
// Input: Binary tree
// Result: In-order traversal of the binary tree.
```

```
inorder(node)
  s ← empty stack
  while (not s.isEmpty() or node ≠ null)
    if (node ≠ null)
      s.push(node)
      node ← node.left
    else
      node ← s.pop()
      visit(node)
      node ← node.right
```

B. Compute 210 x 110 by applying the divide-and-conquer algorithm clearly showing all the steps.

x = 210 and y = 110

xH = 2 and xL = 10
yH = 1 and yL = 10

```
                    xH          xL
      x             yH          yL
    ------------------------------
                  xH yL       xL yL
  + xH yH         xL yH
    ------------------------------
    xH yH    xH yL + xL yH    xL yL
```

Let A = xH yH = 2 * 1 = 2
C = xL yL = 10 * 10 = 100

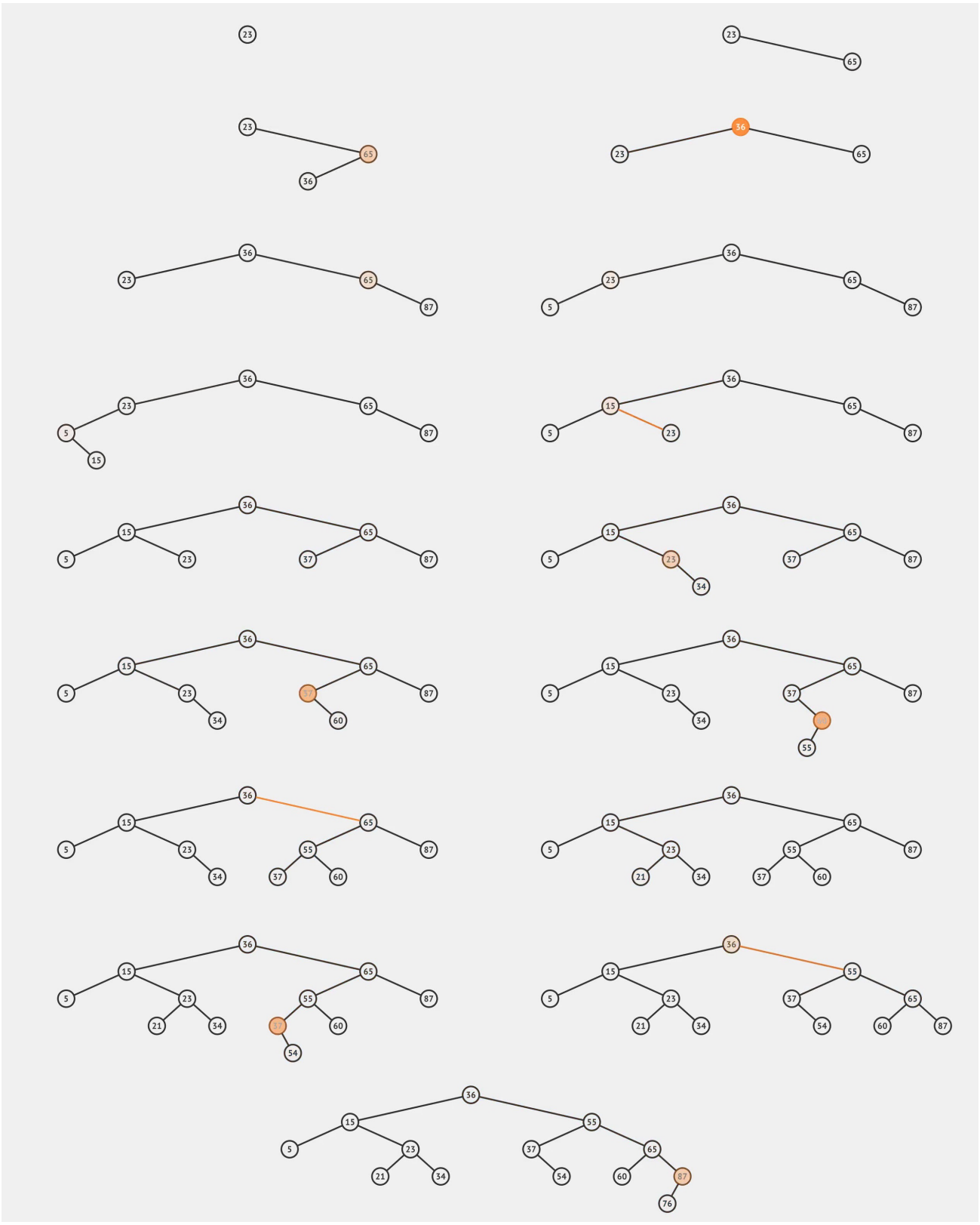
B = xH yL + xL yH
= (xH + xL) (yL + yH) − A − C
= 12 * 11 − 2 − 100
= 30

$$
\begin{aligned}
\text{Result} &= A * 10^4 + B * 10^2 + C * 1 \\
&= 2 * 10000 + B * 100 + C * 1 \\
&= 20000 + 3000 + 100 \\
&= 23100
\end{aligned}
$$

3.

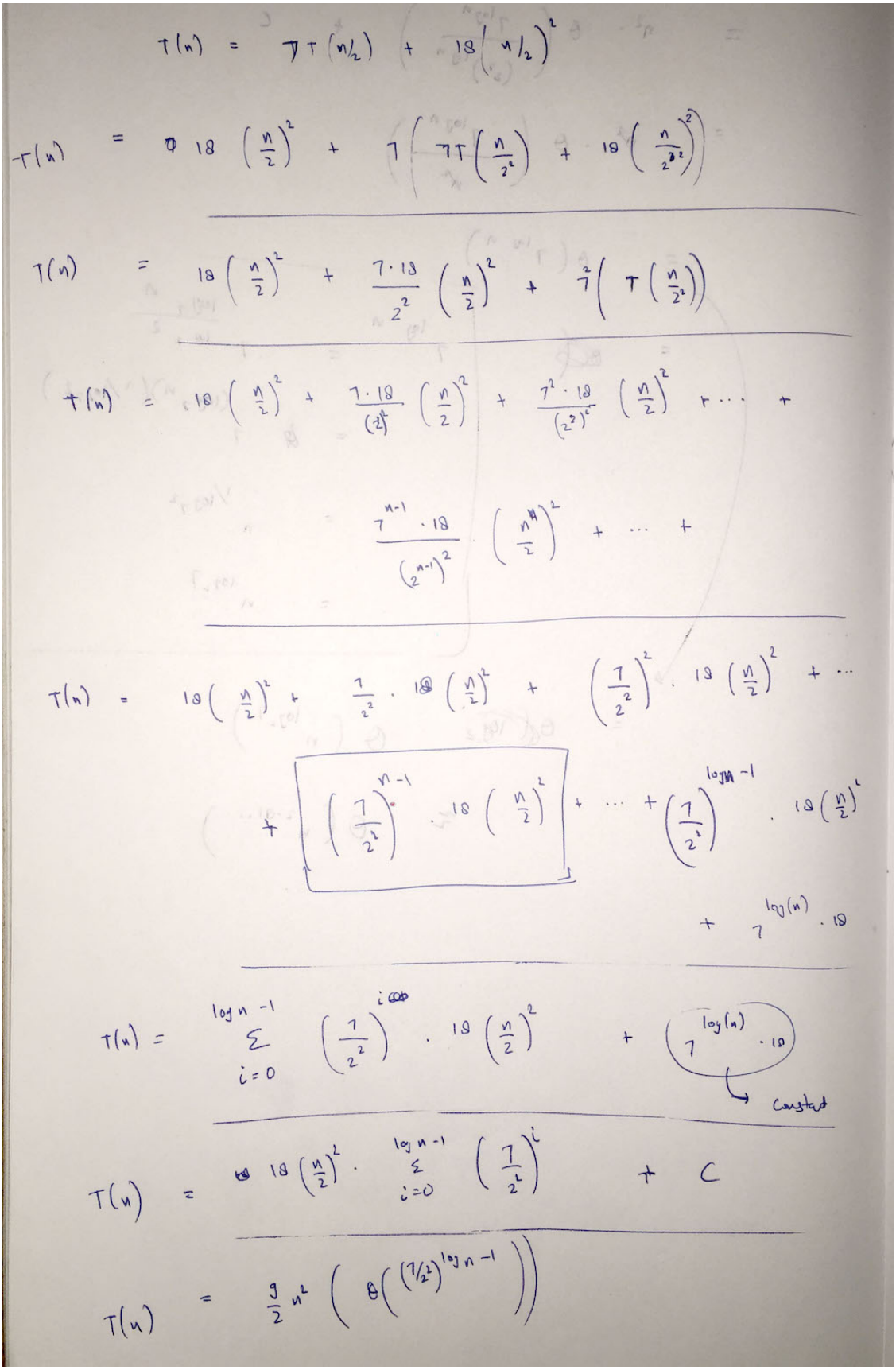
A. Construct an AVL tree for the following data by inserting their elements successively, starting with the empty tree.

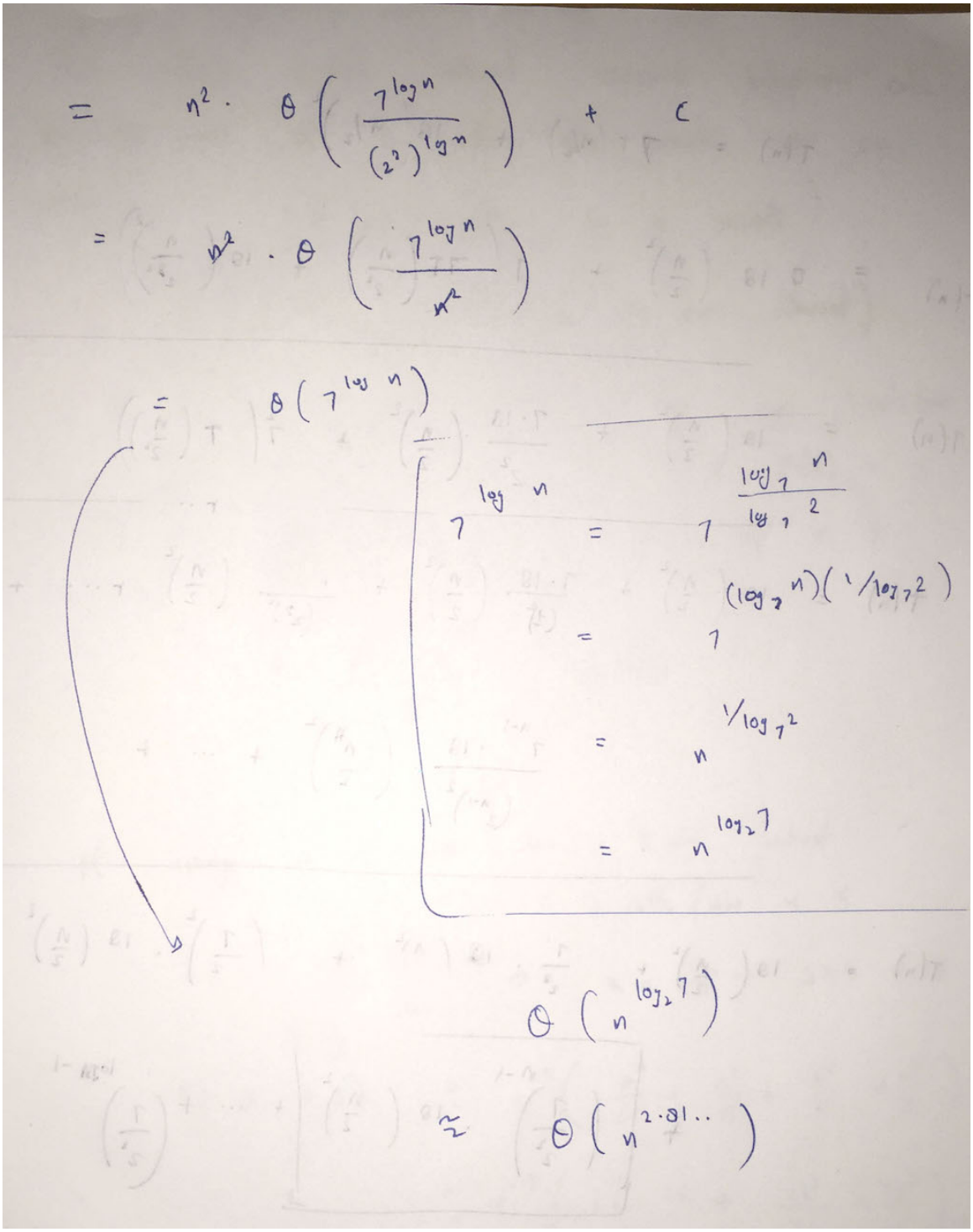
23, 65, 36, 87, 5, 15, 37, 34, 60, 55, 21, 54, 76



B. Solve the following recurrence relation of Strassen's algorithm for multiplying matrices using both back substitution method and Master's Theorem.

$T(n) = 7T(n/2) + 18\,(n/2)^2$
for $n > 1$ and $T(1) = 1$





MASTER THEORM

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

a = 7
b = 2
f(n) = 18 $(n/2)^2 = \Theta(n^d) = \Theta(n^2)$
k = $\log_b(a) = \log_2(7) = 2.81 > (d = 2)$

$T(n) = \Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(7)}) = \Theta(n^{2.81})$

4.

A. Consider the problem of finding distance between the two closest numbers in an array of n numbers. The distance between two numbers x and y is computed as |x-y|. Design a pre-sorting based algorithm for this problem and determine its efficiency.

```
MERGE-SORT (A, p, r)
  IF p < r
     q = FLOOR[(p + r)/2]
     MERGE-SORT (A, p, q)
     MERGE-SORT (A, q + 1, r)
     MERGE (A, p, q, r)
```

```
MERGE (A, p, q, r)
   INPUT: Array A and indices p, q, r such that p ≤ q ≤ r and
   subarray A[p .. q] is sorted and subarray A[q + 1 .. r] is
   sorted. By restrictions on p, q, r, neither subarray is empty.

   OUTPUT: The two subarrays are merged into a single sorted
   subarray in A[p .. r].
```

```
// Algorithm ClosestDistance (A[0..n-1], x, y)
        MERGE-SORT (A, 0, n-1)
        if (y > x)
                SWAP (x, y)
        for i = 0 to n - 1 do
                if (A[i] = x)
                        break
        if (i = n)
                return -1
        for j = i to n - 1 do
                if (A[j] = y)
                        break
        if (j = n)
                return -1
        return j - i
```

Merge sort takes n log(n) time on an average.
Closest distance takes n time at the worst case (elements on both ends or not found)
Total time taken = n log(n) + n = n log(n)

B. For an AVL tree containing real numbers, design an algorithm for computing the range (i.e., the difference between the largest and smallest numbers in the tree) and determine its worst-case efficiency.

```
// Algorithm AVLRange
// Input: AVL tree
// Output: Difference between the ranges

AVLRange (root)
      lnode = rnode = root
      while (lnode.left != null)
            lnode = lnode.left
      while (rnode.right != null)
            rnode = rnode.right
      return rnode.value - lnode.value
```

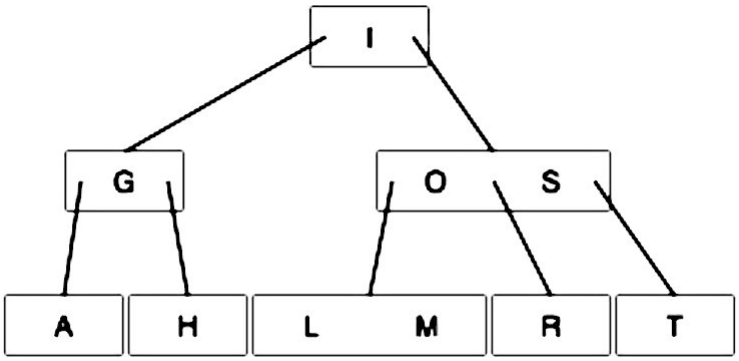
In the worst case, the leftmost and rightmost nodes will be on the last level of the tree.

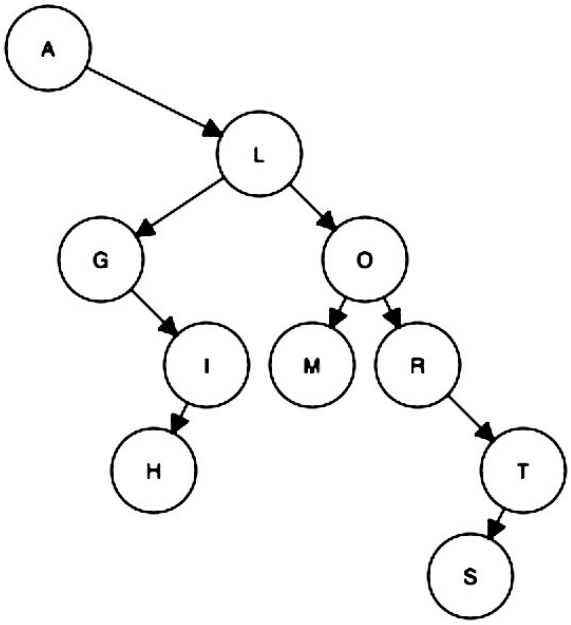
Hence the worst case efficiency

$$= \theta(\log(n)) + \theta(\log(n)) + 1$$
$$\simeq \theta(\log(n))$$

5.

A. Consider a list of elements A, L, G, O, R, I, T, H, M, S. Among the BST and 2-3 representations of the given list, compare the largest number of key comparisons required for a successful search.





At the maximum, the 2-3 tree needs 1 + 2 + 1 = 4 comparisons.

At the maximum the BST needs 6 comparisons.

B. Write an algorithm to check whether a given array A[1..n] is a heap or not. Analyze the complexity.

```
// Algorithm HeapCheck
// Input: Array of size N
// Output: Boolean value determining whether array is heap or not

HeapCheck (A[1..n])
        for i ← 1 to ⌊n/2⌋ do
                if (A[2 * i + 1] > A[i])
                        return NO
                if (A[2 * i + 2] > A[i])
                        return NO
        return YES
```

Basic operation : Comparison

Number of comparisons made = $\lfloor n/2 - 1 \rfloor$.

Complexity = $\theta (n/2)$