2560. House Robber IV

There are several consecutive houses along a street, each of which has some money inside. There is also a robber, who wants to steal money from the homes, but he **refuses to steal from adjacent homes**.

The **capability** of the robber is the maximum amount of money he steals from one house of all the houses he robbed.

You are given an integer array `nums` representing how much money is stashed in each house. More formally, the `ith` house from the left has `nums[i]` dollars.

You are also given an integer `k`, representing the **minimum** number of houses the robber will steal from. It is always possible to steal at least `k` houses.

Return *the* ***minimum*** *capability of the robber out of all the possible ways to steal at least* `k` *houses*.

Example 1:

```
Input: nums = [2,3,5,9], k = 2
Output: 5
Explanation:
There are three ways to rob at least 2 houses:
- Rob the houses at indices 0 and 2. Capability is max(nums[0], nums[2]) = 5.
- Rob the houses at indices 0 and 3. Capability is max(nums[0], nums[3]) = 9.
- Rob the houses at indices 1 and 3. Capability is max(nums[1], nums[3]) = 9.
Therefore, we return min(5, 9, 9) = 5.
```

Example 2:

```
Input: nums = [2,7,9,3,1], k = 2
Output: 2
Explanation: There are 7 ways to rob the houses. The way which leads to minimum
capability is to rob the house at index 0 and 4. Return max(nums[0], nums[4]) = 2.
```

Constraints:

- $1 \leq nums.length \leq 10^5$
- $1 \leq nums[i] \leq 10^9$
- $1 \leq k \leq (nums.length + 1)/2$

2560. House Robber IV

Overview

This is yet another problem based on the **House Robber** series! This article will assume some prior knowledge of the original version, so you may want to solve that before this one. So before diving in, let's quickly recall the core idea behind the original problem.

In the classic House Robber problem, the goal is to maximize the total amount stolen from a row of houses while following one key restriction: the robber cannot rob two consecutive houses. This forces the robber into a branched decision making process that at each house, they must choose whether to rob it or skip it. If they robs it, they must add its value to the best amount stolen from two houses before. If they skips it, they simply takes the best amount stolen from the previous house. This naturally leads to a recursive relationship:

```
maxAmount(houseNumber)=max(maxAmount(houseNumber-1),
maxAmount(houseNumber - 2) + nums[houseNumber])
```

Using dynamic programming, we can store these values and efficiently compute the maximum amount the robber can steal.

In this current problem, the robber still has to follow the restraint that they cannot steal from two consecutive houses. However, this time, instead of maximizing the total reward, they want to maximize the minimum amount stolen amount all houses.

Similar to the original problem, we can think of a recursive relation to solve this. Again, we have two choices:

1. Rob the current house (but then we must skip the next house).
2. Skip the current house and move forward.

However, unlike the original problem, we need an additional condition—ensuring that we rob at least k houses. The dynamic programming solution involves a state `dp[houseIndex][numberOfHousesRobbed]`. Since we iterate over n houses and track up to k robbed houses, the problem becomes more complex, and solving it with dynamic programming takes $O(n \cdot k)$ time.

Problems that require maximizing the minimum or minimizing the maximum often suggest a binary search approach. Instead of searching through indices or subsets directly, we can binary search on the reward value itself. By determining whether a given minimum reward is achievable, we can efficiently narrow down the possible solutions. If you're unfamiliar with this technique, you can refer to this guide to learn more about binary search.

2560. House Robber IV

Approach: Binary Search on the answer

Intuition

Instead of focusing on maximizing a total sum, we need to guarantee that the minimum amount stolen from any robbed house is as large as possible. A brute force approach would involve checking every possible way to rob k houses while obeying the adjacency constraint, but this would be too slow for large inputs.

A more efficient way to approach this problem is to recognize that we are trying to push the minimum stolen amount as high as possible while still satisfying the condition of robbing at least k houses. This naturally leads to using **binary search** on the minimum reward that the robber can secure.

Let $S^{c}_{>=k}(A)$ all possible non-contiguous subsequences of at least size k where all elements are are less or equal to c :

$$S^{c}_{>=k}(A)=\{(a_{i_1},a_{i_2},\ldots,a_{i_m})\,|\,k\leq m\leq n,1\leq i_1<i_2<\ldots<i_m\leq n,i_{j+1}-i_j\geq 2,a_{i_1},a_{i_2},\ldots,a_{i_m}\leq c\}$$

where:

- $S^{c}_{>=k}(A)$: represents the set of all valid subsequences of the list A
- The subsequence size must be at least k ($m\geq k$)
- The indices must be strictly increasing: $1\leq i_1<i_2<\cdots<i_k\leq n$
- The indices must be **non-contiguous**: $i_{j+1}-i_j\geq 2$
- All elements in the subsequence must satisfy $a_{i_j}\leq c$.

Lemma #1:

Let $S^{c}_{>=k}(A)$ be the set of all valid subsequences of at least size k where all elements are $\leq c$.

Let $S^{c'}_{>=k}(A)$ be the set of all valid subsequences where all elements are $\leq c'$, with $c'>c$.

If $S_{start}\subset S^{c}_{>=k}(A)$, then S_{start} is a subset of every sequence in $S^{c'}_{>=k}(A)$, where S_{start} is the first valid subsequences that we can build.

Observations

- Increasing c to c' allows us to add more elements to potential subsequences. This is because increasing the upper bound c **only adds elements** but does not remove any previously valid ones.
- Since S_{start} is already valid under c, every valid subsequence for c' will include at least the elements of S_{start}.

Proof by contradiction

Suppose, for contradiction, that there exists a valid subsequence $T \subset S^{c'}_{>=k}(A)$ **s**uch that S_{start} is not fully contained in T.

This means there exists an element $x \in S_{start}$ such that $x \notin T$. But , since S_{start} is valid for c means $x \leq c$, and since $c' > c$ means $x < c'$, si T should contains x. So, Since valid subsequences for c' must be at least of size k and contain elements $\leq c'$, they must still include all previously valid elements of $S^{c}_{>=k}(A)$. We conclude, that S_{start} **is a subset of every sequence in** $S^{c'}_{>=k}(A)$

Lemma #2

Let $S^{c}_{>=k}(A)$ be the set of all valid subsequences of at least size k where all elements are $\leq c$.

Let $S^{c'}_{>=k}(A)$ be the set of all valid subsequences where all elements are $\leq c'$, with $c' > c$.

We are given that $S_{start} \subset S^{c}_{>=k}(A)$

if $c = min(max(S_{start}))$ **, then** c **is the minimum value of** $max(S^{c'}_{>=k}(A))$

proof

Let T a valid subsequence such that $T \subset S^{c'}_{>=k}(A)$, means it exists max_t, such that $max_t = max(T)$.

From **lemma#1,** if $T = S_{start}$ and $T \subset S^{c'}_{>=k}(A)$ => $max_t = c$,

if $T \neq S_{start}$ and $T \subset S^{c'}_{>=k}(A)$ => $max_t = c'$ or $max_t = x$ such that $c < x < c'$

Since $c' > x > c$, so $min(c, x, c') = c$

So, $min(max(S_{start}), max(S^{c'}_{>=k}(A))) = c$

<u>Conclusion</u>: We just need to find the smallest capability c ***that makes*** S_{start}

2560. House Robber IV

```
/*
    Binary search one the answer
    Time complexity:O(n log hi)
    Space complexity: o(1)
*/
class Solution {
    public:
        int minCapability(std::vector<int>& nums, int k) {
            int n=nums.size();
            int hi=*std::max_element(nums.begin(),nums.end());
            auto is_possible=[&](int capability)->bool{
                int robbed_houses=0;
                int i=0;
                while(i<n){
                    // If we take the current house
                    if(nums[i]<=capability){
                        robbed_houses++; // Count it
                        i+=2; // Skip adjacent house
                    }
                    // If we skip the current house, take the next one
                    else i++;

                    // At least we robbed k houses
                    if(robbed_houses>=k) return true;
                }
                return false;
            };
            int lo=1;
            while(lo<hi){
                int capability=(lo+hi)>>1;
                if(is_possible(capability)) hi=capability;
                else lo=capability+1;
            }
            return lo;
        }
};
```