Assignment 03 Dynamic Programming

By Farhan Ishmam, 180041120

Problem – 1

Atiqur Rahman, 180041123

Mushfiqul Haque, 180041140

Problem – 2 (a)

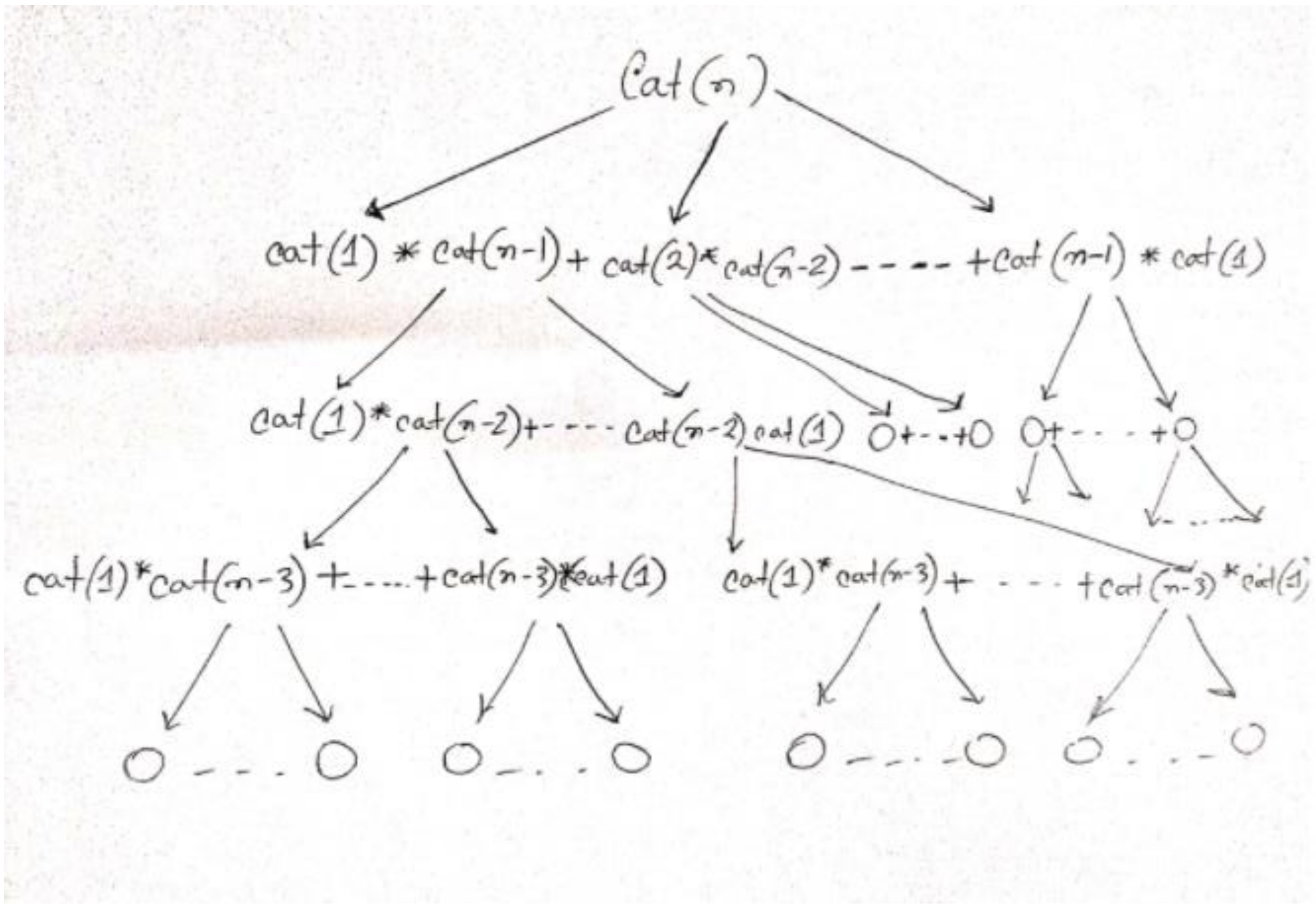
If m = 1, then it would be impossible to divide the *total* among two people and a single person will get the whole *total*. Hence, the maximum amount a friend can get is *total.*

Problem – 2 (b)

In the previous scenario, one friend gets the money while the other gets none. So, the difference will be *(total – zero) = zero*. On the flipside, there can be a scenario where the coins are equally distributed among the two friends (i.e. both gets *(total/2)* amount of money) Then the difference will be *total/2 – total/2 = zero*. Hence, the range of difference is [0, *total*]

Problem – 3 (a)

In the given code, the Cat() function is recursively called and it thus forms a recursive tree.



Cat(n)
cat(1) * cat(n-1) + cat(2) * cat(n-2) - - - - + cat(n-1) * cat(1)
cat(1) * cat(n-2) + - - - cat(n-2) cat(1)
0 + - - + 0
0 + - - - + 0
cat(1) * cat(n-3) + - . . . . + cat(n-3) * cat(1)
cat(1) * cat(n-3) + - - - + cat(n-3) * cat(1)
0 - - - 0
0 - . . 0
0 - - . - 0
0 . . - 0

For each term, Cat(i) is multiplied to Cat(n-i). Let, each term be donoted as T(i), where i is any number in [1,n-1]. So, T(n) = Cat(1) * Cat(n-1)

For Cat(n), we get,

Cat(n) = Cat(1) * Cat(n-1) +Cat(2) * Cat(n-2) + Cat(3) * Cat(n-3) + …… + Cat(n-3) * Cat(3) + Cat(n-2) * Cat(2) + Cat(n-1) * Cat(1)

- ⇨ T(n) = T(1) + T(2) + ……T(n-1)…..T(2) + T(1)
- ⇨ T(n) = 2*(T(1) + T(2) + ….. + T(n-1))
- ⇨ T(n) = 2*(T(1) + T(2) + ... + T(n-2)) + 2*(T(n-1))…………………………………………………………(1)

For Cat(n-1), we get,

Cat(n-1) = Cat(1) * Cat(n-1) +Cat(2) * Cat(n-2) + Cat(3) * Cat(n-3) + …… + Cat(n-3) * Cat(3) + Cat(n-2) * Cat(2) + Cat(n-1) * Cat(1)

- ⇨ T(n-1) = T(1) + T(2) + …….. T(n-2) …… T(2) + T(1)
- ⇨ T(n-1) = 2*(T(1) + T(2) …. + T(n-2)) ………………………………………………………………………………(2)

Replacing the right hand side of equation (2) in equation (1), we get,

T(n) = T(n-1) + 2*T(n-1)

⇨ T(n) = 3*T(n-1)

By breaking down T(n) and represting it in terms of T(n-1), we can see that T(n) is just three times of T(n-1).

Similarly, by replacing n with n-1, for T(n-1), we can write,

T(n-1) = 3*T(n-2)

Combining the equations for T(n) and T(n-1), we can represent T(n) in terms of T(n-2),

T(n) = 3*3*T(n-2)

⇨ T(n) = (3^2)*T(n-2)

Generalizing this form, we get,

T(n) = (3^(n-1))*T(1)

⇨ T(n) ≈ 3^n

T(1) takes constant time and 3^(n-1) takes exponential time.

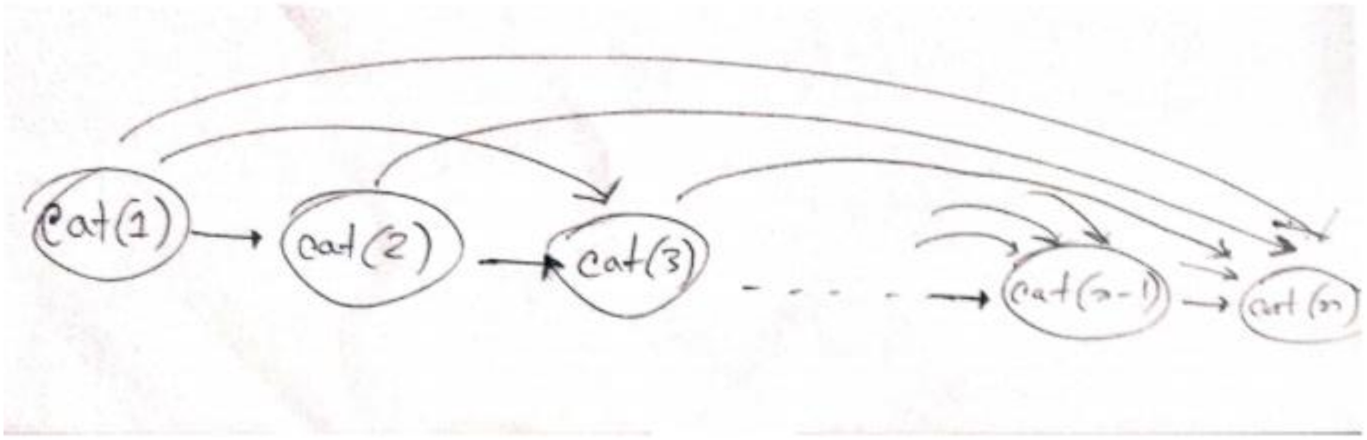
All the other functions before the recursion call takes constant time.

So, the running time of this function is O(3^n) which is exponential time complexity.

Problem – 3 (b)

The DAG for first n Catalan numbers will have nodes Cat(1), Cat(2), Cat(3)….Cat(n-1), Cat(n).

Cat(1) has no dependecy and can be directly computed. Cat(2) depends on Cat(1), Cat(3) depends on Cat(2) and Cat(1) and so on. Cat(n) depends on all the nodes before Cat(n) i.e. from Cat(1) to Cat(n-1). So, we draw edges to represent the dependencies of the graph. Edges go from Cat(1) to Cat(2), Cat(3) and so on. Again for Cat(2), edges go to Cat(3), Cat(4) and so on. In this way the DAG is contructed and visually represented below.



Problem – 4

The problem can be simplified as – there are n plants and each plant has C_n capsicums. If a plant is kept then the neighbors have to uprooted. Plants are to be kept in such an order than number of capsicums is maximum.

The **naïve approach** will be to **try all possible orientations**. Let there be empty spaces after uprooting a plant. We need to form orientations where there are **one or two spaces** between each plant. It can be easily understood why we need a single space between each plant since it's given in the question. But why would we need two empty spaces?

Let's think of a case where the plants 1 to 4 have 100, 2, 2, and 100 capsicums respectively. If we keep plant-1, we need to uproot 2. But uprooting 2 doesn't necessarily mean that we need to keep 3. Because here, plant-4 has a greater value. The optimum solution will be to keep plant-1 and plant-4 which results in 200 capsicums. So, in this case there will be 2 spaces between each plant.

Thus, all the possible orientations are to be found by keeping 1 or 2 spaces between each plant. But is this the most efficient way of solving this problem?

If we look careful, some part of the problems are being repeated. For instance, if we are keeping plant-1 and find out all possible orientations that maximize the value. Again, when we are keeping plant-2 and finding out all possible orientations then same of the orientations were previously calculated when we did the calculation for plant-1. So, if we can **store** that calculation in the

memory and **reuse** it when necessary then a lot of time can be saved. This is fundamental aspect of dynamic programming and we can use DP to solve this problem efficiently.

The whole problem can be divided into subproblems. The subproblems are essentially finding the maximum number of capsicums among a small number of plants; not the whole dataset. For a plant the maximum number is to be found by comparing it with (i-1)th and (i-2)th plant, where is any number in [1,n]. That is, we take the i th and the (i-2) th plant, or we take the (i-1) th plant; whichever yields the maximum value.

This notion for the subproblem clearly solves the single spacing between plants. But when happens when there's double spacing in the optimum structure?

The step by step implementation is given below

i) For our base case we keep the first two plants same in our DP memory. DP[1] = plant[1] and DP[2] = plant[2]. We also predefine two conditions for n = 1 and n = 2. The problem has no relevance for n = 1, so it will return plant[1]. For n = 2, it will return max(plant[1], plant[2])

ii) We define mx and mx_idx to store max value and index of max value respectively and intialize then at mx at plant[1] and mx_idx at 1, by assuming plant 1 having the maximum value. In this bottom up approach, we will use mx_idx to step back and find maximum value from old values.

iii) We keep a sequence array seq[] for prining the sequence of the values. The sequence array is initialized as seq[1] = 0 and seq[2] = 0 for base case. In this sequence array, we store the maximum value indexes. For example if plant 1,4 and 6 are stored for our optimum structure, we store, seq[6] = 4, seq[4] = 1 and seq[1] = 0.

iv) We traverse plant i=3 to n using a for loop

v) Update the DP [i] = plant[i] + mx

vi) Update the Seq[i] = mx_inx

vii) If DP[i-1] is greater than mx then update a new maximum is found. So, we update, mx = DP[i-1] and mx_idx = i

viii) Finally if DP[n] is greater than mx, then mx_idx will be n. This is done for the last node only.

ix) Now, we have a sequence which points to the previous plant in our optimum structure. The sequence can be printed using a while loop starting from m and prints all the index values till the sequence values reaches zero. Because sequence value equal to zero is the base case of our problem. The while loop can start from mx_idx and change the mx_idx = seq[mx_idx] after each iteration and continue till it reaches zero.

Thus the printed sequence is our required order.

In this problem, there are n sub-problems, each taking O(1) time complexity and results in n * O(1) = O(n) time complexity all together.