Maximum non adjacent Sum Elements

RECURSIVE APPROACH

T.C. : O(2^) as each function is giving two calls except the base cases.
S.C : O(n) as the biggest chain in stack space is created by not pick as f(3) -> f(2) -> f(1) -> f(0) recursions

we are given array as {2,1,4,9}, we have to find the maximum non adjacent sum, which is like 2 + 4, 2 + 9, 1 + 9 possibilities and the maximum sum is 11.

We will check all the possibilities and print only those subsequences which follow the condition of non adjacency

f(n) means the maximum sum of non adjacent elements from the index 0 to n.

f(3) -> maximum sum possible from index 0 to 3 by pick or not pick
f(2) -> maximum sum possible from index 0 to 2 by pick or not pick
f(1) -> maximum sum possible from index 0 to 1 by pick and not pick

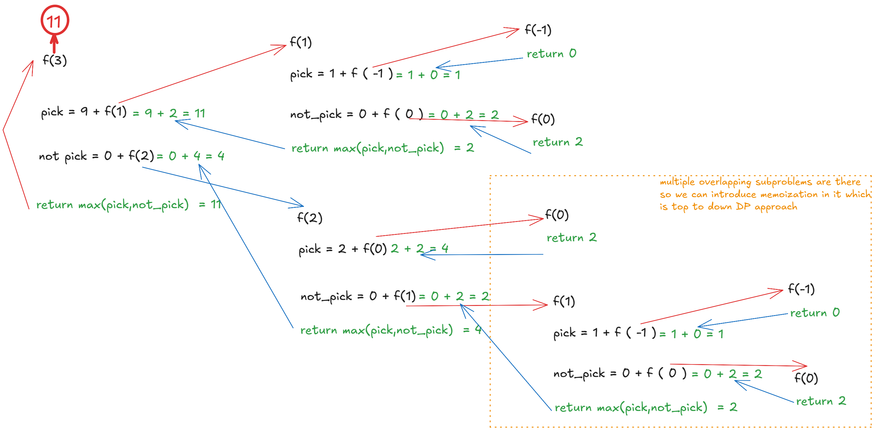
```
int sum( int index, int arr[]){

    if( index == 0){
        // 0 pe pahunch hi gaye ho to sum ko maximise kar lo, arr[0] ko include karke
        return arr[0];
    }

    else if( index < 0){
        return 0;
    }

    int pick = arr[ index ] + sum(index-2, arr);
    int not_pick = 0 + sum( index - 1, arr);

    return max(pick, not_pick);
}

int main(){
    // Here you are given the array
    int arr[4] = {2,1,4,9};
    int n = 4 ;

    int ans = sum(3, arr);

    cout << ans << endl;

}
```

Ham last index se shuru karenge,
f (3) means that maximum non adjacent sum possible, jo ki actually hame hi nikalna hai, ab isme do case possible hain, jis index pe hun which is 3 usey include karu ya exclude karun

include/pick : arr[3] + f(1)
isme pick kar li value 3rd element ki ab mai 2nd element ko pick nahi kar sakta, mtlab f(2) which says ki maximum non adjacent sum possible from index 0 to 2 by pick or not pick, toh pick to nahi kar sakta, so maximum sum is only by not picking 2, so why not directly jump to 1, which says maximum sum possible from index 0 to 1 by picking 1 or not picking 1.

exclude/not pick : 0 + f(2)
0 mtlb mene pick nahi kiya index3 ka element to uski jagah 0 and ab mai issme f(2) add kar dunga.

// base case:
0 pe pahunch gaye : to sum maximise karne ke liye index0 include karlo
-ve pe pahuch gaye : ye to possible nahi hai, to 0 add kar do bas sum me



TOP DOWN APPROACH - DP

T.C. : O(n), you store the result in dp array and each sum is there in the dp array and can be accessed.
S.C : O(n) + O(n) = O(n) as the biggest chain in stack space is created by not pick as f(3) -> f(2) -> f(1) -> f(0) recursions and extra space is of dp array

```
int sum(int n, vector<int> &nums, vector<int> &dp){

    // base case
    dp[0] = nums[0];

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


    // Introduce the dp in this logic


    int pick = nums[n] ;
    if(n > 1){
        pick = pick + sum( n - 2, nums, dp);
    }
    int not_pick = 0 + sum( n - 1, nums, dp);


    // Put the result into the dp
    dp[n] = max(pick, not_pick);

}

int maximumNonAdjacentSum(vector<int> &nums){

    // We can introduce memoization in it
    // Introducing the dp array

    vector<int> dp( nums.size(), - 1) ;

    sum( nums.size() - 1, nums, dp );

    int ans = dp[nums.size()-1];
    return ans;

}
```

- We created the size of dp array same as of the nums array, and initialized all elements with -1.
- Base case is that dp[0] = 0 and there are no negative entries in dp array.
- To use dp, we incorporate the line that if dp[n] != -1 return dp[n].
- Then we write the logic that we developed from the recursion after incorporating dp in it
- then we save that into dp[n] according to the indexes
- your answer is dp[n].
- Everrthing is just same you are now just solving the cases and storing the results in the dp array and you know that you result is in dp[n-1].

BOTTOM UP APPROACH (TABULATION)

T.C. : O(n), you store the result in dp array and each sum is there in the dp array and can be accessed.
S.C : O(n) as this is the dp array legnth.

```
 int main(){

    int arr[4] = {2, 1, 4, 9};

    vector<int> dp( 4, -1 );
    dp[0] = arr[0];

    for( int i = 1; i < 4; i++ ){
        int take = arr[i] + 0;
        if(i > 1){
            take = take + dp[ i - 2];
        }
        int not_take = 0 + dp[i - 1];

        dp[i] = max(take,not_take);

    }

    cout << dp[3];


 }
```

- This is bottom up approach in which we make the answer from the base cases and go up
- we know that dp[0] = 0 is arr[0]
- we manually check the case that negative indexing do not come with help of if
- we build each answer from the bottom.
- Space Complexity is just the dp array O(n) and time complexity is the dp array which is O(n)
- Our answer is dp[n] only.

SPACE OPTIMIZATION

T.C. : O(n), you store the result in dp array and each sum is there in the dp array and can be accessed.
S.C. : O(1), you know that we are taking only three spaces as each time,

```
int main(){

    int arr[4] = {2, 1, 4, 9};

    int prevprev = 0;
    int prev = arr[0];

    int curr;


    for( int i = 1; i < 4; i++ ){

        int take = arr[i];
        if( i > 1){
            take = arr[i] + prevprev;
        }

        int not_take = 0 + prev;

        curr = max(take, not_take);

        prevprev = prev;
        prev = curr;

    }

    cout << curr;


}
```

- This is the space optimized approach in which we just take 3 spaces
- At any point of time, we only require two previous elements to calculate the current, and then we just update those.
- Logic is same.