Problem 1 Order the following list of functions by the big-Oh notation. Group together (for example, by underlining) those functions that are big-Theta of one another.

4^n	n^3	$n^2 \log n$	$4^{\log n}$	$\sqrt{\log n}$
2^{2^n}	$\lceil \sqrt{n} \rceil$	$n^{0.01}$	$1/n$	$4n^{3/2}$
$3n^{0.5}$	$5n$	$\lfloor 2n \log^2 n \rfloor$	2^n	$n \log_4 n$
$6n \log n$	2^{100}	$\log \log n$	$\log^2 n$	$2^{\log n}$

Hint: When in doubt about two functions $f(n)$ and $g(n)$, consider $\log f(n)$ and $\log g(n)$ or $2^{f(n)}$ and $2^{g(n)}$.

Solution

Order Of Complexities:-

1. O(1 / n) <
2. $O(2^{100})$ <
3. log logn <
4. $O(\sqrt{\log n})$ <
5. $O(\log^2 n)$ <
6. $O(n^{0.01})$ <
7. $O(\text{ceil}(\sqrt{n})) = O(3n^{0.5}) = \text{big-Theta}(\sqrt{n})$ <
8. $O(2^{\log n}) = O(5n) = \text{big-Theta}(n)$ <
9. $O(n \log_4 n) = O(6n \log n) = \text{big-Theta}(n \log n)$ <
10. floor(2n $\log^2 n$)
11. $O(4n^{3/2})$ <
12. $O(4^{\log n}) = \text{big-Theta}(n^2)$ <
13. $O(n^2 \log n)$ <
14. $O(n^3)$ <
15. $O(2^n)$ <
16. $O(4^n)$ <
17. $O(2^{2^n})$

1. n^3 $\quad$ $4^{\log n} = x$ $\quad$ $2^{\log n}$
   n^3 $\quad$ $2^{2 * (\log n)} = x$ $\quad$ $2^{\log n} = x$
   n^3 $\quad$ $\log^2 n = \log x$ $\quad$ $\log n = \log x$
   n^3 $\quad > \quad$ $n^2 = x$ $\quad > \quad$ $n = x$

2. $n \log_4 n = (n * \log_2 n) / (\log_2 4) = (n * \log_2 n) / 2$ which is equal to O(nlogn) by big oh notation.

3. 2^{100} = some constant = O(1)

4. $\log(n^{0.01})$ $\quad$ $\log \sqrt{\log n}$
   $0.01 * \log(n)$ $\quad$ 0.50 log(log(n)) that shows $O(\sqrt{\log n}) < O(n^{0.01})$

5. $x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1$ by floor and ceiling property. So O(floor(X)) = O(ceil(x)) = O(X).

Problem 2 Matrix multiplication. Write pseudocode for computing C = A × B, where A, B, C are matrices of integers of size n × m, m × k, and n × k respectively.
Analyze the running time of your algorithm.
Solution

matrix_a, matrix_b list are indexed from 0 to n - 1 here function **matrix_multiplication** takes two matrices as arguments in list form and gives the result of it in the list form.

```
1   def matrix_multiplication(matrix_a: [], matrix_b: []):
2       if matrix_a is None or matrix_b is None or len(matrix_a) == 0 or
len(matrix_b) == 0 or len(matrix_a[0]) != len(matrix_b):
3           return Exception("matrix multiplication not allowed")
4       matrix_c = []

        # for loop will run for i = 0 to len(matrix_a) - 1 which is n =
number of rows in the matrix_a
5       for i in range(0, len(matrix_a)):
6           row = []

            # for loop will run for j = o to len(matrix_b[0]) which is k =
number of columns in the matrix_b
7           for j in range(0, len(matrix_b[0])):
8               row.append(0)
9           matrix_c.append(row)

        # for loop will run for i = 0 to len(matrix_a) - 1 which is n =
number of rows in the matrix_a
10      for i in range(0, len(matrix_a)):

            # for loop will run for k = 0 to len(matrix_b[0]) which is k =
number of columns in the matrix_b
11          for k in range(0, len(matrix_b[0])):
12              matrix_c[i][k] = 0

                # for loop will run for j = 0 to len(matrix_a[0]) which is
m = number of columns in the matrix_a = number of rows in the matrix_b
13              for j in range(0, len(matrix_a[0])):
14                  matrix_c[i][k] += matrix_a[i][j] * matrix_b[j][k]
15      return matrix_c
```

From 5 to 9 we have for loop inside for loop first run n times and second run k times so total complexity here will be O(n * k)

From 11 to 14 we have 3 for loops that are nested and first for loop run n times second for loop run k times and third for loop run m times, so total complexity here will be O(n * k * m)

All other lines have constant time complexity so for them, time complexity will be O(1).

Cumulatively O(n * k * m) is the bottleneck here so **matrix multiplications time complexity will be O(n * k * m)** here.

Problem 3 You are given a tree represented by the following data structure. Nodes have three fields: element, left-child, and right-child. The element is an integer.
Write a procedure (pseudocode) with the input parameter of the location of the root node (or a pointer to the root node), and that computes the average value of the elements in the tree. However, do not use recursion. Use a stack instead.
Analyze the running time of your algorithm. As proven in CS 331, stack operations are O(1).
Solution

```
class Tree:
   def __init__(self, element=0, left_child=None, right_child=None):
       self.element = element
       self.left_child = left_child
       self.right_child = right_child
```

Here **average** function takes the location of the root node of a binary tree as an argument and returns the average of all elements in the tree.

```
1.  def average(root: Tree):
2.      if root is None:
3.          return Exception('Tree not available')
4.      ptr = root
5.      stack = []
6.      total = 0
7.      count = 0
8.      while ptr is not None or len(stack) != 0:
9.          while ptr is not None:
10.             stack.append(ptr)
11.             ptr = ptr.left_child
12.         ptr = stack.pop()
13.         total += ptr.element
14.         count += 1
15.         ptr = ptr.right_child
16.     return total / count
```

Here let’s say n = number of nodes in the tree

In the above solution steps, 2 to 7 are single operations and are not loop so all of them will have constant time complexity let’s say O(c) where c is constant, which will be equivalent to O(1).

Steps 8 to 15 contain 2 while loops where one is inside another while loop. As we can see, the outer while loop at the 8th position to the 15th position will run for all stack elements, which are equivalent to the number of nodes in the tree (n). But inner while loop 9th position to 11th

position will only run for the left element of a node in the tree. Let’s consider every iteration of the outer loop as following:-

1st iteration:-
9th position to 11th position Let’s say Inner loop run for k1 loops so complexity O(k1)
12th position to 15th position run for O(1)
.
.
.
Nth iteration:-
9th position to 11th position Let’s say Inner loop run for kn loops so complexity O(kn)
12th position to 15th position run for O(1)

So total time complexity for 11th position to 15th position will be:-
O(k1 + k2 + k3 + ... + kn) + O(n * 1)
Here k1 + k2 + k3 + . . . + kn will be equivalent to the total number of left nodes present in a tree which is < n.
    so k1 + k2 + k3 + . . . + kn < n
    Ki:- number of loops run by the inner while loop
    N:- number of nodes in the tree
Therefore total complexity will be O(n).

Problem 4 Give pseudocode for the following problem: given an array A[1 . . . n] containing the numbers 1, 2, . . . , n and representing a permutation, modify A to represent the next lexicographic permutation. Analyze running time (which should not exceed O(n)) and argue correctness (why the output is lexicographically bigger than the input? why other permutations lexicographically bigger than the input are bigger than the output?).

Solution

The **next_permutation** function takes the list as an argument and returns the list which is the next permutation of the input list if the next permutation is not possible then it returns an exception.

```
1.  def next_permutation(nums: [int]):
      # here for loop will go from n - 1 to 1
      # here for loop will run for k time and max of k = n - 1
2.    for i in range(len(nums) - 1, 0, -1):
3.         if nums[i] < nums[i + 1]:
4.             replacer = i
               # find_closest function for loop will run for max k - 1
times
5.             closest = find_closest(nums[i + 1:], nums[i])
6.             nums[i], nums[i + closest] = nums[i + closest], nums[i]
               #  reverse function for loop will run for (n - k + n) / 2
times which is about n - k / 2 times
7.             reverse(nums, i + 1)
8.             return nums
9.    return Exception("Already Permutation is at max can't generate next
permutation")
```

The **reverse** function takes a list and a start point from where we want to reverse a list and return a list with the reversed order from the start position to the end position without changing any data before the start position in the list. Here list index will be in form 1 to n.

```
10. def reverse(nums: [int], start: int):
11.    len_array = len(nums)
12.    mid = (start + len_array) // 2

       # for loop will run from i = start to mid from the start point
13.    for i in range(start, mid):
14.        nums[i], nums[len_array - i + start] = nums[len_array - i +
start], nums[i]
15.    return nums
```

P.T.O.(Please Turn Over)

The **find_closest** function will find the smallest value in list which is bigger than replacer_value

```
16. def find_closest(nums: [int], replacer_value: int):
17.     closest = 1
    # for loop will run from i = 2 to n
18.     for i in range(2, len(nums) + 1):
19.         if nums[closest] >= nums[i] > replacer_value:
20.             closest = i
21.     return closest
```

Analysis of algorithm:-

In the **next_permutation** function there is a loop from 2nd position to 8th position. Let's say that run for k times so find the replacer.

After finding the replacer value we can find the closest value in k - 1 times as only there we will be finding closest in **find_closest** function.

After finding closest we will replace values at 6th position which take O(1) time.

After that to reverse the list from the k + 1 th position it will take (k + 1 + len(Array)) / 2 time.

So we can see above next_permutation algo in iteration:-
1st iteration:-
Time complexity O(1) as only comparison happens

Kth iteration:-(let's assume here we find the replacer)
Time complexity of **find_closest + swap(A[i], A[i + closest] + reverse**, which will be O(k - 1) + O(1) + O((k + 1 + len(Array)) / 2) = O(k + (k + 1 + n) / 2) that is O(n) as we can ignore constants.

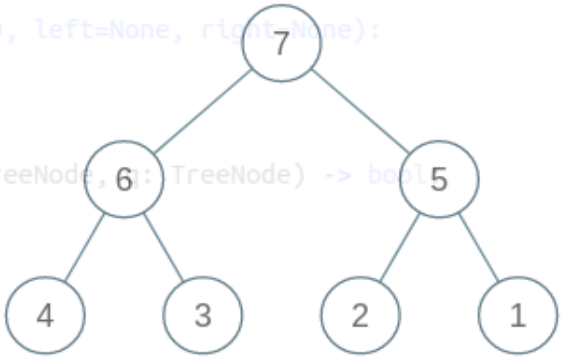
So all other iterations other than the kth iteration will be O(1) so total time complexity will be O(k - 1) + O(n) so total complexity of algo will be O(n) as k < n.

P.T.O.(Please Turn Over)

Problem 5 What is the running time of HEAPSORT on an array A that is sorted in decreasing order? Use an Ω() approximation and justify your answer.

Solution

Array A is already sorted in decreased order here. So Let's say array A is 7, 6, 5, 4, 3, 2, 1. So Array A already has the shape of the max heap so when calling Heapsort(A) first we will call Build_Max_Heap(A).



In Build_Max_Heap(A) there we will call Max_Heapify(A, i) for A.length / 2 times which is n / 2 where n = no. of nodes in the tree

In Max_Heapify(A, i) functions worst conditions Recursive Call in Max_Heapify(A, largest) can go to $\log_2 n$ times but here because the heap is already max-heap there will be no recursive call to Max_Heapify(A, largest)

So time complexity of the Build_Max_Heap(A) function in the current scenario will be omega(n / 2) which is omega(n).

But in steps 2. To 3. where we will run a for loop to extract max value from heap Max_Heapify(A, i) will run for $\log_2 n$ times because when we exchange A[1] to A[i] tree in no more in max heap condition and we will require to run Max_Heapify(A, i) which take time about $\log_2 n$. So total complexity of steps 2. To 3. Will be omega(n $\log_2 n$).

Hence the complexity of all HeapSort will also be omega(n $\log_2 n$) when the array is already sorted in decreasing order.

In HeapSort we have following steps to sort array:-

Heapsort(A)

1. Build_Max_Heap(A)
2. For i = A.length downto 2
3. Exchange A[1] to A[i]
4. A.heap_size = A.heap_size - 1
5. Max_Heapify (A, 1)

Build_Max_Heap(A)

1. A.heap_size = A.length

```
   2. For i = floor(A.length / 2) downto 1
   3.      Max_Heapify(A, i)

Max_Heapify(A, i)
   1. largest = i
   2. left = 2 * i
   3. left = 2 * i + 1
   4. If left <= A.heap_size and A[left] > A[largest]
   5.      largest = left
   6. If right <= A.heap_size and A[right] > A[largest]
   7.      largest = right
   8. If largest != i
   9.      Exchange A[i] to A[largest]
   10.     Max_Heapify(A, largest)
```

End