Süleyman Gölbol
1801042656

1) Master Theorem

Let $T(n)$ be an eventually non decreasing function that satisfies the recurrence relation.

$$T(n) = a.T(\tfrac{n}{b}) + f(n), \quad a = b^k \ (k = 1,2,3....) \text{ and } T(1) = c$$

where $a \geq 1, b \geq 2, c > 0$

$$f(n) = \Theta(n^k \log^p n)$$

$$T(n) \in \begin{cases} \text{case1:} \\ \text{if } \log_b a > k \text{ then } \Theta(n^{\log_b a}) \\ \text{Case2:} \\ \text{if } \log_b a = k \\ \quad \text{if } p > -1 \ \ \Theta(n^k \log^{p+1} n) \\ \quad \text{if } p = -1 \ \ \Theta(n^k . \log\log n) \\ \quad \text{if } p < -1 \ \ \Theta(n^k) \\ \text{case3:} \\ \text{if } \log_b a < k \quad \text{if } p \geq 0 \ \ \Theta(n^k . \log^p n) \\ \quad\quad\quad\quad\quad \text{if } p < 0 \ \ O(n^k) \end{cases}$$

a) $a=2, b=4, k=p=\frac{1}{2}$ $\quad \underline{a=b^k}$ and $p>-1$ So $T(n) = \Theta(n^{\log_4 2} . \log^{(\frac{1}{2}+1)} n) = \Theta(n^{\frac{1}{2}} . \log^{\frac{3}{2}} n)$
$k = \log_b a$

b) $a=9, b=3, k=2, p=0$ $\quad a=b^k$ and $p>-1$ So $T(n) = \Theta(n^{\log_3 9} . \log^{0+1} n) = \Theta(n^2 \log n)$

c) $a=\frac{1}{2}$ is not ≥ 1 so master theorem cannot be applied.

d) $a=5, b=2, k=0, p=1$
$a > b^k$ so $T(n) = \Theta(n^{\log_2 5})$

e) Exponentials doesn't fit with master theorem so cannot be solved.

f) $a=7, b=4, k=1, p=1$
$a > b^k$ so $T(n) = \Theta(n^{\log_4 7})$

g) $a=2, b=3, k=-1$
k is not ≥ 0 so cannot be solved with master theorem.

h) $a=\frac{2}{5}, b=5, k=5, p=0$ $\quad a \ngeq 1$ So cannot be solved with master theorem.

2)
```
for i in range(1, len(A)):      # Start from 2nd element of array, continue until last.
    element = A[i]              # Saving element in ith index to variable so when
                                # array order changes, it will stay the same.
    j = i-1                     # Every time starting j with i-1 so,
    while (j>=0 and A[j]>element):  # So we can compare with previous element
        A[j+1] = A[j]           # while it's bigger than element.
        j -= 1                  # if comparison holds true, move element to left
    A[j+1] = element            # Put element to it's right place.
```

→ 3 2 6 1 4 5
→ 2 3 6 1 4 5
→ 2 3 1 6 4 5
→ 2 1 3 6 4 5
→ 1 2 3 6 4 5
→ 1 2 3 4 6 5
→ 1 2 3 4 5 6

Süleyman 1801042684

3)

a)

i. Linked list holds first element. So, no iteration needed. $O(1)$
Array elements can be accessed through index so $O(1)$.

ii. If linked list doesn't hold it's tail as pointer we need to iterate through all. So $O(n)$
Array elements are indexed. So doesn't matter where, it's $O(1)$.

iii. Even element is in the middle, accessing will be iterating $\frac{n}{2}$ times for linked list, So $O(\frac{n}{2}) = O(n)$
Array elements are indexed. So doesn't matter where, it's $O(1)$.

iv. For linked list, we need to change the pointer of head element to new head without losing other data pointers. $O(1)$
For array, we need to create a new array / move every element so $O(n)$.

v. For linked list, we need to iterate until the end, then add. It's $O(n)$.
For array, because no extra space allocated, we need to create a new array, move all and then add. So $O(n)$

vi. For linked list, we need to iterate half of list, so $O(\frac{n}{2}) \Rightarrow O(n)$
For array, we need to create a new array, move elements until middle, then add new element, and then add rest of it. So $O(n)$

vii: For linked list, moving head to second element and deallocating the first is enough. So, $O(1)$
For array, creating a new array and copying from one to other will take $O(n)$ time.

viii. For linked list, moving iterator till the end, then deallocating is enough so it's $O(n)$.
For array we can just ignore last element. But for getting array without the last element, we need to move every element except last. $O(n)$

ix. Both similar to viii, we need to iterate over elements so it will take $O(n)$ for array and linked list.

b) for extra space needed,

i. No need extra space.
ii. Linked list, we need to create an iterator size of pointer. For array nothing needed.
iii. Linked list needs an iterator to create, for array nothing needed.
iv. Linked list - nothing needed; array - we need to create an array size of n.
v. Linked list - one element needed; array - n element needed.
vi. Linked list - one element needed; array - n element needed.
vii. Linked list - nothing is needed; array - n element needed.
viii. Linked list - nothing is needed; array - n element is needed.
ix. Linked list - nothing is needed; array - n element is needed.

(4)

```
def BinaryTreeToBST(root, n):
    if (root is None):                      # Checking if tree is empty
        return

    tempArray = []
    inorderStorer(root, tempArray)          # Storing values inorderly in an array
    tempArray.sort()                        # Sorting the array (python quicksort)
    convertToBST(tempArray, root)           # Sending from array to BST inorderly

def inorderStorer(root, stored):            # Appending all values of tree to array
    if root is None:                        # inorderly so we can take it back as BST
        return
    inorderStorer(root.left, stored)
    stored.append(root.data)
    inorderStorer(root.right, stored)

def convertToBST(tempArray, root):
    if root is None:                        # Checking if value to add empty recursively
        return
    convertToBST(tempArray, root.left)      # First performing this operation for leftside
    root.data = tempArray[0]                # Add first value to BST, then remove value
    tempArray.pop(0)                        # from array
    convertToBST(tempArray, root.right)     # Perform this for right side
```

bk

= 0

d

b^d

b^d

Time Complexities

Inorderly Storing all n values will take $O(n)$ time. (every case)

- In the worst case, quicksort will sort using pivot which is smallest or largest element of the array. This will happen when input array is already sorted so pivot will be first or last element. So it will take $O(n^2)$ time. At the end, inorderly copying array elements to tree nodes will take $O(n)$ time. (every case)
$O(n+n^2+n) \Rightarrow O(n^2)$
- In the best case, when we partitioned the array to sort, when they are very evenly balanced e.g their size are equal. That makes a balanced binary tree for quicksort. Because of binary tree have height of $(\log n)$, time complexity will be $O(n\log n)$
$O(n+n\lg n+n) = O(n\lg n)$
- In the average case, because average case comparisons required by is recurrence relation, $T(n) = \Theta(n) + 2*T(\frac{n}{2})$. If we use master theorem, because a is 2, b is 2 $\Theta(n^{\log_2 2}) = \Theta(n)$. Every time $\frac{n}{2}$ is makes $\log n$ so at the end $\Theta(n\log n)$.

$\Theta(n + n\lg n + n) = \Theta(n\lg n)$.

smaller than $n\lg n$ so we can ignore.

```
def sort(array, low, high) : #Quicksort
    if low < high:          # Find pivot element and smaller will be put on left
                            # greater will be put in right

        part = partition (array, low, high) # Have partition on array
        sort (array, low, part - 1)         # Operation for left side of pivot
        sort (array, part + 1, high)        # Operation for right side of pivot

def partition (array, low, high] :
    pivot = array [high]            # Element at the right end will be pivot
    i = low - 1                     # bigger element pointer

    for j in range (low, high) :
        if (array [j] <= pivot):
            i = i + 1               # Swap smaller with bigger if found
                                    # but first increment i number

            (array [i], array [j])) = (array [j], array [i])
    (array [i+1], array [high]) = (array [high], array [i+1])
    # Swapping pivot with bigger element
    return i+1                      # Return end of partition value.
```

[illegible]
[illegible]

⑤
```
def pairFinder():
    array = [5, 15, 23, 18, 7, 22]      # Example array
    x = 5                               # Example x

    set = set()                         # Creating empty set so every element
                                        # will be stored there to compare
                                        # with i+x and i-x's absolutes.

    for i in array:                     # For every (element - X), check
        if abs(i-x) in set:             # if that value is already
            print(i, " and ", i-x)      # seen so we can print a
        if abs(i+x) in set:             # valid pair
            print(i+x, " and ", i)
        set.add(i)
```

⑥
a) True.
Because root node value also changes other values' insertion place. (If smaller than root, it will be at left; if bigger, it will be at right.) Every added item will depend on its root for left-hand/right-hand place.

b) True
If tree is skewed, then we will go to one side only, so instead logn, we will get O(n).

c) True
Array's indexes are obvious. No need to iterate to compare any element of array with other element of array.

d) false
If linked list is a singly linked list and we want to examine the last node, we need to perform n-1 operations to access next pointer, Memory allocation is not contiguous so it will take O(n) to find middle element.

e) False.
worst case occurs when array is sorted in reversely order. Because, there are 2 loops. Inner loop iterates from current position to the first element. outer will iterate no matter what so it will take O(n²).