# Question - 1

**Pre-condition for using Master's Theorem:**

$f(n)$ should be eventually non-decreasing

Coefficients, in the $T(n)$ -formula, $a$ should be: $a \geq 1$
$b$ should be: $b > 1$

**Steps for solving:**

Master Theorem:

$$T(n) = aT(n/b) + f(n) \qquad a \geq 1,\ b > 1,\ f(n) \text{ non-dec.}$$

$$T(n) = \begin{cases} \vartheta(n^{\log_b a}) & f(n) = \vartheta(n^{\log_b a - \varepsilon}) \\ \vartheta(n^{\log_b a} \log n) & f(n) = \vartheta(n^{\log_b a}) \\ \vartheta(f(n)) & f(n) = \vartheta(n^{\log_b a + \varepsilon}) \end{cases} \quad \varepsilon > 0,\ c < 1$$

(also $af(n/b) < cf(n)$ (regularity condition))

→ Put the $a$ and $b$ coefficients to formula.
→ Look the relation between result and $f(n)$ asymptotically.
→ Select the proper option in the theorem.

Here, equations are given as recursive equations. They are ready-to-apply. (Not needed to convert them, equations, from the algorithm.)

**Solutions:**

(a) $T(n) = 2 \cdot T(\frac{n}{4}) + \sqrt{n \log n}$, $a = 2$, $b = 4$, $f(n) = \sqrt{n \log n}$

$n^{\log_b a} = n^{\log_4 2} = n^{\log_{(2)^2} 2} = n^{1/2} = \sqrt{n}$

$f(n) = \sqrt{n \log n}$ compare with $\sqrt{n}$ asymptotically.

$\lim_{n \to \infty} \frac{\sqrt{n \log n}}{\sqrt{n}} = \infty$, $f(n)$ is bigger than $\sqrt{n}$. So, case-3 is valid.

We should also control is $af(\frac{n}{b}) < cf(n)$. $a f(\frac{n}{4}) < c f(n)$ ✓

$$T(n) = \vartheta(\sqrt{n \log n})$$

(b) $T(n) = 9 \cdot T(\frac{n}{3}) + 5n^2$, $a = 9$, $b = 3$, $f(n) = 5n^2$

$n^{\log_b a} = n^{\log_3 (3)^2} = n^2$

$f(n) = 5n^2$ compare with $n^2$, they're in same asymptotic class.

case 2: $T(n) = \vartheta(n^2 \log n)$

(c) $T(n) = \frac{1}{2} \cdot T(\frac{n}{2}) + n$, $a = \frac{1}{2}$, $b = 2$, $f(n) = n$

$n^{\log_b a} = n^{\log_2 \frac{1}{2}} = \frac{1}{n}$

$f(n) = n$ compere with $\frac{1}{n}$ $\quad n = \vartheta(n^{-1+\varepsilon})$, $\varepsilon = 2$,

which is case-3. BUT

Due to $a = \frac{1}{2}$, which is not $\geq 1$, Theorem can't be apply

---

(d) $T(n) = 5 \cdot T(\frac{n}{2}) + \log n$, $a = 5$, $b = 2$, $f(n) = \log n$

$n^{\log_b a} = n^{\log_2 5} \cong n^2$

$f(n) = \log n$ comp. with $n^2$ asymptotically. $f(n) = \vartheta(n^{\log_b a - \varepsilon})$,

case-1 is valid. $T(n) = \vartheta(n^{\log_2 5}) \cong \vartheta(n^2)$

---

(e) $T(n) = 4^n \cdot T(\frac{n}{5}) + 1$, $a = 4^n$, $b = 5$, $f(n) = 1$

~~$n^{\log_b a} = n^{\log_5 4^n} \cong n^n$~~

~~$f(n) = 1$ comp. with $n^n$ asymptotically. $f(n) = \vartheta(n^{\log_b a - \varepsilon})$,~~

~~case-1 is valid.~~

Due to $a = 4^n$, which is not $\geq 1$, Theorem can't be apply

---

(f) $T(n) = 7 \cdot T(\frac{n}{4}) + n\log n$, $a = 7$, $b = 4$, $f(n) = n\log n$

There is a limited 4th case for master theorem, to apply polylogarithmic function.

4th case is: If $f(n) = O(n^{\log_b a} \log^k n)$, then $T(n) = O(n^{\log_b a} \log^{k+1} n)$

$f(n) = n\log n$ and $k = 1$, Therefore, $T(n) = \vartheta(n\log^2 n)$

---

(g) $T(n) = 2 \cdot T(\frac{n}{3}) + \frac{1}{n}$, $a = 2$, $b = 3$, $f(n) = \frac{1}{n}$

$n^{\log_b a} = n^{\log_3 2} \cong n^{0.6}$

$f(n) = \frac{1}{n}$ comp. with $n^{0.6}$ $\quad f(n) = \vartheta(n^{\log_b a - \varepsilon})$ but,

in $n^{-1} = \vartheta(n^{0.6-\varepsilon})$, $\varepsilon$ becomes $-1.6$ but it should be

bigger than 0. So, master theorem can't be applied.

---

(h) $T(n) = \frac{2}{5} \cdot T(\frac{n}{5}) + n^5$, $a = \frac{2}{5}$, $b = 5$, $f(n) = n^5$

Due to $a = \frac{2}{5}$, which is not $\geq 1$, Theorem can't apply

# Question - 2

Definition of insertion sort:

The array is virtually split into a sorted and unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted path.

Sorting:

$$A = \{3, 6, 2, 1, 4, 5\}$$

- Assume first element, which is 3, is sorted.

  3 | 6 2 1 4 5

- Select the next element, compare it with sorted part.

  is 3 > 6 ? NO. So 6 comes after 3 → 3 6 | 2 1 4 5

- Select the next element, compare it with sorted part.

  is 6 > 2 ? Yes. Move 6 to next position. 3 □ 6

  Look to the before element, is 3 > 2 ? Yes. Move 3 to next position. □ 3 6

  Due to we come to the start, append 2 to there.

  2 3 6 | 1 4 5

- Select next element. Apply same thing.

  is 6 > 1 ? Yes. Move → 2 3 □ 6

  is 3 > 1 ? Yes. Move → 2 □ 3 6

  is 2 > 1 ? Yes. Move → □ 2 3 6

  we come to the start. Append 1 to array.

  1 2 3 6 | 4 5

- Apply same thing.

  is 6 > 4 ? Yes. Move → 1 2 3 □ 6

  is 3 > 4 ? No. So 4 comes after 3

  1 2 3 4 6 | 5

- Apply same.

  is 6 > 5 ? Yes. Move → 1 2 3 4 □ 6

  is 4 > 5 ? NO. so 5 comes after 4

  1 2 3 4 5 6, all sorted.

Algorithm:

```
InsertionSort(A[0..n-1])
for i = 1 to n-1 do
    v = A[i], j = i-1
    while j >= 0 and A[j] > v do
        A[j+1] = A[j], j--
    A[j+1] = v
```

# Question - 3

(a) + (b)

i. accesing the first element

| Array | Linked List |
|---|---|
| Using the index value, we can access the arry elements in constant time. $\Theta(1)$ , Space = $\Theta(1)$ | Traversing is needed. But for first element, only 1 operation will be done. $\Theta(1)$ , Space = $\Theta(1)$ |

ii. accesing the last element

| | |
|---|---|
| Same for i, $\Theta(1)$ , space = $\Theta(1)$ | Traversing continues until end of the list, so it takes $\Theta(n)$ , Space = $\Theta(1)$ |

iii. accesing any element in the middle

| | |
|---|---|
| Same for i and ii. Regardless of the index, accesing takes $\Theta(1)$, Space $\Theta$(approximately) | Traversing until the middle takes $\Theta(n)$ , Space = $\Theta(1)$ |

iv. adding a new element at the beginning

| | |
|---|---|
| Requires swaping from beginning to the end. Takes $\Theta(n)$ , Space = $\Theta(1)$ | No swap, no replace. Just create a node and assign it as head. $\Theta(1)$ , Space $\Theta(1)$ |

v. adding a new element at the end

| | |
|---|---|
| Assuming the array doesn't need to be resized, adding an elemt to end takes $\Theta(1)$ time. , Space = $\Theta(1)$ | Adding end requires traversing the whole list. So it takes $\Theta(n)$ times. Due to just one node hold, Space = $\Theta(1)$ |

vi. adding a new element in the middle

| | |
|---|---|
| Assuming resize doesn't need, from middle to end, array should be swaped. It takes $\Theta(n)$ , Space = $\Theta(1)$ | Half of the list should be traversing. $\frac{n}{2}$ is in linear class, so time comp. is $\Theta(n)$ , Space = $\Theta(1)$ |

vii. deleting the first element

| | |
|---|---|
| After deletion, swap from end to beginning is required. Takes $\Theta(n)$ time. , space = $\Theta(1)$ | Just a reference change is required in the head element, takes $\Theta(1)$ , Space = $\Theta(1)$ |

viii. deleting the last element

| | |
|---|---|
| Depends on method : should be both $\Theta(1)$ and $\Theta(n)$, but we assume it as $\Theta(1)$, Space $\Theta(1)$ | Just a reference change is required in the last element, takes $\Theta(1)$, Space = $\Theta(1)$ |

ix. deleting any element in the middle

| | |
|---|---|
| Swapping is required. So it takes $\Theta(n)$ time. , Space = $\Theta(1)$ | Just a reference change is required in that element, takes $\Theta(1)$ , Space = $\Theta(1)$ |

There are cases if array need to be resized, the space it doubles itself, even if just one element adding. But in here, I assumed that resizing is not required. And space requirement is decided as "How many extra space is required for that operation?" We all worked on "one" element. So, space comp. is all $\Theta(1)$ in all cases.

# Question - 4

**Approach:** A binary tree is an unsorted tree structure. We want to make it BST, which is same structure but in a sorted way. That means, we need to hold the elements' position, sort them, and put the sorted version to the current binary tree structure one by one. This "sorting" part is left to programmer, to which one he/she choose. Here, due to it is more efficient, I'll select merge sort.

**Pseudo-code:**

```
convertBinaryTree_BST(root)  # takes reference to the root node
    # first count how many node are there
    rootCopyL = root   # copy the root to not to lose it.
    while (rootCopyL is not None) do:
        leftNode++
        rootCopyL = rootCopyL.left
    rootCopyR = root
    while (rootCopyR is not None) do:
        rightNode++
        rootCopyR = rootCopyR.right
    totalNode = leftNode + rightNode
    # Store the binary tree's element into an array in inorder traversal
    # use helper function
    storeBT into Array(root, currentElement)
    # we have an unsorted array, sort it with merge sort.
    MergeSort(Array[0...(n-1)])
    if totalNode > 1
        copy Array[0...(n/2 - 1)] to leftArray[0...(n/2 - 1)]
        copy Array[(n/2)...(n-1)] to rightArray[0...(n/2 - 1)]
        MergeSort(leftArray[0...(n/2 - 1)])
        MergeSort(rightArray[0...(n/2 - 1)])
        Merge(leftArray, rightArray, Array)  # Merge is also helper func.
    # construct BST from sorted array
    constructBST(Array, root)  # it is a helper function
```

**Helper functions:**

```
storeBT into Array(root, curElement)
    if root is None
        return
    storeBT into Array(root.left, curElement)
    curElement.append(root.data)
    storeBT into Array(root.right, curElement)
```

```
Merge(leftA[0...p-1],
      rightA[0...q-1],
      A[0...p+q-1])
i=0, j=0, k=0
while i<p and j<q do
    if leftA[i] ≤ rightA[j]
        A[k] = leftA[i], i++
    else
        A[k] = rightA[j], j++
    k++
if i=p, copy rightA to A
else    copy leftA to A
```

```
constructBST(Array, root)
    if root is None:
        return
    else
        count++
        constructBST(Array, root.left)
        root.data = Array[count]
        constructBST(Array, root.right)
```

## Time Complexity Analysis:

Counting how many node are there,
storing binary tree into Array,
Constructing BST from binary tree
} all takes linear time.

In this algorithm, complexity determined by "sorting algorithm". We used "Merge sort". So let's analyze it.

$$C_{worst}(n) = 2C_{worst}\left(\frac{n}{2}\right) + n - 1$$

from the Master Theorem, $a=2$, $b=2$, $f(n) = n-1$

$$C_{worst}(n) = n \log n$$

Worst case occurs in case of until just one element remaining one of the arrays, other array doesn't becomes empty.

$$C_{Best}(n) = 2C_{Best}\left(\frac{n}{2}\right) + \frac{n}{2}$$

from the Master Theorem, $a=2$, $b=2$, $f(n) = \frac{n}{2}$

$$C_{Best}(n) = n \log n$$

Best case occurs while loop inside "Merge" helper function loops $\frac{n}{2}$ times, where Right or Left array is already almost sorted (always left[i] ≤ right[i] or vice versa.)

Due to best and worst case are same, average case also takes $n \log n$ times.

# Question - 5

If we use Hashing method, there are two alternatives for worst case:

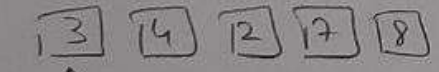
1) There might not be such a pair in the array
2) The pair might be the last one to be compared.

In Hashing, traversing the whole array becomes only once. So at most n comparisons will be make. Worst case is $O(n)$, and best case is where the first pair is provides the condition, $\Omega(1)$.

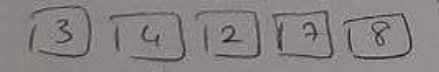
Algorithm:

```
findPairs(A, x)   # A : Integer array
                  # x : Integer (desired pair difference)
    dictionary ← { }   # first create a dictionary
    n ← length of the array
    for i from 0 to n:
        if x + A[i] is in dictionary then
            return (A[i], x + A[i])   # x + A[i] - A[i] = A[i]
        else if -x + A[i] is in dictionary then
            return (A[i], -x + A[i])   # -x + A[i] = A[i]
        else
            append A[i] to dictionary
        end if
    end for
    return -1   # if no such pair found, return -1
```

Solution: Array A = {3, 4, 2, 7, 8}  x = 1  dictionary = { }




↑ Is 1+3 in dictionary? → false → then append 3 to dictionary
~~-1+3 in dictionary? → false → then append 2 to dictionary~~



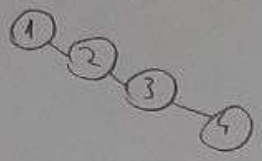

dictionary = {3, 2}

↑ Is 1+4 in dictionary? → false → then append 4 to dictionary
-1+4 in dictionary? → yes → then return (4, 3)

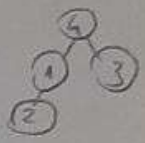
# Question - 6

(a) TRUE. Shape of a BST depends on the insertion order. Insertion order also affects the structure of the tree. for example, If we insert the elements in increasing order, all left nodes becomes empty and only right nodes will be filled, which leads a linked list creating and affects the performance. It is same for inserting in decreasing order, too. Instead of a sorted way, if same elements inserted in random order, shape of the BST will be different.

Insert 1, 2, 3, 4 →




Insert 4 1 3 2 →




(b) TRUE. In general, accesing an element takes logn time, even if that element becomes the last nodes. But, if a tree structure couldn't be implemented as it requires, a linked list similar structure can be occur, which leads to accesing takes linear time.

(c) If we think we didn't construct the array, and ask for input for each element, the answer is TRUE Consider: You asked "enter elements of ary:" to user, and user enters 5. After that user enters 8, and since 8 > 5, your max element will be updated. This goes on and finally, you'll find max only without both any extra storage and time consuming. But If the ary is proper and we'll search on that ary, the answer is FALSE. Since we have to traverse all array for comparing. This is the more general answer since question assumes an array is exist.

(d) FALSE. first, linked list should be sorted. Even if it is sorted, accessing an element didn't takes constant time in linked list. Each time it is required to traverse the list, which takes $O(n)$ time. And in binary search, in each time we need to access middle element.

(e) FALSE. If array reversely sorted, insertion sort takes $O(n^2)$ times in worst case. According to algorithm:

```
for i:= 1 to n-1 do
    v=A[i], j=i-1
    while j≥0 and A[j] > v do
        A[j+1] = A[j], j--
    A[j+1] = v
```

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i$$

$$= \frac{(n-1)n}{2} \in O(n^2)$$