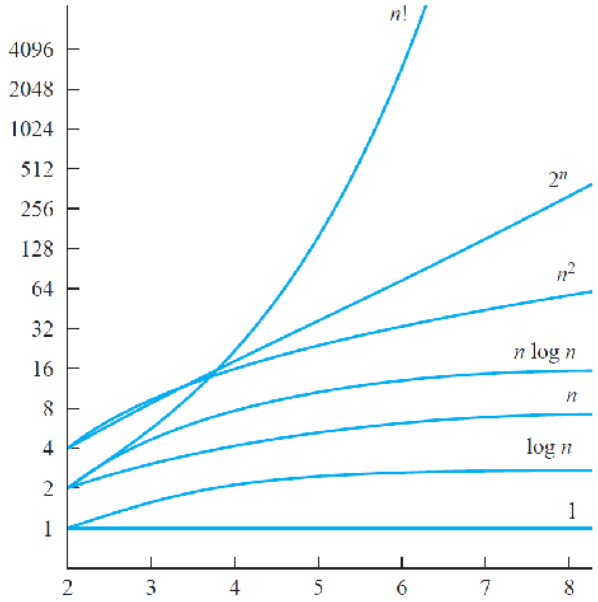
CSCI 570 – ANALYSIS OF ALGORITHMS HOMEWORK – 1

1. Arrange these functions under the Big-$\mathcal{O}$ notation in increasing order of growth rate with $g(n)$ following $f(n)$ in your list if and only if $f(n) = \mathcal{O}(g(n))$ (here, $\log(x)$ is the **natural logarithm**[1] of x, with the base being the Euler's number e) :

$$2^{\log(n)}, 2^{3n}, 3^{2n}, n^{n\log(n)}, \log(n), n\log(n^2), n^{n^2}, \log(n!), \log(\log(n^n)).$$

ANS:



Hierarchies of Functions

We need to list all these terms (functions), in the order of their increasing growth rate for Big O. One of the best possible ways to determine the order is by comparing the given terms or functions with the standard hierarchies of functions (figure above). Although we can create abstract classes like $\sqrt{n}$ if required which are not a part of our standard diagram.

By comparing the given functions according to Standard Hierarchies of Functions above, we can conclude that there are no functions with $O(1)$

We have $log(n)$ which is the 2nd in hierarchies of functions for worst-case time complexity (O). Let us start with $log(n)$ in the output order and compare the similar functions which fall under $O(log(n))$ order or level

$\boldsymbol{log(n)}$

The last term in the given series of terms (functions) which can be written as $log(n(log(n)))$. *As* $log(n(log(n))) > log(n))$ it will be our second term in our output

$\boldsymbol{=> log(n) < log(log(n^n))}$

To determine the next term (function) in our output order, let us find out if we have any terms in the order of n. If we solve $\boldsymbol{2^{log\,(n)}}$ we get,

$$2^{\log_e(n)} = 2^{\log_2(n)} . 2^{\log_e(2)}$$

The value of $log_e(2)$ is equal to 0.693147. We can denote this with a constant c

$$2^{\log_e(n)} = n.c \approx O(n)$$

$\boldsymbol{=> log(n) < log(log(n^n)) < 2^{log\,(n)}}$

Moving on to the next standard level in hierarchies of functions, let us find out if we have any terms in the order of $n(\log(n))$

As $n \text{ –> } \infty$, $log\ (n!) \approx n^* log(n)$ and $n\ log\ (n^2)$ *is represented as*

$2n^* log(n)$

$\boldsymbol{=> log(n) < log(log(n^n)) < 2^{log\,(n)} < log(n!) < n\ log\ (n^2)}$

Now, we need to find out if we have any terms in the order of 2^n

We have 2^{3n} and 3^{2n} which can be represented as 8^n and 9^n respectively. This clearly indicates $3^{2n} > 2^{3n}$

$$\Rightarrow \boldsymbol{log(n) < log(log(n^n)) < 2^{log\,(n)} < log(n!) < n\ log\ (n^2) < 2^{3n} < 3^{2n}}$$

Finally, we are left with 2 terms ($n^{n\,log\,(n)}$ and n^{n^2}) in the order of n^n. Let us use comparison method to determine the final order of growth rate for the given terms

Applying log on both sides

$$n\big(log(n)^{*}log(n)\big) < n^2 * log(n)$$

We have $n^{n\log(n)}$ and n^{n^2} which can be represented as $n\big(log(n)^{*}log(n)\big)$ and $n^2 * log(n)$ respectively

This clearly indicates $n^{n^2} > n\big(log(n)^{*}log(n)\big)$

This concludes our comparisons. Therefore, the final increasing order of growth rate of given functions under Big-O notation is as follows

$$\Rightarrow \boldsymbol{log(n) < log(log(n^n)) < 2^{log\,(n)} < log(n!) < n\ log\ (n^2) < 2^{3n} < 3^{2n} < n^{n\,log\,(n)} < n^{n^2}}$$

2. Show by induction that for any positive integer k, $(k^3 + 5k)$ is divisible by 6.

ANS:

INDUCTIVE HYPOTHESIS:

To prove that $k^3 + 5k$ *is divisible by* 6 *for any value of* k *i.e.*$k \in +ve$ *Integers*

BASE CASE:

Let us assume k = 1

$k^3 + 5k = (1)^3 + 5(1) = 1 + 5 = 6$

6 is exactly divisible by 6 and the hypothesis stands true for $k = 1$

Assume the formula is true for $k = x\ ;\ x \in +ve\ Integers$

$=>(x)^3 + 5(x) = 6\ (q)$ where q is quotient when we divide $(x)^3 + 5(x)$ by 6

Now let us check if the value holds true for $k = x + 1;\ x \in +ve\ Integers$

$= (x + 1)^3 + 5(x + 1)$

$= x^3 + 3x^2 + 3x + 1 + 5x + 5$

$= \boldsymbol{x^3} + \mathbf{5}\boldsymbol{x} + 3x^2 + 3x + \mathbf{1} + \mathbf{5}$ (From inductive hypothesis $(x)^3 + 5(x) = 6\ (q)$

$= 6\ (q) + \mathbf{3}\boldsymbol{x^2} + \mathbf{3}\boldsymbol{x} + 6(1)$

$= 6\ (q) + \mathbf{3}(\boldsymbol{x^2} + \boldsymbol{x}) + 6(1)$

$= 6\ (q) + \mathbf{3}(\boldsymbol{x}(\boldsymbol{x} + \mathbf{1})) + 6(1)$

➔ $x, x + 1 \in +ve\ Integers$

If we multiply any two $+ve$ integers the result is an even number (i.e., divisible by 2). According to the divisibility rule for 6 if any integer is divisible by 2 and 3 then it is divisible by 6. As the expression $\mathbf{3}(\boldsymbol{x}(\boldsymbol{x} + \mathbf{1}))$ **is a multiple of 2 and 3 it is perfectly divisible by 6. As all the terms in the expression** $6\ (q) + \mathbf{3}(\boldsymbol{x}(\boldsymbol{x} + \mathbf{1})) + 6(1)$ is divisible by 6 and the sum is divisible by 6.

$Hence, proved\ that\ k^3 + 5k\ is\ divisible\ by\ 6\ for\ any\ value\ of\ k\ i.e.,$
$k \in +ve\ Integers$

3. Show that $1^3 + 2^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$ for every positive integer n using induction.

INDUCTIVE HYPOTHESIS:

$To\ prove\ that\ 1^3 + 2^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}\ \ for\ n \rightarrow\ n \in +ve\ Integers$

BASE CASE: Let us assume $n = 1$

$=> 1^3 = \frac{1^2(1+1)^2}{4}$

$=> 1 = \frac{1(2)^2}{4}$

$=> 1 = \frac{4}{4}$

L.H.S = R.H.S and the hypothesis stands true for $n = 1$

Assume by inductive hypothesis that the formula is true for $n = k$; $k \in +ve\ Integers$

i.e., $1^3 + 2^3 + \cdots + k^3 = \frac{k^2(k+1)^2}{4}$

Now let us check if the value holds true for $n = k + 1 \rightarrow k \in +ve\ Integers$

$$1^3 + 2^3 + \cdots + k^3 + (k+1)^3 = \frac{(k+1)^2((k+1)+1)^2}{4}$$

=> L.H.S = $\frac{k^2(k+1)^2}{4} + (k+1)^3$

=> $\frac{k^2(k+1)^2}{4} + \frac{4(k+1)^3}{4}$

Taking common terms from numerator and denominator, we get

=> $\frac{(k+1)^2(k^2+4k+4)}{4}$

=> $\frac{(k+1)^2(k+2)^2}{4}$

=> $\frac{(k+1)^2((k+1)+1)^2}{4}$ = R.H.S

$$Hence\ proved\ 1^3 + 2^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4} when\ n \in +ve\ Integers$$

4. Consider the following prime filtering algorithm that outputs all the prime numbers in $2, \ldots, n$ (the pseudo code is presented in Algorithm 1).
   - Please prove this algorithm is correct (that is, a positive integer k that $2 \leq k \leq n$ is a prime if and only if $isPrime(k) = \textbf{True}$).
   - Please calculate the time complexity under the Big-$\mathcal{O}$ notation.

Algorithm 1 Prime Filtering

```
1: Input: a positive integer n ≥ 2
2: initialize the Boolean array isPrime such that isPrime(i) = True for i = 2,...,n
3: for i = 2...n do
4:    for j = 2...⌊n/i⌋ do
5:       if i × j ≤ n then
6:          isPrime(i × j) ← False
7:       end if
8:    end for
9: end for
```

The given prime filtering algorithm finds all the prime numbers up to the given $+ve$ integer n. It iteratively marks the multiples of each prime number $\leq n$ as non-prime.

Let us prove this algorithm is correct i.e., $k \in +ve$ integer such that $2 \leq k \leq n$ is a prime if and only if ($isPrime(k)$ = True)

ASSUMPTION BY PROOF OF CONTRADICTION:

The algorithm marks at least one prime as non-prime (Case 1) and marks at least one non-prime as prime (Case 2).

Case 1:

For a prime number k such that $2 \leq k \leq n$ the algorithm has marked ($isPrime(k)$ = False)

According to the given algorithm, $isPrime(k)$ = True is an initialization for all values of k such that $2 \leq k \leq n$. The only possibility for $isPrime(k)$ = False is when k can be written as $k = i \text{x } j$ $(i, j \in +ve\ integers\ 2 \leq i, j \leq n)$

This contradicts the primary definition of a prime number which states that a prime number is only divisible by 1 and itself. Hence, our assumption is proved to be wrong in Case 1.

Case 2:

For a non-prime number k such that $2 \leq k \leq n$ the algorithm has marked ($isPrime(k)$ = True)

According to the given algorithm, $isPrime(k)$ = True is an initialization for all values of k such that $2 \leq k \leq n$. The only possibility for $isPrime(k)$ to remain True is when k cannot be represented as $k = i$x j $(i, j \in +ve\ integers\ and\ 2 \leq i, j < k)$

=> k does not have any factors between 2 (inclusive) and k (exclusive) => k is only divided by 1 and itself which makes it a Prime number. Hence, our assumption is proved to be wrong as k turned out to be a prime number in Case 2.

Time Complexity under Big O Notation:

initialize the Boolean array $isPrime$ such that $isPrime(i)$ = True for i = 2, . . . , n

Initializing the $isPrime$ array takes $O(n)$ time, where n is a $+ve$ integer

for i = 2 . . . n do – Outer Loop

The outer loop iterates from 2 to n (inclusive), so it runs n - 1 times = $O(n-1)$.

for j = 2 . . . $\left\lfloor \frac{n}{i} \right\rfloor$ do – Inner Loop

The inner loop iterates from 2 to n/i, where i is the current value of the outer loop variable. On average the inner loop will run for n/i times for each value of i. Therefore, the total number of iterations for the inner loop can be approximated as follows

$[\frac{n}{2} + \frac{n}{3} + \frac{n}{4} \cdots + \frac{n}{n}]$ => $n\ [\frac{1}{2} + \frac{1}{3} + \frac{1}{4} \cdots + \frac{1}{n}]$

=> Using the approximation for the harmonic series, we get ln (n) + constant

So, the time complexity of the given algorithm is $\approx \boldsymbol{O(nlogn)}$

5. Amy usually walks from Amy's house ("H") to SGM ("S") for CSCI 570. On her way, there are six crossings named from A to F. After taking the first course, Amy denotes the six crossings, the house, and SGM as 8 nodes, and write down the roads together with their time costs (in minutes) in Figure 1. Could you find the shortest path from Amy's house to SGM? You need to calculate the shortest length, and write down all the valid paths.

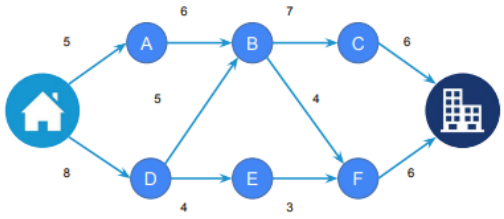


Figure 1: Problem 4's Graph

ANS:

SHORTEST PATHS:

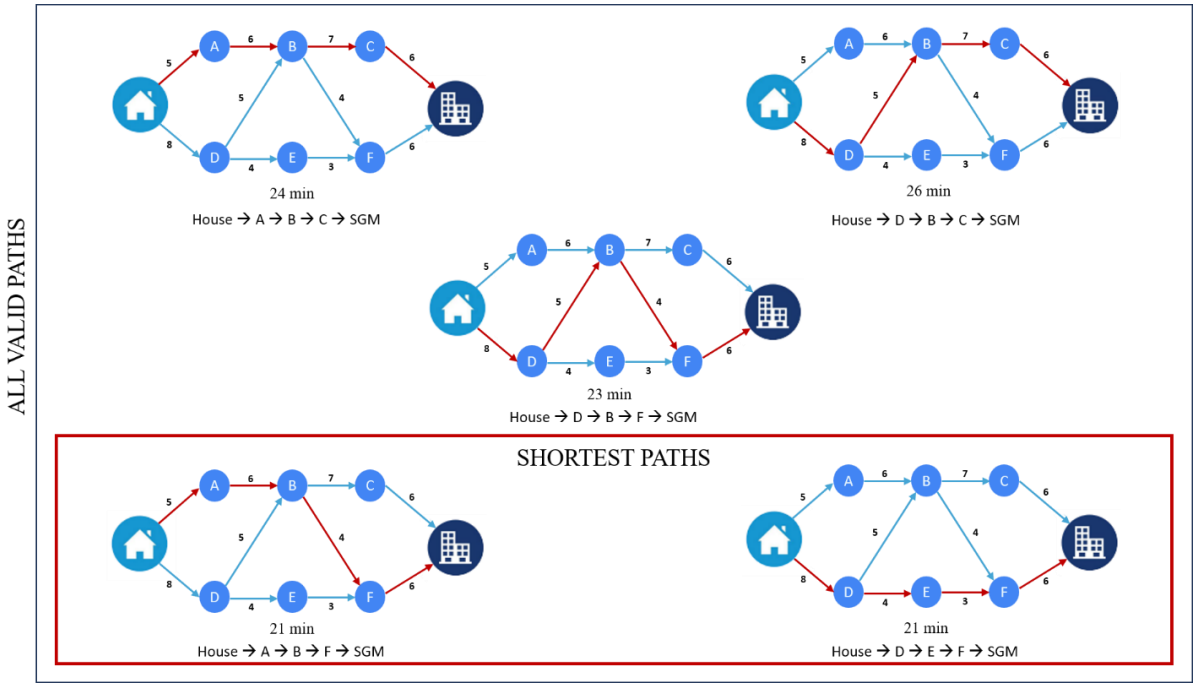
H → A → B → F → S & H → D → E → F → S = 21 min

OTHER VALID PATHS:

H → A → B → C → S = 24 min

H → D → B → C → S = 26 min

H → D → B → F → S = 23 min



6. According to the Topological Sort for DAG described in Lecture 1, please find one possible topological order of the graph in Figrue 2. In addition, could you find all the possible topological orders?

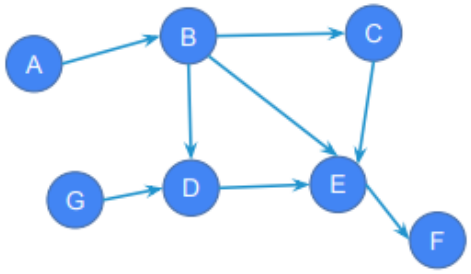


Figure 2: Problem 5's Graph

ANS:

TOPOLOGICAL SORT:

STEP 1: Select a vertex that has **"0"** In-Degree

STEP 2: Add the vertex to the output array

STEP 3: Remove the vertex and its outgoing edges from the graph

STEP 4: Repeat the process

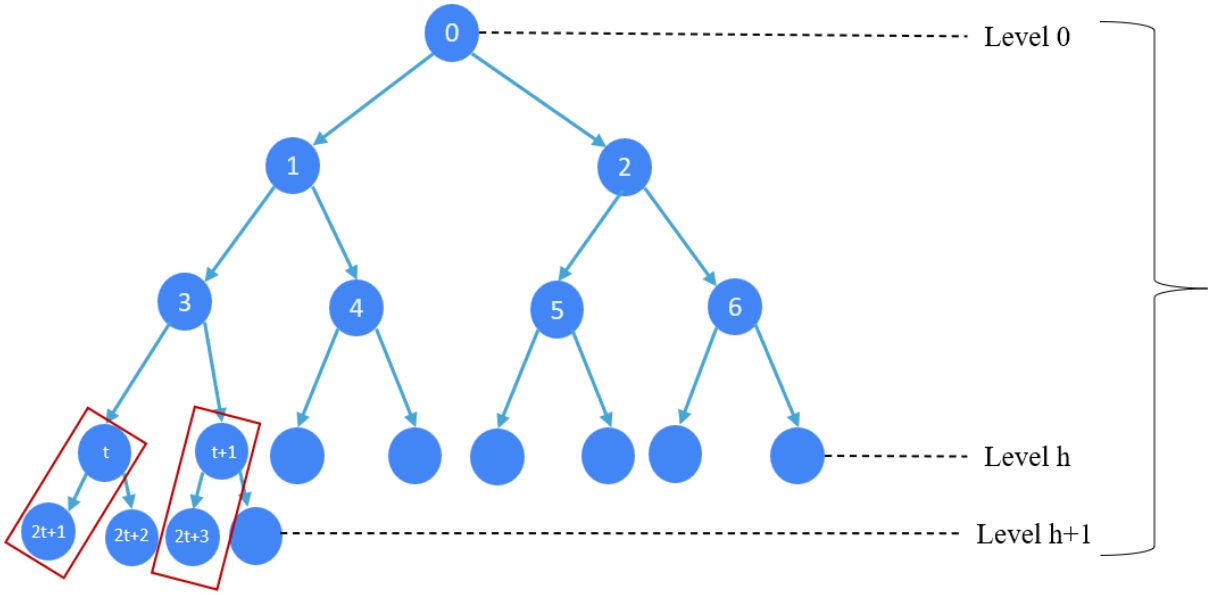
ALL TOPOLOGICAL ORDERS:

1) **A → B → C → G → D → E → F**
2) **A → B → G → C → D → E → F**
3) **A → B → G → D → C → E → F**
4) **A → G → B → C → D → E → F**
5) **A → G → B → D → C → E → F**
6) **G → A → B → C → D → E → F**
7) **G → A → B → D → C → E → F**

7. A binary tree[2] is a rooted tree in which each node has two children at most. A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible. For a complete binary tree T with k nodes, suppose we number the node from top to down, from left to right with 0, 1, 2, ..., $(k-1)$. Please solve the following two questions:

   - For any of the left most node of a layer with label t, suppose it has at least one child, prove that its left child is $2t+1$.
   - For a node with label t and suppose it has at least one child, prove that its left child is $2t+1$.

ANS:



To prove any left most node t in a complete binary tree if it has at least one child node, the left child will be equal to $2t+1$. NOTE: In diagrams above, I have indicated arrows in trees just to establish parent-child hierarchy.

The number of nodes in any level of a full binary tree is 2^h where h is height or level of the tree. As per the problem statement, let us consider a left most node with label t which can be written as $2^h - 1$ ("-1" as labelling started from 0) => $t = 2^h - 1 => t + 1 = 2^h$ ... (1)

The left child of t (which is already a leftmost node of level h) = tlc, will be a leftmost child for the next level i.e., $height = h + 1$ => $tlc = 2^{h+1} - 1$ … (2)

Solving (1) and (2)

$=> tlc = 2^{h+1} - 1$

$=> tlc = 2(t + 1) - 1$ ... From (1)

$=> tlc = 2t + 2 - 1$

$=> tlc = 2t + 1$

INDUCTIVE HYPOTHESIS:

Assume that for any node with label t, if it has at least one child node, then the left child node is $2t + 1$. Now we must prove this for the node with label $t + 1$

BASE CASE:

For t = 0 the left node must be 2(0) + 1 = 1, which is true. Hence, hypothesis proved for the base case.

Now let us prove for the node with label $t + 1$. As it is a complete binary tree the node with label t must have 2 children for $t + 1$ to have at least 1 child.

From (1) & (2) we have proved that given t is a left most node in a complete binary tree with at least 1 child then the left child of *t is* $2t + 1$ => right child of *t is* $2t + 2$ as the naming follows Top →Down and Left → Right approach.

The right child of t is followed by our required left child of $t + 1$ => $2t + 2 + 1 = \mathbf{2(t + 1) + 1}$ which proves our hypothesis is true.

8. Consider a full binary tree (all nodes have zero or two children) with k nodes. Two operations are defined: 1) *removeLastNodes()*: removes nodes whose distance equals the largest distance among all nodes to the root node; 2) *addTwoNodes()*: adds two children to all leaf nodes. The cost of either adding or removing one node is 1. What is the time complexity of these two operations, respectively? Suppose the time complexity to obtain the list of nodes with the largest distance to the root and the list of leaf nodes is both $O(1)$.

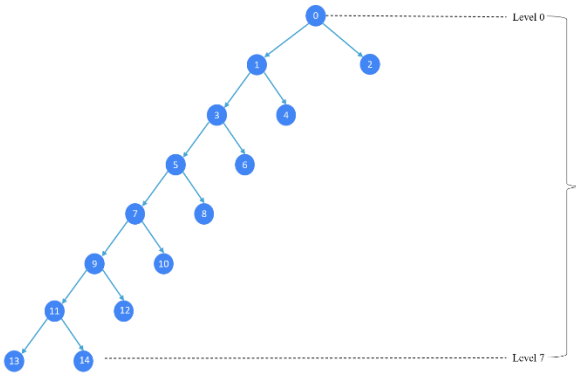
Two operations are defined:

1) removeLastNodes(): Removes nodes whose distance equals the largest distance among all nodes to the root node;

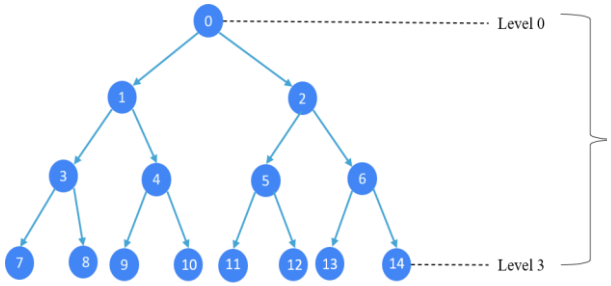
2) addTwoNodes(): Adds two children to all leaf nodes. The cost of either adding or removing one node is 1. The time complexity to obtain the list of nodes with the largest distance to the root and the list of leaf nodes is both $O(1)$.

AIM: To find the time complexity of these two operations, respectively

ANS:



Full Binary Tree -1



Full Binary Tree -2

Example: If we have a full binary tree (all nodes have zero or two children) of k = 15 nodes, it can be interpreted in many ways. Above are 2 such ways where we can represent a full binary tree with 15 nodes (there are many other ways as well). NOTE: In diagrams above, I have indicated arrows in trees just to establish parent-child hierarchy.

Hence, time complexity will be different in different cases. The best way to find the time complexity is to find all the possibilities for Best Case (Ω), Worst Case (O) and Average Case (θ). Note that in a full binary tree k (Number of nodes) is always an odd number.

1) removeLastNodes():

For operation removeLastNodes() the best case is Full Binary Tree – 1 with k nodes where only 2 nodes are the farthest from the root and must be removed (i.e.., cost for finding the elements with the farthest distance is 1 [as given in the problem statement] and cost for deleting 2 nodes is 2). Hence the best-case time complexity (Ω) = $\Omega(3) \approx \Omega(1)$.

For operation removeLastNodes() the worst case is Full Binary Tree – 2 with k nodes where the tree is a complete binary tree and have $\frac{k+1}{2}$ nodes farthest from the root and must be removed (i.e.., cost for finding the elements with the farthest distance is 1 [as given in the problem statement] and cost for deleting $\frac{k+1}{2}$ nodes is $\frac{k+1}{2}$). Hence, the worst-case time complexity (O) = $O\left(1 + \frac{k+1}{2}\right) \approx O(k)$.

For operation removeLastNodes() the average case is where the tree have k nodes and (2, $\frac{k+1}{2}$) nodes are farthest from the root and must be removed (i.e.., cost for finding the elements with the farthest distance is 1 [as given in the problem statement] and cost for deleting $\frac{k+1}{2} - x$ nodes is $\frac{k+1}{2} - x$ where $1 < x < \frac{k+1}{2} - 1$ & $x\ is + ve\ even\ integer$). Hence, average case time complexity (θ) $\approx \theta(k)$ if we consider k is a large value and hence, average input leans to worst-case input) $\approx \theta(k)$.

2) addTwoNodes():

For operation addTwoNodes(), the best case, worst case and average case remains same as the number of leaf nodes remain constant irrespective of arrangement of nodes in a full binary tree. Hence,

Best Case ($\boldsymbol{\Omega}$) = **Worst Case** ($\boldsymbol{O}$) = **Average Case** ($\boldsymbol{\theta}$) $\boldsymbol{for\ this\ operation}$

Number of leaf nodes in a full binary tree with k nodes = $\frac{k+1}{2}$, To fetch this list of leaf nodes it costs 1 unit [as given in the problem statement] – (1)

Now we need to add 2 nodes to each of these leaf nodes which will cost 2 units i.e., total cost to add all these nodes = $[\frac{k+1}{2}]2 = k + 1$ which makes the total cost $k + 2$ units (from (1)).

=> Time complexity for this operation is $\boldsymbol{\Omega(k) \approx O(k) \approx \theta(k)}$

9. Given a sequence of n operations, suppose the i-th operation cost 2^{j-1} if $i = 2^j$ for some integer j; otherwise, the cost is 1. Prove that the amortized cost per operation is $O(1)$.

ANS:

AGGREGATE METHOD:

$$\textbf{Amortized Cost} = \frac{\textbf{Total Cost}}{\textbf{Number of Operations}}$$

Number of Operations = n

$if\ i = 2^j\ then\ cost\ of\ cost\ of\ i^{th} operation\ is\ 2^{j-1}$

$=> 2^j(2^{-1}) = \frac{2^j}{2} = \frac{i}{2}$ - (1)

$j \geq 0\ and\ j \in +ve\ integers\ (as$
$-ve\ integer\ will\ yield\ decimal\ value\ for\ i)$

$if\ i \neq 2^j\ then\ cost\ of\ cost\ of\ i^{th} operation\ is\ 1 - (2)$

From on (1) and (2) the cost of operations:

i	1	2	3	4	5	…	n
$Cost\ of\ i^{th}\ Operation$	$\frac{1}{2}$	$\frac{2}{2}$	1	$\frac{4}{2}$	1	…	1 or $\frac{n}{2}$

$Number\ of\ values\ of\ i\ such\ that\ i = 2^j => (log_2 n)$

$Number\ of\ values\ of\ i\ such\ that\ i \neq 2^j => (n - log_2 n - 1)$

$=> \frac{1}{2} + (1 + 2 + 4 + 8 + 16 + \cdots) + (n - log_2 n * (1))$

$$=> Total\ Cost = \frac{1}{2} + \frac{(2^{log_2 n} - 1)}{2 - 1} + (n - log_2 n - 1)$$

$$= \frac{1 + (2n - 2) + (2n - 2log_2 n - 2)}{2}$$

$$=> \frac{4n - 2\,log_2 n - 3}{2}$$

$$Amortized\ Cost = \frac{Total\ Cost}{Number\ of\ Operations} = \frac{4n - 2\,log_2 n - 3}{2(n)}$$

$$= 2 - \frac{log_2 n}{n} - \frac{1.5}{n}$$

As n –> ∞ *the Amortized time complexity is* $\approx O(1)$

10. Consider a singly linked list as a dictionary that we always insert at the beginning of the list. Now assume that you may perform n insert operations but will only perform one last lookup operation (of a random item in the list after n insert operations). What is the amortized cost per operation?

ANS:

ACCOUNTING METHOD: (BANKER'S METHOD)

In this method, we assign tokens to each operation and total sum of the tokens covers the actual cost of all operations.

INSERT OPERATION:

Cost of each insert Operation at the beginning of a singly linked list is $O(1)$ as we create new nodes and update the pointers.

Let us assign 2 tokens for each insert operation. Which means paying 1 extra token for each insert operation and storing them for future use.

LOOKUP OPERATION:

The actual cost of the lookup operation is *O(n)* because in the worst case, we need to traverse the entire list to find the item.

Since we already have 1 token in our reserve (bank) for each of the *n* insert operations, we have *n* tokens available to pay for the lookup operation.

CALCULATION:

*Total cost of n inserts operations = n * (Actual Cost of Insert Operation) = n * (1) = n*

Cost of Lookup Operation = O(n) ……. (1)

*Tokens assigned for n insert operations = n * (1(Actual cost) + 1 (Reserve Token)) = 2n*

Total tokens available for the lookup operation after insertions = 2n – n (for insertion) = n

=> We have enough tokens stored to perform a lookup operation (ref (1)) as the worst-case complexity for a lookup operation is O(n) (Complexity = 2n – n+O(n)) which suggests that it is constant time.

To add.. we can support it using Aggregate method

AMORTIZED COST PER OPERATION:

$$Amortized\ Cost\ per\ operation\ = \frac{Total\ Cost}{Number\ of\ Operations}$$

Amortized Cost per operation = $\frac{[(n+O(n))]}{n+1}$

Amortized Cost per operation = $\frac{n+O(n)}{n+1}$

Therefore, the amortized cost per operation is approximately $\frac{n+O(n)}{n+1}$ which simplifies to $\frac{n+O(n)}{n+1}$

As $n \rightarrow \infty$ *the Amortized time complexity is* $\approx \boldsymbol{O(1)}$

********** **THE END** **********