Member 1 : Kishan Sinha
2019428
Member 2 : Abhishek Chaturvedi
2019401

PROBLEM 1 :

Given the array C[1 .. n], where C[i] is the type of candy that the ith animal is holding.

Let 'x' denote the kind the kind of candy an animal is holding in numeric form : x = 0 implies , the animal is holding '*Circus Peanuts*' x = 1 implies *'Health Bars*' and x = 2 implies '*Cioccolateria Gardini chocolate truffles*'

So the array C is filled with the three integers 0, 1 , 2

Subproblem:

CandyDis(i,x) : Maximum possible score after visiting the the array C[i..n] i.e., *the maximum possible score for all C[k] for k = i to k = n* **(**taking size of the candy array to be **n**) starting with the candy type **x**

Recurrence Relation:

Base Case i == n-1 for all types of candy , i.e x = 0,1,2 .

CandyDis(i,x) = 1 if x == C[n-1] , else 0

Recursion Step :

CandyDis(i,x) = Max(CandyDis(i+1 ,C[i]) + point, CandyDis(i+1, x))

here **point** is +1 if same candies are swapped and -1 is dissimilar candies are swapped

Final Answer : CandyDis(0,0)

Code :

```cpp
#include <bits/stdc++.h>
using namespace  std;

int dp[100][3];
int C[100];
int n;

int CandyDis(int i, int x){

    if(i==n-1){
        if(C[i] == x)
            return 1;
        return 0;

    }

    if(dp[i][x] != -1)
        return dp[i][x];

    if(x == C[i])
        return dp[i][x] = CandyDis(i+1 ,C[i]) + 1;

    return dp[i][x] = max(CandyDis(i+1, x) , CandyDis(i+1 , C[i]) -1);

}

int main(){


    cin>>n;
    for(int i = 0 ; i < n ; i++)
        cin>>C[i];


   for(int i = 0 ; i < n ; i++)
       for(int j = 0 ; j < 3; j++)
           dp[i][j] = -1;

    cout<<CandyDis(0,0)<<endl;



}
```

Correctness : CandyDis(i,x) gives the maximum possible score possible for all the animals in the range arr[i..n]

Now there are two cases for the ith animal , whether a candy is swapped or skipped.

If the current candy in hand is same as the candy of the animal , it is quite easy to see that since we have the answer for all elements in the range i+1 .. n , The answer will be that Plus 1 , because we are swapping with the same candy , for answer possibility , the answer will be lower than CandyDis(i,x).

If candy in hand is not same , then we can do one of two things ,

1. Skip (in this case , the answer for arr[i+1..n] will be our answer
2. Swap with different candy type (in this case answer will CandyDis(i+1..n) -1) we get a penalty of MINUS -1 to it.

And we take the **maximum** of these two cases

Proof : Let us suppose for the sake of contradiction that there was some different set S other than i+1 .. n ,(*which means S doesn't have all elements from i+1 to n , say S is arr[k..n] where k is not i+1)* which gave the correct answer for all elements arr[i..n] ,

But according to the question all the animals need to be visited once, so S has to contain i+1 , hence by Contradiction . [i+1 .. n] gives the correct answer.

Induction Proof :

Hypothesis : P(n) -> CandyDis(0,0) gives the correct answer for all n elements of the array . (i.e CandyDis(0,0) refers for problem of size n , hence we take the notation P(n))

Induction : P(k) is true for all k < n [Strong Induction]

Statement : Since CandyDis(0,0) or P(n) calls CandyDis(1,0) / CandyDis(1,1) / CandyDis(1,2) or in other words , CandyDis(i,x) calls CandyDis(i+1, x)
Or in other words, P(n) calls for P(n - 1) and so on , since n - 1 < n , i.e., k < n (Assumed in Induction Step)
So P(n) is true by induction , because P(n) depends on P(i) where all i < n .

Hence , P(n) is true.

Time Complexity : In the CandyDis function , the parameter ‘i’ goes from 0 to n-1 , calls upon its next index , i+1 ,

This is equivalent to one loop of O(n) (after DP optimisation)
[done in class]

And x goes from 0 to 2 , which O(3)

Hence, T(n) = 3 * O(n) which is asymptotically , O(n)

Problem 2 :

Subproblem:

BestScore(i) : Maximum score achieved and the optimal solution obtained on dancing to the ith song in the array of songs[i…...n-1]

Recurrence Relation:

Base Case*:*

i >= n means that there are no more songs left to dance so 0 is returned

Recursion Step :

BestScore(i) = Max(Score[i] + BestScore(i+1 + Wat[i]), BestScore(i+1))

Final Answer : BestScore(0)

Code:

```cpp
#include <bits/stdc++.h>
using namespace  std;

int dp[100];
int Score[100];
int Wait[100];
int n;

int BestScore(int i){

    if(i>=n)
        return 0;

    if(dp[i] != -1)
        return dp[i];

    int option1 = Score[i] + BestScore(i+1+Wait[i]);
    int option2 = BestScore(i+1);

    return dp[i] = max(option1, option2);

}

int main(){

    cin>>n;
    for(int i = 0 ; i < n ; i++)
        cin>>Score[i];
    for(int i = 0 ; i < n ; i++)
        cin>>Wait[i];

   for(int i = 0 ; i < n ; i++)
       dp[i] = -1;

    cout<<BestScore(0)<<endl;
}
```

Correctness :

Case 1: In this case we are taking the option that the person **IS** dancing on the ith song , so his score will be Score[i] + BestScore(next song he will dance on)

Since, he cannot dance for Wait[i] songs, the next song song he can dance will be , i + Wait[i] + 1

BestScore = Score[i] + BestScore(i+ Wait[i] + 1)
From the recurrence we got that adding score at the ith index to the BestScore(i+1 + Wait[i]) gives us the optimal solution. Assume that this is not true for the score of contradiction and thus assume that BestScore(i+1 + Wait[i]) gives us a much better optimal solution.
But, there we can see that if we add Score[i] to this optimal solution of BestScore(i+1 + Wait[i]), it will generate a greater score than BestScore(i) contradicting the optimality of the latter.

Case 2: In this case we are taking the option that the person **IS NOT** dancing on the ith song , so his score will be BestScore(next song he will dance on) , in this case the next song he will dance on will (i + 1)

BestScore(i) = BestScore(i+1)

From the recurrence we see that BestSCore(i+1) gives an optimal solution. Suppose this is not true for the sake of contradiction thus BestScore(i+1) generates a much better optimal solution.
Now since this is true then,
BestScore(i+1) > BestScore(i)
Here, subproblem is more optimal thus the actual problem thus contradicting the optimality of latter.

Time Complexity : In the BestScore function , the parameter ‘i’ goes from 0 to n-1 , calls upon its next index , i+1 or some i + k , where k is > 1

This is equivalent to one loop of O(n) (after DP optimisation) [done in class]

Hence, **T(n) = O(n) which is asymptotically , O(n)**

Problem 3:

Assuming *arr[0..n-1]* to be the array of volumes of the drops

Subproblem : solveDrops (i,j) gives the Maximum possible Energy generated in arr[i .. j] (both inclusive)

Recurrence Relation :

Base Case : i>=j **i.e.,** left boundary has become more or equal to the right boundary

Recursion Step :

solveDrops (i,j) =

Max for all k in (k = i to k = j - 1)**{**

solveDrops (i,k) +

solveDrops (k+1, j) +

Csum(i, k)^2 + Csum (k+1, j)^2

}

Csum(i, k)^2 + Csum (k+1, j)^2 is the energy generated in combining the left and right resultant drops.

CSum(x,y) calculates the sum of all values in the array between index x and y

Final Answer : solveDrops(0,n-1)

Code :

```
#include<bits/stdc++.h>
using namespace std;
long long arr[1000];
long long dp[1000][1000];

long long Csum(int si , int ei){
    long long ans = 0;
    for(int i = si ; i <=ei ; i++)
        ans+=arr[i];

    return ans;
}

long long solveDrops(int i , int j){

    if(i>=j)
        return 0;
```

```
    if(dp[i][j]!=-1)
        return dp[i][j];

    long long bestAns = -INT_MAX;
    for(int k = i ;  k <j ; k++){
        long long possibility = solveDrops(i,k) + solveDrops(k+1,j) +
Csum(i,k)*Csum(i,k) + Csum(k+1,j)*Csum(k+1,j);
        if(possibility > bestAns) {
            bestAns = possibility;
        }

    }

    dp[i][j] = bestAns;

    return bestAns;

}









int main() {
    int n;
    cin>>n;
    for(int i = 0 ; i < n ; i++)
        cin>>arr[i];




    for(int i = 0 ; i <=n ; i++)
        for(int j = 0 ; j <=n ; j++)
            dp[i][j] = -1;

    cout << solveDrops(0,n-1) << endl;


}
```

Correctness : This solution is basically to try out all the intervals possible in the array and taking the best of all .
The loop in the function solveDrops , basically divides the array into two parts: arr(i….k) and arr(k+1….j) , we get the best answer of the two parts by recursion and then combining them will generate energy = (Volume of the left resultant drop) ^2 + (Volume of the right resultant drop) ^2

Now the Volume of the left resultant drop will be the sum of all elements in the left part of the array . Similarly for the right part.
This is due to the basic observational fact that no matter what order we combine the drops the resultant drop will have the same final volume because order doesn't matter in addition.

Claim : There is no other type of the last drops other than the drops generated by arr(i….k) and arr(k+1….j)
Justification : This claim is true because we cannot change the order of the elements in the array, since we can only combine adjacent elements.

Time Complexity :

Number of subproblems : depends on i and j , since in the recursion , i and j both go from 0 to n - 1 .
Hence , it is equivalent to two loops of O(n)
Hence, O(n^2)

Time in each Subproblem : For each i and j , the value of k goes from i to j , which has worst case O(n)

Hence Total , **T(n) = O(n^2) * O(n) = O(n^3**)