1.
(i) True
(ii) False

2.
(a) We have a directed weighted graph with negative edge weights but no negative cycles.

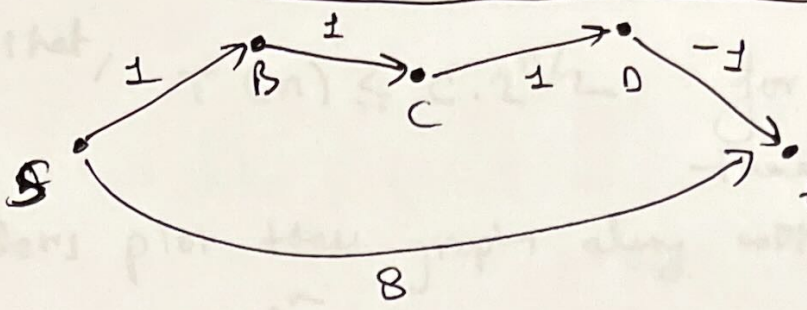
We know that Dijkstra's algorithm works on =>
→ weighted graph
→ directed graph (G=(V,E,w))
→ real-valued weights (non-negative)
→ can handle cycles.

The current graph follows the 1st two characteristics above and has no negative cycles which would mean that there will be some positive weights too.

Thus, only problematic thing to apply Dijkstra for our current kind of graph will be negative edge weights.

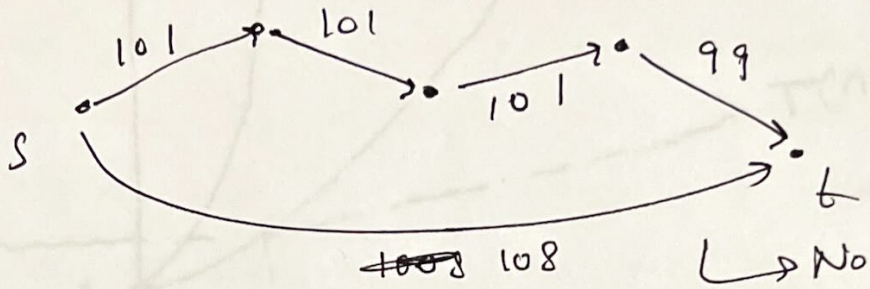
But if we just add a very large constant to all the edge weights such that all edge weights become positive, then all the edge weight would become quite large and if we previously had a shortest distance along a path that had lot more edges, then adding high constant to each of these edge weights could potentially increase the total weight along a path such that it would no longer remain the shortest path. This would mean that Dijkstra's cannot be applied there!

Ex:- Chandan Suri, CS4090



→ SBCDt was the shortest here.

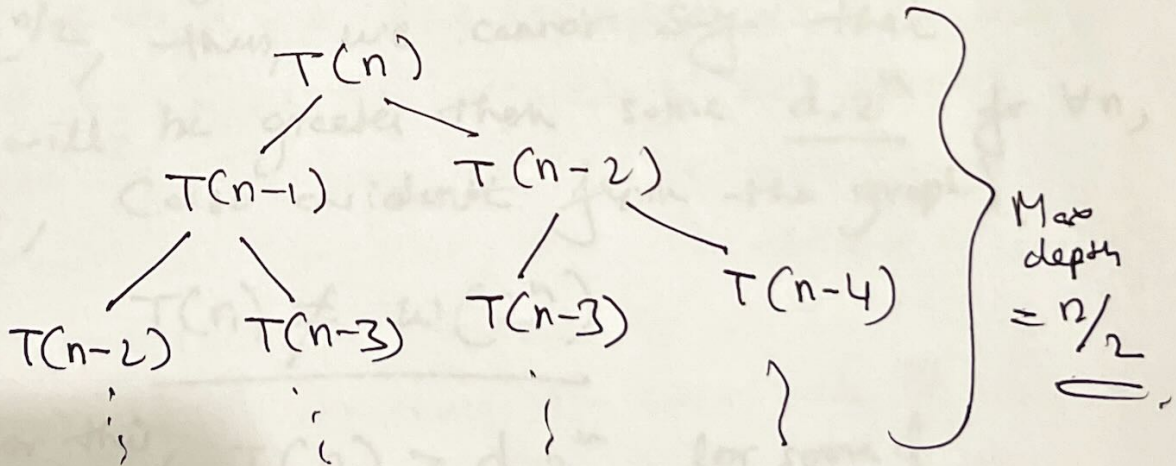
Let's add 100,



↳ Now this becomes the shortest, thus the shortest path isn't preserved.

Thus, False. Ans.

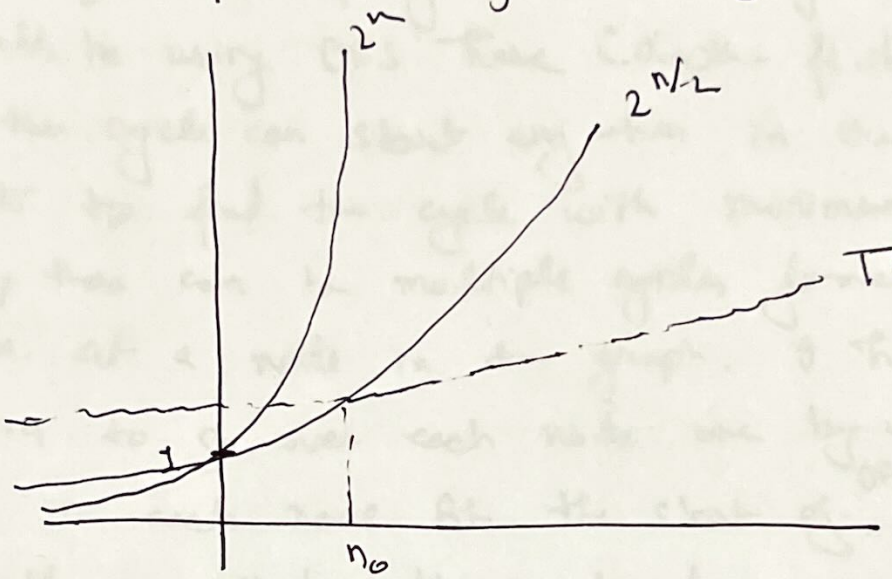
(b).



The recurrence given here is for fibonacci. Acc to classroom discussion, this T(n) would result in ∴ $T(n) = O(2^{n/2})$

As this is true, we can say that, $T(n) \leq c.2^{n/2}$ for $\forall n$, $n \geq n_0$ there exists a c.

Now, lets plot these graphs along with 2^n below,



As we can follow from above, that $T(n)$ is always less than $2^{n/2}$ for some c and all $n \geq n_0$, and as rate of change of 2^n is much higher than $2^{n/2}$, thus, we cannot say that $T(n)$ will be greater than some $\underline{d.2^n}$ for $\forall n$, $n \geq n_0$, (also evident from the graph).

So, $T(n) \neq \omega(2^n)$

as for this, $T(n) > d.2^n$ for some d $\forall n, n \geq n_0$.

So, this is not true.

Thus, False. ∴

(3.)

Chandan Devi, CS4090

Description:) To find the minimum weight cycle in the graph, I will keep a global minimum weight value initialized to infinity. for traversing the graph, I will be using DFS here (depth-first search). As the cycle can start anywhere in the graph and I want to find the cycle with minimum total weight. Also, there can be multiple cycles formed that starts and ends at a node in the graph. I there all the more reason to go over each node one by one and run DFS on each node. At the start of DFS of each node, I will re-initialize the explored array with zeros.

Then, I simply run DFS, going over all the nodes starting at a node 's'. If the next node wasn't explored before, I just keep traversing while updating the current cycle weight by adding the current edge weight. I can pass this as a variable in the recursive call, so, that this is updated accordingly while the backtracking happens.

However, if I encounter the same node again that had been ofcourse visited before, I update the global minimum weight value by comparing the current cycle weight = current cycle weight + current edge weight with the global minimum weight.

if it's less than the global minimum weight with the current cycle weight. Chandan Dwi, CS4090

This else condition only happens if there was a cycle in the graph. Otherwise, when the traversal has ended for all the nodes in the graph (starting at all the nodes in the graph) and ~~min~~ global minimum weight is infinity, then I will return "no" meaning that G was acyclic.

Correctness:

I am basically running DFS on each node in the graph which is of course correct. I have just added an else condition which doesn't change the flow of the algorithm. It's just that I update the global minimum weight with the current cycle weight if I visit a node again. That tells us about the presence of cycle in the graph.

Also, I am just updating the current weight recursively while traversing the nodes, so, this will just add up the weights.

At the end of DFS for a node, I will get the minimum cycle weight and after doing DFS for all the nodes, I will get minimum cycle weight for the full graph.

Thus, the algorithm is correct. //

Runtime Analysis:

Charalen Suri, CS4090

As we are using DFS which has a worst case time complexity of $O(m+n)$ where m is the number of edges & n is the number of vertices/nodes. Although, when we have a dense graph with $m = n^2$, then, worst case time complexity for DFS (with one else condition) could reach $n^2 \Rightarrow O(n^2)$.

As I am doing this (DFS) for all the nodes in the graph, then total worst-case time complexity will be $O(n^3)$.

Ans

Chandan Suri, CS4090

Description =>

For the case here, we need to first find the optimal point in the row of cards where the sum of the values of those cards comes out to be the maximum. Once we have found those 2 cards from all the cards, this will be our starting point. For doing this it will just iterate over all the values of the cards the find those 2 cards.

Then, I start at the position of those 2 cards found and compare the value of the sum (profit at that point) to the ~~left card~~ card on the left and the card on the right as we can only do the profit aggregation for adjacent cards. If the value of the left card is found to be more, than the value of the right card, I will add the left card to calculate the profit and decrement the left pointer by 1. Similarly, if the value of the right card is found to be greater, then I will add the right card and increment the right pointer by 1. Doing this over and over again iteratively till we have only one card left, will give us the total maximum profit. This is a greedy approach to the problem.

Correctness:)

We can do this by Induction.

Base case:- $n=2$.

When we have 2 cards, then for maximum total profit we will just add the value of those 2 cards.

Hypothesis:- when $n=k$,

(8 step)

$C_1\ C_2\ C_3\ C_4, \ldots\ C_{k-2}\ C_{k-1}\ C_k\ C_{k+1} \ldots C_{n-1}\ C_n$

$C_2\ C_3$: Sum = B

$C_{k-1}\ C_k$: Sum = A

As at K, the optimal point would have been there, assuming $A > B$,

Now, we can either add the value of left card or the right card,

$A + C_{k-2} > B$

$A + C_{k+1} > B$

we will get the maximum total profit at this point.

Cases

if $C_{k-2} > C_{k+1}$, then we add the left card,

otherwise, we add the right card.

Induction:- for $n=k+1$,

As we would have replaced the card at kth index, then we again do the same thing of looking at the left card and the right card. As till now we found the maximum possible total profit,

In the next step, if we add the maximum of the left card or the right card, then the maximum total profit is then, maximized again.

Chandan Suri
CS4090

Also, as we had stated at the maximum optimal value of the total profit, this maximum value is added to all the profits computed.

Thus, till ~~nth card~~ (n-1)th card profit aggregation, the same is proved by induction.

Thus, the algorithm is correct.

Runtime analysis ⇒

For the first loop to find the first optimal point where the value of the adjacent cards is the maximum, we go over all the cards, this step takes $\Theta(n)$ time.

In the next step, where I update 2 pointers for the left card and right card, adding up the values for calculating the total profit, this loop will go on till I have added the values for (n-1) cards, thus, this loop will also take $\Theta(n)$ time.

Thus the total time complexity of the algorithm is ⇒ $\Theta(2n) \Rightarrow \boxed{\Theta(n)}$ (linear time for worst-case)

Ans

5. Chandan Suri, CS4090

$A = \{a, b, c, d, e\}$

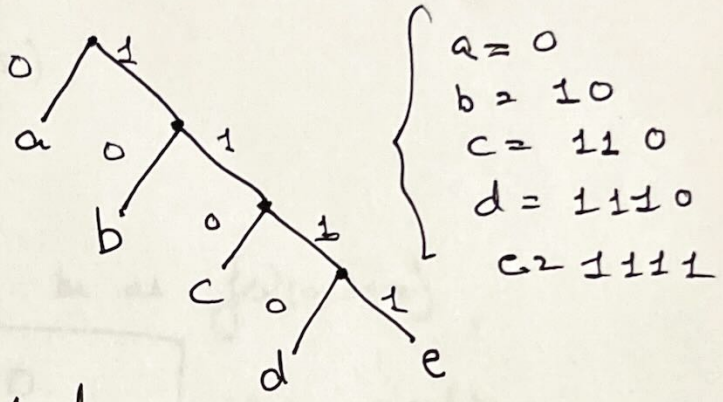
$P = \{p_a, p_b, p_c, p_a, p_e\}$

(a) (i) $C_1 \rightarrow \{1, 2, 3, 4, 4\}$

Thus, for $p_a = 0.5$, $p_b = 0.25$, $p_c = 0.125$, $p_d = 0.0625$

$p_e = 0.0625$

Tree formed is →



a = 0
b = 10
c = 110
d = 1110
e = 1111

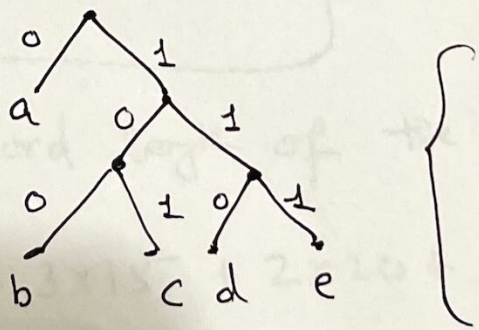
We can get valid encoding here.

(ii) $C_2 = \{1, 3, 3, 3, 3\}$

Thus for $p_a = 0.6$, $p_b = 0.1$, $p_c = 0.1$, $p_d = 0.1$,

$p_e = 0.1$.

Tree →



a = 0
b = 100
c = 101
d = 110
e = 111

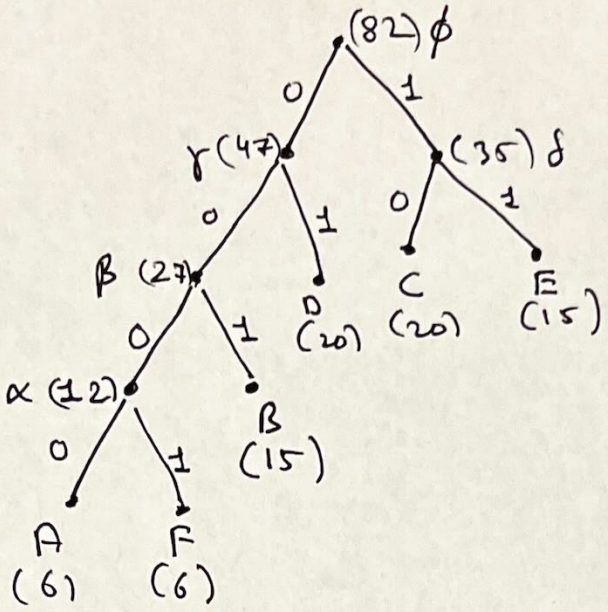
We can get valid encoding here.

(b) A → 6, B → 15, C → 20, D → 20, E → 15,

F → 6.

Huffman Tree:

Chandan Dutta, CS4090



Thus, encoding will be as follows:

A →	0000
B →	001
C →	10
D →	01
E →	11
F →	0001

Huffman codes.

Average codeword length of the Huffman code:

$$\frac{1}{82}\left[4\times6 + 3\times15 + 2\times20 + 2\times20 + 2\times15 + 4\times6\right]$$

$$= \frac{1}{82} \times 203 = \frac{203}{82} = 2.4756$$

$$\approx 2.48$$

Ans.