

# LeetCode



# Dynamic Programming



## SOLUTION IN C++



WWW.LEARNLONER.COM



# Dynamic Programming Questions

[Download PDF](#)

- [Dynamic Programming Questions](#)
  - [5. Longest Palindromic Substring](#)
  - [53. Maximum Subarray](#)
  - [62. Unique Paths](#)
  - [63. Unique Paths II](#)
  - [64. Minimum Path Sum](#)
  - [70. Climbing Stairs](#)
  - [72. Edit Distance](#)
  - [121. Best Time to Buy and Sell Stock](#)
  - [198. House Robber](#)
  - [256. Paint House](#)
  - [276. Paint Fence](#)
  - [303. Range Sum Query - Immutable](#)
  - [338. Counting Bits](#)
  - [392. Is Subsequence](#)
  - [647. Palindromic Substrings](#)
  - [650. 2 Keys Keyboard](#)
  - [877. Stone Game](#)
  - [931. Minimum Falling Path Sum](#)
  - [1035. Uncrossed Lines](#)
  - [1277. Count Square Submatrices with All Ones](#)
  - [1314. Matrix Block Sum](#)
  - [1326. Minimum Number of Taps to Open to Water a Garden](#)
  - [1335. Minimum Difficulty of a Job Schedule](#)
  - [1458. Max Dot Product of Two Subsequences](#)
  - [1473. Paint House III](#)
  - [1478. Allocate Mailboxes](#)

## 5. Longest Palindromic Substring

*Description*

Given a string *s*, find the longest palindromic substring in *s*. You may assume that the maximum length of *s* is 1000.

Example 1:

Input: "babad"

Output: "bab"

Note: "aba" is also a valid answer.

Example 2:

Input: "cbbd"

Output: "bb"

*Solution*

01/13/2020 (Dynamic Programming):

```
class Solution {
public:
    string longestPalindrome(string s) {
        int n = s.size(), start, max_len = 0;
        if (n == 0) return "";
        vector<vector<bool>> dp(n, vector<bool>(n, false));
        for (int i = 0; i < n; ++i) dp[i][i] = true;
        for (int i = 0; i < n - 1; ++i) dp[i][i + 1] = s[i] == s[i + 1];
        for (int i = n - 3; i >= 0; --i) {
            for (int j = i + 2; j < n; ++j) {
                dp[i][j] = dp[i + 1][j - 1] && s[i] == s[j];
            }
        }
        for (int i = 0; i < n; ++i) {
            for (int j = i; j < n; ++j) {
                if (dp[i][j] && j - i + 1 > max_len) {
                    max_len = j - i + 1;
                    start = i;
                }
            }
        }
        return s.substr(start, max_len);
    }
};
```

01/13/2020 (Expand Around Center):

```
class Solution {
public:
    string longestPalindrome(string s) {
```

```

int n = s.size(), start = 0, max_len = n > 0 ? 1 : 0;
for(int i = 0; i < n; ++i) {
    for (int l = i - 1, r = i; l >= 0 && r < n && s[l] == s[r]; --l, ++r) {
        if (r - l + 1 > max_len) {
            max_len = r - l + 1;
            start = l;
        }
    }
    for (int l = i - 1, r = i + 1; l >= 0 && r < n && s[l] == s[r]; --l, ++r)
    {
        if (r - l + 1 > max_len) {
            max_len = r - l + 1;
            start = l;
        }
    }
}
return max_len == 0 ? "" : s.substr(start, max_len);
};

```

## 53. Maximum Subarray

### Description

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Example:

Input: `[-2,1,-3,4,-1,2,1,-5,4]`,

Output: 6

Explanation: `[4,-1,2,1]` has the largest sum = 6.

Follow up:

If you have figured out the  $O(n)$  solution, try coding another solution using the divide and conquer approach, which is more subtle.

### Solution

01/13/2020 (Dynamic Programming):

```

class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int n = nums.size(), max_sum = nums[0];
        for (int i = 1; i < n; ++i) {
            if (nums[i - 1] > 0) nums[i] += nums[i - 1];
            max_sum = max(max_sum, nums[i]);
        }
        return max_sum;
    }
};

```

## 62. Unique Paths

### *Description*

A robot is located at the top-left corner of a  $m \times n$  grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

Above is a  $7 \times 3$  grid. How many possible unique paths are there?

Note:  $m$  and  $n$  will be at most 100.

Example 1:

Input:  $m = 3, n = 2$

Output: 3

Explanation:

From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:

1. Right -> Right -> Down
2. Right -> Down -> Right
3. Down -> Right -> Right

Example 2:

Input:  $m = 7, n = 3$

Output: 28

Solution

01/29/2020:

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<vector<long long>> dp(m + 1, vector<long long>(n + 1, 0));
        dp[0][1] = 1;
        for (int i = 1; i < m + 1; ++i) {
            for (int j = 1; j < n + 1; ++j) {
                dp[i][j] = dp[i][j - 1] + dp[i - 1][j];
            }
        }
        return dp[m][n];
    }
};
```

## 63. Unique Paths II

*Description*

A robot is located at the top-left corner of a  $m \times n$  grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

Note:  $m$  and  $n$  will be at most 100.

Example 1:

Input:

```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```

Output: 2

Explanation:

There is one obstacle in the middle of the 3x3 grid above.

There are two ways to reach the bottom-right corner:

1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right

*Solution*

01/27/2020:

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size(), n = obstacleGrid[0].size();
        if (m == 0 || n == 0) return 1;
        vector<vector<long>> dp(m, vector<long>(n, 0));
        dp[0][0] = obstacleGrid[0][0] == 1 ? 0 : 1;
        // dp[i][j] the total number of unique moves
        // dp[i][j] = dp[i - 1][j] * (grid[i - 1][j] != 1) + dp[i][j - 1] * (grid[i]
[j - 1] != 1)
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (obstacleGrid[i][j] == 1) {
                    dp[i][j] = 0;
                    continue;
                }
                if (i - 1 >= 0) dp[i][j] += dp[i - 1][j] * (obstacleGrid[i - 1][j] != 1
? 1 : 0);
                if (j - 1 >= 0) dp[i][j] += dp[i][j - 1] * (obstacleGrid[i][j - 1] != 1
? 1 : 0);
            }
        }
        return (m - 1 >= 0 && n - 1 >= 0) ? dp[m - 1][n - 1] : 0;
    }
};
```

## 64. Minimum Path Sum

*Description*

Given a  $m \times n$  grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Example:

Input:

```
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
```

Output: 7

Explanation: Because the path 1→3→1→1→1 minimizes the sum.

*Solution*

01/27/2020:

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size(), n = grid[0].size();
        if (m == 0 || n == 0) return 0;
        const int INF = 1e9 + 5;
        vector<vector<int>> dp(m, vector<int>(n, INF));
        dp[0][0] = grid[0][0];
        // dp[i][j]: the minimum sum from grid[0][0] to grid[i][j]
        // dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j]
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (i - 1 >= 0) dp[i][j] = min(grid[i][j] + dp[i - 1][j], dp[i][j]);
                if (j - 1 >= 0) dp[i][j] = min(grid[i][j] + dp[i][j - 1], dp[i][j]);
            }
        }
        return dp[m - 1][n - 1];
    }
};
```

## 70. Climbing Stairs

*Description*

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?



Note: Given n will be a positive integer.

Example 1:

Input: 2

Output: 2

Explanation: There are two ways to climb to the top.

1. 1 step + 1 step
2. 2 steps

Example 2:

Input: 3

Output: 3

Explanation: There are three ways to climb to the top.

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

*Solution*

01/14/2020 (Dynamic Programming):

```
class Solution {
public:
    int climbStairs(int n) {
        int s1 = 1, s2 = 2;
        for (int i = 2; i < n; ++i) {
            s1 = s1 + s2;
            swap(s1, s2);
        }
        return n >= 2 ? s2 : s1;
    }
};
```

## 72. Edit Distance

*Description*

Given two words word1 and word2, find the minimum number of operations required to convert word1 to word2.

You have the following 3 operations permitted on a word:

Insert a character

Delete a character

Replace a character

Example 1:

Input: word1 = "horse", word2 = "ros"

Output: 3

Explanation:

horse -> rorse (replace 'h' with 'r')

rorse -> rose (remove 'r')

rose -> ros (remove 'e')

Example 2:

Input: word1 = "intention", word2 = "execution"

Output: 5

Explanation:

intention -> inention (remove 't')

inention -> enention (replace 'i' with 'e')

enention -> exention (replace 'n' with 'x')

exention -> exection (replace 'n' with 'c')

exection -> execution (insert 'u')

*Solution*

01/27/2020:

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        int m = word1.size(), n = word2.size();
        if (m == 0) return n;
        if (n == 0) return m;
        const int INF = 1e9 + 5;
        vector<vector<int>> dp(m + 1, vector(n + 1, INF));
        dp[0][0] = 0;
        // dp[i][j]: the edit distance between word1[0..i] and word2[0..j]
        // dp[i][j] = dp[i - 1][j - 1]      if word1[i] == word2[j]: no operations
        // needed
        // dp[i][j] = min(
        //     dp[i - 1][j - 1] + 1,      if word1[i] != word2[j]
        //     dp[i - 1][j] + 1,          replace word1[i] by word2[j]
        //     dp[i][j - 1] + 1,          delete character word1[i]
        //     dp[i][j - 1] + 1,          delete character word2[j]
        // )
        for (int i = 1; i <= m; ++i) dp[i][0] = dp[i - 1][0] + 1;
        for (int j = 1; j <= n; ++j) dp[0][j] = dp[0][j - 1] + 1;
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (word1[i - 1] == word2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
```

```

        dp[i][j] = min(dp[i - 1][j], min(dp[i][j - 1], dp[i - 1][j - 1])) + 1;
    }
}
}
return dp[m][n];
}
};

```

## 121. Best Time to Buy and Sell Stock

### Description

Say you have an array for which the  $i$ th element is the price of a given stock on day  $i$ .

If you were only permitted to complete at most one transaction (i.e., buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Note that you cannot sell a stock before you buy one.

Example 1:

Input: [7,1,5,3,6,4]

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6 - 1 = 5.

Not 7 - 1 = 6, as selling price needs to be larger than buying price.

Example 2:

Input: [7,6,4,3,1]

Output: 0

Explanation: In this case, no transaction is done, i.e. max profit = 0.

### Solution

01/13/2020 (Dynamic Programming):

```

class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        if (n <= 1) return 0;
        vector<int> diff(n - 1, 0);
        for (int i = 0; i < n - 1; ++i) {

```

```

    diff[i] = prices[i + 1] - prices[i];
}
int max_sum = max(0, diff[0]);
for (int i = 1; i < n - 1; ++i) {
    if (diff[i - 1] > 0) diff[i] += diff[i - 1];
    max_sum = max(diff[i], max_sum);
}
return max_sum;
}
};

```

## 198. House Robber

### Description

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

Example 1:

Input: [1,2,3,1]

Output: 4

Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).

Total amount you can rob = 1 + 3 = 4.

Example 2:

Input: [2,7,9,3,1]

Output: 12

Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).

Total amount you can rob = 2 + 9 + 1 = 12.

### Solution

01/14/2020 (Dynamic Programming):

```

class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() >= 2) nums[1] = max(nums[0], nums[1]);
        for (int i = 2; i < nums.size(); ++i)
            nums[i] = max(nums[i - 1], nums[i - 2] + nums[i]);
        return nums.size() > 0 ? nums.back() : 0;
    }
};

```

## 256. Paint House

### *Description*

There are a row of  $n$  houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a  $n \times 3$  cost matrix. For example,  $costs[0][0]$  is the cost of painting house 0 with color red;  $costs[1][2]$  is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

Note:

All costs are positive integers.

Example:

Input:  $[[17,2,17],[16,16,5],[14,3,19]]$

Output: 10

Explanation: Paint house 0 into blue, paint house 1 into green, paint house 2 into blue.

Minimum cost:  $2 + 5 + 3 = 10$ .

### *Solution*

05/26/2020:

```

class Solution {
public:
    int minCost(vector<vector<int>>& costs) {
        if (costs.empty()) return 0;
        int n = costs.size();
        for (int i = 1; i < n; ++i) {
            costs[i][0] += min(costs[i - 1][1], costs[i - 1][2]);
            costs[i][1] += min(costs[i - 1][0], costs[i - 1][2]);
            costs[i][2] += min(costs[i - 1][0], costs[i - 1][1]);
        }
        return *min_element(costs.back().begin(), costs.back().end());
    }
};

```

```

class Solution {
public:
    int minCost(vector<vector<int>>& costs) {
        for (int i = 1; i < costs.size(); ++i) {
            for (int j = 0; j < costs[0].size(); ++j) {
                int cur_min = INT_MAX;
                for (int k = 0; k < costs[0].size(); ++k)
                    if (k != j)
                        cur_min = min(cur_min, costs[i - 1][k]);
                costs[i][j] += cur_min;
            }
        }
        return costs.size() > 0 ? *min_element(costs.back().begin(),
costs.back().end()) : 0;
    }
};

```

## 276. Paint Fence

### *Description*

There is a fence with  $n$  posts, each post can be painted with one of the  $k$  colors.

You have to paint all the posts such that no more than two adjacent fence posts have the same color.

Return the total number of ways you can paint the fence.

Note:

n and k are non-negative integers.

Example:

Input: n = 3, k = 2

Output: 6

Explanation: Take c1 as color 1, c2 as color 2. All possible ways are:

	post1	post2	post3
1	c1	c1	c2
2	c1	c2	c1
3	c1	c2	c2
4	c2	c1	c1
5	c2	c1	c2
6	c2	c2	c1

*Solution*

01/14/2020 (Dynamic Programming):

```
class Solution {
public:
    int numWays(int n, int k) {
        if (n == 0 || k == 0) return 0;
        vector<vector<int>> dp(n, vector<int>(2, 0));
        dp[0][0] = k;
        for (int i = 1; i < n; ++i) {
            dp[i][0] = (dp[i - 1][0] + dp[i - 1][1]) * (k - 1);
            dp[i][1] = dp[i - 1][0];
        }
        return dp.back()[0] + dp.back()[1];
    }
};
```

01/14/2020: (Dynamic Programming, Improve Space Complexity):

```
class Solution {
public:
    int numWays(int n, int k) {
        if (n == 0 || k == 0) return 0;
        vector<int> dp(2, 0);
        dp[0] = k;
        for (int i = 1; i < n; ++i) {
            int tmp = dp[0];
            dp[0] = (dp[0] + dp[1]) * (k - 1);
            dp[1] = tmp;
        }
    }
};
```

```

    }
    return dp[0] + dp[1];
}
};

```

## 303. Range Sum Query - Immutable

### Description

Given an integer array `nums`, find the sum of the elements between indices `i` and `j` ( $i \leq j$ ), inclusive.

Example:

Given `nums = [-2, 0, 3, -5, 2, -1]`

`sumRange(0, 2) -> 1`

`sumRange(2, 5) -> -1`

`sumRange(0, 5) -> -3`

Note:

You may assume that the array does not change.

There are many calls to `sumRange` function.

### Solution

01/14/2020 (Dynamic Programming):

```

class NumArray {
public:
    vector<int> nums;
    NumArray(vector<int>& nums) {
        for (int i = 1; i < nums.size(); ++i) nums[i] += nums[i - 1];
        this->nums = nums;
    }

    int sumRange(int i, int j) {
        return i > 0 ? nums[j] - nums[i - 1] : nums[j];
    }
};

/**
 * Your NumArray object will be instantiated and called as such:
 * NumArray* obj = new NumArray(nums);
 * int param_1 = obj->sumRange(i,j);

```



\*/

## 338. Counting Bits

### Description

Given a non negative integer number num. For every numbers i in the range  $0 \leq i \leq \text{num}$  calculate the number of 1's in their binary representation and return them as an array.

Example 1:

Input: 2

Output: [0,1,1]

Example 2:

Input: 5

Output: [0,1,1,2,1,2]

Follow up:

It is very easy to come up with a solution with run time  $O(n * \text{sizeof}(\text{integer}))$ .

But can you do it in linear time  $O(n)$  /possibly in a single pass?

Space complexity should be  $O(n)$ .

Can you do it like a boss? Do it without using any builtin function like `__builtin_popcount` in c++ or in any other language.

### Solution

01/15/2020 (Dynamic Programming):

```
class Solution {
public:
    vector<int> countBits(int num) {
        vector<int> dp(num + 1, 0);
        for (int i = 1, m = 1; i <= num; ++i) {
            if (i == m << 1) m <<= 1;
            dp[i] = dp[i % m] + 1;
        }
        return dp;
    }
};
```

## 392. Is Subsequence

### Description

Given a string *s* and a string *t*, check if *s* is subsequence of *t*.

You may assume that there is only lower case English letters in both *s* and *t*. *t* is potentially a very long (length  $\sim 500,000$ ) string, and *s* is a short string ( $\leq 100$ ).

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ace" is a subsequence of "abcde" while "aec" is not).

Example 1:

*s* = "abc", *t* = "ahbgdc"

Return true.

Example 2:

*s* = "axc", *t* = "ahbgdc"

Return false.

Follow up:

If there are lots of incoming *S*, say *S*<sub>1</sub>, *S*<sub>2</sub>, ... , *S*<sub>*k*</sub> where *k*  $\geq 1B$ , and you want to check one by one to see if *T* has its subsequence. In this scenario, how would you change your code?

Credits:

Special thanks to @pbrother for adding this problem and creating all test cases.

### Solution

01/14/2020 (Dynamic Programming, Memory Limit Exceeded):

```
class Solution {
public:
    bool isSubsequence(string s, string t) {
        if (s.size() >= t.size()) return s == t;
        return s.back() == t.back() ? isSubsequence(s.substr(0, s.size() - 1),
t.substr(0, t.size() - 1)) :
        isSubsequence(s, t.substr(0, t.size() - 1));
    }
};
```

```
class Solution {
public:
    bool isSubsequence(string s, string t) {
        int ps = 0, pt = 0;
        for (; ps < s.size() && pt < t.size(); ++pt)
            if (s[ps] == t[pt]) ++ps;
        return ps == s.size();
    }
};
```

## 647. Palindromic Substrings

### *Description*

Given a string, your task is to count how many palindromic substrings in this string.

The substrings with different start indexes or end indexes are counted as different substrings even they consist of same characters.

Example 1:

Input: "abc"

Output: 3

Explanation: Three palindromic strings: "a", "b", "c".

Example 2:

Input: "aaa"

Output: 6

Explanation: Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".

Note:

The input string length won't exceed 1000.

### *Solution*

```

class Solution {
public:
    int countSubstrings(string s) {
        int ret = 0;
        for (int i = 0; i < s.size(); ++i) {
            for (int l = i, r = i; l >= 0 && r < s.size() && s[l] == s[r]; --l, ++r)
                ++ret;
            for (int l = i, r = i + 1; l >= 0 && r < s.size() && s[l] == s[r]; --l,
                ++r) ++ret;
        }
        return ret;
    }
};

```

## 650. 2 Keys Keyboard

### Description

Initially on a notepad only one character 'A' is present. You can perform two operations on this notepad for each step:

Copy All: You can copy all the characters present on the notepad (partial copy is not allowed).

Paste: You can paste the characters which are copied last time.

Given a number n. You have to get exactly n 'A' on the notepad by performing the minimum number of steps permitted. Output the minimum number of steps to get n 'A'.

Example 1:

Input: 3

Output: 3

Explanation:

Initially, we have one character 'A'.

In step 1, we use Copy All operation.

In step 2, we use Paste operation to get 'AA'.

In step 3, we use Paste operation to get 'AAA'.

Note:

The n will be in the range [1, 1000].

Solution

06/09/2020:

```
class Solution {
public:
    int minSteps(int n) {
        // dp[i]: the minimum steps to obtain i A's
        // dp[i] = min_j(dp[j] + 1 + (i - j) / j) = min_j(dp[j] + 1 + i / j - 1) =
        min_j(dp[j] + i / j)
        vector<int> dp(n + 1, INT_MAX);
        dp[1] = 0;
        for (int i = 2; i <= n; ++i)
            for (int j = 1; j < i; ++j)
                if (i % j == 0)
                    dp[i] = min(dp[i], dp[j] + i / j);
        return dp[n];
    }
};
```

## 877. Stone Game

### *Description*

Alex and Lee play a game with piles of stones. There are an even number of piles arranged in a row, and each pile has a positive integer number of stones  $piles[i]$ .

The objective of the game is to end with the most stones. The total number of stones is odd, so there are no ties.

Alex and Lee take turns, with Alex starting first. Each turn, a player takes the entire pile of stones from either the beginning or the end of the row. This continues until there are no more piles left, at which point the person with the most stones wins.

Assuming Alex and Lee play optimally, return True if and only if Alex wins the game.

Example 1:

Input: [5,3,4,5]

Output: true

Explanation:

Alex starts first, and can only take the first 5 or the last 5.  
Say he takes the first 5, so that the row becomes [3, 4, 5].  
If Lee takes 3, then the board is [4, 5], and Alex takes 5 to win with 10 points.  
If Lee takes the last 5, then the board is [3, 4], and Alex takes 4 to win with 9 points.  
This demonstrated that taking the first 5 was a winning move for Alex, so we return true.

Note:

```
2 <= piles.length <= 500
piles.length is even.
1 <= piles[i] <= 500
sum(piles) is odd.
```

*Solution*

01/15/2020 (Mathematics):

```
class Solution {
public:
    bool stoneGame(vector<int>& piles) {
        return true;
    }
};
```

01/15/2020 (Dynamic Programming):

```
class Solution {
public:
    bool stoneGame(vector<int>& piles) {
        // dp[i][j]: the largest number of stones Alex can pick
        int n = piles.size();
        vector<vector<int>> dp(n, vector<int>(n, 0));
        for (int i = 0; i < n; ++i) dp[i][i] = piles[i];
        for (int i = 0; i < n - 1; ++i) {
            for (int j = i + 1; j < n; ++j) {
                dp[i][j] = max(piles[i] + dp[i + 1][j], piles[j] + dp[i][j - 1]);
            }
        }
        return dp[0][n - 1] > max(dp[1][n - 1], dp[0][n - 2]);
    }
};
```

# 931. Minimum Falling Path Sum

## Description

Given a square array of integers A, we want the minimum sum of a falling path through A.

A falling path starts at any element in the first row, and chooses one element from each row. The next row's choice must be in a column that is different from the previous row's column by at most one.

Example 1:

Input: [[1,2,3],[4,5,6],[7,8,9]]

Output: 12

Explanation:

The possible falling paths are:

[1,4,7], [1,4,8], [1,5,7], [1,5,8], [1,5,9]

[2,4,7], [2,4,8], [2,5,7], [2,5,8], [2,5,9], [2,6,8], [2,6,9]

[3,5,7], [3,5,8], [3,5,9], [3,6,8], [3,6,9]

The falling path with the smallest sum is [1,4,7], so the answer is 12.

Note:

$1 \leq A.length == A[0].length \leq 100$

$-100 \leq A[i][j] \leq 100$

## Solution

01/15/2020 (Dynamic Programming):

```
class Solution {
public:
    int minFallingPathSum(vector<vector<int>>& A) {
        int n = A[0].size(), K = 1;
        if (n == 0) return 0;
        // dp[i][j]: the current smallest sum from row 0 to row i at column j.
        vector<vector<int>> dp(n, vector<int>(n, 0));
        for (int i = 0; i < n; ++i) dp[0][i] = A[0][i];
        for (int i = 1; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                int cur_min = INT_MAX;
                for (int k = -K; k <= K; ++k) {
```

```

    int l = j + k;
    if(l >= 0 && l < n) {
        cur_min = min(cur_min, dp[i - 1][l] + A[i][j]);
    }
}
dp[i][j] = cur_min;
}
}
return *min_element(dp.back().begin(), dp.back().end());
}
};

```

## 1035. Uncrossed Lines

### Description

We write the integers of A and B (in the order they are given) on two separate horizontal lines.

Now, we may draw connecting lines: a straight line connecting two numbers A[i] and B[j] such that:

$A[i] == B[j]$ ;

The line we draw does not intersect any other connecting (non-horizontal) line. Note that a connecting lines cannot intersect even at the endpoints: each number can only belong to one connecting line.

Return the maximum number of connecting lines we can draw in this way.

Example 1:

Input: A = [1,4,2], B = [1,2,4]

Output: 2

Explanation: We can draw 2 uncrossed lines as in the diagram.

We cannot draw 3 uncrossed lines, because the line from A[1]=4 to B[2]=4 will intersect the line from A[2]=2 to B[1]=2.

Example 2:

Input: A = [2,5,1,2,5], B = [10,5,2,1,5,2]

Output: 3

Example 3:



Input: A = [1,3,7,1,7,5], B = [1,9,2,5,1]

Output: 2

Note:

1 <= A.length <= 500

1 <= B.length <= 500

1 <= A[i], B[i] <= 2000

*Solution*

05/25/2020:

```
class Solution {
public:
    int maxUncrossedLines(vector<int>& A, vector<int>& B) {
        if (A.empty() && B.empty()) return 0;
        // dp[i][j]: the maximum number of lines can be drawn of A[0..i] and B[0..j]
        // same as longest common subsequence
        int m = A.size(), n = B.size();
        vector<vector<int>> dp(m, vector<int>(n, 0));
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                dp[i][j] = A[i] == B[j];
                if (i > 0 && j > 0) dp[i][j] = dp[i][j] + dp[i - 1][j - 1];
                if (i > 0) dp[i][j] = max(dp[i][j], dp[i - 1][j]);
                if (j > 0) dp[i][j] = max(dp[i][j], dp[i][j - 1]);
            }
        }
        return dp.back().back();
    }
};
```

## 1277. Count Square Submatrices with All Ones

*Description*

Given a  $m \times n$  matrix of ones and zeros, return how many square submatrices have all ones.

Example 1:

Input: matrix =

```
[
  [0,1,1,1],
  [1,1,1,1],
  [0,1,1,1]
]
```

Output: 15

Explanation:

There are 10 squares of side 1.

There are 4 squares of side 2.

There is 1 square of side 3.

Total number of squares = 10 + 4 + 1 = 15.

Example 2:

Input: matrix =

```
[
  [1,0,1],
  [1,1,0],
  [1,1,0]
]
```

Output: 7

Explanation:

There are 6 squares of side 1.

There is 1 square of side 2.

Total number of squares = 6 + 1 = 7.

Constraints:

1 <= arr.length <= 300

1 <= arr[0].length <= 300

0 <= arr[i][j] <= 1

*Solution*

01/14/2020 (Dynamic Programming):

```
class Solution {
public:
    int countSquares(vector<vector<int>>& matrix) {
        int m = matrix.size(), n = matrix[0].size(), ret = 0;;
        vector<vector<int>> dp(m, vector<int>(n, 0));
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (i == 0 || j == 0)
                    dp[i][j] = matrix[i][j];
                else
```

```

        dp[i][j] = matrix[i][j] == 0 ? 0 : min(min(dp[i - 1][j - 1], dp[i - 1]
[j]), dp[i][j - 1]) + 1;
        ret += dp[i][j];
    }
}
return ret;
}
};

```

## 1314. Matrix Block Sum

### Description

Given a  $m \times n$  matrix `mat` and an integer `K`, return a matrix `answer` where each `answer[i][j]` is the sum of all elements `mat[r][c]` for  $i - K \leq r \leq i + K$ ,  $j - K \leq c \leq j + K$ , and  $(r, c)$  is a valid position in the matrix.

Example 1:

Input: `mat = [[1,2,3],[4,5,6],[7,8,9]]`, `K = 1`

Output: `[[12,21,16],[27,45,33],[24,39,28]]`

Example 2:

Input: `mat = [[1,2,3],[4,5,6],[7,8,9]]`, `K = 2`

Output: `[[45,45,45],[45,45,45],[45,45,45]]`

Constraints:

`m == mat.length`

`n == mat[i].length`

`1 <= m, n, K <= 100`

`1 <= mat[i][j] <= 100`

### Solution

01/14/2020 (Definition):

```

class Solution {
public:
    vector<vector<int>> matrixBlockSum(vector<vector<int>>& mat, int K) {
        int m = mat.size(), n = mat[0].size();
        vector<vector<int>> ret(m, vector<int>(n, 0));
        for (int i = 0; i < m; ++i) {

```

```

    for (int j = 0; j < n; ++j) {
        for (int k = -K; k <= K; ++k) {
            for (int l = -K; l <= K; ++l) {
                if (i + k >= 0 && i + k < m && j + l >= 0 && j + l < n) {
                    ret[i][j] += mat[i + k][j + l];
                }
            }
        }
    }
    return ret;
}
};

```

01/14/2020 (Dynamic Programming):

```

class Solution {
public:
    vector<vector<int>> matrixBlockSum(vector<vector<int>>& mat, int K) {
        int m = mat.size(), n = mat[0].size();
        vector<vector<int>> dp(m, vector<int>(n, 0));
        vector<vector<int>> ret(m, vector<int>(n, 0));
        dp[0][0] = mat[0][0];
        for (int i = 1; i < m; ++i) dp[i][0] += dp[i - 1][0] + mat[i][0];
        for (int j = 1; j < n; ++j) dp[0][j] += dp[0][j - 1] + mat[0][j];
        for (int i = 1; i < m; ++i)
            for (int j = 1; j < n; ++j)
                dp[i][j] = mat[i][j] + dp[i][j - 1] + dp[i - 1][j] - dp[i - 1][j - 1];
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                int r1 = max(i - K, 0);
                int c1 = max(j - K, 0);
                int r2 = min(i + K, m - 1);
                int c2 = min(j + K, n - 1);
                ret[i][j] = dp[r2][c2];
                if (r1 > 0) ret[i][j] -= dp[r1 - 1][c2];
                if (c1 > 0) ret[i][j] -= dp[r2][c1 - 1];
                if (r1 > 0 && c1 > 0) ret[i][j] += dp[r1 - 1][c1 - 1];
            }
        }
        return ret;
    }
};

```

1326. Minimum Number of Taps to Open to Water a Garden

## Description

There is a one-dimensional garden on the x-axis. The garden starts at the point 0 and ends at the point n. (i.e The length of the garden is n).

There are n + 1 taps located at points [0, 1, ..., n] in the garden.

Given an integer n and an integer array ranges of length n + 1 where ranges[i] (0-indexed) means the i-th tap can water the area [i - ranges[i], i + ranges[i]] if it was open.

Return the minimum number of taps that should be open to water the whole garden, If the garden cannot be watered return -1.

Example 1:

Input: n = 5, ranges = [3,4,1,1,0,0]

Output: 1

Explanation: The tap at point 0 can cover the interval [-3,3]

The tap at point 1 can cover the interval [-3,5]

The tap at point 2 can cover the interval [1,3]

The tap at point 3 can cover the interval [2,4]

The tap at point 4 can cover the interval [4,4]

The tap at point 5 can cover the interval [5,5]

Opening Only the second tap will water the whole garden [0,5]

Example 2:

Input: n = 3, ranges = [0,0,0,0]

Output: -1

Explanation: Even if you activate all the four taps you cannot water the whole garden.

Example 3:

Input: n = 7, ranges = [1,2,1,0,2,1,0,1]

Output: 3

Example 4:

Input: n = 8, ranges = [4,0,0,0,0,0,0,0,4]

Output: 2

Example 5:

Input: n = 8, ranges = [4,0,0,0,4,0,0,0,4]

Output: 1

Constraints:

```
1 <= n <= 10^4
ranges.length == n + 1
0 <= ranges[i] <= 100
```

*Solution*

01/19/2020 (Dynamic Programming):

```
const int INF = 1e9 + 5;
class Solution {
public:
    int minTaps(int n, vector<int>& ranges) {
        vector<pair<int, int>> intervals;
        for (int i = 0; i <= n; ++i) intervals.push_back({max(i - ranges[i], 0),
min(i + ranges[i], n)});
        // dp[i]: the minimum number of taps cover from 0 to point i
        int best = INF;
        vector<int> dp(n + 1, INF);
        for (int i = 0; i <= n; ++i) {
            if (intervals[i].first <= 0) dp[i] = 1;
            for (int j = i + 1; j <= n; ++j) {
                if (intervals[j].first <= intervals[i].second) {
                    dp[j] = min(dp[j], dp[i] + 1);
                }
            }
            if (intervals[i].second >= n)
                best = min(best, dp[i]);
        }
        return best < INF ? best : -1;
    }
};
```

01/19/2020 (Dynamic Programming):

```
class Solution {
public:
    int minTaps(int n, vector<int>& ranges) {
        const int INF = 1e9 + 5;
        vector<int> dp(n + 1, INF);
        for (int i = 0; i < n + 1; ++i) {
            int left = max(i - ranges[i], 0);
            int best = i - ranges[i] <= 0 ? 1 : dp[left] + 1;
            for (int j = i; j <= min(i + ranges[i], n); ++j) {
                dp[j] = min(dp[j], best);
            }
        }
    }
};
```

```
    }  
    return dp.back() == INF ? -1 : dp.back();  
  }  
};
```

## 1335. Minimum Difficulty of a Job Schedule

### *Description*

You want to schedule a list of jobs in  $d$  days. Jobs are dependent (i.e. To work on the  $i$ -th job, you have to finish all the jobs  $j$  where  $0 \leq j < i$ ).

You have to finish at least one task every day. The difficulty of a job schedule is the sum of difficulties of each day of the  $d$  days. The difficulty of a day is the maximum difficulty of a job done in that day.

Given an array of integers `jobDifficulty` and an integer  $d$ . The difficulty of the  $i$ -th job is `jobDifficulty[i]`.

Return the minimum difficulty of a job schedule. If you cannot find a schedule for the jobs return  $-1$ .

Example 1:

Input: `jobDifficulty = [6,5,4,3,2,1]`,  $d = 2$

Output: 7

Explanation: First day you can finish the first 5 jobs, total difficulty = 6.

Second day you can finish the last job, total difficulty = 1.

The difficulty of the schedule =  $6 + 1 = 7$

Example 2:

Input: `jobDifficulty = [9,9,9]`,  $d = 4$

Output:  $-1$

Explanation: If you finish a job per day you will still have a free day. you cannot find a schedule for the given jobs.

Example 3:

Input: `jobDifficulty = [1,1,1]`,  $d = 3$

Output: 3

Explanation: The schedule is one job per day. total difficulty will be 3.

Example 4:

Input: `jobDifficulty = [7,1,7,1,7,1]`,  $d = 3$

Output: 15

Example 5:

Input: jobDifficulty = [11,111,22,222,33,333,44,444], d = 6

Output: 843

Constraints:

1 <= jobDifficulty.length <= 300

0 <= jobDifficulty[i] <= 1000

1 <= d <= 10

*Solution*

01/26/2020:

```
class Solution {
public:
    int minDifficulty(vector<int>& jobDifficulty, int d) {
        int n = jobDifficulty.size();
        if (n < d) return -1;
        vector<vector<int>> diff(n, vector<int>(n, 0));
        // diff[i][j]: max diff from jobDifficulty[i] to jobDifficulty[j]
        for (int i = 0; i < n; ++i) {
            diff[i][i] = jobDifficulty[i];
            for (int j = i + 1; j < n; ++j) {
                diff[i][j] = max(diff[i][j - 1], jobDifficulty[j]);
            }
        }
        const int INF = 1e9 + 5;
        // dp[i][j]: min diff of i jobs in j days
        // dp[i][j] = min(dp[j - 1][j - 1] + diff[j][i], dp[j][j - 1] + diff[j + 1]
        [i], ..., dp[i - 1][j - 1] + diff[i][i])
        vector<vector<int>> dp(n, vector<int>(d, INF));
        for (int j = 0; j < d; ++j) {
            for (int i = 0; i < n; ++i) {
                if (j == 0) {
                    dp[i][j] = diff[j][i];
                } else {
                    for (int k = j - 1; k < i; ++k) {
                        dp[i][j] = min(dp[i][j], dp[k][j - 1] + diff[k + 1][i]);
                    }
                }
            }
        }
        return dp[n - 1][d - 1];
    }
}
```



```
};
```

## 1458. Max Dot Product of Two Subsequences

### Description

Given two arrays `nums1` and `nums2`.

Return the maximum dot product between non-empty subsequences of `nums1` and `nums2` with the same length.

A subsequence of a array is a new array which is formed from the original array by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, `[2,3,5]` is a subsequence of `[1,2,3,4,5]` while `[1,5,3]` is not).

Example 1:

Input: `nums1 = [2,1,-2,5]`, `nums2 = [3,0,-6]`

Output: 18

Explanation: Take subsequence `[2,-2]` from `nums1` and subsequence `[3,-6]` from `nums2`.

Their dot product is  $(2*3 + (-2)*(-6)) = 18$ .

Example 2:

Input: `nums1 = [3,-2]`, `nums2 = [2,-6,7]`

Output: 21

Explanation: Take subsequence `[3]` from `nums1` and subsequence `[7]` from `nums2`.

Their dot product is  $(3*7) = 21$ .

Example 3:

Input: `nums1 = [-1,-1]`, `nums2 = [1,1]`

Output: -1

Explanation: Take subsequence `[-1]` from `nums1` and subsequence `[1]` from `nums2`.

Their dot product is  $-1$ .

Constraints:

$1 \leq \text{nums1.length}, \text{nums2.length} \leq 500$

$-1000 \leq \text{nums1}[i], \text{nums2}[i] \leq 1000$

### Solution

05/23/2020:

```
class Solution {
public:
    int maxDotProduct(vector<int>& nums1, vector<int>& nums2) {
        // dp[i][j]: the maximum dot product of two subsequences
        // nums1[0..i], nums2[0..j];
        // dp[i][j] = max(max(0, nums1[i] * nums2[j]) + dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1])
        int n1 = nums1.size(), n2 = nums2.size();
        vector<vector<int>> dp(n1, vector<int>(n2, INT_MIN));
        dp[0][0] = nums1[0] * nums2[0];
        for (int i = 1; i < n1; ++i) dp[i][0] = max(nums1[i] * nums2[0], dp[i - 1][0]);
        for (int j = 1; j < n2; ++j) dp[0][j] = max(nums1[0] * nums2[j], dp[0][j - 1]);
        for (int i = 1; i < n1; ++i) {
            for (int j = 1; j < n2; ++j) {
                dp[i][j] = max(nums1[i] * nums2[j], 0) + dp[i - 1][j - 1];
                dp[i][j] = max(dp[i][j], dp[i - 1][j]);
                dp[i][j] = max(dp[i][j], dp[i][j - 1]);
                dp[i][j] = max(dp[i][j], nums1[i] * nums2[j]);
            }
        }
        return dp[n1 - 1][n2 - 1];
    }
};
```

```
class Solution {
public:
    int maxDotProduct(vector<int>& nums1, vector<int>& nums2) {
        int n1 = nums1.size(), n2 = nums2.size();
        vector<vector<int>> dp(n1, vector<int>(n2));
        for (int i = 0; i < n1; ++i) {
            for (int j = 0; j < n2; ++j) {
                dp[i][j] = nums1[i] * nums2[j];
                if (i > 0 && j > 0) dp[i][j] = max(dp[i][j], max(nums1[i] * nums2[j], 0)
+ dp[i - 1][j - 1]);
                if (i > 0) dp[i][j] = max(dp[i][j], dp[i - 1][j]);
                if (j > 0) dp[i][j] = max(dp[i][j], dp[i][j - 1]);
            }
        }
        return dp[n1 - 1][n2 - 1];
    }
};
```

# 1473. Paint House III

## Description

There is a row of  $m$  houses in a small city, each house must be painted with one of the  $n$  colors (labeled from 1 to  $n$ ), some houses that has been painted last summer should not be painted again.

A neighborhood is a maximal group of continuous houses that are painted with the same color. (For example: houses = [1,2,2,3,3,2,1,1] contains 5 neighborhoods [{1}, {2,2}, {3,3}, {2}, {1,1}]).

Given an array houses, an  $m * n$  matrix cost and an integer target where:

houses[i]: is the color of the house  $i$ , 0 if the house is not painted yet.

cost[i][j]: is the cost of paint the house  $i$  with the color  $j+1$ .

Return the minimum cost of painting all the remaining houses in such a way that there are exactly target neighborhoods, if not possible return -1.

Example 1:

Input: houses = [0,0,0,0,0], cost = [[1,10],[10,1],[10,1],[1,10],[5,1]],  $m = 5$ ,  $n = 2$ , target = 3

Output: 9

Explanation: Paint houses of this way [1,2,2,1,1]

This array contains target = 3 neighborhoods, [{1}, {2,2}, {1,1}].

Cost of paint all houses (1 + 1 + 1 + 1 + 5) = 9.

Example 2:

Input: houses = [0,2,1,2,0], cost = [[1,10],[10,1],[10,1],[1,10],[5,1]],  $m = 5$ ,  $n = 2$ , target = 3

Output: 11

Explanation: Some houses are already painted, Paint the houses of this way [2,2,1,2,2]

This array contains target = 3 neighborhoods, [{2,2}, {1}, {2,2}].

Cost of paint the first and last house (10 + 1) = 11.

Example 3:

Input: houses = [0,0,0,0,0], cost = [[1,10],[10,1],[1,10],[10,1],[1,10]],  $m = 5$ ,  $n = 2$ , target = 5

Output: 5

Example 4:

Input: houses = [3,1,2,3], cost = [[1,1,1],[1,1,1],[1,1,1],[1,1,1]],  $m = 4$ ,  $n = 3$ , target = 3

Output: -1

Explanation: Houses are already painted with a total of 4 neighborhoods  $\{\{3\}, \{1\}, \{2\}, \{3\}\}$  different of target = 3.

Constraints:

```
m == houses.length == cost.length
n == cost[i].length
1 <= m <= 100
1 <= n <= 20
1 <= target <= m
0 <= houses[i] <= n
1 <= cost[i][j] <= 10^4
```

*Solution*

06/06/2020:

```
int dp[101][101][20];
class Solution {
public:
    int minCost(vector<int>& houses, vector<vector<int>>& cost, int m, int n, int
target) {
        fill_n(dp[0][0], 101 * 101 * 20, INT_MAX);
        fill_n(dp[0][0], 20, 0);
        for (int i = 1; i <= m; ++i) {
            int hi = houses[i - 1] - 1;
            for (int k = 1; k <= target; ++k) {
                for (int j = 0; j < n; ++j) {
                    if (hi != -1) {
                        dp[k][i][hi] = hi == j ? min(dp[k][i][hi], dp[k][i - 1][hi]) :
min(dp[k][i][hi], dp[k - 1][i - 1][j]);
                    } else {
                        if (dp[k][i - 1][j] != INT_MAX)
                            dp[k][i][j] = min(dp[k][i][j], dp[k][i - 1][j] + cost[i - 1][j]);
                        for (int l = 0; l < n; ++l)
                            if (l != j && dp[k - 1][i - 1][l] != INT_MAX)
                                dp[k][i][j] = min(dp[k][i][j], dp[k - 1][i - 1][l] + cost[i - 1]
[j]);
                    }
                }
            }
        }
        int ret = *min_element(dp[target][m], dp[target][m] + n);
        return ret == INT_MAX ? -1 : ret;
    }
};
```

```

int dp[101][20][101];
class Solution {
public:
    int minCost(vector<int>& houses, vector<vector<int>>& cost, int m, int n, int
target) {
        for (int k = 0; k <= target; ++k)
            for (int i = 0; i <= m; ++i)
                for (int j = 0; j < n; ++j)
                    dp[i][j][k] = INT_MAX;
        for (int j = 0; j < n; ++j) dp[0][j][0] = 0; // no houses, with target 0
        for (int i = 1; i <= m; ++i) {
            int house_i = houses[i - 1] - 1;
            if (house_i != -1) {
                for (int k = 1; k <= target; ++k) {
                    dp[i][house_i][k] = min(dp[i][house_i][k], dp[i - 1][house_i][k]);
                    for (int j = 0; j < n; ++j) {
                        if (j == house_i) continue;
                        dp[i][house_i][k] = min(dp[i][house_i][k], dp[i - 1][j][k - 1]);
                    }
                }
            } else {
                for (int k = 1; k <= target; ++k) {
                    for (int j = 0; j < n; ++j) {
                        if (dp[i - 1][j][k] != INT_MAX) dp[i][j][k] = min(dp[i][j][k], dp[i
- 1][j][k] + cost[i - 1][j]);
                        for (int l = 0; l < n; ++l) {
                            if (l == j || dp[i - 1][l][k - 1] == INT_MAX) continue;
                            dp[i][j][k] = min(dp[i][j][k], dp[i - 1][l][k - 1] + cost[i - 1]
[j]);
                        }
                    }
                }
            }
        }
        int ret = INT_MAX;
        for (int j = 0; j < n; ++j)
            ret = min(ret, dp[m][j][target]);
        return ret == INT_MAX ? -1 : ret;
    }
};

```

## 1478. Allocate Mailboxes

*Description*

Given the array `houses` and an integer `k`. where `houses[i]` is the location of the `i`th house along a street, your task is to allocate `k` mailboxes in the street.

Return the minimum total distance between each house and its nearest mailbox.

The answer is guaranteed to fit in a 32-bit signed integer.

Example 1:

Input: `houses = [1,4,8,10,20]`, `k = 3`

Output: 5

Explanation: Allocate mailboxes in position 3, 9 and 20.

Minimum total distance from each houses to nearest mailboxes is  $|3-1| + |4-3| + |9-8| + |10-9| + |20-20| = 5$

Example 2:

Input: `houses = [2,3,5,12,18]`, `k = 2`

Output: 9

Explanation: Allocate mailboxes in position 3 and 14.

Minimum total distance from each houses to nearest mailboxes is  $|2-3| + |3-3| + |5-3| + |12-14| + |18-14| = 9$ .

Example 3:

Input: `houses = [7,4,6,1]`, `k = 1`

Output: 8

Example 4:

Input: `houses = [3,6,14,10]`, `k = 4`

Output: 0

Constraints:

`n == houses.length`

`1 <= n <= 100`

`1 <= houses[i] <= 10^4`

`1 <= k <= n`

Array `houses` contain unique integers.

*Solution*

06/13/2020:

```

int dp[105][105];

int solve(vector<int>& v, int n, int k) {
    if (n == 0) return 0;
    if (k == 0) return 1e9;
    if (dp[n][k] >= 0) return dp[n][k];
    int ret = 1e9;
    for (int take = 1; take <= n; ++take) {
        int rhs = n - 1;
        int lhs = rhs - take + 1;
        int mid = (lhs + rhs) / 2;
        int candidate = 0;
        for (int i = lhs; i <= rhs; ++i) {
            candidate += abs(v[mid] - v[i]);
        }
        ret = min(ret, candidate + solve(v, n - take, k - 1));
    }
    dp[n][k] = ret;
    return ret;
}

class Solution {
public:
    int minDistance(vector<int>& houses, int k) {
        memset(dp, -1, sizeof(dp));
        sort(houses.begin(), houses.end());
        return solve(houses, houses.size(), k);
    }
};

```