House Robber IV

---

leetcode link : Click

Approach - 1

idea - backtracking approach

Time : $O(2^n)$

Space : $O(1)$

code :-

```
class Solution {
    /*[TLE]⚠✔ Approach - 1 (backtracking & recursive i.etake or dont take approach
)

        explanation :- we choose 1 house, or we dont choose it

        ⚠Time : O(2^n) because we have 2 choices for every n elements to pick or not
pick
        Space : O(n)


    */
private:
    // Fun.3 : isSafeToInsert()
    bool isSafeToInsert(vector<int> &currIndexes, int index){

        // we need to check if all will be fine if we insert the 'index' into our
currINdexes vector, so insert 'index' at its end, make sure to pop it before the
function ends
        currIndexes.push_back(index);

        int size = currIndexes.size();
        // if currIndexes[] is of size = 1 or 0, then return true
        if(size == 0 || size == 1){
            currIndexes.pop_back();
            return true;
        }

        // check last 2 elements of the currIndexes[] if their difference is less then
or equal to 1, then return false
        if((currIndexes[size - 1] - currIndexes[size - 2]) <= 1){
            currIndexes.pop_back();
            return false;
        }

        else{
            currIndexes.pop_back();
            return true;
        }
    }

    // Fun.2 : solve()
    void solve(vector<int> &houses, int k, vector<int> &currIndexes, int &ans, int
index){

        // base case - if we explored all the houses, then find the max of all the
currHouses and then store the min(ans,maxMoney) in ans
        if(index == houses.size()){
            // only save the ans when the 'Currindexes' i.e houses explored are more
then or equal to k,
            if(currIndexes.size() >= k){

                int maxMoney = INT_MIN;
                for(int i = 0; i < currIndexes.size(); i++)
                    maxMoney = max(maxMoney, houses[currIndexes[i]]);
                ans = min(ans, maxMoney);
            }

            return;
        }

        // choice number - 1 (if it is same to explore a house = 'index', then explore it)
        if(isSafeToInsert(currIndexes, index)){
            currIndexes.push_back(index);
            solve(houses, k, currIndexes, ans, index + 1);
            currIndexes.pop_back();   // backtrack
        }

        // choice number-2 (dont explore the house = 'index')
        solve(houses, k, currIndexes, ans, index + 1);
    }


public:
    int minCapability(vector<int>& houses, int k) {

        // we need to check all possible cobminations of the indexes of houses , so we need to maintain that in a 'currIndexes' vector, also we will be maintaining the 'index = 0' and a int 'ans = INT_MAX' (coz we will return the minimum value of ans)
        int index = 0;
        vector<int> currIndexes;
        int ans = INT_MAX;

        solve(houses, k, currIndexes, ans, index);

        return ans;
    }
};
```

Approach - 2

idea - using binary search

Time : $O(n * logn)$

Space : $O(1)$

code :-

```
class Solution {
private:
    /*✔⭐Approach - 2 (using binary search)

        Explanation :-

                ->// Fun2. : returns true if 'mid' can be a possoble ans, else return false
                bool isMidPossileAns(vector<int>&nums, int k, int mid){

                    step1 :  we need to maintain a index (to traverse the whole array), and also maintain a k = k, so that we do not pick elements more then k
                        step2  : if the ith element of nums array is smaller or equal to mid, that means it can be added to the robery, so do i += 2, k-- (because 1 house is now robbed)
                        step 3 :  else if ith element is greater then mid, then this element can not be included, i.e this ith house cant be robbed , so skip it i.e i++
                        step 4 : when all k are robbed then return true
                    step 5 : loop ends and  when all k are robbed then return true, if not then return false

                ->// Main fun
                    step 1 :  find min and max element of the vector nums, and then store them as low and high
                    step 2 :  do a binary search traversal for low to high
                        step 3 : Find mid
                        step 4 : call fun.2 isMidPossibleAns(nums, k, mid), it it returns yes then store mid in 'ans' coz mid can be a possoble ans, and search for a smaller ans in left half
                        step 5 :  else if not possible ans, then find it in the right half
                    step 6 :  return ans

            ☑T : O(NlogN)
            ☑S : O(1)

    */

    // Fun2. : returns true if 'mid' can be a possoble ans, else return false
    bool isMidPossileAns(vector<int>&nums, int k, int mid){

        // we need to maintain a index (to traverse the whole array), and also maintain a k = k, so that we do not pick elements more then k
        int index = 0;
        while(index < nums.size()){

            // if the ith element of nums array is smaller or equal to mid, that means it can be added to the robery, so do i += 2, k-- (because 1 house is now robbed)
            if(nums[index] <= mid){
                index += 2;
                k--;
            }

            // else if ith element is greater then mid, then this element can not be
included, i.e this ith house cant be robbed , so skip it i.e i++
            else index++;

            // when all k are robbed then return true
            if(k == 0) return true;
        }

        // loop ends and  when all k are robbed then return true, if not then return
false
        return k==0;
    }

public:
    // Main fun
    int minCapability(vector<int>& nums, int k) {

        int ans = INT_MAX;
        // find min and max element of the vector nums, and then store them as low and
high
        int low, high;
        for(int i = 0; i < nums.size(); i++){
            low = min(low,nums[i]);
            high = max(high,nums[i]);
        }

        // do a binary search traversal for low to high
        while(low <= high){

            int mid = low + (high - low)/2;

            // call fun.2 isMidPossibleAns(nums, k, mid), it it returns yes then store
mid in 'ans' coz mid can be a possoble ans, and search for a smaller ans in left half
            if(isMidPossileAns(nums, k, mid)){
                ans = mid;
                high = mid - 1;
            }

            // else if not possible ans, then find it in the right half
            else low = mid + 1;
        }

        return ans;
    }
};
```