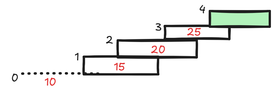
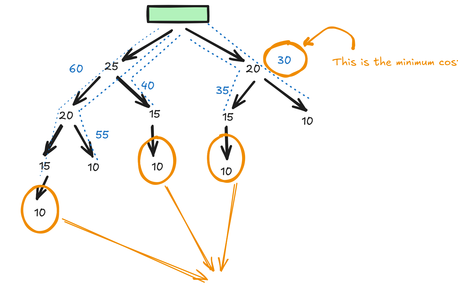
Minimum Cost of Climbing the Stairs (#746)



- Cost array is given such that [10,15,20,25]
- The steps are 0 , 1, 2 , 3
- The destination is 4th step.
- We need to reach the destination and there is no cost to stand at the destination step
- You can start from 0 step or 1 step
- pay 10 for starting from step 0.
- pay 15 for starting from step 1.
- Pay to stand on a particular step and now you from there can take one step or two steps to reach further stairs, depend on you, reach that step and pay the price for that step, unless it is destination step.
- Your goal is to reach destination step with minimum cost.



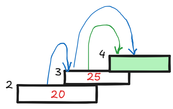
Why this node does not make any sense ?

I can start from either 0 or 1, If I am at 1, I have paid the cost of being at 15, If I go from 15 to 10, or actually if I have arrived from 0 to 1, I need to pay for 0 as well as 1, 15 + 10 = 25.

From 0 I can be at 2, by paying just 10 rupee, From 1, I can be at 2, by paying just 15 rupee. I would never want to reach 2 from 0 via 1, I will go directly from 0 or directly from 1.

Simple Recursion Logic

The logic is that, at wherever step you are you have reached there, only via two possibilities



- Either from the (n - 1)th stair or that (n - 2) th stair

- Cost is always adding, never getting subtracted based on the steps so only thing we can do is choose minimum cost at each step

- So reach the destination from that step, where cost was less, that is you should jump from Stair 2 (20) , pay 20 and use two steps

- Now you are at stair 2, Choose that stair from stair 1 or stair 0, whichever is less.

- Base Case : You Started from stair 1 or Stair 0.

- Very Important thing is that, there is the cost for each step but as you see the array is of size n only, there is no cost of the destination step, hence we will code accordingly

```
int solve( int n, vector<int> cost){

    // base case
    if( n == 0 ){
        return cost[0];
    }
    else if( n == 1){
        return cost[1];
    }

    // recursion logic
    int ans = cost[n] + min(solve( n - 1, cost),solve(n-2,cost);
    return ans;
}

int minCost(vector<int> cost){


    n = cost.size();
    int ans  = 0 + min(solve( n -1, cost),solve(n-2,cost) );
    return ans;

}
```

- From the main function, ans = 0 (no cost for destination) + min (cost of previous two stairs 3, 2)
- min(solve(3),solve(2)) gets called
- solve(3) is called, solve(3) = 25 + min(solve2,solve1)
- solve(2) is called, solve(2) = 20 + min(solve(1),solve(0))
- solve (1) is 15 , solve0 is called which is 10,
- solve2 : 20 + min (1solve1,solve0) = 20 + 10 = 30;
- solve(3) : 25 + min(30,15) = 40
- min(40, 30) is 30
- main function : 0 + 30

- return 30

ANS

- This is a bit optimized recursion logic, as we are just rejecting other path and we are not actually required to take all the different branches, but here there is same problem that it would need a large amount of compute as we are solving the terms like solve2, solve 1, solve 3. So there is a scope of applying DP.

Dynamic Programming : Recursion + Memoization [TOP DOWN APPROACH]

T.C : O(n)
S.C : O(n) + O(n)

```
int solve( int n , vector<int> &cost, vector<int> & dp){

    // base cases
    dp[0] = cost[0];
    dp[1] = cost[1];

    // Memoization
    if( dp[n] != -1 ) {
        return dp[n];
    }

    // Recursion incorporation with dp
    dp[n] = cost[n] + min(solve(n-1, cost,dp),solve(n-2,cost,dp) );
    return dp[n];

}

int minCost( vector<int> & cost ){
    int n = cost.size();

    // creating a dp array
    vector<int> dp(n + 1, -1);

    dp[n] = 0 + min( solve( n - 1, cost,dp), solve(n -2, cost,dp) );
    return dp[n];
}
```

- Explanation is same as above, it just stores the values in the dp array, and we can fetch that value and we not need to make recursive tree of each subproblem again and again

Dynamic Programming : Tabular Method (BOTTOM UP Approach)

T.C : O(n)
S.C : O(n)

```
int solve(vector<int> & cost, int n){
    // step 1 : create the dp array
    vector<int> dp(n+1);
    // step 2 : base case analysis
    dp[0] = cost[0];
    dp[1] = cost[1];

    // 3rd step : for loop
    for( int i = 2; i <n ; i++){
        dp[i] = cost[i] + min(dp[i - 1] , dp[i - 2]);
    }

    return min(dp[n-1],dp[n-2]);
}

int main( ){
    vector<int> vec = {10,2,4,50,2};
    int output = solve(vec,5);
    cout << output << endl;
    return 0 ;
}
```

Dynamic Programming : Space Optimization

T.C : O(n)
S.C : O(1)

```
int Solve(vector<int> &cost, int n){

    prev = cost[1];
    prevprev = cost[0];

    for( int i = 2; i < n; i ++ ){
        curr = cost[i] + min(prev, prevprev);
        prevprev = prev;
        prev = curr;

    }

    return min(prev,prevprev);


}
```

using only two variables, we can optimize the space complexity