LEETCODE | 198 |

DATE

House Robber

Input: nums = [2, 7, 9, 3, 1]
Output: 12

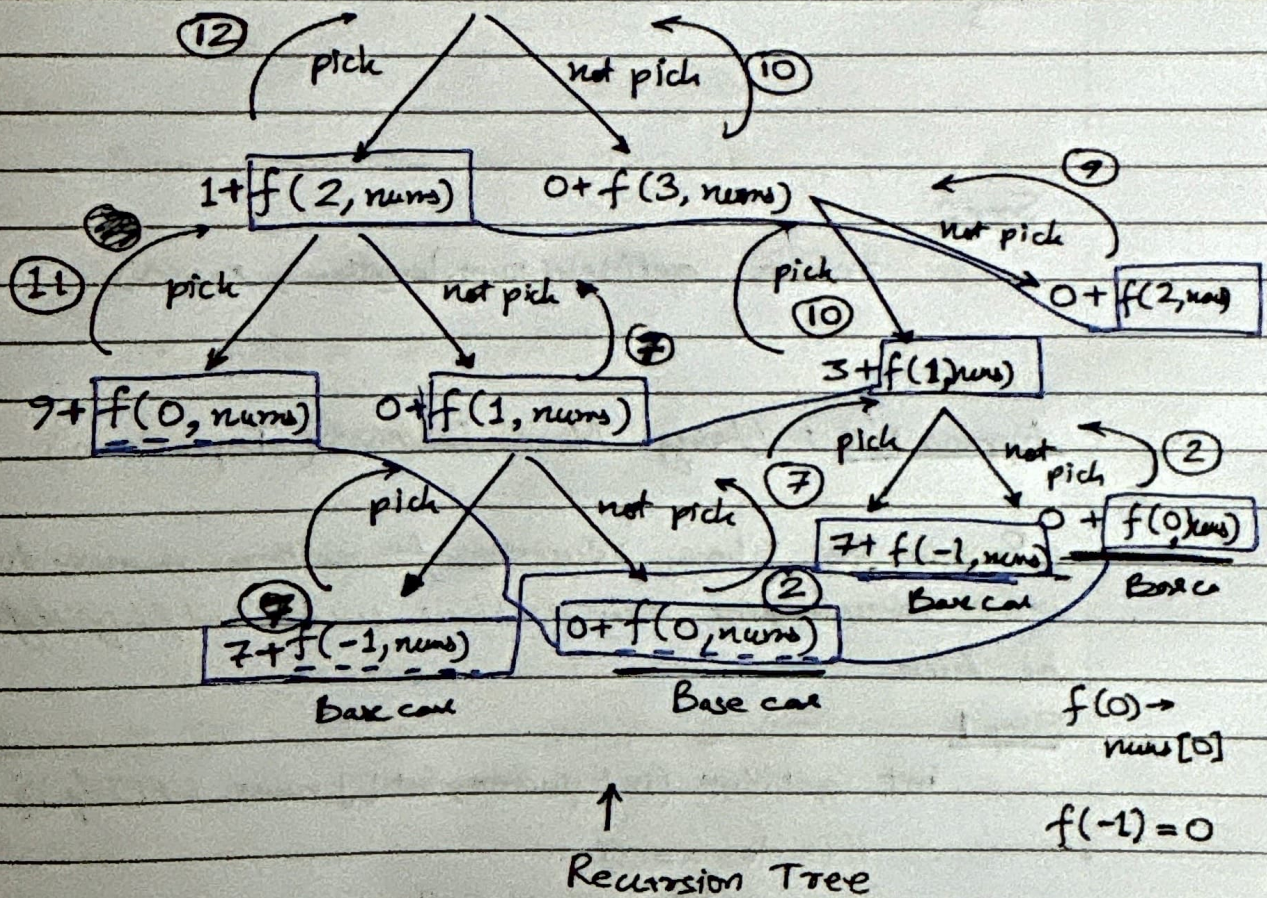
Approach 1: As robber needs to pick alternate houses so we need to try all possibilities i.e. Recursion
We can start from last element index

int n = nums.length nums = [2, 7, 9, 3, 1] (indices 0 1 2 3 4)

n = 5

f(index, nums) → f(n-1, nums) → f(4, nums)

f(4, nums) → 12 ← Answer



f(0) → nums[0]

f(-1) = 0

↑

Recursion Tree

Step 1

The recursion fn looks like below:

```
int getMax(int index, int[] nums){
    if(index==0){
        return nums[0];
    }
    if(index<0){
        return 0;
    }
    //pick
    int pick = nums[index] + getMax(index-2, nums);
    //not pick
    int notpick = 0 + getMax(index-1, nums);
    return Math.max(pick, notpick);
}
```

Step 2

```
return getMax(num.length-1, nums);
```

Approach 2: Using Memoization (Top down)

Rewrite the above recursive fn with a memory to save overlapping sub-problems underlined / highlighted in blue

Step 1

```
int getMax(int index, int[] nums, int[] dp){
    if(index==0)
        return nums[0];
    if(index<0)
        return 0;
```

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

// pick
int pick = nums[index] + getMax(index-2, nums, dp);

//notpick
int notpick = 0+ getMax(index-1, nums, dp);

dp[index] = Math.max(pick, notpick);

return dp[index];

}
```

Step2

Now call the above memoized fn as below:

```
int n = nums.length;                          TC: O(n)
int[] dp = new int[n];                        SC: O(n)
                                                  +O(n)
Arrays.fill(dp,-1);                               ↑
                                              Recursion
return getMax(n-1, nums, dp);                 Stack
```

Approach 3: Using Tabulation (Bottom Up)

We can optimize above approach by not calling methods recursively. Then we can save space of O(n)

Step1 Let's re-write the above function as below:

```
int getMax(int[] nums, int[] dp){
    dp[0] = nums[0];
    int n = nums.length;

    for(int i=0; i<n; i++){
        //pick
        int pick = nums[i];
        if (i>1){
            pick += dp[i-2];
        }
        //not pick
        int notpick = 0;
        if(i>0){
            notpick += dp[i-1];
        }
        dp[i] = Math.max(pick, notpick);
    }
return dp[n-1];
```

Step2 Form dp = new int [n];

Return getMax(nums, dp);

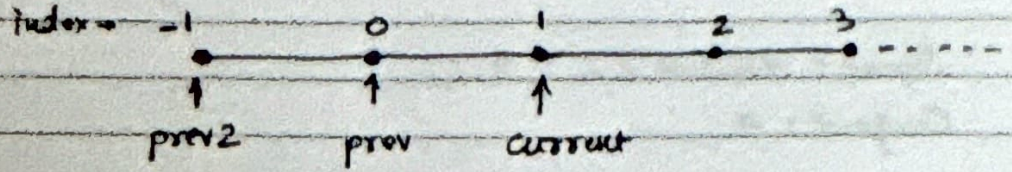
TC = O(n)

SC - O(n)

We can further optimize it by saving extra space of O(n) by doing space optimization by below approach

DATE

Approach 4 With Space optimization



Step1 Rewrite the above f^n with three variables/states

```
int getMax(int[] nums) {
    int prev2 = 0;
    int prev = nums[0];
    int n = nums.length;

    for(int i=0; i<n; i++) {
        //pick
        int pick = nums[i];
        if(i>1) {
            pick += prev2;
        }
        //not pick
        int notpick = 0;
        if(i>0) {
            notpick += prev;
        }
        int current = Math.max(pick, notpick);

        prev2 = prev;
        prev = current;
    }
    return prev;
}
```

Step2 Return getMax(nums);

TC: O(n)
SC: O(1)