DYNAMIC PROGRAMMING

MNNIT Computer Coding Club

Is Dynamic Programming an Algorithm

No

It is an Algorithm Paradigm i.e., a general idea on which various algorithms are based.

Layman understanding of DP

Consider you have 1000 sticks with you. If I add one more to the collection, how many do I have now. Most of you would have instantly said 1001.

Why is that so? Because of a single reasoning, there were already 1000 sticks so adding 1 more would mean (1000 + 1) sticks = 1001 sticks.

Another approach would have been after adding one more stick, we start counting from the beginning 1, 2 ,3 ... all the way up to 1001.

Which of the two methods was better and Why?

It is pretty obvious that the first method was way better.

Why is it so? Well the answer to this question forms the foundation of the idea on which Dynamic Programming is based.

The first method is better simply because we don’t re-count i.e., we **make use of the existing information** to find the solution of the current problem.

So what is Dynamic Programming

Dynamic Programming is an algorithm paradigm, which aims at storing and re-using the information that is already calculated and not calculate the values every time it is needed.

Dynamic programming is just recursion with cache.
CHANGE MY MIND

Technical Definition

A DP is an algorithmic technique which is usually based on a recurrent formula and one (or some) starting states. A sub-solution of the problem is constructed from previously found ones. DP solutions have a polynomial complexity which assures a much faster running time than other techniques like backtracking, brute-force etc.

In other words if the problem has following properties then it can be solved using DP:

- Optimal Substructure
- Overlapping Subproblems

Terminologies

- **Optimal Substructure** : It means that the problem can be divided in to various sub problems and the optimal solutions of those sub problems will give us the optimal solution of the main problem.
- **Overlapping Subproblem** : It means that the subproblems which will be generated might overlap, i.e., we might need to calculate the value for the same multiple times.
- **Memoization** : A technique of caching (storing) previously calculated values to ensure re-use and avoid unnecessary re calculations.

Divide and Conquer AND Dynamic Programming

Both Divide and Conquer and Dynamic Programming are really similar algorithm paradigms. In other words Divide and Conquer is Dynamic Programming where the subproblems don’t overlap.

How to solve problems using Dynamic Programming

- Start by analysing the main problem and represent it as the final **state** using some **state variables.**
- Then determine the **transitions** between the states, i.e., how does the subproblems help in determining the solution of the actual problem.
- Identify the base conditions.
- This would give us a recursive function. **Memoise** the states so that we don’t have to calculate their values again and again.

Complexity Analysis of DP problems

The complexity of a problem solved using DP is

> **O(number of states * Time complexity of deriving one state)**.

Understanding DP with a problem

Problem : Find the Nth fibonacci number.

Can this problem be solved using DP?

For a problem to be solvable using DP, it has to satisfy the properties mentioned before.

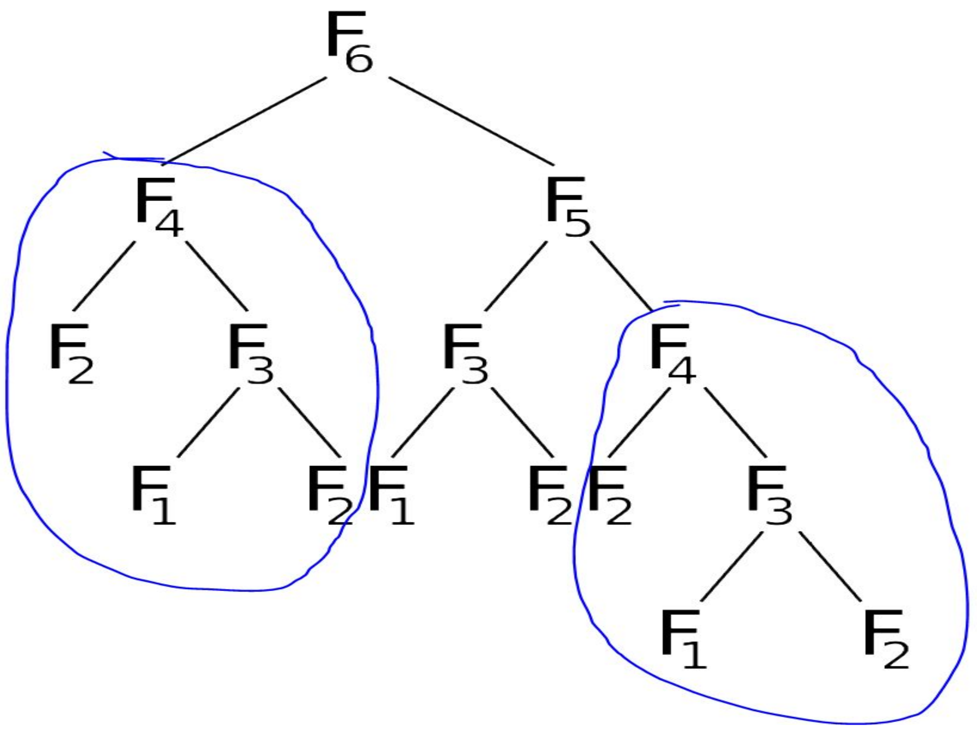
- **Optimal Substructure** : This problem can indeed be remodelled in a structure such that the optimal solution of the subproblems will derive the optimal solution of the main problem.

  The structure of the problem would be f(n) where n is the state variable, and it depends on its sub problems in the following way:

  $f(n) = f(n-1) + f(n-2)$

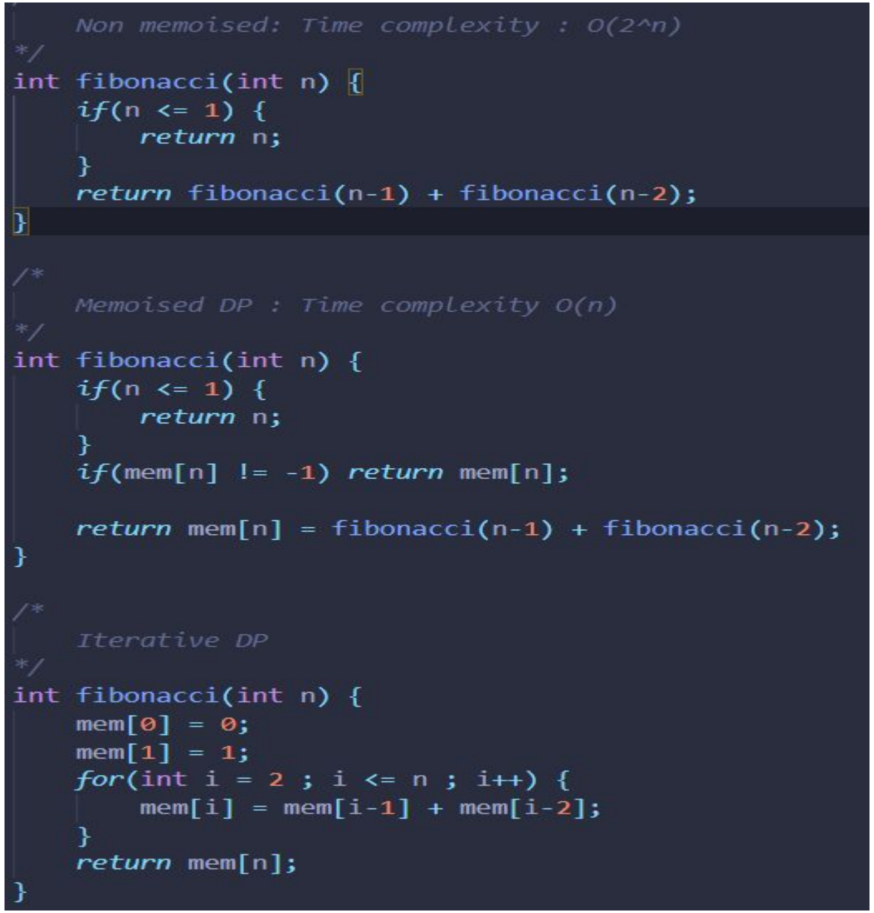
  So the first requirement is met.

- **Overlapping Subproblems** : The recursive tree of this function indicates that many subproblems do overlap. Eg. Here F4 has been calculated 2 times.



So DP can indeed be used to solve this problem.

Solution



```
    Non memoised: Time complexity : O(2^n)
*/
int fibonacci(int n) {
    if(n <= 1) {
        return n;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}

/*
    Memoised DP : Time complexity O(n)
*/
int fibonacci(int n) {
    if(n <= 1) {
        return n;
    }
    if(mem[n] != -1) return mem[n];

    return mem[n] = fibonacci(n-1) + fibonacci(n-2);
}

/*
    Iterative DP
*/
int fibonacci(int n) {
    mem[0] = 0;
    mem[1] = 1;
    for(int i = 2 ; i <= n ; i++) {
        mem[i] = mem[i-1] + mem[i-2];
    }
    return mem[n];
}
```

PROBLEMS

Problem 1

Given infinite number of coins of denominations a , b , c ($a < b < c$) what is the minimum number of coins which you need to give if you want to pay a bill of N.

Also called the Coin Change Problem.

Initial Thoughts

The most intuitive thought that comes to our mind is to use the highest denomination as many times as it can be used, then move to the next highest and so on.

Will it work for every case ?

The answer is **No**.

example : If a = 1 , b = 3 , c = 4 and N = 6 then the answer should be 2 (3 + 3), but the greedy solution will give 3 (4 + 1 + 1).

General DP solution

Since a greedy solution won’t be helpful for many cases, we resort to DP.

Does this problem satisfies the requirements for using DP ie., does this problem uses has optimal substructure and overlapping subproblems?

Or in simpler terms can this problem be expressed in states (which depend upon previous states) and do the values of those states need to be re calculated multiple times?

The answer is YES

What are the state variables?

There is only one state variable, i.e., the amount to be spent. Considering a,b,c as constants, no other factor influences the number of coins to give. So, the only state variable needed is amt i.e., **the amount to be spent.**

So we can express the problem as a function **fun(amt)** which takes in the current state as the input (the amount to be spent) and returns the minimum number of coins to spend.

What are the transitions?

fun(amt) = min(fun(amt - a) , fun(amt - b) , fun(amt - c)) + 1

That’s it. A single transition would be sufficient for our case. How did we derive this transition? It’s simple.

We initially have to spend amt money. If we give a coin of denomination a then we are left with (amt - a) money to spend and we have spent 1 coin. Same is the case with b , c. So the optimal number of coins for spending amt money would be the minimum coins we have to spend for (amt - a) or (amt - b) or (amt - c).

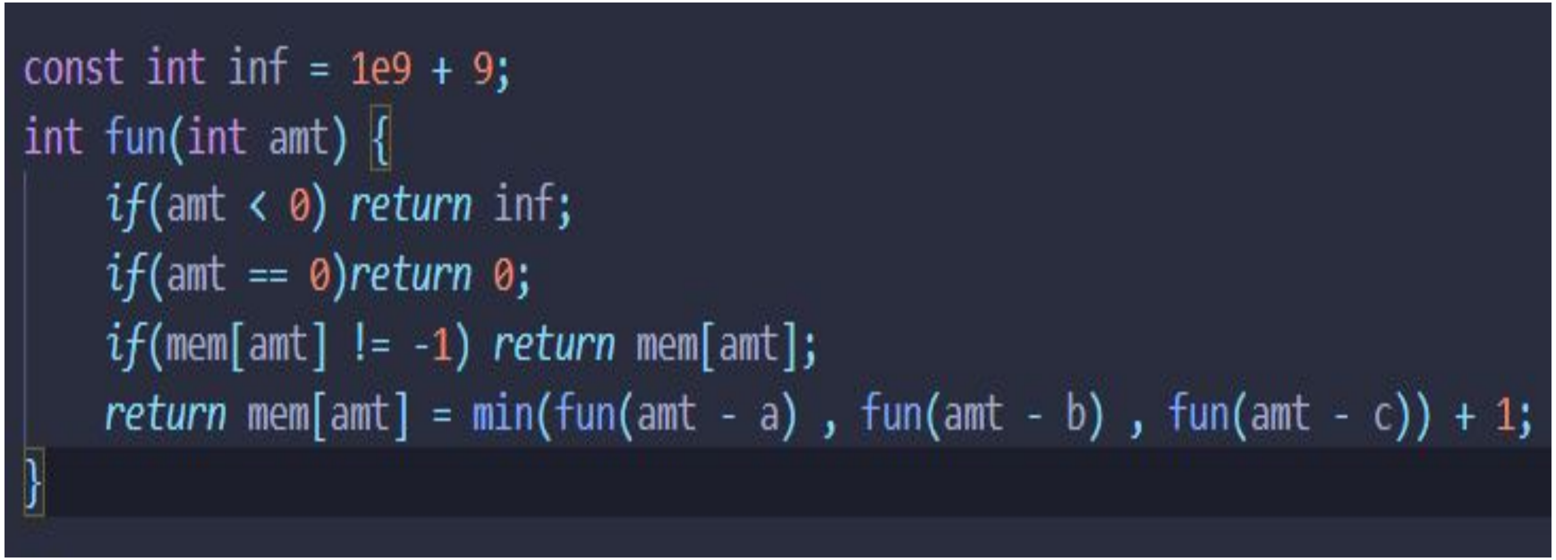
What is the Base case?

If n == 0 the obviously we can’t spend any money , so fun(0) = 0

If n < 0, this can be inferred as we being in a debt rather than having any money to spend. So this is an invalid case and any transaction that leads to this case is invalid.

However for ease of coding and calculation we can say if we have n < 0 then we have to spend an infinite number of coins. This is because if any transaction leads us to this case we’ll return an insane amount of coins to be spent which will be eventually rejected by our algorithm since any valid transaction is bound to have a finite (less than this case) answer.

Code



```
const int inf = 1e9 + 9;
int fun(int amt) {
    if(amt < 0) return inf;
    if(amt == 0)return 0;
    if(mem[amt] != -1) return mem[amt];
    return mem[amt] = min(fun(amt - a) , fun(amt - b) , fun(amt - c)) + 1;
}
```

Time complexity : O(amt)

Problem 2

SPOJ problem (Code : FARIDA)

What state variables to use?

Since the only thing important to represent a state is the number of monsters seen so far, so we'll take only one state.

fun(n) -> maximum number of coins that we can get considering we have only considered first n monsters.

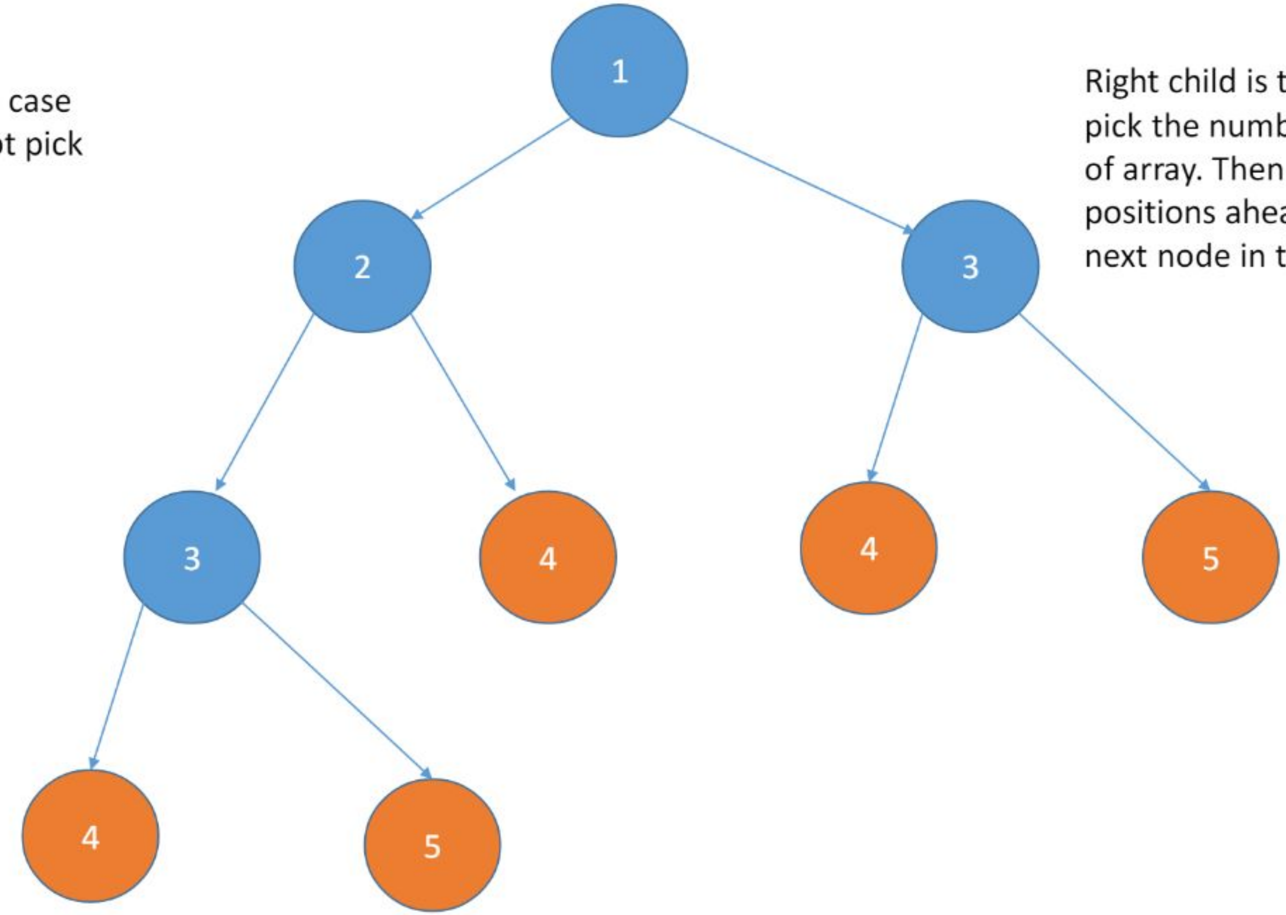
What would be the transitions?

- If we decide not to take coins from the nth monster, then fun(n) = fun(n+1)
- If we take coins from the current monster then we will only be able to consider monsters from (n+2)th position, so in this case fun(n) = fun(n+2) + coins[n]

So the transition :

fun(n) = max(fun(n+1) , fun(n+2) + coins[n])

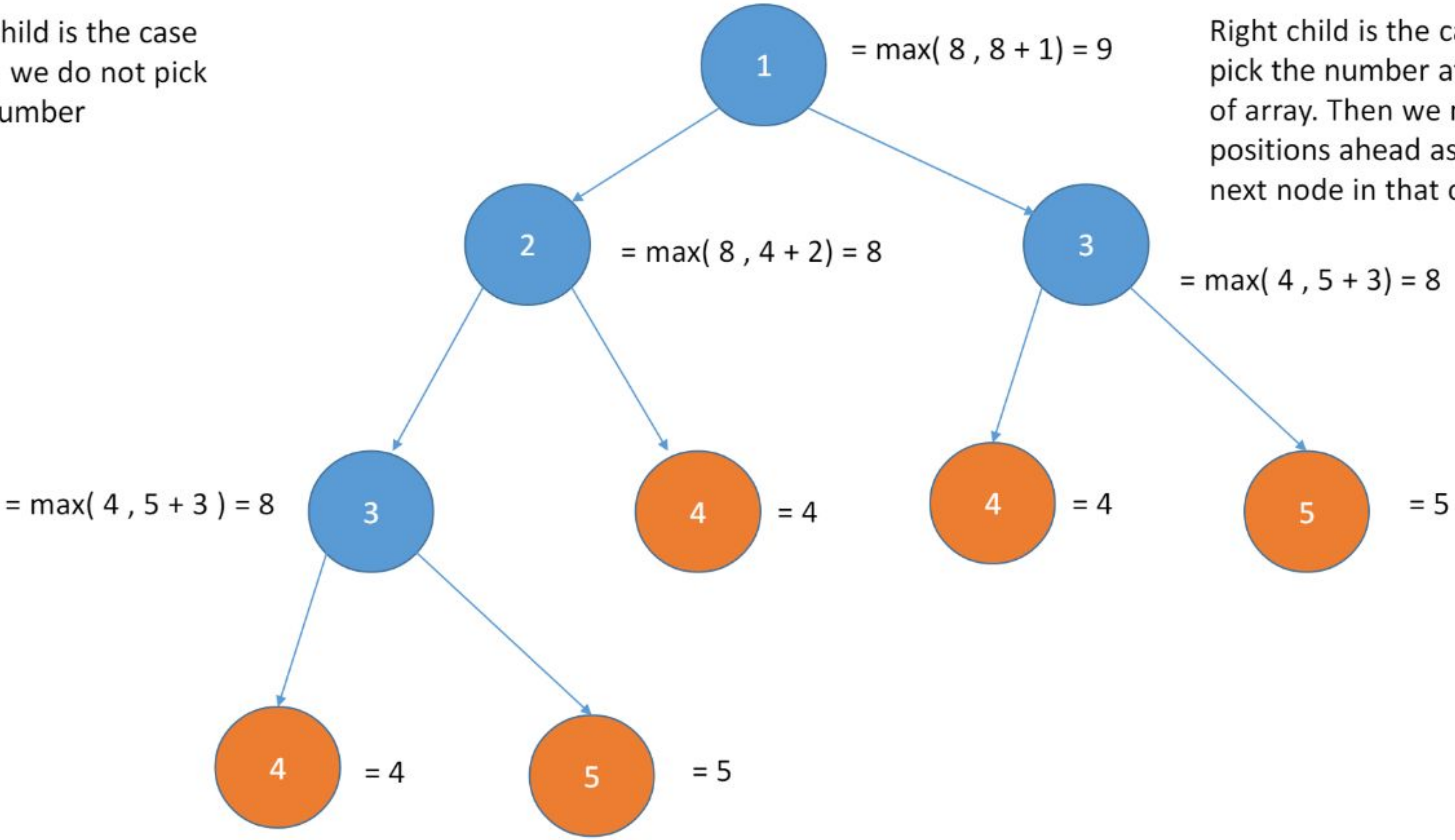
Left child is the case when we do not pick the number



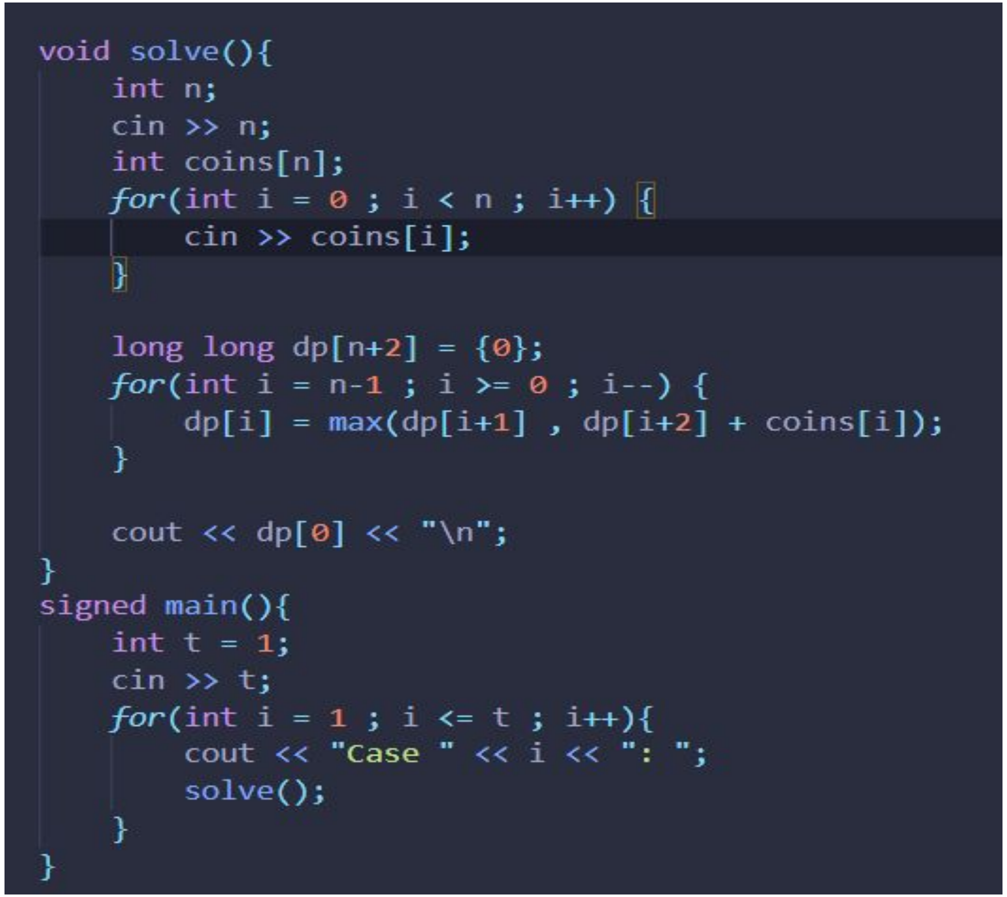
Right child is the case when we pick the number at that position of array. Then we move two positions ahead as we cannot pick next node in that case.

Left child is the case when we do not pick the number

Right child is the case when we pick the number at that position of array. Then we move two positions ahead as we cannot pick next node in that case.



Code



```
void solve(){
    int n;
    cin >> n;
    int coins[n];
    for(int i = 0 ; i < n ; i++) {
        cin >> coins[i];
    }

    long long dp[n+2] = {0};
    for(int i = n-1 ; i >= 0 ; i--) {
        dp[i] = max(dp[i+1] , dp[i+2] + coins[i]);
    }

    cout << dp[0] << "\n";
}
signed main(){
    int t = 1;
    cin >> t;
    for(int i = 1 ; i <= t ; i++){
        cout << "Case " << i << ": ";
        solve();
    }
}
```

Problem 3

SPOJ Problem (ACODE)

What will be the state variables?

Again we need only one state variable i.e., the number of indices processed.

Transitions?

Since a character can correspond to at most 2 digits, so there will be at most two branches originating from our recursive tree for every state.

fun(n) = fun(n-1) + fun(n-2) , but various conditions have to be checked.

Problems to practice

- https://atcoder.jp/contests/dp/tasks
- https://cses.fi/problemset/ (DP section)
- https://www.spoj.com/problems/tag/dynamic-programming