<https://leetcode.com/problems/house-robber-iv/description/>

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 <= nums.length <= 10^5

1 <= nums[i] <= 10^9

1 <= k <= (nums.length + 1)/2

**Attempt 1: 2025-06-02**

**Solution 1: Native DFS (60 min, TLE 28/64)**

**Style 1: void return**

class Solution {

    int minCap = Integer.MAX\_VALUE;

    public int minCapability(int[] nums, int k) {

        helper(nums, k, 0, 0, Integer.MIN\_VALUE, false);

        return minCap;

    }

    private void helper(int[] nums, int k, int index, int robbedCount, int curMax, boolean prevRobbed) {

        // Base case: All houses passed

        if(index == nums.length) {

            if(robbedCount >= k) {

                minCap = Math.min(curMax, minCap);

            }

            return;

        }

        // Option 1: Skip current

        helper(nums, k, index + 1, robbedCount, curMax, false);

        // Option 2: Rob current

        if(!prevRobbed) {

            int newMax = Math.max(curMax, nums[index]);

            helper(nums, k, index + 1, robbedCount + 1, newMax, true);

        }

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

**Style 2: void return without 'prevRobbed'**

class Solution {

    int minCap = Integer.MAX\_VALUE;

    public int minCapability(int[] nums, int k) {

        helper(nums, k, 0, 0, Integer.MIN\_VALUE);

        return minCap;

    }

    private void helper(int[] nums, int k, int index, int robbedCount, int curMax) {

        if(index >= nums.length) {

            if(robbedCount >= k) {

                minCap = Math.min(curMax, minCap);

            }

            return;

        }

        // Option 1: Skip current

        helper(nums, k, index + 1, robbedCount, curMax);

        // Option 2: Rob current

        int newMax = Math.max(curMax, nums[index]);

        helper(nums, k, index + 2, robbedCount + 1, newMax);

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

**Style 3: int return without 'prevRobbed'**

class Solution {

    public int minCapability(int[] nums, int k) {

        return helper(nums, k, 0, 0, Integer.MIN\_VALUE);

    }

    private int helper(int[] nums, int k, int index, int robbedCount, int curMax) {

        // Base case 1: Robbed enough houses

        // Why put 'robbedCount >= k' ahead of 'index >= nums.length' ?

        // Because in native DFS version its more strict condition

        // in base case as nested inside 'index >= nums.length'

        if(robbedCount >= k) {

            return curMax;

        }

        // Base case 2: No more houses to consider

        if(index >= nums.length) {

            // Invalid, didn't rob enough houses till the

            // end hence return potential largest value

            return Integer.MAX\_VALUE;

        }

        // Option 1: Skip current

        int skip = helper(nums, k, index + 1, robbedCount, curMax);

        // Option 2: Rob current

        int newMax = Math.max(curMax, nums[index]);

        int rob = helper(nums, k, index + 2, robbedCount + 1, newMax);

        return Math.min(skip, rob);

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

**Style 4: int return without 'prevRobbed' and 'curMax'**

class Solution {

    public int minCapability(int[] nums, int k) {

        return helper(nums, k, 0, 0);

    }

    private int helper(int[] nums, int k, int index, int robbedCount) {

        // Base case 1: Robbed enough houses

        // Why put 'robbedCount >= k' ahead of 'index >= nums.length' ?

        // Because in native DFS version its more strict condition

        // in base case as nested inside 'index >= nums.length'

        if(robbedCount >= k) {

            return 0;

        }

        // Base case 2: No more houses to consider

        if(index >= nums.length) {

            // Invalid, didn't rob enough houses till the

            // end hence return potential largest value

            return Integer.MAX\_VALUE;

        }

        // Option 1: Skip current

        int skip = helper(nums, k, index + 1, robbedCount);

        // Option 2: Rob current

        // 'nums[index]' means current house's value becomes the new local max

        // 'helper(nums, k, index + 2, robbedCount + 1)' means recurse on remaining houses

        int rob = Math.max(nums[index], helper(nums, k, index + 2, robbedCount + 1));

        return Math.min(skip, rob);

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

**Solution 2: Memoization (60 min)**

**Style 1: Based on void return (TLE, 33/64)**

class Solution {

    int minCap = Integer.MAX\_VALUE;

    public int minCapability(int[] nums, int k) {

        Map<String, Integer> memo = new HashMap<>();

        helper(nums, k, 0, 0, Integer.MIN\_VALUE, false, memo);

        return minCap;

    }

    private void helper(int[] nums, int k, int index, int robbedCount, int curMax, boolean prevRobbed, Map<String, Integer> memo) {

        // Base case: All houses passed

        if(index == nums.length) {

            if(robbedCount >= k) {

                minCap = Math.min(curMax, minCap);

            }

            return;

        }

        String key = index + "\_" + robbedCount + "\_" + curMax + "\_" + prevRobbed;

        if(memo.containsKey(key)) {

            return;

        }

        // Option 1: Skip current

        helper(nums, k, index + 1, robbedCount, curMax, false, memo);

        // Option 2: Rob current

        if(!prevRobbed) {

            int newMax = Math.max(curMax, nums[index]);

            helper(nums, k, index + 1, robbedCount + 1, newMax, true, memo);

        }

        memo.put(key, minCap);

    }

}

Time Complexity: O(n⋅k⋅m)

Number of Unique States:

The memoization key is index\_robbedCount\_curMax\_prevRobbed, where:

index ranges from 0 to n-1 (n = nums.length).

robbedCount ranges from 0 to k.

curMax can take up to m unique values (m = max(nums)).

prevRobbed is a boolean (true/false).

Total unique states: O(n⋅k⋅m⋅2) = O(n⋅k⋅m).

Work per State:

Each state performs O(1) operations (excluding recursive calls).

Recursive calls are memoized, so each state is computed only once.

Overall Time Complexity:O(n⋅k⋅m)

We compute each of the O(n⋅k⋅m) states exactly once.

Space Complexity

Memoization Storage:

The HashMap stores up to O(n⋅k⋅m) entries.

Recursion Stack:

Maximum recursion depth: O(n) (when skipping all houses).

Each stack frame uses O(1) space.

Global Variable:

minCap uses O(1) space.

Overall Space Complexity:O(n⋅k⋅m)

Dominated by the memoization table.

**Style 2: Based on void return without 'prevRobbed' (TLE, 33/64)**

class Solution {

    int minCap = Integer.MAX\_VALUE;

    public int minCapability(int[] nums, int k) {

        Map<String, Integer> memo = new HashMap<>();

        helper(nums, k, 0, 0, Integer.MIN\_VALUE, memo);

        return minCap;

    }

    private void helper(int[] nums, int k, int index, int robbedCount, int curMax, Map<String, Integer> memo) {

        if(index >= nums.length) {

            if(robbedCount >= k) {

                minCap = Math.min(curMax, minCap);

            }

            return;

        }

        String key = index + "\_" + robbedCount + "\_" + curMax;

        if(memo.containsKey(key)) {

            return;

        }

        // Option 1: Skip current

        helper(nums, k, index + 1, robbedCount, curMax, memo);

        // Option 2: Rob current

        int newMax = Math.max(curMax, nums[index]);

        helper(nums, k, index + 2, robbedCount + 1, newMax, memo);

        memo.put(key, minCap);

    }

}

Time Complexity: O(n⋅k⋅m)

Number of Unique States:

The memoization key is index\_robbedCount\_curMax, where:

index ranges from 0 to n-1 (n = nums.length).

robbedCount ranges from 0 to k.

curMax can take up to m unique values (m = max(nums)).

Total unique states: O(n⋅k⋅m)

Work per State:

Each state performs O(1) operations (excluding recursive calls).

Recursive calls are memoized, so each state is computed only once.

Overall Time Complexity:O(n⋅k⋅m)

We compute each of the O(n⋅k⋅m) states exactly once.

Space Complexity

Memoization Storage:

The HashMap stores up to O(n⋅k⋅m) entries.

Recursion Stack:

Maximum recursion depth: O(n) (when skipping all houses).

Each stack frame uses O(1) space.

Overall Space Complexity:O(n⋅k⋅m)

Dominated by the memoization table.

**Style 3: Based on int return without 'prevRobbed' (TLE, 33/64)**

class Solution {

    public int minCapability(int[] nums, int k) {

        Map<String, Integer> memo = new HashMap<>();

        return helper(nums, k, 0, 0, Integer.MIN\_VALUE, memo);

    }

    private int helper(int[] nums, int k, int index, int robbedCount, int curMax, Map<String, Integer> memo) {

        // Base case 1: Robbed enough houses

        // Why put 'robbedCount >= k' ahead of 'index >= nums.length' ?

        // Because in native DFS version its more strict condition

        // in base case as nested inside 'index >= nums.length'

        if(robbedCount >= k) {

            return curMax;

        }

        // Base case 2: No more houses to consider

        if(index >= nums.length) {

            // Invalid, didn't rob enough houses till the

            // end hence return potential largest value

            return Integer.MAX\_VALUE;

        }

        String key = index + "\_" + robbedCount + "\_" + curMax;

        if(memo.containsKey(key)) {

            return memo.get(key);

        }

        // Option 1: Skip current

        int skip = helper(nums, k, index + 1, robbedCount, curMax, memo);

        // Option 2: Rob current

        int newMax = Math.max(curMax, nums[index]);

        int rob = helper(nums, k, index + 2, robbedCount + 1, newMax, memo);

        int result = Math.min(skip, rob);

        memo.put(key, result);

        return result;

    }

}

Time Complexity: O(n⋅k⋅m)

Number of Unique States:

The memoization key is index\_robbedCount\_curMax, where:

index ranges from 0 to n-1 (n = nums.length).

robbedCount ranges from 0 to k.

curMax can take up to m unique values (m = max(nums)).

Total unique states: O(n⋅k⋅m)

Work per State:

Each state performs O(1) operations (excluding recursive calls).

Recursive calls are memoized, so each state is computed only once.

Overall Time Complexity:O(n⋅k⋅m)

We compute each of the O(n⋅k⋅m) states exactly once.

Space Complexity

Memoization Storage:

The HashMap stores up to O(n⋅k⋅m) entries.

Recursion Stack:

Maximum recursion depth: O(n) (when skipping all houses).

Each stack frame uses O(1) space.

Overall Space Complexity:O(n⋅k⋅m)

Dominated by the memoization table.

**Style 4: Based on int return without 'prevRobbed' and 'curMax' (TLE, 43/64)**

class Solution {

    public int minCapability(int[] nums, int k) {

        Map<String, Integer> memo = new HashMap<>();

        return helper(nums, k, 0, 0, memo);

    }

    private int helper(int[] nums, int k, int index, int robbedCount, Map<String, Integer> memo) {

        // Base case 1: Robbed enough houses

        // Why put 'robbedCount >= k' ahead of 'index >= nums.length' ?

        // Because in native DFS version its more strict condition

        // in base case as nested inside 'index >= nums.length'

        if(robbedCount >= k) {

            return 0;

        }

        // Base case 2: No more houses to consider

        if(index >= nums.length) {

            // Invalid, didn't rob enough houses till the

            // end hence return potential largest value

            return Integer.MAX\_VALUE;

        }

        String key = index + "\_" + robbedCount;

        if(memo.containsKey(key)) {

            return memo.get(key);

        }

        // Option 1: Skip current

        int skip = helper(nums, k, index + 1, robbedCount, memo);

        // Option 2: Rob current

        // 'nums[index]' means current house's value becomes the new local max

        // 'helper(nums, k, index + 2, robbedCount + 1)' means recurse on remaining houses

        int rob = Math.max(nums[index], helper(nums, k, index + 2, robbedCount + 1, memo));

        int result = Math.min(skip, rob);

        memo.put(key, result);

        return result;

    }

}

Time Complexity: O(n\*k)

Number of Unique States:

The memoization key is index\_robbedCount, where:

index ranges from 0 to n-1 (n = nums.length).

robbedCount ranges from 0 to k.

Total unique states: O(n⋅k).

Work per State:

Each state performs O(1) operations (excluding recursive calls).

Recursive calls are memoized, so each state is computed only once.

Overall Time Complexity: O(n⋅k)

We compute each of the O(n⋅k) states exactly once.

Space Complexity: O(n\*k)

Memoization Storage:

The HashMap stores up to O(n⋅k) entries.

Recursion Stack:

Maximum recursion depth: O(n) (when skipping all houses).

Each stack frame uses O(1) space.

Overall Space Complexity:O(n⋅k)

Dominated by the memoization table.

**Solution 3: Binary Search + Greedy (60 min, refer to** [L875.Koko Eating Bananas (Ref.L410,L1011,L1283,L1482,L1802,L2064)](note://DBE895393C1D470AAEAB2675DBC07208)**)**

class Solution {

    public int minCapability(int[] nums, int k) {

        int lo = Integer.MAX\_VALUE;

        int hi = Integer.MIN\_VALUE;

        for(int num : nums) {

            lo = Math.min(lo, num);

            hi = Math.max(hi, num);

        }

        // Create a greedy algorithm: always steal from the current house

        // if its value is ≤ capability, then skip the adjacent house.

        // If we can steal at least k houses with this capability, try a

        // lower capability (search in the left half).

        // Otherwise, search in the right half.

        // Find lower boundary

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(canStealKHouses(nums, k, mid)) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

    private boolean canStealKHouses(int[] nums, int k, int capability) {

        int count = 0;

        int i = 0;

        while(i < nums.length) {

            if(nums[i] <= capability) {

                count++;

                i += 2;

            } else {

                i++;

            }

        }

        return count >= k;

    }

}

Time complexity: O(n\*log m) - where n is the length of nums and m is the maximum value in nums

Space complexity: O(1)

**Refer to**

<https://leetcode.com/problems/house-robber-iv/solutions/6537473/binary-search-python-c-java-js-c-go-swift-rust-kotlin/>

**Intuition**

The robber's capability is defined as the maximum amount of money he steals from one house. We want to minimize this capability while ensuring the robber steals from at least k houses. This is a problem where we can apply **binary search** on the answer.

The key insight is that if the robber can steal from at least k houses with a capability of x, then he can also do so with any capability > x. Conversely, if he cannot steal from k houses with capability x, then he cannot do so with any capability < x. This monotonicity makes the problem suitable for **binary search**.

**Approach**

Use binary search on the possible capability values, which range from the minimum to maximum value in the nums array.

For each capability value mid, check if it's possible to steal at least k houses:

Create a greedy algorithm: always steal from the current house if its value is ≤ capability, then skip the adjacent house.

If we can steal at least k houses with this capability, try a lower capability (search in the left half).

Otherwise, search in the right half.

Return the minimum capability that allows stealing at least k houses.

**Complexity**

Time complexity: O(n log m) - where n is the length of nums and m is the maximum value in nums

Space complexity: O(1)

**Code**

class Solution {

private boolean canStealKHouses(int[] nums, int k, int capability) {

int count = 0;

int i = 0;

while (i < nums.length) {

if (nums[i] <= capability) {

count++;

i += 2;

} else {

i++;

}

}

return count >= k;

}

public int minCapability(int[] nums, int k) {

int left = Arrays.stream(nums).min().getAsInt();

int right = Arrays.stream(nums).max().getAsInt();

while (left < right) {

int mid = left + (right - left) / 2;

if (canStealKHouses(nums, k, mid)) {

right = mid;

} else {

left = mid + 1;

}

}

return left;

}

}

**Refer to Deepseek**

For **LeetCode 2560. House Robber IV**, a **native DFS (brute-force) solution** is impractical due to the problem's constraints (up to (10^5) houses), which would lead to exponential time complexity ((O(2^n))). However, for educational purposes, here's how a native DFS approach would look, along with an explanation of why it's unsuitable for this problem.

**Native DFS Solution (Brute-Force)**

class Solution {

private int minCapability = Integer.MAX\_VALUE;

public int minCapability(int[] nums, int k) {

dfs(nums, k, 0, 0, Integer.MIN\_VALUE, false);

return minCapability;

}

private void dfs(int[] nums, int k, int index, int robbedCount, int currentMax, boolean prevRobbed) {

// Base case: All houses processed

if (index == nums.length) {

if (robbedCount >= k) {

minCapability = Math.min(minCapability, currentMax);

}

return;

}

// Option 1: Skip current house

dfs(nums, k, index + 1, robbedCount, currentMax, false);

// Option 2: Rob current house (if previous wasn't robbed)

if (!prevRobbed) {

int newMax = Math.max(currentMax, nums[index]);

dfs(nums, k, index + 1, robbedCount + 1, newMax, true);

}

}

}

**Why This Fails for LeetCode 2560**

**Exponential Time Complexity**:

The DFS explores all possible combinations of robbing/skipping houses, leading to (O(2^n)) time.

For (n = 10^5), this is computationally infeasible.

**Constraints**:

The problem requires (O(n \log m)) or better (where (m) is the max house value).

The binary search + greedy approach is optimal for large inputs.

**Correct Output**:

While this DFS would eventually find the correct answer for small inputs, it **times out** on LeetCode.

**Key Takeaways**

**Native DFS is Only for Learning**:

Demonstrates the brute-force logic but is **not usable for submission**.

Helps understand the problem's decision space.

**Optimal Approach**:

**Binary Search + Greedy** (as shown in the initial solution) is required for large inputs.

Achieves (O(n log m)) time by intelligently narrowing down the search space.

**When to Use DFS**:

Small inputs (e.g., (n <= 20)).

Problems where the constraints explicitly allow exponential solutions.

**Comparison with Optimal Solution**

|  |  |  |
| --- | --- | --- |
| Aspect | Native DFS (Brute-Force) | Binary Search + Greedy (Optimal) |
| **Time Complexity** | (O(2^n)) (Exponential) | (O(n log m)) (Linearithmic) |
| **Space Complexity** | (O(n)) (Recursion stack) | (O(1)) (Constant) |
| **Feasibility** | Fails for (n >= 20) | Handles (n <= 10^5) |

For LeetCode 2560, always prefer the **binary search + greedy approach**. The native DFS is included here purely to illustrate the brute-force logic.

**Modified Native DFS Solution (Without prevRobbed)**

To remove the prevRobbed variable from the native DFS solution while maintaining correctness, we need to **implicitly enforce the adjacency constraint** by skipping the next house whenever we rob the current one. Here's the modified DFS solution without prevRobbed:

class Solution {

private int minCapability = Integer.MAX\_VALUE;

public int minCapability(int[] nums, int k) {

dfs(nums, k, 0, 0, Integer.MIN\_VALUE);

return minCapability;

}

private void dfs(int[] nums, int k, int index, int robbedCount, int currentMax) {

// Base case: All houses processed

if (index >= nums.length) {

if (robbedCount >= k) {

minCapability = Math.min(minCapability, currentMax);

}

return;

}

// Option 1: Skip current house

dfs(nums, k, index + 1, robbedCount, currentMax);

// Option 2: Rob current house (skip next house)

int newMax = Math.max(currentMax, nums[index]);

dfs(nums, k, index + 2, robbedCount + 1, newMax);

}

}

**Key Changes**

**Removed prevRobbed**:

Instead of tracking whether the previous house was robbed, we **skip the next house** (index + 2) when robbing the current one.

This implicitly ensures no two adjacent houses are robbed.

**Simplified Base Case**:

Terminates changes to if index >= nums.length (not ==)

**Behavior**:

**Rob**: Adds nums[index] to currentMax and jumps to index + 2.

**Skip**: Moves to index + 1 without changing currentMax or robbedCount.

**Why This Works**

**Adjacency Constraint**: Skipping the next house (index + 2) when robbing ensures no two adjacent houses are selected.

**Correctness**: All valid robbery combinations are still explored, just with a different state representation.

**Example Walkthrough**

For nums = [1, 2, 3, 1], k = 2:

**Path 1**: Rob 0 (1), skip 1, rob 2 (3) → [1, 3] (max = 3)

**Path 2**: Rob 0 (1), skip 1, skip 2, rob 3 (1) → [1, 1] (max = 1)

**Path 3**: Skip 0, rob 1 (2), skip 2, rob 3 (1) → [2, 1] (max = 2)

**Path 4**: Skip 0, rob 1 (2), skip 3 → [2] (invalid, count < k)

**Result**: Minimum max is 1 (from Path 2).

**Limitations**

**Time Complexity**: Still (O(2^n)) (exponential), so **not suitable for large inputs**.

**Use Case**: Only for learning or very small inputs ((n <= 20)).

**Summary**

**Removing prevRobbed** is possible by **skipping the next house** when robbing.

**DFS remains exponential**—use only for small inputs or learning.

This modification preserves correctness while simplifying state tracking.

**I capture a difference when remove 'prevRobbed' from native DFS solution:**

In base condition, the original native DFS solution just have 'index == nums.length', but for the removed 'prevRobbed' native DFS solution if still keep 'index == nums.length', it will stack overflow for below input when hitting logic 'helper(nums, k, index + 1, robbedCount, curMax);'

nums = [2,3,5,9]

k = 2

looks like if change the base condition to 'index >= nums.length' will fix the issue, how and why ?

The difference in base conditions (index == nums.length vs. index >= nums.length) arises from how the recursion handles house indices when skipping vs. robbing. Let's break down why this matters and how it affects the stack overflow.

**1. Original DFS (With prevRobbed)**

private void helper(int[] nums, int k, int index, int robbedCount, int curMax, boolean prevRobbed) {

if (index == nums.length) { // Base case

if (robbedCount >= k) minCap = Math.min(curMax, minCap);

return;

}

// Option 1: Skip current (move to index + 1)

helper(nums, k, index + 1, robbedCount, curMax, false);

// Option 2: Rob current (move to index + 1)

if (!prevRobbed) {

helper(nums, k, index + 1, robbedCount + 1, Math.max(curMax, nums[index]), true);

}

}

**Behavior**:

When robbing or skipping, the recursion **always moves to index + 1**.

The base case index == nums.length works because index increments by exactly 1 in all cases.

**No Stack Overflow**:

The recursion depth is exactly nums.length (no overshooting).

**2. Modified DFS (Without prevRobbed)**

private void helper(int[] nums, int k, int index, int robbedCount, int curMax) {

if (index >= nums.length) { // Base case

if (robbedCount >= k) minCap = Math.min(curMax, minCap);

return;

}

// Option 1: Skip current (move to index + 1)

helper(nums, k, index + 1, robbedCount, curMax);

// Option 2: Rob current (move to index + 2)

helper(nums, k, index + 2, robbedCount + 1, Math.max(curMax, nums[index]));

}

**Behavior**:

When robbing, the recursion jumps to index + 2 (to skip the next house).

This can overshoot nums.length (e.g., index = nums.length - 1 → index + 2 = nums.length + 1).

**Stack Overflow Risk**:

If the base case only checks index == nums.length, the recursion might **miss the termination condition** when index > nums.length, leading to infinite recursion (and stack overflow).

**Fix**:

Change the base case to index >= nums.length to catch all overshooting indices.

**Why index >= nums.length Fixes the Issue**

**Covers All Cases**:

Catches both index == nums.length (normal termination) and index > nums.length (overshooting due to index + 2).

Example: For nums = [2, 3, 5, 9] and k = 2:

Robbing house 5 (index 2) → jumps to index = 4 (out of bounds).

index >= nums.length correctly terminates the recursion.

**Prevents Infinite Recursion**:

Ensures the recursion stops even if index exceeds the array bounds.

**Example Walkthrough**

For nums = [2, 3, 5, 9], k = 2:

**Path 1**: Rob 2 (index = 0), rob 5 (index = 2):

Rob 2 → jumps to index = 2 (skips 3).

Rob 5 → jumps to index = 4 (terminates at index >= nums.length).

Result: max(2, 5) = 5.

**Path 2**: Rob 2 (index = 0), skip 5, rob 9 (index = 3):

Rob 2 → jumps to index = 2.

Skip 5 → moves to index = 3.

Rob 9 → jumps to index = 5 (terminates at index >= nums.length).

Result: max(2, 9) = 9.

**Final Minimum**: min(5, 9, ...) = 5.

**Key Takeaways**

**Base Case Logic**:

index == nums.length works when indices increment by 1.

index >= nums.length is needed when indices can jump by 2 (to handle overshooting).

**Stack Overflow Cause**:

Missing the termination condition for index > nums.length leads to infinite recursion.

**General Rule**:

If recursion can skip indices (e.g., index + 2), use index >= nums.length.

If recursion always moves sequentially (index + 1), index == nums.length suffices.

**Instead of void return native DFS, can you write a version with int return DFS ?**

Here's a **native DFS solution with int return type** for LeetCode 2560 (House Robber IV), which avoids using a global variable and instead returns the minimum capability directly:

**Int-Return Native DFS Solution**

class Solution {

public int minCapability(int[] nums, int k) {

return dfs(nums, k, 0, 0, Integer.MIN\_VALUE);

}

private int dfs(int[] nums, int k, int index, int robbedCount, int currentMax) {

// Base case 1: Robbed enough houses

if (robbedCount >= k) {

return currentMax;

}

// Base case 2: No more houses to consider

if (index >= nums.length) {

return Integer.MAX\_VALUE; // Invalid, didn't rob enough houses

}

// Option 1: Skip current house

int skip = dfs(nums, k, index + 1, robbedCount, currentMax);

// Option 2: Rob current house (skip next house)

int newMax = Math.max(currentMax, nums[index]);

int rob = dfs(nums, k, index + 2, robbedCount + 1, newMax);

return Math.min(skip, rob);

}

}

**Key Improvements**

**No Global Variable**:

Uses return values to propagate results up the call stack.

Eliminates the need for minCapability as a class field.

**Explicit State Tracking**:

currentMax: Tracks the maximum value in the current robbery sequence.

robbedCount: Tracks the number of houses robbed so far.

**Base Cases**:

**Valid**: Returns currentMax if robbedCount >= k.

**Invalid**: Returns Integer.MAX\_VALUE if we run out of houses before robbing k houses.

**Recursive Logic**:

**Skip**: Moves to next house (index + 1) without changing state.

**Rob**: Updates currentMax and skips the next house (index + 2).

**Why This Works**

**Implicit Adjacency Constraint**: Jumping to index + 2 when robbing ensures no two adjacent houses are selected.

**Minimization**: The Math.min(skip, rob) ensures we find the smallest possible maximum value among all valid robbery sequences.

**Example Walkthrough**

For nums = [1, 2, 3, 1], k = 2:

**Path 1**: Rob 0 (1), rob 2 (3) → [1, 3] (max = 3)

**Path 2**: Rob 0 (1), skip 2, rob 3 (1) → [1, 1] (max = 1)

**Path 3**: Skip 0, rob 1 (2), rob 3 (1) → [2, 1] (max = 2)

**Result**: Minimum max is 1 (from Path 2).

**Limitations**

**Time Complexity**: (O(2^n)) (exponential), so it **fails for large inputs** (e.g., (n \geq 20)).

**Purpose**: Demonstrates the brute-force logic but is **not suitable for LeetCode submission**.

**Summary**

**Int-return DFS** is cleaner than void-return (no global state).

**Still exponential**, so only for learning/small inputs.

This version maintains correctness while better adhering to functional programming principles.

**Can we remove variable 'currentMax' ?**

To remove the currentMax variable from the native DFS solution while maintaining correctness, we can **restructure the logic to track the minimum capability implicitly** by comparing results during the recursion. Here's the modified solution:

**Native DFS Solution (Without currentMax)**

class Solution {

public int minCapability(int[] nums, int k) {

return dfs(nums, k, 0, 0);

}

private int dfs(int[] nums, int k, int index, int robbedCount) {

// Base case: Robbed enough houses

if (robbedCount >= k) {

return 0; // No need to track max here; handled by parent calls

}

// Base case: No more houses to rob

if (index >= nums.length) {

return Integer.MAX\_VALUE; // Invalid path

}

// Option 1: Skip current house

int skip = dfs(nums, k, index + 1, robbedCount);

// Option 2: Rob current house (skip next house)

int rob = Math.max(

nums[index], // Current house's value becomes the new local max

dfs(nums, k, index + 2, robbedCount + 1) // Recurse on remaining houses

);

return Math.min(skip, rob); // Track the minimal maximum value

}

}

**Key Changes**

**Removed currentMax**:

Instead of passing currentMax down the recursion, we **compute it on the way back up** using Math.max.

When robbing a house, its value becomes the new local maximum for that path.

**Base Case Adjustment**:

If robbedCount >= k, return 0 (since we only care about the maximum in the path, handled by Math.max).

If we run out of houses (index >= nums.length), return Integer.MAX\_VALUE to mark invalid paths.

**Implicit Max Tracking**:

The rob case now computes the maximum between:

The current house's value (nums[index]).

The result of the subproblem (recursive call).

**Why This Works**

**Correctness**: The Math.max ensures the local maximum is propagated up the call stack.

**Adjacency Constraint**: Skipping the next house (index + 2) when robbing maintains the no-adjacent-houses rule.

**Minimization**: The Math.min(skip, rob) ensures we find the smallest possible maximum value across all valid paths.

**Example Walkthrough**

For nums = [1, 2, 3, 1], k = 2:

**Path 1**: Rob 0 (1), rob 2 (3) → max(1, max(3, 0)) = 3

**Path 2**: Rob 0 (1), skip 2, rob 3 (1) → max(1, max(1, 0)) = 1

**Path 3**: Skip 0, rob 1 (2), rob 3 (1) → max(2, max(1, 0)) = 2

**Result**: min(3, 1, 2) = 1 (correct answer).

**Limitations**

**Time Complexity**: Still (O(2^n)) (exponential), so **not suitable for large inputs**.

**Redundant Calculations**: Repeats work for overlapping subproblems (e.g., dfs(nums, k, i, j) is called multiple times).

**Summary**

**Removed currentMax** by computing the maximum during recursion.

**Still exponential**, so only for learning/small inputs.

This version simplifies state tracking while preserving correctness. Use it to understand the problem's decision space, but switch to the optimal approach for actual submissions.

**Memoization Solution (Based on Native DFS)**

For **LeetCode 2560. House Robber IV**, a **memoization solution** based on the native DFS approach is still impractical due to the problem's constraints (up to (10^5) houses). However, for educational purposes, here's how you could attempt to memoize the DFS solution, along with an explanation of why it remains unsuitable for this problem.

import java.util.HashMap;

import java.util.Map;

class Solution {

private int minCapability = Integer.MAX\_VALUE;

public int minCapability(int[] nums, int k) {

Map<String, Integer> memo = new HashMap<>();

dfs(nums, k, 0, 0, Integer.MIN\_VALUE, false, memo);

return minCapability;

}

private void dfs(int[] nums, int k, int index, int robbedCount, int currentMax, boolean prevRobbed, Map<String, Integer> memo) {

// Base case: All houses processed

if (index == nums.length) {

if (robbedCount >= k) {

minCapability = Math.min(minCapability, currentMax);

}

return;

}

// Generate a unique key for memoization

String key = index + "," + robbedCount + "," + currentMax + "," + prevRobbed;

if (memo.containsKey(key)) {

return;

}

// Option 1: Skip current house

dfs(nums, k, index + 1, robbedCount, currentMax, false, memo);

// Option 2: Rob current house (if previous wasn't robbed)

if (!prevRobbed) {

int newMax = Math.max(currentMax, nums[index]);

dfs(nums, k, index + 1, robbedCount + 1, newMax, true, memo);

}

memo.put(key, minCapability);

}

}

**Why This Still Fails for LeetCode 2560**

**State Explosion**:

The memoization key combines index, robbedCount, currentMax, and prevRobbed.

For (n = 10^5) houses, the number of unique states becomes astronomically large (O(n \* k \* m \* 2)), where (m) is the max house value.

This exceeds memory limits and is computationally infeasible.

**Time Complexity**:

Even with memoization, the worst-case time complexity remains **exponential** due to the problem's constraints.

**Correct Output but Impractical**:

While this solution would eventually find the correct answer for small inputs, it **times out** or **runs out of memory** on LeetCode for large inputs.

**Key Takeaways**

**Memoization Doesn't Save the DFS**:

The DFS + memoization approach is still **too slow** for (n >= 20) due to state explosion.

The problem requires an **optimized approach** (binary search + greedy).

**Optimal Approach**:

**Binary Search + Greedy** (as shown in the initial solution) is the only feasible way to solve this problem for large inputs.

Achieves (O(n log m)) time by intelligently narrowing down the search space.

**When to Use Memoization**:

Small inputs (e.g., (n <= 20)).

Problems where the state space is manageable (e.g., House Robber I/II/III).

**Comparison with Optimal Solution**

|  |  |  |
| --- | --- | --- |
| Aspect | Memoization (Based on DFS) | Binary Search + Greedy (Optimal) |
| **Time Complexity** | Still exponential (state explosion) | (O(n log m)) (Linearithmic) |
| **Space Complexity** | (O(n \* k \* m)) (Unbounded) | (O(1)) (Constant) |
| **Feasibility** | Fails for (n >= 20) | Handles (n <= 10^5) |

For LeetCode 2560, always prefer the **binary search + greedy approach**. The memoized DFS is included here purely to illustrate why brute-force methods fail for large inputs.

Would you like a detailed walkthrough of the optimal solution?

**Binary Search + Greedy Solution**

**Binary Search Solution for House Robber IV**

**Intuition**

The problem requires finding the **minimum capability** (maximum value in any robbery sequence) to rob at least k houses without robbing adjacent houses.

Key observations:

**Capability Range**: The answer must lie between the smallest (min(nums)) and largest (max(nums)) house values.

**Feasibility Check**: For a given capability mid, we can check if it's possible to rob at least k houses where no house value exceeds mid and no two are adjacent.

**Approach**

**Binary Search Setup**:

**Left Bound (left)**: Minimum possible capability (1 or min(nums)).

**Right Bound (right)**: Maximum possible capability (max(nums)).

**Feasibility Function (canRob)**:

Traverse the array and greedily count houses that can be robbed (value ≤ mid), skipping the next house to avoid adjacency.

If the count ≥ k, mid is a feasible capability.

**Binary Search Execution**:

Adjust left or right based on whether mid is feasible until convergence.

**Solution Code**

class Solution {

public int minCapability(int[] nums, int k) {

int left = 1;

int right = Arrays.stream(nums).max().getAsInt();

int result = right;

while (left <= right) {

int mid = left + (right - left) / 2;

if (canRob(nums, mid, k)) {

result = mid;

right = mid - 1;

} else {

left = mid + 1;

}

}

return result;

}

private boolean canRob(int[] nums, int maxVal, int k) {

int count = 0;

for (int i = 0; i < nums.length; i++) {

if (nums[i] <= maxVal) {

count++;

i++; // Skip next house

if (count >= k) return true;

}

}

return count >= k;

}

}

**Explanation**

**Binary Search Initialization**:

left = 1 (smallest possible capability).

right = max(nums) (largest capability needed).

**Feasibility Check (canRob)**:

Iterate through nums, counting houses where nums[i] ≤ maxVal.

Skip the next house (i++) to avoid adjacency.

If count ≥ k, return true (feasible).

**Binary Search Execution**:

If mid is feasible (canRob returns true), search for a smaller capability (right = mid - 1).

Otherwise, search for a larger capability (left = mid + 1).

**Result**:

The smallest feasible mid is stored in result and returned.

**Complexity**

**Time**: (O(n \log m)), where (n = \text{nums.length}) and (m = \max(\text{nums})).

Binary search runs (O(\log m)) times.

Each canRob check takes (O(n)).

**Space**: (O(1)) (no extra space beyond variables).

**Why Binary Search Works**

**Monotonicity**: If a capability mid is feasible, any ≥ mid is also feasible. Binary search efficiently finds the minimal feasible value.

**Greedy Selection**: The canRob function ensures optimal house selection without adjacency.

**Example**

For nums = [2, 3, 5, 9], k = 2:

**Binary Search**:

left = 1, right = 9.

mid = 5 → canRob counts [2, 5] (valid). Update right = 4.

mid = 2 → canRob counts [2] (invalid). Update left = 3.

mid = 3 → canRob counts [2, 3] (valid). Update right = 2.

Exit with result = 3.

This approach efficiently narrows down the minimal capability. 🚀

**Refer to**

[L11.Container With Most Water](note://C0F878EFCE5B47B3A30E26B9AB7A114E)

[L198.House Robber (Ref.L213,L337,L2320,L2560)](note://WEB3a5c4401c63ae93266b0b603cb20076c)

[L213.House Robber II (Ref.L198,L337,L2320,L2560)](note://A5D6119BDEB54084A860F30A71A0A8C2)

[L337.House Robber III (Ref.L198,L213)](note://WEBdfa028bc04290ef6bce0fdf40c470cfe)

[L875.Koko Eating Bananas (Ref.L410,L1011,L1283,L1482,L1802,L2064)](note://DBE895393C1D470AAEAB2675DBC07208)