Problem 1 :

The new swap-puzzle game Candy Swap Saga XIII involves n cute animals numbered from 1 to n. Each animal holds one of three types of candy: circus peanuts, Heath bars, and Cioccolateria Gardini chocolate trues. You also have a candy in your hand; at the start of the game, you have a circus peanut.

To earn points, you visit each of the animals in order from 1 to n. For each animal, you can either keep the candy in your hand or exchange it with the candy the animal is holding.

- **If you swap your candy for another candy of the same type, you earn one point.**
- **If you swap your candy for a candy of a different type, you lose one point. (Yes, your score can be negative.)**
- **If you visit an animal and decide not to swap candy, your score does not change.**

You must visit the animals in order, and once you visit an animal, you can never visit it again.

Describe and analyze an efficient algorithm to compute your maximum possible score. Your input is an array C[1.. n], where C[i] is the type of candy that the ith animal is holding.

Solution:

Assuming that values of candies are:
circus peanuts = 1;
Heath bars = 2;
Cioccolateria Gardini chocolate trues = 3;
Assumption: We are going from index 1 to n, i.e from left to right.

1.Subproblem:

SP(i,t):optimum solution from index i to n, holding candy type t at index i

2.Recurrence relation:

SP(i,t) = if(c[i] == t)

max(**(**0+SP(i+1,t)**)** , **(**1+SP(i+1,c[i]**)**)

else

max(0+SP(i+1,t)) , (-1 + SP(i+1,c[i]))

Base Case:

```
if(i==n){
        if(t == c[i])
                return 1;
        else
                return 0;
}
```

3.Final Answer:

SP(1,1)

4.Proof of Correctness of recurrence:

Lemma 1: The RHS of the recurrence is SP(i,t)

By proving the following two claims, we prove the above lemma.

Claim 1:RHS is always the value of feasible solution for SP(i,t)

Claim 2: There is no feasible solution for SP(i,t) of larger value than RHS

Proof of Claim 1:

There can be two cases at each step:

Either we replace or not.

When we don't replace we just add 0 to our score.

Now consider i+1,value of SP(i+1,t) is the solution for index i+1 to n holding the same candy t.Hence appending the score for no replacement to it gives us the score for index i as 0+SP(i+1,t).

When we replace and c[i] == t we just add 1 to our score.Now consider i+1,value of SP(i+1,t) is the solution for index i+1 to n holding the same candy type t.Hence appending the score for replacing with the same type to it gives us the score for index i as 1+SP(i+1,t)

When we replace and c[i] != t we just add -1 to our score. Now consider i+1,value of SP(i+1,c[i]) is the solution for index i+1 to n holding the candy type c[i].Hence appending the score for replacing with a different type to it gives us the score for index i as -1+SP(i+1,c[i]).

Since RHS takes the max over all 3 such possibilities,RHS is always the value of a feasible solution to SP(i,t).

Proof of Claim 2:

Consider any feasible solution S for SP(i,t).This is the score for index i to n holding candy t at i.

Now either we replace our candy or not.

Now let us consider the case when we don't replace our current candy.Now for index i+1,S\score(not replacing) is is the solution for index i+1 to n holding candy of type t(Since we did not replace).Since SP(i+1,t) is the most optimum score for index i+1 to n holding the candy t at i+1,S\score(not replacing) <= SP(i+1,t).Since score(not replacing) = 0.

S <= 0+SP(i+1,t) which is at most RHS.

Similarly we can prove the two cases for when replace.

Since RHS takes the max over many possibilities including this one,RHS is the most optimal solution for SP(i).

5.Pseudo-code:

Assumption:

Create a 2D array(D) with dimensions n,3.

Initialize the array with -infinity.

```
SP(D[1...n][1...3], i,t){
    if(i==n){
        if(t == c[i]){
```

```
                        D[i][t] = 1;
                        return D[i][t] ;
        }
        else{           D[i][t] = 0;
                        return D[i][t];
        }
if(D[i][t] != -infinity){
                return D[i][t];
        }
D[i][t] =
         if(c[i] == t){
                max( (0+SP(i+1,t)) , (1+SP(i+1,c[i]))
                }
        else{
                max(0+SP(i+1,t)) , (-1 + SP(i+1,c[i]))
        }
return D[i][t];
```

6.Time-Complexity:

Possible values for D[n][3] are 1….n and 1...3, and work done at each level O(1).
Hence the time complexity is O(n).

Problem 2:

It's almost time to show o(your flippin' sweet dancing skills! Tomorrow is the big dance contest you've been training for your entire life, except for that summer you spent with your uncle in Alaska hunting wolverines. You've obtained an advance copy of the list of n songs that the judges will play during the contest, in chronological order. Yesssssssss!

You know all the songs, all the judges, and your own dancing ability extremely well. For each integer k, you know that if you dance to the kth song on the schedule, you will be awarded exactly Score[k] points, but then you will be physically unable to dance for the next Wait[k] songs (that is, you cannot dance to songs k + 1 through k + Wait[k]). The dancer with the highest total score at the end of the night wins the contest, so you want your total score to be as high as possible.

Describe and analyze an e)cient algorithm to compute the maximum total score you can achieve. The input to your sweet algorithm is the pair of arrays Score[1 .. n] and Wait[1.. n].

1.Subproblem:

SP(i):optimum solution from index i to n.

2.Recurrence relation:

SP(i) = max((0+SP(i+1)) , (Score[i] + SP((i+1)+Wait[i]))

Base Case:

```
if(i==n)
        return Score[i]
if((i+1) + Wait[i] > n)
        return Score[i]
```

3.Final Answer:

SP(1)

4.Proof of Correctness:

Lemma 1: The RHS of the recurrence is SP(i)

By proving the following two claims, we prove the above lemma.

Claim 1:RHS is always the value of feasible solution for SP(i)

Claim 2: There is no feasible solution for SP(i) of larger value than RHS

Proof of Claim 1:

There can be two cases at each step:

Either we dance to current song or not

When we don't dance we just add 0 to our score.

Now consider i+1,value of SP(i+1) is the solution for index i+1 to n .Hence appending the score for not dancing to it gives us the score for index i as 0+SP(i+1).
When we dance to the current song we just add score[i] to our score.Now consider i+1 + wait[i],value of SP(i+1+wait[i]) is the solution for index i+1+wait[i] to n.Hence appending the current score for not dancing to the current song gives us the score for index i as 0+SP(i+1+Wait[i]);
Since RHS takes the max overall 2 such possibilities, RHS is always the value of a feasible solution to SP(i).

Proof of Claim 2:

Consider any feasible solution S for SP(i).This is the score for index i to n.
Now either we dance to the current song or not.
Now let us consider the case when we don't dance to the current song.Now for index i+1,S\score(not dancing) is is the solution for index i+1 to n.Since SP(i+1) is the most optimum score for index i+1 to n,S\score(not dancing) <= SP(i+1).Since score(not dancing) = 0.
S <= 0+SP(i+1) which is at most RHS.
Similarly, let us consider the case when we dance to the current song.Now for index i+1,S\score(dancing to the current song i) is the solution for index i+1+Wait[i](Since cannot dance for i+1+Wait[i] songs).Since SP(i+1+wait[i]) is the most optimum score for index i+1+Wait[i] to n,S\score(dancing to the current song i) <= SP(i+1+Wait[i]).Since score(dancing to the current song) = Score[i].
S<=Score[i] + SP(i+1+wait[i]) which is at most RHS.
Since RHS takes the max over many possibilities including this one,RHS is the most optimal solution for SP(i).

5.Pseudo-code:

Assumption:

Create a 1D array(D) with dimensions n
Initialize the array with -1.

```
SP(D[1...n], i){
    if(i==n)
    {
        D[i] = Score[i];
        return D[i];
    }
    if((i+1 + Wait[i]) > n)
    {
        D[i] = Score[i];
        return D[i];
    }

    if(D[i] != -1)
        {
```

```
                    return D[i];
                }
            D[i] = max( (0+SP(i+1)) , (Score[i] + SP((i+1)+Wait[i]) )
            return D[i];
        }
```

6.Time-Complexity:

Possible values for D[n] are 1….n, and work done at each level O(1).
Hence the time complexity is O(n).

Problem 3:

Upon time traveling to an ancient laboratory, you find n drops of a mystical liquid arranged in an order. The volume of each drop is given to you. You find a scroll with the following instructions.

You can do the following operation as many times as you like until just one drop remains:

Combine any two adjacent drops and place the combined drop at the same place in the order

When you combine two drops of volumes x and y respectively, you get a new drop whose volume is the sum x + y of the two original volumes (i.e. volume is conserved). Also, you gain an amount of energy for your time traveling device equal to $x^2 + y^2$.

For example,

1. Start with volumes
1 4 2 3 5
2. Combine the second and the third drop to get volumes
1 6 3 5
and gain energy $4^2 + 2^2 = 20$
3. Combine the third and the fourth drop to get volumes
1 6 8
and gain energy $3^2 + 5^2 = 34$
4. Combine the first and the second drop to get volumes
7 8
and gain energy $1^2 + 6^2 = 37$
5. Combine the remaining two drops to get volumes
15
and gain energy $7^2 + 8^2 = 113$
Hence, you gain a total energy of 20 + 34 + 37 + 113 = 204.

Eager to time travel back to 2021, you'd like to maximize the energy gained for your time traveling device. Give an efficient algorithm to find the sequence of operations to achieve this

1.Subproblem:

SP(i,j):optimum energy from index i to j and total volume from index i to j.

2.Recurrence relation:

SP(i,j) = max(SP(i,k).energy + SP(k+1,j).energy + (SP(i,k).volume) ^ 2 + SP(k+1,j).volume ^ 2)

for i<=k<j

Base Case:

if(i==j)

return 0,A[i] //energy and volume respectively

3.Final Answer:

SP(1,n)

4.Proof of Correctness:

Lemma 1: The RHS of the recurrence is SP(i,j)

By proving the following two claims, we prove the above lemma.

Claim 1:RHS is always the value of feasible solution for SP(i,j)

Claim 2: There is no feasible solution for SP(i,j) of larger value than RHS

Proof of Claim 1:

When we have input with one element(namely the element i) in that case the output is 0, which can be a potential solution.

Now consider any k such that i<=k<j. Value of SP(i,k) and SP(k+1,j) gives us the energy obtained by combining elements from i to k and k+1 to j respectively. Now we just add the energy at this level by adding to squares of their values to the energies obtained above.

Since RHS takes the max over all these possible values of k for i<=k<j, RHS is always the value of a feasible solution to SP(i,j).

Proof of Claim 2:

For every value of k in i<=k<j, we are taking the max of (SP(i,k).energy + SP(k+1,j).energy + (SP(i,k).volume) ^ 2 + SP(k+1,j).volume ^ 2). Now, since we are taking the max of all such possibilities of k we only have to prove that SP(i,k) and SP(k+1,j) are the most optimal solution for i...k and k+1...j.

But SP(i,j) is the most optimal solution for i...j, hence the values of energy of SP(i,k) and SP(k+1,j) has to be the most optimal for i...k and k+1...j

5.Pseudo-code:

Assumption:

Create a 2D array(D) with dimensions n,n

Initialize the array with -1(energy) and 0(volume).

```
SP(D[1...n][1...n], i,j){
    if(i==j)
```

```
{       D[i][j].volume = A[i];
        D[i][j].energy = 0;
        return D[i][j] ;
}

if(D[i][j].energy != -1)
        {
                return D[i][j];
        }
temp.volume = 0; temp.energy = -1;
for(k=i; k<j; k++){
        v1 = SP(i,k);
        v2 = SP(k+1,j);
        score = v1.energy + v2.energy + (v1.volume ^ 2) + (v2.volume ^ 2)
        if(temp.energy < score){
                temp.volume = SP(i,k).volume + SP(k+1,j).volume;
                temp.energy = score;

}
D[i][j].volume = temp.volume;
D[i][j].energy = temp.energy;
return D[i][j]
```

6.Time-Complexity:

Possible values for D[n][n] are 1….n,1…n and work done at each level O(n).
Hence the time complexity is O(n^3).