目录

[629. K Inverse Pairs Array 4](#_Toc64621869)

[630. Course Schedule III 6](#_Toc64621870)

[632. Smallest Range Covering Elements from K Lists 8](#_Toc64621871)

[664. Strange Printer 10](#_Toc64621872)

[668. Kth Smallest Number in Multiplication Table 12](#_Toc64621873)

[675. Cut Off Trees for Golf Event 14](#_Toc64621874)

[679. 24 Game 17](#_Toc64621875)

[685. Redundant Connection II 19](#_Toc64621876)

[689. Maximum Sum of 3 Non-Overlapping Subarrays 22](#_Toc64621877)

[691. Stickers to Spell Word 24](#_Toc64621878)

[699. Falling Squares 27](#_Toc64621879)

[710. Random Pick with Blacklist 29](#_Toc64621880)

[715. Range Module 33](#_Toc64621881)

[719. Find K-th Smallest Pair Distance 34](#_Toc64621882)

[726. Number of Atoms 36](#_Toc64621883)

[730. Count Different Palindromic Subsequences 40](#_Toc64621884)

[732. My Calendar III 42](#_Toc64621885)

[741. Cherry Pickup 44](#_Toc64621886)

[745. Prefix and Suffix Search 47](#_Toc64621887)

[757. Set Intersection Size At Least Two 49](#_Toc64621888)

[761. Special Binary String 51](#_Toc64621889)

[765. Couples Holding Hands 53](#_Toc64621890)

[768. Max Chunks To Make Sorted II 55](#_Toc64621891)

[773. Sliding Puzzle 59](#_Toc64621892)

[778. Swim in Rising Water 61](#_Toc64621893)

[786. K-th Smallest Prime Fraction 64](#_Toc64621894)

[793. Preimage Size of Factorial Zeroes Function 66](#_Toc64621895)

[798. Smallest Rotation with Highest Score 68](#_Toc64621896)

[803. Bricks Falling When Hit 70](#_Toc64621897)

[805. Split Array With Same Average 74](#_Toc64621898)

[815. Bus Routes 76](#_Toc64621899)

[818. Race Car 78](#_Toc64621900)

[827. Making A Large Island 81](#_Toc64621901)

[828. Count Unique Characters of All Substrings of a Given String 83](#_Toc64621902)

[834. Sum of Distances in Tree 85](#_Toc64621903)

[839. Similar String Groups 87](#_Toc64621904)

[847. Shortest Path Visiting All Nodes 89](#_Toc64621905)

[850. Rectangle Area II 91](#_Toc64621906)

[854. K-Similar Strings 93](#_Toc64621907)

[857. Minimum Cost to Hire K Workers 94](#_Toc64621908)

[862. Shortest Subarray with Sum at Least K 96](#_Toc64621909)

[864. Shortest Path to Get All Keys 99](#_Toc64621910)

[871. Minimum Number of Refueling Stops 101](#_Toc64621911)

[878. Nth Magical Number 105](#_Toc64621912)

[879. Profitable Schemes 107](#_Toc64621913)

[887. Super Egg Drop 109](#_Toc64621914)

[891. Sum of Subsequence Widths 112](#_Toc64621915)

[895. Maximum Frequency Stack 114](#_Toc64621916)

[902. Numbers At Most N Given Digit Set 117](#_Toc64621917)

[903. Valid Permutations for DI Sequence 120](#_Toc64621918)

[913. Cat and Mouse 121](#_Toc64621919)

[920. Number of Music Playlists 123](#_Toc64621920)

[924. Minimize Malware Spread 125](#_Toc64621921)

[927. Three Equal Parts 128](#_Toc64621922)

[928. Minimize Malware Spread II 130](#_Toc64621923)

[936. Stamping The Sequence 133](#_Toc64621924)

[940. Distinct Subsequences II 135](#_Toc64621925)

[943. Find the Shortest Superstring 137](#_Toc64621926)

[952. Largest Component Size by Common Factor 139](#_Toc64621927)

[956. Tallest Billboard 142](#_Toc64621928)

[960. Delete Columns to Make Sorted III 145](#_Toc64621929)

[964. Least Operators to Express Number 148](#_Toc64621930)

[968. Binary Tree Cameras 151](#_Toc64621931)

[975. Odd Even Jump 154](#_Toc64621932)

[980. Unique Paths III 159](#_Toc64621933)

[992. Subarrays with K Different Integers 162](#_Toc64621934)

[995. Minimum Number of K Consecutive Bit Flips 164](#_Toc64621935)

[996. Number of Squareful Arrays 166](#_Toc64621936)

[1000. Minimum Cost to Merge Stones 168](#_Toc64621937)

[1001. Grid Illumination 171](#_Toc64621938)

[1012. Numbers With Repeated Digits 174](#_Toc64621939)

[1028. Recover a Tree From Preorder Traversal 176](#_Toc64621940)

[1032. Stream of Characters 179](#_Toc64621941)

[1036. Escape a Large Maze 182](#_Toc64621942)

[1044. Longest Duplicate Substring 183](#_Toc64621943)

[1074. Number of Submatrices That Sum to Target 186](#_Toc64621944)

[1092. Shortest Common Supersequence 189](#_Toc64621945)

[1095. Find in Mountain Array 191](#_Toc64621946)

[1096. Brace Expansion II 193](#_Toc64621947)

[1106. Parsing A Boolean Expression 196](#_Toc64621948)

[1125. Smallest Sufficient Team 198](#_Toc64621949)

[1147. Longest Chunked Palindrome Decomposition 200](#_Toc64621950)

[1157. Online Majority Element In Subarray 202](#_Toc64621951)

[1172. Dinner Plate Stacks 203](#_Toc64621952)

[1178. Number of Valid Words for Each Puzzle 207](#_Toc64621953)

[1187. Make Array Strictly Increasing 209](#_Toc64621954)

[1192. Critical Connections in a Network 211](#_Toc64621955)

[1203. Sort Items by Groups Respecting Dependencies 212](#_Toc64621956)

[1206. Design Skiplist 214](#_Toc64621957)

[1210. Minimum Moves to Reach Target with Rotations 218](#_Toc64621958)

[1220. Count Vowels Permutation 221](#_Toc64621959)

[1224. Maximum Equal Frequency 223](#_Toc64621960)

[1235. Maximum Profit in Job Scheduling 225](#_Toc64621961)

### 629. K Inverse Pairs Array★★

Hard

Given two integers n and k, find how many different arrays consist of numbers from 1 to n such that there are exactly k inverse pairs.

We define an inverse pair as following: For ith and jth element in the array, if i < j and a[i] > a[j] then it's an inverse pair; Otherwise, it's not.

Since the answer may be very large, the answer should be modulo 109 + 7.

**Example 1:**

**Input:** n = 3, k = 0

**Output:** 1

**Explanation:**

Only the array [1,2,3] which consists of numbers from 1 to 3 has exactly 0 inverse pair.

**Example 2:**

**Input:** n = 3, k = 1

**Output:** 2

**Explanation:**

The array [1,3,2] and [2,1,3] have exactly 1 inverse pair.

**Note:**

1. The integer n is in the range [1, 1000] and k is in the range [0, 1000].

class Solution {

public:

    int kInversePairs(int n, int k) {

        vector<vector<long long>> dp(2, vector<long long>(k+1));

        dp[0][0] = 1;

        int p = 0;

        for(int i = 1; i <= n; ++i){

            auto &pre = dp[p], &cur = dp[p^1];

            cur[0] = 1;

            for(int j = 1; j <= k; ++j){

                cur[j] = (cur[j-1] - (j >= i ? pre[j-i]:0)

+ pre[j] + mod) % mod;

            }

            p ^= 1;

        }

        return static\_cast<int> (dp[p][k]);

    }

private:

    const long long mod = 1e9 + 7;

};

### 630. Course Schedule III★★

Hard

There are n different online courses numbered from 1 to n. Each course has some duration(course length) t and closed on dth day. A course should be taken **continuously** for t days and must be finished before or on the dth day. You will start at the 1st day.

Given n online courses represented by pairs (t,d), your task is to find the maximal number of courses that can be taken.

**Example:**

**Input:** [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]

**Output:** 3

**Explanation:**

There're totally 4 courses, but you can take 3 courses at most:

First, take the 1st course, it costs 100 days so you will finish it on the 100th day, and ready to take the next course on the 101st day.

Second, take the 3rd course, it costs 1000 days so you will finish it on the 1100th day, and ready to take the next course on the 1101st day.

Third, take the 2nd course, it costs 200 days so you will finish it on the 1300th day.

The 4th course cannot be taken now, since you will finish it on the 3300th day, which exceeds the closed date.

**Note:**

1. The integer 1 <= d, t, n <= 10,000.
2. You can't take two courses simultaneously.

class Solution {

public:

    int scheduleCourse(vector<vector<int>>& courses) {

        sort(courses.begin(), courses.end(), [](auto& p, auto& q) {

return p[1] < q[1];

});

        priority\_queue<int> q;

        int sum = 0;

        for (auto& c : courses) {

            q.push(c[0]);

            sum += c[0];

            if (sum > c[1]) {

                sum -= q.top();

                q.pop();

            }

        }

        return q.size();

    }

};

### 632. Smallest Range Covering Elements from K Lists

Hard

You have k lists of sorted integers in **non-decreasing order**. Find the **smallest** range that includes at least one number from each of the k lists.

We define the range [a, b] is smaller than range [c, d] if b - a < d - c **or** a < c if b - a == d - c.

**Example 1:**

**Input:** nums = [[4,10,15,24,26],[0,9,12,20],[5,18,22,30]]

**Output:** [20,24]

**Example 2:**

**Input:** nums = [[1,2,3],[1,2,3],[1,2,3]]

**Output:** [1,1]

**Example 3:**

**Input:** nums = [[10,10],[11,11]]

**Output:** [10,11]

**Example 4:**

**Input:** nums = [[10],[11]]

**Output:** [10,11]

**Example 5:**

**Input:** nums = [[1],[2],[3],[4],[5],[6],[7]]

**Output:** [1,7]

**Constraints:**

* nums.length == k
* 1 <= k <= 3500
* 1 <= nums[i].length <= 50
* -105 <= nums[i][j] <= 105
* nums[i] is sorted in **non-decreasing** order.

class Solution {

public:

    vector<int> smallestRange(vector<vector<int>>& nums) {

        using vi = vector<int>::iterator;

        using pvv = pair<vi, vi>;

        auto cmp = [](const pvv &p1, const pvv &p2) {

            return \*p1.first > \*p2.first;

        };

        int lo = INT\_MAX, hi = INT\_MIN;

        priority\_queue<pvv, vector<pvv>, decltype(cmp)> pq(cmp);

        for (auto &row : nums) {

            lo = min(lo, row[0]);

            hi = max(hi, row[0]);

            pq.emplace(row.begin(), row.end());

        }

        vector<int> res{lo, hi};

        while (1) {

            auto p = pq.top();

            pq.pop();

            ++p.first;

            if (p.first == p.second) break;

            pq.push(p);

            lo = \*pq.top().first;

            hi = max(hi, \*p.first);

            if (hi - lo < res[1] - res[0])

                res = {lo, hi};

        }

        return res;

    }

};

### 664. Strange Printer★★

Hard

There is a strange printer with the following two special requirements:

1. The printer can only print a sequence of the same character each time.
2. At each turn, the printer can print new characters starting from and ending at any places, and will cover the original existing characters.

Given a string consists of lower English letters only, your job is to count the minimum number of turns the printer needed in order to print it.

**Example 1:**

**Input:** "aaabbb"

**Output:** 2

**Explanation:** Print "aaa" first and then print "bbb".

**Example 2:**

**Input:** "aba"

**Output:** 2

**Explanation:** Print "aaa" first and then print "b" from the second place of the string, which will cover the existing character 'a'.

**Hint**: Length of the given string will not exceed 100.

class Solution {

public:

    int strangePrinter(string t) {

        if (t.empty()) return 0;

        string s;

        for (auto c : t) {

            if (s.empty() || c != s.back()) s += c;

        }

        int n = s.size();

        vector<vector<int>> dp(n, vector<int>(n));

        for (int len = 0; len < n; ++len) {

            for (int i = 0; i < n - len; ++i) {

                int j = i + len;

                dp[i][j] = len + 1;

                for (int k = i; k < j; ++k) {

                    if (s[k] == s[j])

dp[i][j] = min(dp[i][j],dp[i][k]+dp[k+1][j]-1);

                    else dp[i][j] = min(dp[i][j], dp[i][k]+dp[k+1][j]);

                }

            }

        }

        return dp[0][n-1];

    }

};

### 668. Kth Smallest Number in Multiplication Table

Hard

Nearly every one have used the [Multiplication Table](https://en.wikipedia.org/wiki/Multiplication_table). But could you find out the k-th smallest number quickly from the multiplication table?

Given the height m and the length n of a m \* n Multiplication Table, and a positive integer k, you need to return the k-th smallest number in this table.

**Example 1:**

**Input:** m = 3, n = 3, k = 5

**Output:**

**Explanation:**

The Multiplication Table:

1 2 3

2 4 6

3 6 9

The 5-th smallest number is 3 (1, 2, 2, 3, 3).

**Example 2:**

**Input:** m = 2, n = 3, k = 6

**Output:**

**Explanation:**

The Multiplication Table:

1 2 3

2 4 6

The 6-th smallest number is 6 (1, 2, 2, 3, 4, 6).

**Note:**

1. The m and n will be in the range [1, 30000].
2. The k will be in the range [1, m \* n]

class Solution {

public:

    int findKthNumber(int m, int n, int k) {

        int left = 1, right = m \* n + 1;

        while (left < right) {

            int mid = left + (right - left) / 2, cnt = 0;

            for (int i = 1; i <= m; ++i) {

                cnt += min(mid / i, n);

            }

            if (cnt < k) left = mid + 1;

            else right = mid;

        }

        return left;

    }

};

### 675. Cut Off Trees for Golf Event

Hard

You are asked to cut off all the trees in a forest for a golf event. The forest is represented as an m x n matrix. In this matrix:

* 0 means the cell cannot be walked through.
* 1 represents an empty cell that can be walked through.
* A number greater than 1 represents a tree in a cell that can be walked through, and this number is the tree's height.

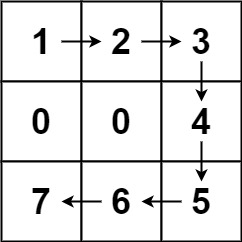
In one step, you can walk in any of the four directions: north, east, south, and west. If you are standing in a cell with a tree, you can choose whether to cut it off.

You must cut off the trees in order from shortest to tallest. When you cut off a tree, the value at its cell becomes 1 (an empty cell).

Starting from the point (0, 0), return *the minimum steps you need to walk to cut off all the trees*. If you cannot cut off all the trees, return -1.

You are guaranteed that no two trees have the same height, and there is at least one tree needs to be cut off.

**Example 1:**

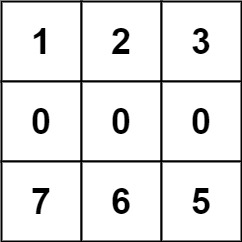


**Input:** forest = [[1,2,3],[0,0,4],[7,6,5]]

**Output:** 6

**Explanation:** Following the path above allows you to cut off the trees from shortest to tallest in 6 steps.

**Example 2:**



**Input:** forest = [[1,2,3],[0,0,0],[7,6,5]]

**Output:** -1

**Explanation:** The trees in the bottom row cannot be accessed as the middle row is blocked.

**Example 3:**

**Input:** forest = [[2,3,4],[0,0,5],[8,7,6]]

**Output:** 6

**Explanation:** You can follow the same path as Example 1 to cut off all the trees.

Note that you can cut off the first tree at (0, 0) before making any steps.

**Constraints:**

* m == forest.length
* n == forest[i].length
* 1 <= m, n <= 50
* 0 <= forest[i][j] <= 109

class Solution {

public:

    const int dir[5] = {0, -1, 0, 1, 0};

    int n, m;

    int cutOffTree(vector<vector<int>>& v) {

        m = v.size(), n = v[0].size();

        vector<vector<int>> heights;

        for (int i = 0; i < m; i++)

            for (int j = 0; j < n; j++) if (v[i][j] > 1)

                heights.push\_back({v[i][j], i, j});

        sort(heights.begin(), heights.end());

        int x = 0, y = 0, res = 0;

        for (int i = 0; i < heights.size(); i++) {

            int cnt = BFS(v, x, y, heights[i][1], heights[i][2]);

            if (cnt == -1) return -1;

            res += cnt;

            x = heights[i][1], y = heights[i][2];

        }

        return res;

    }

    int BFS(vector<vector<int>>& v, int x0, int y0, int x1, int y1){

        if (x0 == x1 && y0 == y1) return 0;

        int cnt = 0;

        queue<pair<int, int>> q;

        q.emplace(x0, y0);

        vector<vector<bool>> vis(m, vector<bool>(n));

        vis[x0][y0] = true;

        while(!q.empty()) {

            int sz = q.size(); cnt++;

            while(sz--) {

                auto [cur\_x, cur\_y] = q.front(); q.pop();

                for(int k = 0; k < 4; k++) {

                    int x = cur\_x + dir[k], y = cur\_y + dir[k+1];

                    if (x>=0&&x<m&&y>=0&&y<n && v[x][y] && !vis[x][y]){

                        if (x == x1 && y == y1) return cnt;

                        vis[x][y] = true;

                        q.emplace(x,y);

                    }

                }

            }

        }

        return -1;

    }

};

### 679. 24 Game

Hard

You have 4 cards each containing a number from 1 to 9. You need to judge whether they could operated through \*, /, +, -, (, ) to get the value of 24.

**Example 1:**

**Input:** [4, 1, 8, 7]

**Output:** True

**Explanation:** (8-4) \* (7-1) = 24

**Example 2:**

**Input:** [1, 2, 1, 2]

**Output:** False

**Note:**

1. The division operator / represents real division, not integer division. For example, 4 / (1 - 2/3) = 12.
2. Every operation done is between two numbers. In particular, we cannot use - as a unary operator. For example, with [1, 1, 1, 1] as input, the expression -1 - 1 - 1 - 1 is not allowed.
3. You cannot concatenate numbers together. For example, if the input is [1, 2, 1, 2], we cannot write this as 12 + 12.

double MyAdd(double a, double b) {return a+b;}

auto MySub = std::minus<double>();

struct MyMul {

    double operator () (double a, double b) {

        return a\*b;

    }

};

auto MyDiv = [](double a, double b) -> double {return a/b;};

unordered\_map<char, function<double(double, double)>> MyOperations{

    {'+', MyAdd}, {'-', MySub}, {'\*', MyMul()}, {'/', MyDiv}

};

class Solution {

public:

    bool judgePoint24(vector<int>& nums) {

       vector<double> pre(nums.begin(), nums.end());

       return find24(pre, 4);

    }

private:

    bool find24(const vector<double> &pre, int n) {

        if (n == 1) return abs(pre[0] - 24) < 1e-6;

        for (int i = 0; i < n; ++i) {

            for (int j = 0; j < n; ++j) {

                if (i == j) continue;

                vector<double> cur;

                for(int k = 0; k < n; ++k) {

                   if (k != i && k != j) cur.push\_back(pre[k]);

                }

                for (auto op : string("+-\*/")) {

                    if ((op == '+' || op == '\*') && i > j) continue;

                    else if (op == '/'  && abs(pre[j]) < 1e-6)

continue;

                    cur.push\_back(MyOperations[op](pre[i], pre[j]));

                    if (find24(cur, n-1)) return true;

                    cur.pop\_back();

                }

            }

        }

        return false;

    }

};

### 685. Redundant Connection II★★

Hard

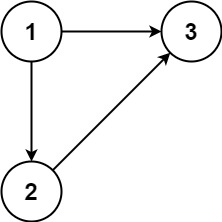
In this problem, a rooted tree is a **directed** graph such that, there is exactly one node (the root) for which all other nodes are descendants of this node, plus every node has exactly one parent, except for the root node which has no parents.

The given input is a directed graph that started as a rooted tree with n nodes (with distinct values from 1 to n), with one additional directed edge added. The added edge has two different vertices chosen from 1 to n, and was not an edge that already existed.

The resulting graph is given as a 2D-array of edges. Each element of edges is a pair [ui, vi] that represents a **directed** edge connecting nodes ui and vi, where ui is a parent of child vi.

Return *an edge that can be removed so that the resulting graph is a rooted tree of* n *nodes*. If there are multiple answers, return the answer that occurs last in the given 2D-array.

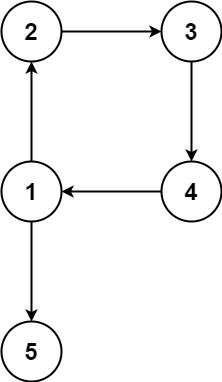
**Example 1:**



**Input:** edges = [[1,2],[1,3],[2,3]]

**Output:** [2,3]

**Example 2:**



**Input:** edges = [[1,2],[2,3],[3,4],[4,1],[1,5]]

**Output:** [4,1]

**Constraints:**

* n == edges.length
* 3 <= n <= 1000
* edges[i].length == 2
* 1 <= ui, vi <= n

class Solution {

public:

    vector<int> findRedundantDirectedConnection(vector<vector<int>>& edges) {

        //给一棵有向树添加一条边，情况有四种(注意指向root情况)

        int n = edges.size();

        vector<int> fa(n+1), candA, candB;

        // step 1, 找到使得某一结点入度为2的边

        for (auto &edge : edges) {

            if (fa[edge[1]] == 0)

                fa[edge[1]] = edge[0];

            else {

                candA = {fa[edge[1]], edge[1]};

                candB = edge;

                //作用 不把candB加入是否满足

                edge[1] = 0;

            }

        }

        // step 2, union find

        for (int i = 1; i <= n; i++) fa[i] = i;

        for (auto &edge:edges) {

            if (edge[1] == 0) continue;

            int fax = find(edge[0], fa);

            int fay = find(edge[1], fa);

            if (fax == fay) {

                return candA.empty() ? edge : candA;

            }

            fa[fax] = fay;

        }

        return candB;

    }

private:

    int find(int x, vector<int>& fa) {

        return x == fa[x] ? x : fa[x] = find(fa[x], fa);

    }

};

### 689. Maximum Sum of 3 Non-Overlapping Subarrays★★

Hard

In a given array nums of positive integers, find three non-overlapping subarrays with maximum sum.

Each subarray will be of size k, and we want to maximize the sum of all 3\*k entries.

Return the result as a list of indices representing the starting position of each interval (0-indexed). If there are multiple answers, return the lexicographically smallest one.

**Example:**

**Input:** [1,2,1,2,6,7,5,1], 2

**Output:** [0, 3, 5]

**Explanation:** Subarrays [1, 2], [2, 6], [7, 5] correspond to the starting indices [0, 3, 5].

We could have also taken [2, 1], but an answer of [1, 3, 5] would be lexicographically larger.

**Note:**

* nums.length will be between 1 and 20000.
* nums[i] will be between 1 and 65535.
* k will be between 1 and floor(nums.length / 3).

class Solution {

public:

/\* 1，定义如下：

    sums[i]代表以nums[i]结尾的前k个数的和

    dp[i][j]代表截止到nums[i]形成的第j个无重叠子数组的最大和

    path[i][j]代表截止到nums[i]形成的第j个无重叠子数组以哪个下标为结尾，用来回溯路径

  2，状态转移方程为

    dp[i][j] = max{dp[i - 1][j], sums[i] + dp[i - k][j - 1]};

    path[i][j] = path[i - 1][j]或i\*/

    vector<int> maxSumOfThreeSubarrays(vector<int>& nums, int k) {

        return maxSumOfNSubarrays(nums, k, 3);

    }

private:

    vector<int> maxSumOfNSubarrays(vector<int>& nums, int k, int m) {

        vector<int> res;

        if (k < 1 || m \* k > nums.size()) return res;

        int n=nums.size(), s=accumulate(nums.begin(),nums.begin()+k,0);

        vector<int> preSum;

        partial\_sum(nums.begin(), nums.end(), back\_inserter(preSum));

        vector<vector<int>> dp(n, vector<int>(m + 1, 0));

        vector<vector<int>> path(n, vector<int>(m + 1, 0));

        dp[k-1][1] = preSum[k - 1];

        path[k-1][1] = k - 1;

        for (int i = k; i < n; ++i) {

            for (int j = 1; j <= m; ++j) {

                dp[i][j] = dp[i-1][j];

                path[i][j] = path[i-1][j];

                if (dp[i][j] < preSum[i] - preSum[i-k] + dp[i-k][j-1]){

                    dp[i][j] = preSum[i] - preSum[i-k] + dp[i-k][j-1];

                    path[i][j] = i;

                }

            }

        }

        int pos = path[n-1][m];

        res.push\_back(pos - k + 1);

        for (int i = m - 1; i > 0; --i) {

            pos = path[pos - k][i];

            res.push\_back(pos - k + 1);

        }

        reverse(res.begin(), res.end());

        return res;

    }

};

### 691. Stickers to Spell Word★★

Hard

We are given N different types of stickers. Each sticker has a lowercase English word on it.

You would like to spell out the given target string by cutting individual letters from your collection of stickers and rearranging them.

You can use each sticker more than once if you want, and you have infinite quantities of each sticker.

What is the minimum number of stickers that you need to spell out the target? If the task is impossible, return -1.

**Example 1:**

Input:

["with", "example", "science"], "thehat"

Output:

3

Explanation:

We can use 2 "with" stickers, and 1 "example" sticker.

After cutting and rearrange the letters of those stickers, we can form the target "thehat".

Also, this is the minimum number of stickers necessary to form the target string.

**Example 2:**

Input:

["notice", "possible"], "basicbasic"

Output:

-1

Explanation:

We can't form the target "basicbasic" from cutting letters from the given stickers.

**Note:**

 stickers has length in the range [1, 50].

 stickers consists of lowercase English words (without apostrophes).

 target has length in the range [1, 15], and consists of lowercase English letters.

 In all test cases, all words were chosen randomly from the 1000 most common US English words, and the target was chosen as a concatenation of two random words.

 The time limit may be more challenging than usual. It is expected that a 50 sticker test case can be solved within 35ms on average.

class Solution {

public:

    int minStickers(vector<string>& stickers, string target) {

        int n = target.size(), N = 1 << n;

        vector<uint> dp(N, -1);

        dp[0] = 0;

        for (int i = 0; i < N; i++) if (dp[i] != -1) {

            for (string& s : stickers) {

                int now = i;

                for (char c : s) {

                    for (int r = 0; r < n; r++) {

                        if (target[r] == c && !((now >> r) & 1)) {

                            now |= 1 << r;

                            break;

                        }

                    }

                }

                dp[now] = min(dp[now], dp[i] + 1);

            }

        }

        return dp[N-1];

    }

};

### 699. Falling Squares

Hard

On an infinite number line (x-axis), we drop given squares in the order they are given.

The i-th square dropped (positions[i] = (left, side\_length)) is a square with the left-most point being positions[i][0] and sidelength positions[i][1].

The square is dropped with the bottom edge parallel to the number line, and from a higher height than all currently landed squares. We wait for each square to stick before dropping the next.

The squares are infinitely sticky on their bottom edge, and will remain fixed to any positive length surface they touch (either the number line or another square). Squares dropped adjacent to each other will not stick together prematurely.

Return a list ans of heights. Each height ans[i] represents the current highest height of any square we have dropped, after dropping squares represented by positions[0], positions[1], ..., positions[i].

**Example 1:**

**Input:** [[1, 2], [2, 3], [6, 1]]

**Output:** [2, 5, 5]

**Explanation:**

After the first drop of positions[0] = [1, 2]: \_aa \_aa ------- The maximum height of any square is 2.

After the second drop of positions[1] = [2, 3]: \_\_aaa \_\_aaa \_\_aaa \_aa\_\_ \_aa\_\_ -------------- The maximum height of any square is 5. The larger square stays on top of the smaller square despite where its center of gravity is, because squares are infinitely sticky on their bottom edge.

After the third drop of positions[1] = [6, 1]: \_\_aaa \_\_aaa \_\_aaa \_aa \_aa\_\_\_a -------------- The maximum height of any square is still 5. Thus, we return an answer of [2, 5, 5].

**Example 2:**

**Input:** [[100, 100], [200, 100]]

**Output:** [100, 100]

**Explanation:** Adjacent squares don't get stuck prematurely - only their bottom edge can stick to surfaces.

**Note:**

* 1 <= positions.length <= 1000.
* 1 <= positions[i][0] <= 10^8.
* 1 <= positions[i][1] <= 10^6.

### 710. Random Pick with Blacklist★

Hard

Given a blacklist B containing unique integers from [0, N), write a function to return a uniform random integer from [0, N) which is **NOT** in B.

Optimize it such that it minimizes the call to system’s Math.random().

**Note:**

1. 1 <= N <= 1000000000
2. 0 <= B.length < min(100000, N)
3. [0, N) does NOT include N. See [interval notation](https://en.wikipedia.org/wiki/Interval_(mathematics)).

**Example 1:**

**Input:**

["Solution","pick","pick","pick"]

[[1,[]],[],[],[]]

**Output:** [null,0,0,0]

**Example 2:**

**Input:**

["Solution","pick","pick","pick"]

[[2,[]],[],[],[]]

**Output:** [null,1,1,1]

**Example 3:**

**Input:**

["Solution","pick","pick","pick"]

[[3,[1]],[],[],[]]

**Output:** [null,0,0,2]

**Example 4:**

**Input:**

["Solution","pick","pick","pick"]

[[4,[2]],[],[],[]]

**Output:** [null,1,3,1]

**Explanation of Input Syntax:**

The input is two lists: the subroutines called and their arguments. Solution's constructor has two arguments, N and the blacklist B. pick has no arguments. Arguments are always wrapped with a list, even if there aren't any.

//黑名单向白名单做映射

class Solution {

public:

    unordered\_map<int, int> mp;

    int wlen;

    Solution(int n, vector<int> b) {

        wlen = n - b.size();

        unordered\_set<int> w;

        for (int i = wlen; i < n; i++) w.insert(i);

        for (int x : b) w.erase(x);

        auto wi = w.begin();

        for (int x : b)

            if (x < wlen)

                mp[x] = \*wi++;

    }

    int pick() {

        int k = rand() % wlen;

        return mp.count(k) ? m[k] : k;

    }

};

//二分

//在B中找到第一个比k大的数字

class Solution {

public:

    int m;

    vector<int> b;

    Solution(int N, vector<int> blacklist) {

        m = N - blacklist.size();

        sort(blacklist.begin(), blacklist.end());

        blacklist.push\_back(N);

        b = std::move(blacklist);

    }

    int pick() {

        int k = rand() % m;

        int lo = 0, hi = b.size();

        while (lo < hi) {

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

            if (b[mid] - mid <= k) lo = mid + 1;

            else hi = mid;

        }

        return b[lo] - (b[lo] - lo - k);

    }

};

### 715. Range Module

Hard

A Range Module is a module that tracks ranges of numbers. Your task is to design and implement the following interfaces in an efficient manner.

 addRange(int left, int right) Adds the half-open interval [left, right), tracking every real number in that interval. Adding an interval that partially overlaps with currently tracked numbers should add any numbers in the interval [left, right) that are not already tracked.

 queryRange(int left, int right) Returns true if and only if every real number in the interval [left, right) is currently being tracked.

 removeRange(int left, int right) Stops tracking every real number currently being tracked in the interval [left, right).

**Example 1:**

**addRange(10, 20)**: null

**removeRange(14, 16)**: null

**queryRange(10, 14)**: true (Every number in [10, 14) is being tracked)

**queryRange(13, 15)**: false (Numbers like 14, 14.03, 14.17 in [13, 15) are not being tracked)

**queryRange(16, 17)**: true (The number 16 in [16, 17) is still being tracked, despite the remove operation)

**Note:**

 A half open interval [left, right) denotes all real numbers left <= x < right.

 0 < left < right < 10^9 in all calls to addRange, queryRange, removeRange.

 The total number of calls to addRange in a single test case is at most 1000.

 The total number of calls to queryRange in a single test case is at most 5000.

 The total number of calls to removeRange in a single test case is at most 1000.

### 719. Find K-th Smallest Pair Distance

Hard

Given an integer array, return the k-th smallest **distance** among all the pairs. The distance of a pair (A, B) is defined as the absolute difference between A and B.

**Example 1:**

**Input:**

nums = [1,3,1]

k = 1

**Output: 0**

**Explanation:**

Here are all the pairs:

(1,3) -> 2

(1,1) -> 0

(3,1) -> 2

Then the 1st smallest distance pair is (1,1), and its distance is 0.

**Note:**

1. 2 <= len(nums) <= 10000.
2. 0 <= nums[i] < 1000000.
3. 1 <= k <= len(nums) \* (len(nums) - 1) / 2.

class Solution {

public:

    int smallestDistancePair(vector<int>& nums, int k) {

        sort(nums.begin(), nums.end());

        int n = nums.size(), low = 0, high = 1e6;

        while (low < high) {

            int mid = low + (high - low) / 2, cnt = 0;

            for (int i = 0, j = 0; i < n; i++) {

                while (j < n && nums[j]-nums[i] <= mid) j++;

                cnt += j-i-1;

            }

            if (cnt < k) low = mid + 1;

            else high = mid;

        }

        return low;

    }

};

### 726. Number of Atoms

Hard

Given a chemical formula (given as a string), return the count of each atom.

The atomic element always starts with an uppercase character, then zero or more lowercase letters, representing the name.

One or more digits representing that element's count may follow if the count is greater than 1. If the count is 1, no digits will follow. For example, H2O and H2O2 are possible, but H1O2 is impossible.

Two formulas concatenated together to produce another formula. For example, H2O2He3Mg4 is also a formula.

A formula placed in parentheses, and a count (optionally added) is also a formula. For example, (H2O2) and (H2O2)3 are formulas.

Given a formula, return *the count of all elements as a string in the following form*: the first name (in sorted order), followed by its count (if that count is more than 1), followed by the second name (in sorted order), followed by its count (if that count is more than 1), and so on.

**Example 1:**

**Input:** formula = "H2O"

**Output:** "H2O"

**Explanation:** The count of elements are {'H': 2, 'O': 1}.

**Example 2:**

**Input:** formula = "Mg(OH)2"

**Output:** "H2MgO2"

**Explanation:** The count of elements are {'H': 2, 'Mg': 1, 'O': 2}.

**Example 3:**

**Input:** formula = "K4(ON(SO3)2)2"

**Output:** "K4N2O14S4"

**Explanation:** The count of elements are {'K': 4, 'N': 2, 'O': 14, 'S': 4}.

**Example 4:**

**Input:** formula = "Be32"

**Output:** "Be32"

**Constraints:**

* 1 <= formula.length <= 1000
* formula consists of English letters, digits, '(', and ')'.
* formula is always valid.

class Solution {

public:

    using mp = map<string, int>;

    stack<mp> stk;

    mp cur;

    string pre, res;

    bool flag = false;

    string countOfAtoms(string formula) {

        stringstream ss(formula);

        char c;

        while (ss >> c) {

            if (c == '(') {

                mp tmp;

                swap(tmp, cur);

                stk.push(std::move(tmp));

            }

            else if (c == ')') flag = true;

            else if (isdigit(c)) {

                int cnt = c - '0';

                while (isdigit(ss.peek())) {

                    ss >> c;

                    cnt = cnt\*10 + c-'0';

                }

                if (flag) {

                    for (auto &[s, n] : cur) n \*= cnt;

                    for (auto &[s, n] : stk.top()) cur[s] += n;

                    stk.pop();

                    flag = false;

                }

                else cur[pre] += cnt-1;

            }

            else {

                string t = string(1, c);

                while (islower(ss.peek())) {

                    ss >> c;

                    t += c;

                }

                ++cur[pre = t];

            }

        }

        while (!stk.empty()) {

            for (auto &[s, n] : stk.top()) cur[s] += n;

            stk.pop();

        }

        for (auto &[s, n] : cur) {

            res += s + (n == 1 ? "" : to\_string(n));

        }

        return res;

    }

};

### 730. Count Different Palindromic Subsequences★★★

Hard

Given a string S, find the number of different non-empty palindromic subsequences in S, and **return that number modulo 10^9 + 7.**

A subsequence of a string S is obtained by deleting 0 or more characters from S.

A sequence is palindromic if it is equal to the sequence reversed.

Two sequences A\_1, A\_2, ... and B\_1, B\_2, ... are different if there is some i for which A\_i != B\_i.

**Example 1:**

**Input:**

S = 'bccb'

**Output:** 6

**Explanation:**

The 6 different non-empty palindromic subsequences are 'b', 'c', 'bb', 'cc', 'bcb', 'bccb'.

Note that 'bcb' is counted only once, even though it occurs twice.

**Example 2:**

**Input:**

S = 'abcdabcdabcdabcdabcdabcdabcdabcddcbadcbadcbadcbadcbadcbadcbadcba'

**Output:** 104860361

**Explanation:**

There are 3104860382 different non-empty palindromic subsequences, which is 104860361 modulo 10^9 + 7.

**Note:**

 The length of S will be in the range [1, 1000].

 Each character S[i] will be in the set {'a', 'b', 'c', 'd'}.

/\*

if s[i]!=s[j] dp(i,j)=dp(i,j-1)+dp(i+1,j)-dp(i+1,j-1)

else 定义left(i)表示第一个和s(i)相同的下标 right(j)表示第一个和s(j)相同的下标有：

if left(i)>right(j) 即i和j之间不存在和s(i)相同的元素 有 dp(i,j)=dp(i+1,j-1)\*2+2 (+2是加单个s（i）和一对s（i）)

if left(i)=right(j) 即i和j之间有且只有一个元素和s(i)相同 有 dp(i,j)=dp(i+1,j-1)\*2+1 (+1是加一对s(i) 单个s（i）已经有了)

if left(i)<right(j) 即i和j之间有一个区间也是s(i)元素包裹的 有 dp(i,j)=dp(i+1,j-1)\*2-dp(left(i)+1,right(j)-1) (left+1和right-1之间的回文串不需要再叠加)

\*/

class Solution {

public:

    int countPalindromicSubsequences(string s) {

        const int n = s.length(), mod = 1e9+7;

        vector<vector<long long>> dp(n, vector<long long>(n, 0));

        for (int i = 0; i < n; ++i) dp[i][i] = 1;

        for (int i = n-1; i >= 0; i--) {

            for (int j = i+1; j < n; j++) {

                if (s[i] == s[j]) {

                    int left = i+1,right = j-1;

                    while (left <= right && s[left] != s[i]) ++left;

                    while (right >= left && s[right] != s[j]) --right;

                    if (left > right) dp[i][j] = dp[i+1][j-1]\*2 + 2;

                    else if (left == right)

dp[i][j] = dp[i+1][j-1]\*2 + 1;

                    else

dp[i][j] = dp[i+1][j-1]\*2 - dp[left+1][right-1];

                }

                else {

                    dp[i][j] = dp[i][j-1] + dp[i+1][j] - dp[i+1][j-1];

                }

                dp[i][j] %= mod;

            }

        }

        return static\_cast<int> ((dp[0][n-1]+mod) % mod);

    }

};

### 732. My Calendar III

Hard

Implement a MyCalendarThree class to store your events. A new event can **always** be added.

Your class will have one method, book(int start, int end). Formally, this represents a booking on the half open interval [start, end), the range of real numbers x such that start <= x < end.

A *K-booking* happens when **K** events have some non-empty intersection (ie., there is some time that is common to all K events.)

For each call to the method MyCalendar.book, return an integer K representing the largest integer such that there exists a K-booking in the calendar.

Your class will be called like this: MyCalendarThree cal = new MyCalendarThree(); MyCalendarThree.book(start, end)

**Example 1:**

MyCalendarThree();

MyCalendarThree.book(10, 20); // returns 1

MyCalendarThree.book(50, 60); // returns 1

MyCalendarThree.book(10, 40); // returns 2

MyCalendarThree.book(5, 15); // returns 3

MyCalendarThree.book(5, 10); // returns 3

MyCalendarThree.book(25, 55); // returns 3

**Explanation:**

The first two events can be booked and are disjoint, so the maximum K-booking is a 1-booking.

The third event [10, 40) intersects the first event, and the maximum K-booking is a 2-booking.

The remaining events cause the maximum K-booking to be only a 3-booking.

Note that the last event locally causes a 2-booking, but the answer is still 3 because

eg. [10, 20), [10, 40), and [5, 15) are still triple booked.

**Note:**

* The number of calls to MyCalendarThree.book per test case will be at most 400.
* In calls to MyCalendarThree.book(start, end), start and end are integers in the range [0, 10^9].

class MyCalendarThree {

public:

    MyCalendarThree() {}

    map<int, int> mp;

   int book(int start, int end) {

        ++mp[start];

        --mp[end];

        int res = 1, cnt = 0;

        for (auto it : mp) res = max(res, cnt += it.second);

        return res;

    }

};

### 741. Cherry Pickup

Hard

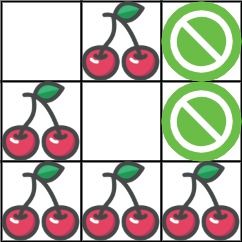
You are given an n x n grid representing a field of cherries, each cell is one of three possible integers.

* 0 means the cell is empty, so you can pass through,
* 1 means the cell contains a cherry that you can pick up and pass through, or
* -1 means the cell contains a thorn that blocks your way.

Return *the maximum number of cherries you can collect by following the rules below*:

* Starting at the position (0, 0) and reaching (n - 1, n - 1) by moving right or down through valid path cells (cells with value 0 or 1).
* After reaching (n - 1, n - 1), returning to (0, 0) by moving left or up through valid path cells.
* When passing through a path cell containing a cherry, you pick it up, and the cell becomes an empty cell 0.
* If there is no valid path between (0, 0) and (n - 1, n - 1), then no cherries can be collected.

**Example 1:**



**Input:** grid = [[0,1,-1],[1,0,-1],[1,1,1]]

**Output:** 5

**Explanation:** The player started at (0, 0) and went down, down, right right to reach (2, 2).

4 cherries were picked up during this single trip, and the matrix becomes [[0,1,-1],[0,0,-1],[0,0,0]].

Then, the player went left, up, up, left to return home, picking up one more cherry.

The total number of cherries picked up is 5, and this is the maximum possible.

**Example 2:**

**Input:** grid = [[1,1,-1],[1,-1,1],[-1,1,1]]

**Output:** 0

**Constraints:**

* n == grid.length
* n == grid[i].length
* 1 <= n <= 50
* grid[i][j] is -1, 0, or 1.
* grid[0][0] != -1
* grid[n - 1][n - 1] != -1

class Solution {

public:

    int cherryPickup(vector<vector<int>>& grid) {

        int n = grid.size(), p = 0;

        vector<vector<vector<int>>> dp(2, vector<vector<int>> (n,

vector<int>(n, -1)));

        dp[0][0][0] = grid[0][0];

        const int maxK = 2 \* (n - 1);

        for (int k = 1; k <= maxK; k++) {

            auto &pre = dp[p], &cur = dp[p^1];

            for (auto &v : cur) for (auto &i : v) i = -1;

            for (int i = 0; i < n && i <= k; i++) {

                if (k - i >= n || grid[i][k-i] < 0) continue;

                for (int j = 0; j < n && j <= k; j++) {

                    if (k - j >= n || grid[j][k - j] < 0) continue;

                    int cherries = pre[i][j];

                    if (i > 0) cherries = max(cherries, pre[i - 1][j]);

                    if (j > 0) cherries = max(cherries, pre[i][j - 1]);

                    if (i>0&&j>0) cherries=max(cherries,pre[i-1][j-1]);

                    if (cherries < 0) continue;

                    cherries += grid[i][k - i] + (i==j?0:grid[j][k-j]);

                    cur[i][j] = cherries;

                }

            }

            p ^= 1;

        }

        return max(dp[p][n-1][n-1], 0);

    }

};

### 745. Prefix and Suffix Search★★

Hard

Design a special dictionary which has some words and allows you to search the words in it by a prefix and a suffix.

Implement the WordFilter class:

* WordFilter(string[] words) Initializes the object with the words in the dictionary.
* f(string prefix, string suffix) Returns *the index of the word in the dictionary* which has the prefix prefix and the suffix suffix. If there is more than one valid index, return **the largest** of them. If there is no such word in the dictionary, return -1.

**Example 1:**

**Input**

["WordFilter", "f"]

[[["apple"]], ["a", "e"]]

**Output**

[null, 0]

**Explanation**

WordFilter wordFilter = new WordFilter(["apple"]);

wordFilter.f("a", "e"); // return 0, because the word at index 0 has prefix = "a" and suffix = 'e".

**Constraints:**

* 1 <= words.length <= 15000
* 1 <= words[i].length <= 10
* 1 <= prefix.length, suffix.length <= 10
* words[i], prefix and suffix consist of lower-case English letters only.
* At most 15000 calls will be made to the function f.

class Trie {

public:

    void insert(const string &s, int id) {

        auto p = this;

        for (auto c : s) {

            if (!p->mp.count(c)) {

                p->mp[c] = new Trie();

            }

            p->id = id;

            p = p->mp[c];

        }

        p->id = id;

    }

    int search(const string &s) {

        auto p = this;

        for (auto c : s) {

            if (!p->mp.count(c)) {

                return -1;

            }

            p = p->mp[c];

        }

        return p->id;

    }

    int id = -1;

    unordered\_map<char, Trie\*> mp;

};

class WordFilter {

public:

    Trie\* trie = new Trie();

    WordFilter(vector<string>& words) {

        for (int i = 0; i < words.size(); ++i) {

            for (int j = 0; j <= words[i].size(); ++j) {

                string t = words[i].substr(j) + "#" + words[i];

                trie->insert(t, i);

            }

        }

    }

    int f(string prefix, string suffix) {

        return trie->search(suffix + "#" + prefix);

    }

};

### 757. Set Intersection Size At Least Two

Hard

An integer interval [a, b] (for integers a < b) is a set of all consecutive integers from a to b, including a and b.

Find the minimum size of a set S such that for every integer interval A in intervals, the intersection of S with A has a size of at least two.

**Example 1:**

**Input:** intervals = [[1,3],[1,4],[2,5],[3,5]]

**Output:** 3

**Explanation:** Consider the set S = {2, 3, 4}. For each interval, there are at least 2 elements from S in the interval.

Also, there isn't a smaller size set that fulfills the above condition.

Thus, we output the size of this set, which is 3.

**Example 2:**

**Input:** intervals = [[1,2],[2,3],[2,4],[4,5]]

**Output:** 5

**Explanation:** An example of a minimum sized set is {1, 2, 3, 4, 5}.

**Constraints:**

* 1 <= intervals.length <= 3000
* intervals[i].length == 2
* 0 <= ai < bi <= 108

class Solution {

public:

    int intersectionSizeTwo(vector<vector<int>>& intervals) {

        auto cmp = [](vector<int>& a, vector<int>& b) {

            return a[1] < b[1] || (a[1] == b[1] && a[0] > b[0]);

        };

        sort(intervals.begin(), intervals.end(), cmp);

        int n = intervals.size(), res = 0, p1 = -1, p2 = -1;

        for (const auto &v : intervals) {

            if (v[0] > p2) {

                res += 2;

                p2 = v[1];

                p1 = p2-1;

            }

            else if (v[0] > p1) {

                res++;

                p1 = p2;

                p2 = v[1];

            }

        }

        return res;

    }

};

### 761. Special Binary String

Hard

*Special* binary strings are binary strings with the following two properties:

 The number of 0's is equal to the number of 1's.

 Every prefix of the binary string has at least as many 1's as 0's.

Given a special string S, a *move* consists of choosing two consecutive, non-empty, special substrings of S, and swapping them. *(Two strings are consecutive if the last character of the first string is exactly one index before the first character of the second string.)*

At the end of any number of moves, what is the lexicographically largest resulting string possible?

**Example 1:**

**Input:** S = "11011000"

**Output:** "11100100"

**Explanation:**

The strings "10" [occuring at S[1]] and "1100" [at S[3]] are swapped.

This is the lexicographically largest string possible after some number of swaps.

**Note:**

1. S has length at most 50.
2. S is guaranteed to be a *special* binary string as defined above.

class Solution {

public:

    string makeLargestSpecial(string S) {

        vector<string> v;

        for (int j = 0, i = 0, cnt = 0; j < S.size(); ++j) {

            S[j] == '1' ? ++cnt : --cnt;

            if (cnt < 0) return S;

            if (cnt == 0) {

string t = S.substr(i + 1, j - i - 1);

                v.push\_back('1' + makeLargestSpecial(move(t)) + '0');

                i = j + 1;

            }

        }

        sort(v.begin(), v.end(), greater<>());

        string res;

        for (auto &s : v) res += s;

        return res;

    }

};

### 765. Couples Holding Hands

Hard

N couples sit in 2N seats arranged in a row and want to hold hands. We want to know the minimum number of swaps so that every couple is sitting side by side. A *swap* consists of choosing **any** two people, then they stand up and switch seats.

The people and seats are represented by an integer from 0 to 2N-1, the couples are numbered in order, the first couple being (0, 1), the second couple being (2, 3), and so on with the last couple being (2N-2, 2N-1).

The couples' initial seating is given by row[i] being the value of the person who is initially sitting in the i-th seat.

**Example 1:**

**Input:** row = [0, 2, 1, 3]

**Output:** 1

**Explanation:** We only need to swap the second (row[1]) and third (row[2]) person.

**Example 2:**

**Input:** row = [3, 2, 0, 1]

**Output:** 0

**Explanation:** All couples are already seated side by side.

**Note:**

1. len(row) is even and in the range of [4, 60].
2. row is guaranteed to be a permutation of 0...len(row)-1.

class Solution {

public:

    int minSwapsCouples(vector<int>& row) {

        int n = row.size(), res = 0;

        vector<int> fa(n, -1);

        for (int i = 0; i < n; i += 2) {

            Union(i, i+1, fa);

            Union(row[i], row[i+1], fa);

        }

        for (int i = 0; i < n; ++i) {

            if (find(i, fa) == i) {

                res -= fa[i]/2 + 1;

            }

        }

        return res;

    }

    void Union(int x, int y, vector<int> &fa) {

        int fax = find(x, fa);

        int fay = find(y, fa);

        if (fax != fay) {

            fa[fay] += fa[fax];

            fa[fax] = fay;

        }

    }

    int find(int x, vector<int> &fa) {

        return fa[x] < 0 ? x : (fa[x] = find(fa[x], fa));

    }

};

### 768. Max Chunks To Make Sorted II★★

Hard

*This question is the same as "Max Chunks to Make Sorted" except the integers of the given array are not necessarily distinct, the input array could be up to length 2000, and the elements could be up to 10\*\*8.*

Given an array arr of integers (**not necessarily distinct**), we split the array into some number of "chunks" (partitions), and individually sort each chunk.  After concatenating them, the result equals the sorted array.

What is the most number of chunks we could have made?

**Example 1:**

**Input:** arr = [5,4,3,2,1]

**Output:** 1

**Explanation:**

Splitting into two or more chunks will not return the required result.

For example, splitting into [5, 4], [3, 2, 1] will result in [4, 5, 1, 2, 3], which isn't sorted.

**Example 2:**

**Input:** arr = [2,1,3,4,4]

**Output:** 4

**Explanation:**

We can split into two chunks, such as [2, 1], [3, 4, 4].

However, splitting into [2, 1], [3], [4], [4] is the highest number of chunks possible.

**Note:**

* arr will have length in range [1, 2000].
* arr[i] will be an integer in range [0, 10\*\*8].

//O(nlogn)

class Solution {

public:

    int maxChunksToSorted(vector<int>& arr) {

        auto sort\_arr(arr);

        sort(sort\_arr.begin(), sort\_arr.end());

        int n = arr.size(), res = 0, nonzero = 0;

        unordered\_map<int, int> cnt;

        for (int i = 0; i < n; ++i) {

            ++cnt[arr[i]];

            if (cnt[arr[i]] == 0) --nonzero;

            else if (cnt[arr[i]] == 1) ++nonzero;

            --cnt[sort\_arr[i]];

            if (cnt[sort\_arr[i]] == 0) --nonzero;

            else if (cnt[sort\_arr[i]] == -1) ++nonzero;

            if (nonzero == 0) ++res;

        }

        return res;

    }

};

//O(n)

class Solution {

public:

    int maxChunksToSorted(vector<int>& arr) {

        stack<int> stk;

        for (auto &x : arr) {

            if (stk.empty() || x >= stk.top()) stk.push(x);

            else {

                auto top = stk.top();

                s.pop();

                while (!stk.empty() && x < stk.top()) {

                    stk.pop();

                }

                stk.push(top);

            }

        }

        return stk.size();

    }

};

//O(nlogn)

class Solution {

public:

    int maxChunksToSorted(vector<int>& arr) {

        auto sort\_arr(arr);

        sort(sort\_arr.begin(), sort\_arr.end());

        int n = arr.size(), res = 0;

        long long sum0 = 0, sum1 = 0;

        for (int i = 0; i < n; ++i) {

            sum0 += arr[i];

            sum1 += sort\_arr[i];

            if (sum0 == sum1) ++res;

        }

        return res;

    }

};

### 773. Sliding Puzzle

Hard

On a 2x3 board, there are 5 tiles represented by the integers 1 through 5, and an empty square represented by 0.

A move consists of choosing 0 and a 4-directionally adjacent number and swapping it.

The state of the board is *solved* if and only if the board is [[1,2,3],[4,5,0]].

Given a puzzle board, return the least number of moves required so that the state of the board is solved. If it is impossible for the state of the board to be solved, return -1.

**Examples:**

**Input:** board = [[1,2,3],[4,0,5]]

**Output:** 1

**Explanation:** Swap the 0 and the 5 in one move.

**Input:** board = [[1,2,3],[5,4,0]]

**Output:** -1

**Explanation:** No number of moves will make the board solved.

**Input:** board = [[4,1,2],[5,0,3]]

**Output:** 5

**Input:** board = [[3,2,4],[1,5,0]]

**Output:** 14

**Note:**

* board will be a 2 x 3 array as described above.
* board[i][j] will be a permutation of [0, 1, 2, 3, 4, 5].

class Solution {

public:

    const vector<vector<int>> dirs {

        {1,3}, {0,2,4}, {1,5},

        {0,4}, {1,3,5}, {2,4}

    };

    string vecToStr(vector<vector<int>> board) {

        string s;

        for (auto &v : board) {

            for (auto i : v) {

                s += char('0' + i);

            }

        }

        return s;

    }

    int slidingPuzzle(vector<vector<int>>& board) {

        auto start = vecToStr(board);

        auto goal = vecToStr({{1,2,3}, {4,5,0}});

        if (start == goal) return 0;

        set<string> st{start};

        queue<string> q;

        q.push(start);

        int cnt = 0;

        while (!q.empty()) {

            ++cnt;

            int sz = q.size();

            while (sz--) {

                auto u = q.front();

                q.pop();

                int pos = 0;

                while (u[pos] != '0') ++pos;

                for (auto newPos : dirs[pos]) {

                    auto t = u;

                    swap(t[newPos], t[pos]);

                    if (t == goal) return cnt;

                    if (st.count(t)) continue;

                    st.insert(t);

                    q.push(t);

                }

            }

        }

        return -1;

    }

};

### 778. Swim in Rising Water

Hard

On an N x N grid, each square grid[i][j] represents the elevation at that point (i,j).

Now rain starts to fall. At time t, the depth of the water everywhere is t. You can swim from a square to another 4-directionally adjacent square if and only if the elevation of both squares individually are at most t. You can swim infinite distance in zero time. Of course, you must stay within the boundaries of the grid during your swim.

You start at the top left square (0, 0). What is the least time until you can reach the bottom right square (N-1, N-1)?

**Example 1:**

**Input:** [[0,2],[1,3]]

**Output:** 3

**Explanation:**

At time 0, you are in grid location (0, 0).

You cannot go anywhere else because 4-directionally adjacent neighbors have a higher elevation than t = 0.

You cannot reach point (1, 1) until time 3.

When the depth of water is 3, we can swim anywhere inside the grid.

**Example 2:**

**Input:** [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,7,6]]

**Output:** 16

**Explanation:**

**0 1 2 3 4**

24 23 22 21 **5**

**12 13 14 15 16**

**11** 17 18 19 20

**10 9 8 7 6**

The final route is marked in bold.

We need to wait until time 16 so that (0, 0) and (4, 4) are connected.

**Note:**

1. 2 <= N <= 50.
2. grid[i][j] is a permutation of [0, ..., N\*N - 1].

class Solution {

public:

    using pii = pair<int, int>;

    const vector<int> dirs{0, 1, 0, -1, 0};

    int swimInWater(vector<vector<int>>& grid) {

        priority\_queue<pii, vector<pii>, greater<>> pq;

        int n = grid.size(), m = grid[0].size();

        for (int i = 0; i < n; ++i) {

            for (int j = 0; j < m; ++j) {

                pq.emplace(grid[i][j], i\*m+j);

            }

        }

        vector<int> fa(n\*m, -1);

        while (!pq.empty()) {

            auto t = pq.top();

            pq.pop();

            int x = t.second / m, y = t.second % m;

            for (int i = 0; i < 4; ++i) {

                int xx = x + dirs[i], yy = y + dirs[i+1];

                if (xx < 0 || yy < 0 || xx >= n || yy >= m) continue;

                if (grid[x][y] > grid[xx][yy]) {

                    Union(t.second, xx \* m + yy, fa);

                    if (find(0, fa) == find(n\*m-1, fa)) return t.first;

                }

            }

        }

        return -1;

    }

    void Union(int x, int y, vector<int> &fa) {

        int fax = find(x, fa);

        int fay = find(y, fa);

        if (fax != fay) {

            fa[fay] += fa[fax];

            fa[fax] = fay;

        }

    }

    int find(int x, vector<int> &fa) {

        return fa[x] < 0 ? x : (fa[x] = find(fa[x], fa));

    }

};

### 786. K-th Smallest Prime Fraction★★

Hard

A sorted list A contains 1, plus some number of primes.  Then, for every p < q in the list, we consider the fraction p/q.

What is the K-th smallest fraction considered?  Return your answer as an array of ints, where answer[0] = p and answer[1] = q.

**Examples:**

**Input:** A = [1, 2, 3, 5], K = 3

**Output:** [2, 5]

**Explanation:**

The fractions to be considered in sorted order are:

1/5, 1/3, 2/5, 1/2, 3/5, 2/3.

The third fraction is 2/5.

**Input:** A = [1, 7], K = 1

**Output:** [1, 7]

**Note:**

* A will have length between 2 and 2000.
* Each A[i] will be between 1 and 30000.
* K will be between 1 and A.length \* (A.length - 1) / 2.

class Solution {

public:

    vector<int> kthSmallestPrimeFraction(vector<int>& A, int K) {

        double left = 0, right = 1.0;

        int n = A.size();

        while (left < right) {

            double mid = (right + left) / 2;

            int cnt = 0;

            vector<int> res{0, 1};

            for(int i = 0, j = 0; i < n; i++) {

                while (j < n && A[i] >= mid\*A[j]) ++j;

                cnt += n - j;

                if (j < n && res[0]\*A[j] < A[i]\*res[1])

res = {A[i], A[j]};

            }

            if (cnt == K)  return res;

            else if (cnt < K)   left = mid;

            else right = mid;

        }

        return {};

    }

};

### 793. Preimage Size of Factorial Zeroes Function★

Hard

Let f(x) be the number of zeroes at the end of x!. (Recall that x! = 1 \* 2 \* 3 \* ... \* x, and by convention, 0! = 1.)

For example, f(3) = 0 because 3! = 6 has no zeroes at the end, while f(11) = 2 because 11! = 39916800 has 2 zeroes at the end. Given K, find how many non-negative integers x have the property that f(x) = K.

**Example 1:**

**Input:** K = 0

**Output:** 5

**Explanation:** 0!, 1!, 2!, 3!, and 4! end with K = 0 zeroes.

**Example 2:**

**Input:** K = 5

**Output:** 0

**Explanation:** There is no x such that x! ends in K = 5 zeroes.

**Note:**

* K will be an integer in the range [0, 10^9].

class Solution {

public:

    using ll = long long;

    ll check(ll n) {

        ll res = 0;

        while (n) {

            res += (n /= 5);

        }

        return res;

    }

    int preimageSizeFZF(int K) {

        int a = low(K);

        if (a == -1) return 0;

        int b = upper(K);

        return b - a;

    }

    ll low(int K) {

        ll l = 0, r = LLONG\_MAX;

        while (l < r) {

            ll mid = l + (r-l) / 2;

            if (check(mid) < K) l = mid+1;

            else r = mid;

        }

        return check(l) == K ? l : -1;

    }

    ll upper(int K) {

        ll l = 0, r = LLONG\_MAX;

        while (l < r) {

            ll mid = l + (r-l) / 2;

            if (check(mid) <= K) l = mid+1;

            else r = mid;

        }

        return l;

    }

};

### 798. Smallest Rotation with Highest Score

Hard

 Given an array A, we may rotate it by a non-negative integer K so that the array becomes A[K], A[K+1], A{K+2], ... A[A.length - 1], A[0], A[1], ..., A[K-1].  Afterward, any entries that are less than or equal to their index are worth 1 point.

For example, if we have [2, 4, 1, 3, 0], and we rotate by K = 2, it becomes [1, 3, 0, 2, 4].  This is worth 3 points because 1 > 0 [no points], 3 > 1 [no points], 0 <= 2 [one point], 2 <= 3 [one point], 4 <= 4 [one point].

Over all possible rotations, return the rotation index K that corresponds to the highest score we could receive.  If there are multiple answers, return the smallest such index K.

**Example 1:**

**Input:** [2, 3, 1, 4, 0]

**Output:** 3

**Explanation:**

Scores for each K are listed below:

K = 0, A = [2,3,1,4,0], score 2

K = 1, A = [3,1,4,0,2], score 3

K = 2, A = [1,4,0,2,3], score 3

K = 3, A = [4,0,2,3,1], score 4

K = 4, A = [0,2,3,1,4], score 3

So we should choose K = 3, which has the highest score.

**Example 2:**

**Input:** [1, 3, 0, 2, 4]

**Output:** 0

**Note:**

* A will have length at most 20000.
* A[i] will be in the range [0, A.length].

class Solution {

public:

    int bestRotation(vector<int>& A) {

        int N = A.size(), res = 0;

        vector<int> mark(N, 0);

        for (int i = 0; i < N; ++i) {

            if (A[i] >= N) continue;

            int k\_in = (i + 1) % N;

            int k\_out = (i + 1 + N - A[i]) % N;

            if (i > A[i]) {

                ++mark[0];      --mark[k\_out];

                ++mark[k\_in];   //--mark[N];

            }

            else {

                ++mark[k\_in];

                --mark[k\_out];

            }

        }

        int score = 0, max\_score = -1;

        for (int i = 0; i < N; ++i) {

            score += mark[i];

            if (score > max\_score) {

                res = i;

                max\_score = score;

            }

        }

        return res;

    }

};

### 803. Bricks Falling When Hit★★

Hard

You are given an m x n binary grid, where each 1 represents a brick and 0 represents an empty space. A brick is **stable** if:

* It is directly connected to the top of the grid, or
* At least one other brick in its four adjacent cells is **stable**.

You are also given an array hits, which is a sequence of erasures we want to apply. Each time we want to erase the brick at the location hits[i] = (rowi, coli). The brick on that location (if it exists) will disappear. Some other bricks may no longer be stable because of that erasure and will **fall**. Once a brick falls, it is **immediately** erased from the grid (i.e., it does not land on other stable bricks).

Return *an array*result*, where each*result[i]*is the number of bricks that will****fall****after the*ith*erasure is applied.*

**Note** that an erasure may refer to a location with no brick, and if it does, no bricks drop.

**Example 1:**

**Input:** grid = [[1,0,0,0],[1,1,1,0]], hits = [[1,0]]

**Output:** [2]

**Explanation:** Starting with the grid:

[[1,0,0,0],

[1,1,1,0]]

We erase the underlined brick at (1,0), resulting in the grid:

[[1,0,0,0],

[0,1,1,0]]

The two underlined bricks are no longer stable as they are no longer connected to the top nor adjacent to another stable brick, so they will fall. The resulting grid is:

[[1,0,0,0],

[0,0,0,0]]

Hence the result is [2].

**Example 2:**

**Input:** grid = [[1,0,0,0],[1,1,0,0]], hits = [[1,1],[1,0]]

**Output:** [0,0]

**Explanation:** Starting with the grid:

[[1,0,0,0],

[1,1,0,0]]

We erase the underlined brick at (1,1), resulting in the grid:

[[1,0,0,0],

[1,0,0,0]]

All remaining bricks are still stable, so no bricks fall. The grid remains the same:

[[1,0,0,0],

[1,0,0,0]]

Next, we erase the underlined brick at (1,0), resulting in the grid:

[[1,0,0,0],

[0,0,0,0]]

Once again, all remaining bricks are still stable, so no bricks fall.

Hence the result is [0,0].

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 200
* grid[i][j] is 0 or 1.
* 1 <= hits.length <= 4 \* 104
* hits[i].length == 2
* 0 <= xi<= m - 1
* 0 <= yi <= n - 1
* All (xi, yi) are unique.

class Solution {

public:

    const vector<int> dirs{0,1,0,-1,0};

    vector<int> hitBricks(vector<vector<int>>& grid, vector<vector<int>>& hits) {

        vector<int> res(hits.size());

        for (int i = hits.size()-1; i >= 0; --i) {

            if (!grid[hits[i][0]][hits[i][1]]) hits[i][0] = -1;

            else grid[hits[i][0]][hits[i][1]] = 0;

        }

        int n = grid.size(), m = grid[0].size();

        vector<int> fa(n\*m+1, -1);

        for (int i = 0; i < n; ++i) {

            for (int j = 0; j < m; ++j) {

                if (grid[i][j]) {

                    if (i == 0) Union(i\*m+j, n\*m, fa);

                    for (int k = 0; k < 4; ++k) {

                        int x = i + dirs[k], y = j + dirs[k+1];

                        if (x<0 || y<0 || x>=n || y>=m) continue;

                        if (!grid[x][y]) continue;

                        Union(i\*m+j, x\*m+y, fa);

                    }

                }

            }

        }

        for (int i = hits.size()-1; i >= 0; --i) {

            if (hits[i][0] == -1) continue;

            int cnt = 0, pos = hits[i][0]\*m+hits[i][1];

            bool ok = false;

            grid[hits[i][0]][hits[i][1]] = 1;

            if (hits[i][0] == 0) {

                Union(pos, n\*m, fa);

                ok = true;

            }

            set<int> st;

            for (int k = 0; k < 4; ++k) {

                int x = hits[i][0] + dirs[k];

                int y = hits[i][1] + dirs[k+1];

                if (x < 0 || y < 0 || x >= n || y >= m) continue;

                if (!grid[x][y]) continue;

                st.insert(find(x\*m+y, fa));

            }

            for (auto id : st) {

                if (find(id, fa) != n\*m) {

                    cnt -= fa[find(id, fa)];

                }

                else ok = true;

                Union(pos, id, fa);

            }

            if (ok) res[i] = cnt;

        }

        return res;

    }

    void Union(int x, int y, vector<int> &fa) {

        int fax = find(x, fa), fay = find(y, fa);

        if (fax != fay) {

            fa[max(fax, fay)] += fa[min(fax, fay)];

            fa[min(fax, fay)] = max(fax, fay);

        }

    }

    int find(int x, vector<int> &fa) {

        return fa[x] < 0 ? x : (fa[x] = find(fa[x], fa));

    }

};

### 805. Split Array With Same Average★★

Hard

In a given integer array A, we must move every element of A to either list B or list C. (B and C initially start empty.)

Return true if and only if after such a move, it is possible that the average value of B is equal to the average value of C, and B and C are both non-empty.

**Example :**

**Input:**

[1,2,3,4,5,6,7,8]

**Output:** true

**Explanation:** We can split the array into [1,4,5,8] and [2,3,6,7], and both of them have the average of 4.5.

**Note:**

* The length of A will be in the range [1, 30].
* A[i] will be in the range of [0, 10000].

class Solution {

public:

    unordered\_map<int, int> f(vector<int> &v) {

        int n = v.size();

        unordered\_map<int, int> mp;

        for (int i = (1 << n) -1; i >= 0; --i) {

            int sum = 0;

            for (int j = 0; j < n; ++j) {

                if (i & (1<<j)) sum += v[j];

            }

            ++mp[sum];

        }

        return mp;

    }

    bool splitArraySameAverage(vector<int>& A) {

        if (A.size() <= 1) return false;

        int sum = accumulate(A.begin(), A.end(), 0);

        int N = A.size(), cnt = 1;

        for(int& x : A) x = N\*x - sum;

        if (accumulate(A.begin(), A.end(), 0) == 0) ++cnt;

        vector<int> v0{A.begin(), A.begin() + N/2};

vector<int> v1{A.begin() + N/2, A.end()};

        auto mp0 = f(v0), mp1 = f(v1);

        for (auto [i, j] : mp0) {

            if (mp1.count(-i)) {

                if ((cnt -= j\*mp1[-i]) < 0) {

                    return true;

                }

            }

        }

        return false;

    }

};

### 815. Bus Routes

Hard

We have a list of bus routes. Each routes[i] is a bus route that the i-th bus repeats forever. For example if routes[0] = [1, 5, 7], this means that the first bus (0-th indexed) travels in the sequence 1->5->7->1->5->7->1->... forever.

We start at bus stop S (initially not on a bus), and we want to go to bus stop T. Travelling by buses only, what is the least number of buses we must take to reach our destination? Return -1 if it is not possible.

**Example:**

**Input:**

routes = [[1, 2, 7], [3, 6, 7]]

S = 1

T = 6

**Output:** 2

**Explanation:**

The best strategy is take the first bus to the bus stop 7, then take the second bus to the bus stop 6.

**Constraints:**

* 1 <= routes.length <= 500.
* 1 <= routes[i].length <= 10^5.
* 0 <= routes[i][j] < 10 ^ 6.

class Solution {

public:

    int numBusesToDestination(vector<vector<int>>&routes,int S,int T) {

        if (S == T) return 0;

        unordered\_map<int, vector<int>> mp;

        for (int i = 0; i < routes.size(); ++i)

            for (int j : routes[i])

                mp[j].push\_back(i);

        queue<int> q;

        q.emplace(S);

        unordered\_set<int> vis{S};

        int cnt = 0;

        while (!q.empty()) {

            ++cnt;

            int sz = q.size();

            while (sz--) {

                auto stop = q.front();

                q.pop();

                for (int i : mp[stop]) {

                    for (int j : routes[i]) {

                        if (j == T) return cnt;

                        if (vis.find(j) == vis.end()) {

                            vis.insert(j);

                            q.emplace(j);

                        }

                    }

                    routes[i].clear();

                }

            }

        }

        return -1;

    }

};

### 818. Race Car

Hard

Your car starts at position 0 and speed +1 on an infinite number line.  (Your car can go into negative positions.)

Your car drives automatically according to a sequence of instructions A (accelerate) and R (reverse).

When you get an instruction "A", your car does the following: position += speed, speed \*= 2.

When you get an instruction "R", your car does the following: if your speed is positive then speed = -1 , otherwise speed = 1.  (Your position stays the same.)

For example, after commands "AAR", your car goes to positions 0->1->3->3, and your speed goes to 1->2->4->-1.

Now for some target position, say the **length** of the shortest sequence of instructions to get there.

**Example 1:**

**Input:** target = 3

**Output:** 2

**Explanation:**

The shortest instruction sequence is "AA".

Your position goes from 0->1->3.

**Example 2:**

**Input:** target = 6

**Output:** 5

**Explanation:**

The shortest instruction sequence is "AAARA".

Your position goes from 0->1->3->7->7->6.

**Note:**

* 1 <= target <= 10000.

class Solution {

public:

    int dp[10001];

    int racecar(int t) {

        if (dp[t] > 0) return dp[t];

        int n = ceil(log2(t+1)), res;

        if (1<<n == t + 1) dp[t] = n;

        else {

            //超过再返回

            dp[t] = racecar((1 << n) - 1 - t) + n + 1;

            //没超过，折返加速m次，再掉头

            for (int m = 0; m < n - 1; ++m)

                dp[t] = min(dp[t], racecar(t - (1 << (n - 1))

+ (1 << m)) + n + m + 1);

        }

        return dp[t];

    }

};

class Solution {

public:

    using pii = pair<int, int>;

    int racecar(int T) {

        int depth = 0;

        auto myhash = [](const pii& lhs)->size\_t {

            return hash<int>()(lhs.first) ^ hash<int>()(lhs.second);

        };

        unordered\_set<pii, decltype(myhash)> vis(20, myhash);

        queue<pii> q{{{0,1}}};

        vis.emplace(0, 1);

        while(1) {

            int sz = q.size();

            while (sz--) {

                auto [pos, vel] = q.front();

                q.pop();

                if (pos == T) return depth;

                vector<pii> cand;

                if (abs(T - (pos + vel)) < T)

                    cand.emplace\_back(pos + vel, 2 \* vel);

                cand.emplace\_back(pos, vel < 0 ? 1 : -1);

                for (auto [pos, vel]: cand)

                    if (!vis.count({pos, vel})) {

                        q.emplace(pos, vel);

                        vis.emplace(pos,vel);

                    }

            }

            ++depth;

        }

        return -1;

    }

};

### 827. Making A Large Island

Hard

In a 2D grid of 0s and 1s, we change at most one 0 to a 1.

After, what is the size of the largest island? (An island is a 4-directionally connected group of 1s).

**Example 1:**

**Input:** [[1, 0], [0, 1]]

**Output:** 3

**Explanation:** Change one 0 to 1 and connect two 1s, then we get an island with area = 3.

**Example 2:**

**Input:** [[1, 1], [1, 0]]

**Output:** 4

**Explanation:** Change the 0 to 1 and make the island bigger, only one island with area = 4.

**Example 3:**

**Input:** [[1, 1], [1, 1]]

**Output:** 4

**Explanation:** Can't change any 0 to 1, only one island with area = 4.

Notes:

* 1 <= grid.length = grid[0].length <= 50.
* 0 <= grid[i][j] <= 1.

class Solution {

public:

    int largestIsland(vector<vector<int>>& grid) {

        int n = grid.size(), m = grid[0].size(), index = 2;

        unordered\_map<int, int> mp;

        int res = 1;

        for (int i = 0; i < n; ++i) {

            for (int j = 0; j < m; ++j) {

                if (grid[i][j] == 1) {

                    mp[index] = dfs(i, j, n, m, index, grid);

                    res = max(mp[index++], res);

                }

            }

        }

        for (int i = 0; i < n; ++i) {

            for (int j = 0; j < m; ++j) if (!grid[i][j]) {

                set<int> st;

                for (auto [dx, dy] : dir) {

                    int x = i + dx, y = j + dy;

                    if (x < 0 || y < 0 || x >= n || y >= m) continue;

                    if (!grid[x][y]) continue;

                    st.insert(grid[x][y]);

                }

                int sum = 1;

                for (auto i : st) sum += mp[i];

                res = max(res, sum);

            }

        }

        return res;

    }

private:

    const vector<pair<int, int>> dir{{0, 1},{1, 0},{-1, 0},{0,-1}};

   int dfs(int i,int j,int n,int m,int index,vector<vector<int>>&grid){

        int ret = 1;

        grid[i][j] = index;

        for (auto [dx, dy] : dir) {

            int x = i + dx, y = j + dy;

            if (x >= 0 && y >= 0 && x < n && y < m && grid[x][y] == 1){

                ret += dfs(x, y, n, m, index, grid);

            }

        }

        return ret;

    }

};

### 828. Count Unique Characters of All Substrings of a Given String★★

Hard

Let's define a function countUniqueChars(s) that returns the number of unique characters on s, for example if s = "LEETCODE" then "L", "T","C","O","D" are the unique characters since they appear only once in s, therefore countUniqueChars(s) = 5.  
  
On this problem given a string s we need to return the sum of countUniqueChars(t) where t is a substring of s. Notice that some substrings can be repeated so on this case you have to count the repeated ones too.

Since the answer can be very large, return the answer modulo 10 ^ 9 + 7.

**Example 1:**

**Input:** s = "ABC"

**Output:** 10

**Explanation:** All possible substrings are: "A","B","C","AB","BC" and "ABC".

Evey substring is composed with only unique letters.

Sum of lengths of all substring is 1 + 1 + 1 + 2 + 2 + 3 = 10

**Example 2:**

**Input:** s = "ABA"

**Output:** 8

**Explanation:** The same as example 1, except countUniqueChars("ABA") = 1.

**Example 3:**

**Input:** s = "LEETCODE"

**Output:** 92

**Constraints:**

* 0 <= s.length <= 10^4
* s contain upper-case English letters only.

class Solution {

public:

    int uniqueLetterString(string s) {

        int n = s.size();

        vector<vector<int>> v(26, vector<int> {-1});

        for (int i = 0; i < n; ++i) {

            v[s[i]-'A'].push\_back(i);

        }

        long long res = 0, mod = 1e9 + 7;

        for (auto &t : v) {

            t.push\_back(n);

            for (int i = 1; i < t.size()-1; ++i) {

                res = (res + 1ll\*(t[i]-t[i-1])\*(t[i+1]-t[i])) % mod;

            }

        }

        return static\_cast<int> (res);

    }

};

### 834. Sum of Distances in Tree★★

Hard

An undirected, connected tree with N nodes labelled 0...N-1 and N-1 edges are given.

The ith edge connects nodes edges[i][0] and edges[i][1] together.

Return a list ans, where ans[i] is the sum of the distances between node i and all other nodes.

**Example 1:**

**Input:** N = 6, edges = [[0,1],[0,2],[2,3],[2,4],[2,5]]

**Output:** [8,12,6,10,10,10]

**Explanation:**

Here is a diagram of the given tree:

0

/ \

1 2

/|\

3 4 5

We can see that dist(0,1) + dist(0,2) + dist(0,3) + dist(0,4) + dist(0,5)

equals 1 + 1 + 2 + 2 + 2 = 8. Hence, answer[0] = 8, and so on.

Note: 1 <= N <= 10000

class Solution {

public:

    vector<unordered\_set<int>> tree;

    vector<int> res, count;

    vector<int> sumOfDistancesInTree(int N,vector<vector<int>>& edges){

        tree.resize(N);

        res.assign(N, 0);

        count.assign(N, 1);

        for (auto e : edges) {

            tree[e[0]].insert(e[1]);

            tree[e[1]].insert(e[0]);

        }

        dfs(0, -1);

        dfs2(0, -1);

        return res;

    }

    void dfs(int root, int pre) {

        for (auto i : tree[root]) {

            if (i == pre) continue;

            dfs(i, root);

            count[root] += count[i];

            res[root] += res[i] + count[i];

        }

    }

    void dfs2(int root, int pre) {

        for (auto i : tree[root]) {

            if (i == pre) continue;

            res[i] = res[root] - count[i] + count.size() - count[i];

            dfs2(i, root);

        }

    }

};

### 839. Similar String Groups

Hard

Two strings X and Y are similar if we can swap two letters (in different positions) of X, so that it equals Y. Also two strings X and Y are similar if they are equal.

For example, "tars" and "rats" are similar (swapping at positions 0 and 2), and "rats" and "arts" are similar, but "star" is not similar to "tars", "rats", or "arts".

Together, these form two connected groups by similarity: {"tars", "rats", "arts"} and {"star"}.  Notice that "tars" and "arts" are in the same group even though they are not similar.  Formally, each group is such that a word is in the group if and only if it is similar to at least one other word in the group.

We are given a list strs of strings where every string in strs is an anagram of every other string in strs. How many groups are there?

**Example 1:**

**Input:** strs = ["tars","rats","arts","star"]

**Output:** 2

**Example 2:**

**Input:** strs = ["omv","ovm"]

**Output:** 1

**Constraints:**

* 1 <= strs.length <= 100
* 1 <= strs[i].length <= 1000
* sum(strs[i].length) <= 2 \* 104
* strs[i] consists of lowercase letters only.
* All words in strs have the same length and are anagrams of each other.

class Solution {

public:

    int numSimilarGroups(vector<string>& strs) {

        strs.erase(unique(strs.begin(), strs.end()), strs.end());

        int n = strs.size(), res = 0;

        vector<int> fa(n, -1);

        for (int i = 0; i < n; ++i) {

            for (int j = 0; j < i; ++j) {

                if (check(strs[i], strs[j])) {

                    Union(i, j, fa);

                }

            }

        }

        for (int i = 0; i < n; ++i) {

            if (find(i, fa) == i) ++res;

        }

        return res;

    }

    bool check(string &a, string &b) {

        int n = a.size(), cnt = 0;

        for (int i = 0; i < n; ++i)

            if (a[i] != b[i])

                if (++cnt > 2) return false;

        return true;

    }

    void Union(int x, int y, vector<int> &fa) {

        int fax = find(x, fa);

        int fay = find(y, fa);

        if (fax != fay) {

            fa[fay] += fa[fax];

            fa[fax] = fay;

        }

    }

    int find(int x, vector<int> &fa) {

        return fa[x] < 0 ? x : (fa[x] = find(fa[x], fa));

    }

};

### 847. Shortest Path Visiting All Nodes

Hard

An undirected, connected graph of N nodes (labeled 0, 1, 2, ..., N-1) is given as graph.

graph.length = N, and j != i is in the list graph[i] exactly once, if and only if nodes i and j are connected.

Return the length of the shortest path that visits every node. You may start and stop at any node, you may revisit nodes multiple times, and you may reuse edges.

**Example 1:**

**Input:** [[1,2,3],[0],[0],[0]]

**Output:** 4

**Explanation**: One possible path is [1,0,2,0,3]

**Example 2:**

**Input:** [[1],[0,2,4],[1,3,4],[2],[1,2]]

**Output:** 4

**Explanation**: One possible path is [0,1,4,2,3]

**Note:**

1. 1 <= graph.length <= 12
2. 0 <= graph[i].length < graph.length

class Solution {

public:

    int shortestPathLength(vector<vector<int>>& graph) {

        n = graph.size();

        dis.resize(n, vector<int> (n, inf));

        dp.resize(n, vector<int> (1<<(n+1), inf));

        for (int i = 0; i < n; ++i) {

            dis[i][i] = 0;

            dp[i][1<<i] = 0;

            for (auto j : graph[i]) {

                dis[i][j] = dis[j][i] = 1;

            }

        }

        floyd(n);

        return DP();

    }

private:

    const int inf = 100;

    int n;

    vector<vector<int>> dis, dp;

    int DP() {

        for (int group = 1; group < (1<<n); group++) {

            for (int u = 0; u < n; u++) {

                for (int v = 0; v < n; v++) {

                    int src = 1 << u, dst = 1 << v;

                    if ((group & src) && !(group & dst))

                        dp[v][group|dst] = min(dp[u][group] + dis[u][v]

, dp[v][group|dst]);

                }

            }

        }

        int ret = inf;

        for (int i = 0; i < n; i++)

            ret = min(dp[i][(1<<n)-1], ret);

        return ret;

    }

    void floyd(int n) {

        for (int k = 0; k < n; k++)

            for (int i = 0; i < n; i++)

                for (int j = 0; j < n; j++)

                    dis[i][j] = min(dis[i][j], dis[i][k]+dis[k][j]);

    }

};

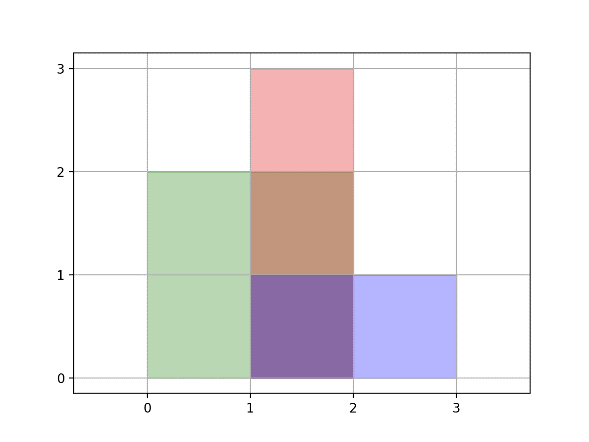
### 850. Rectangle Area II

Hard

We are given a list of (axis-aligned) rectangles. Each rectangle[i] = [xi1, yi1, xi2, yi2] , where (xi1, yi1) are the coordinates of the bottom-left corner, and (xi2, yi2) are the coordinates of the top-right corner of the ith rectangle.

Find the total area covered by all rectangles in the plane. Since the answer may be too large, return it **modulo** 109 + 7.

**Example 1:**



**Input:** rectangles = [[0,0,2,2],[1,0,2,3],[1,0,3,1]]

**Output:** 6

**Explanation:** As illustrated in the picture.

**Example 2:**

**Input:** rectangles = [[0,0,1000000000,1000000000]]

**Output:** 49

**Explanation:** The answer is 1018 modulo (109 + 7), which is (109)2 = (-7)2 = 49.

**Constraints:**

* 1 <= rectangles.length <= 200
* rectanges[i].length = 4
* 0 <= rectangles[i][j] <= 109
* The total area covered by all rectangles will never exceed 263 - 1 and thus will fit in a **64-bit** signed integer.

class Solution {

public:

    int rectangleArea(vector<vector<int>>& rect) {

        vector<int> X, Y;

        for (auto v : rect) {

            X.push\_back(v[0]); X.push\_back(v[2]);

            Y.push\_back(v[1]); Y.push\_back(v[3]);

        }

        sort(X.begin(), X.end());

        sort(Y.begin(), Y.end());

        int n = X.size(), m = Y.size();

        vector<vector<bool>> t(n, vector<bool> (m, false));

        for (auto v : rect) {

            int x0 = lower\_bound(X.begin(), X.end(), v[0]) - X.begin();

            int y0 = lower\_bound(Y.begin(), Y.end(), v[1]) - Y.begin();

            int x1 = lower\_bound(X.begin(), X.end(), v[2]) - X.begin();

            int y1 = lower\_bound(Y.begin(), Y.end(), v[3]) - Y.begin();

            for (int i = x0; i < x1; ++i) {

                for (int j = y0; j < y1; ++j) {

                    t[i][j] = true;

                }

            }

        }

        long long res = 0, mod = 1e9 + 7;

        for (int i = 0; i < n; ++i) {

            for (int j = 0; j < m; ++j) {

                if (t[i][j]) {

                    long long x = X[i+1] - X[i];

                    long long y = Y[j+1] - Y[j];

                    res = (res + x\*y) % mod;

                }

            }

        }

        return static\_cast<int> (res);

    }

};

### 854. K-Similar Strings

Hard

Strings A and B are K-similar (for some non-negative integer K) if we can swap the positions of two letters in A exactly K times so that the resulting string equals B.

Given two anagrams A and B, return the smallest K for which A and B are K-similar.

**Example 1:**

**Input:** A = "ab", B = "ba"

**Output:** 1

**Example 2:**

**Input:** A = "abc", B = "bca"

**Output:** 2

**Example 3:**

**Input:** A = "abac", B = "baca"

**Output:** 2

**Example 4:**

**Input:** A = "aabc", B = "abca"

**Output:** 2

**Note:**

1. 1 <= A.length == B.length <= 20
2. A and B contain only lowercase letters from the set {'a', 'b', 'c', 'd', 'e', 'f'}

### 857. Minimum Cost to Hire K Workers

Hard

There are N workers.  The i-th worker has a quality[i] and a minimum wage expectation wage[i].

Now we want to hire exactly K workers to form a *paid group*.  When hiring a group of K workers, we must pay them according to the following rules:

1. Every worker in the paid group should be paid in the ratio of their quality compared to other workers in the paid group.
2. Every worker in the paid group must be paid at least their minimum wage expectation.

Return the least amount of money needed to form a paid group satisfying the above conditions.

**Example 1:**

**Input:** quality = [10,20,5], wage = [70,50,30], K = 2

**Output:** 105.00000

**Explanation**: We pay 70 to 0-th worker and 35 to 2-th worker.

**Example 2:**

**Input:** quality = [3,1,10,10,1], wage = [4,8,2,2,7], K = 3

**Output:** 30.66667

**Explanation**: We pay 4 to 0-th worker, 13.33333 to 2-th and 3-th workers seperately.

**Note:**

1. 1 <= K <= N <= 10000, where N = quality.length = wage.length
2. 1 <= quality[i] <= 10000
3. 1 <= wage[i] <= 10000
4. Answers within 10^-5 of the correct answer will be considered correct.

class Solution {

public:

    double mincostToHireWorkers(vector<int> q, vector<int> w, int K) {

        vector<pair<double,int>> workers;

        for (int i = 0; i < q.size(); ++i)

            workers.emplace\_back((double)(w[i]) / q[i], q[i]);

        sort(workers.begin(), workers.end());

        double res = DBL\_MAX, sum = 0;

        priority\_queue<int> pq;

        for (const auto &worker: workers) {

            sum += worker.second;

            pq.push(worker.second);

            if (pq.size() > K) {

                sum -= pq.top();

                pq.pop();

            }

            if (pq.size() == K) res = min(res, sum \* worker.first);

        }

        return res;

    }

};

### 862. Shortest Subarray with Sum at Least K

Hard

Return the **length** of the shortest, non-empty, contiguous subarray of A with sum at least K.

If there is no non-empty subarray with sum at least K, return -1.

**Example 1:**

**Input:** A = [1], K = 1

**Output:** 1

**Example 2:**

**Input:** A = [1,2], K = 4

**Output:** -1

**Example 3:**

**Input:** A = [2,-1,2], K = 3

**Output:** 3

**Note:**

1. 1 <= A.length <= 50000
2. -10 ^ 5 <= A[i] <= 10 ^ 5
3. 1 <= K <= 10 ^ 9

//O(nlogn)

class Solution {

public:

    using pii = pair<int, int>;

    int shortestSubarray(vector<int>& A, int K) {

        priority\_queue<pii, vector<pii>, greater<>> pq;

        int N = A.size(), sum = 0, res = N+1;

        for (int i = 0; i < N; i++) {

            sum += A[i];

            if (sum >= K) res = min(res, i+1);

            while (!pq.empty() && sum - pq.top().first >= K) {

                res = min(res, i - pq.top().second);

                pq.pop();

            }

            pq.emplace(sum, i);

        }

        return res <= N ? res : -1;

    }

};

//O(n)

class Solution {

public:

    int shortestSubarray(vector<int> A, int K) {

        int N = A.size(), res = N + 1;

        deque<int> deq;

        for (int i = 0; i < N; i++) {

            if (i > 0) A[i] += A[i - 1];

            if (A[i] >= K) res = min(res, i + 1);

            while (!deq.empty() && A[i] <= A[deq.back()])

                deq.pop\_back();

            while (!deq.empty() && A[i] - A[deq.front()] >= K) {

                res = min(res, i - deq.front());

                deq.pop\_front();

            }

            deq.push\_back(i);

        }

        return res <= N ? res : -1;

    }

};

### 864. Shortest Path to Get All Keys

Hard

We are given a 2-dimensional grid. "." is an empty cell, "#" is a wall, "@" is the starting point, ("a", "b", ...) are keys, and ("A", "B", ...) are locks.

We start at the starting point, and one move consists of walking one space in one of the 4 cardinal directions.  We cannot walk outside the grid, or walk into a wall.  If we walk over a key, we pick it up.  We can't walk over a lock unless we have the corresponding key.

For some 1 <= K <= 6, there is exactly one lowercase and one uppercase letter of the first K letters of the English alphabet in the grid.  This means that there is exactly one key for each lock, and one lock for each key; and also that the letters used to represent the keys and locks were chosen in the same order as the English alphabet.

Return the lowest number of moves to acquire all keys.  If it's impossible, return -1.

**Example 1:**

**Input:** ["@.a.#","###.#","b.A.B"]

**Output:** 8

**Example 2:**

**Input:** ["@..aA","..B#.","....b"]

**Output:** 6

**Note:**

1. 1 <= grid.length <= 30
2. 1 <= grid[0].length <= 30
3. grid[i][j] contains only '.', '#', '@', 'a'-'f' and 'A'-'F'
4. The number of keys is in [1, 6].  Each key has a different letter and opens exactly one lock.

class Solution {

public:

    const vector<int> dirs{0,1,0,-1,0};

    bool vis[31][31][64];

    int shortestPathAllKeys(vector<string>& grid) {

        int n = grid.size(), m = grid[0].size(), cnt = 0, keys = 0;

        tuple<int, int, int> start;

        for (int i = 0; i < n; ++i) {

            for (int j = 0; j < m; ++j) {

                if (grid[i][j] == '@') {

                    vis[i][j][0] = true;

                    start = {i, j, 0};

                }

                else if (islower(grid[i][j]))

                    keys |= 1 << (grid[i][j] - 'a');

            }

        }

        if (keys == 0) return 0;

        queue<tuple<int, int, int>> q; q.push(start);

        while (!q.empty()) {

            int sz = q.size();

            ++cnt;

            while (sz--) {

                auto [x, y, k] = q.front(); q.pop();

                for (int i = 0; i < 4; ++i) {

                    int xx = x + dirs[i], yy = y + dirs[i+1], kk = k;

                    if (xx<0 || yy<0 || xx>=n || yy>=m) continue;

                    if (grid[xx][yy] == '#') continue;

                    else if (islower(grid[xx][yy])) {

                        kk |= 1 << (grid[xx][yy] - 'a');

                        if (kk == keys) return cnt;

                    }

                    else if (isupper(grid[xx][yy])) {

                        if (!(kk & (1 << (grid[xx][yy]-'A'))))continue;

                    }

                    if (vis[xx][yy][kk]) continue;

                    vis[xx][yy][kk] = true;

                    q.emplace(xx, yy, kk);

                }

            }

        }

        return -1;

    }

};

### 871. Minimum Number of Refueling Stops

Hard

A car travels from a starting position to a destination which is target miles east of the starting position.

Along the way, there are gas stations.  Each station[i] represents a gas station that is station[i][0] miles east of the starting position, and has station[i][1] liters of gas.

The car starts with an infinite tank of gas, which initially has startFuel liters of fuel in it.  It uses 1 liter of gas per 1 mile that it drives.

When the car reaches a gas station, it may stop and refuel, transferring all the gas from the station into the car.

What is the least number of refueling stops the car must make in order to reach its destination?  If it cannot reach the destination, return -1.

Note that if the car reaches a gas station with 0 fuel left, the car can still refuel there.  If the car reaches the destination with 0 fuel left, it is still considered to have arrived.

**Example 1:**

**Input:** target = 1, startFuel = 1, stations = []

**Output:** 0

**Explanation:** We can reach the target without refueling.

**Example 2:**

**Input:** target = 100, startFuel = 1, stations = [[10,100]]

**Output:** -1

**Explanation:** We can't reach the target (or even the first gas station).

**Example 3:**

**Input:** target = 100, startFuel = 10, stations = [[10,60],[20,30],[30,30],[60,40]]

**Output:** 2

**Explanation:**

We start with 10 liters of fuel.

We drive to position 10, expending 10 liters of fuel. We refuel from 0 liters to 60 liters of gas.

Then, we drive from position 10 to position 60 (expending 50 liters of fuel),

and refuel from 10 liters to 50 liters of gas. We then drive to and reach the target.

We made 2 refueling stops along the way, so we return 2.

**Note:**

1. 1 <= target, startFuel, stations[i][1] <= 10^9
2. 0 <= stations.length <= 500
3. 0 < stations[0][0] < stations[1][0] < ... < stations[stations.length-1][0] < target

//O(n^2)

class Solution {

public:

    int minRefuelStops(int target, int startFuel, vector<vector<int>>& stations) {

        int N = stations.size();

        vector<long> dp(N+1);

        dp[0] = startFuel;

        for (int i = 0; i < N; ++i) {

            for (int t = i; t >= 0; --t)

                if (dp[t] >= stations[i][0])

                    dp[t+1] = max(dp[t+1], dp[t]+(long)stations[i][1]);

        }

        for (int i = 0; i <= N; ++i)

            if (dp[i] >= target) return i;

        return -1;

    }

};

//O(NlogN)

class Solution {

public:

    int minRefuelStops(int target, int startFuel, vector<vector<int>>& stations) {

        priority\_queue<int> pq;

        int farest = startFuel;

        int cnt = 0, i = 0, n = stations.size();

        while (farest < target) {

            while (i < n && stations[i][0] <= farest) {

                pq.push(stations[i++][1]);

            }

            if (farest >= target || pq.empty()) break;

            ++cnt;

            farest += pq.top();

            pq.pop();

        }

        return farest >= target ? cnt : -1;

    }

};

### 878. Nth Magical Number

Hard

A positive integer is *magical* if it is divisible by either A or B.

Return the N-th magical number.  Since the answer may be very large, **return it modulo**10^9 + 7.

**Example 1:**

**Input:** N = 1, A = 2, B = 3

**Output:** 2

**Example 2:**

**Input:** N = 4, A = 2, B = 3

**Output:** 6

**Example 3:**

**Input:** N = 5, A = 2, B = 4

**Output:** 10

**Example 4:**

**Input:** N = 3, A = 6, B = 4

**Output:** 8

**Note:**

1. 1 <= N <= 10^9
2. 2 <= A <= 40000
3. 2 <= B <= 40000

class Solution {

public:

    int nthMagicalNumber(int n, int a, int b) {

        using ll = long long;

        const ll mod = 1e9 +7;

        ll t = a\*b / \_\_gcd(a, b);

        ll l = 1, r = LONG\_MAX;

        while (l < r) {

            ll mid = l + (r-l) / 2;

            ll rk = mid/a + mid/b - mid/t;

            if (rk < n) l = mid+1;

            else r = mid;

        }

        return static\_cast<int> (l % mod);

    }

};

### 879. Profitable Schemes

Hard

There is a group of G members, and a list of various crimes they could commit.

The ith crime generates a profit[i] and requires group[i] members to participate in it.

If a member participates in one crime, that member can't participate in another crime.

Let's call a *profitable scheme* any subset of these crimes that generates at least P profit, and the total number of members participating in that subset of crimes is at most G.

How many schemes can be chosen?  Since the answer may be very large, **return it modulo** 10^9 + 7.

**Example 1:**

**Input:** G = 5, P = 3, group = [2,2], profit = [2,3]

**Output:** 2

To make a profit of at least 3, the group could either commit crimes 0 and 1, or just crime 1.

In total, there are 2 schemes.

**Example 2:**

**Input:** G = 10, P = 5, group = [2,3,5], profit = [6,7,8]

**Output:** 7

To make a profit of at least 5, the group could commit any crimes, as long as they commit one.

There are 7 possible schemes: (0), (1), (2), (0,1), (0,2), (1,2), and (0,1,2).

**Note:**

1. 1 <= G <= 100
2. 0 <= P <= 100
3. 1 <= group[i] <= 100
4. 0 <= profit[i] <= 100
5. 1 <= group.length = profit.length <= 100

class Solution {

public:

    using ll = long long;

    const ll mod = 1e9 + 7;

    int profitableSchemes(int n, int minProfit, vector<int>& group, vector<int>& profit) {

        vector<vector<ll>> pre(n+1, vector<ll>(minProfit+1, 0));

        pre[0][0] = 1;

        for (int i = 0; i < group.size(); ++i) {

            vector<vector<ll>> cur = pre;

            for (int a = 0, aa = group[i]; aa <= n; ++a, ++aa) {

                for (int b = 0; b <= minProfit; ++b) {

                    int bb = min(b+profit[i], minProfit);

                    cur[aa][bb] = (cur[aa][bb] + pre[a][b]) % mod;

                }

            }

            pre = std::move(cur);

        }

        ll res = 0;

        for (int i = 0; i <= n; ++i)

res = (res + pre[i][minProfit]) % mod;

        return static\_cast<int> (res);

    }

};

### 887. Super Egg Drop

Hard

You are given K eggs, and you have access to a building with N floors from 1 to N.

Each egg is identical in function, and if an egg breaks, you cannot drop it again.

You know that there exists a floor F with 0 <= F <= N such that any egg dropped at a floor higher than F will break, and any egg dropped at or below floor F will not break.

Each *move*, you may take an egg (if you have an unbroken one) and drop it from any floor X (with 1 <= X <= N).

Your goal is to know **with certainty** what the value of F is.

What is the minimum number of moves that you need to know with certainty what F is, regardless of the initial value of F?

**Example 1:**

**Input:** K = 1, N = 2

**Output:** 2

Drop the egg from floor 1. If it breaks, we know with certainty that F = 0.Otherwise, drop the egg from floor 2. If it breaks, we know with certainty that F = 1.If it didn't break, then we know with certainty F = 2.Hence, we needed 2 moves in the worst case to know what F is with certainty.

**Example 2:**

**Input:** K = 2, N = 6

**Output:** 3

**Example 3:**

**Input:** K = 3, N = 14

**Output:** 4

**Note:**

1. 1 <= K <= 100
2. 1 <= N <= 10000

class Solution {

public:

    int superEggDrop(int K, int N) {

        vector<vector<int>> dp(K + 1, vector<int> (N+1, 0));

// 当前有 k 个鸡蛋，可以尝试扔 m 次鸡蛋

// 这个状态下，最坏情况下最多能确切测试一栋 n 层的楼

        int m = 0;

        while (dp[K][m] < N) {

            m++;

            for (int k = 1; k <= K; k++)

                dp[k][m] = dp[k][m - 1] + dp[k - 1][m - 1] + 1;

       }

       return m;

   }

};

class Solution {

public:

    int dp[105][100005];

    int superEggDrop(int K, int N) {

        return helper(K, N);

    }

    int helper(int K, int N) {

        if (K == 1) return N;

        if (N <= 1) return 1;

        if (dp[K][N] != 0) return dp[K][N];

        int ret = INT\_MAX;

        //以下注释掉的算法的时间复杂度O(K\*N^2) 超时

        //for (int i = 1; i <= N; ++i) {

        // ret = min(ret, 1+max(helper(K-1, i-1),helper(K, N-i)));

        //}

        int left = 0, right = N;

        while (left <= right) {

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

            int broken = helper(K-1, mid-1);

            int not\_broken = helper(K, N-mid);

            if (broken > not\_broken) {

                right = mid-1;

                ret = min(ret, broken+1);

            }

            else {

                left = mid+1;

                ret = min(ret, not\_broken+1);

            }

        }

        return dp[K][N] = ret;

    }

};

### 891. Sum of Subsequence Widths

Hard

Given an array of integers A, consider all non-empty subsequences of A.

For any sequence S, let the *width* of S be the difference between the maximum and minimum element of S.

Return the sum of the widths of all subsequences of A.

As the answer may be very large, **return the answer modulo 10^9 + 7**.

**Example 1:**

**Input:** [2,1,3]

**Output:** 6

**Explanation:**

Subsequences are [1], [2], [3], [2,1], [2,3], [1,3], [2,1,3].

The corresponding widths are 0, 0, 0, 1, 1, 2, 2.

The sum of these widths is 6.

**Note:**

* 1 <= A.length <= 20000
* 1 <= A[i] <= 20000

class Solution {

public:

    using ll = long long;

    int sumSubseqWidths(vector<int>& arr) {

        sort(arr.begin(), arr.end());

        ll res = 0, mod = 1e9+7, n = arr.size();

        vector<ll> pow2(n, 1);

        for(int i = 1; i < n; i++) {

            pow2[i] = (pow2[i-1] \* 2) % mod;

        }

        for(int i = 0; i < n; i++){

            res = (res + arr[i]\*((pow2[i]-pow2[n-1-i]+mod)%mod)) % mod;

        }

        return static\_cast<int> (res);

    }

};

### 895. Maximum Frequency Stack

Hard

Implement FreqStack, a class which simulates the operation of a stack-like data structure.

FreqStack has two functions:

* push(int x), which pushes an integer x onto the stack.
* pop(), which **removes** and returns the most frequent element in the stack.
  + If there is a tie for most frequent element, the element closest to the top of the stack is removed and returned.

**Example 1:**

**Input:**

["FreqStack","push","push","push","push","push","push","pop","pop","pop","pop"],

[[],[5],[7],[5],[7],[4],[5],[],[],[],[]]

**Output:** [null,null,null,null,null,null,null,5,7,5,4]

**Explanation**:

After making six .push operations, the stack is [5,7,5,7,4,5] from bottom to top. Then:

pop() -> returns 5, as 5 is the most frequent.

The stack becomes [5,7,5,7,4].

pop() -> returns 7, as 5 and 7 is the most frequent, but 7 is closest to the top.

The stack becomes [5,7,5,4].

pop() -> returns 5.

The stack becomes [5,7,4].

pop() -> returns 4.

The stack becomes [5,7].

**Note:**

* Calls to FreqStack.push(int x) will be such that 0 <= x <= 10^9.
* It is guaranteed that FreqStack.pop() won't be called if the stack has zero elements.
* The total number of FreqStack.push calls will not exceed 10000 in a single test case.
* The total number of FreqStack.pop calls will not exceed 10000 in a single test case.
* The total number of FreqStack.push and FreqStack.pop calls will not exceed 150000 across all test cases.

class FreqStack {

public:

    map<int, int> cntCache;

    map<int, stack<int>> group;

    int maxFreq = 0;

    FreqStack() {}

    void push(int x) {

        int freq = ++cntCache[x];

        if (freq > maxFreq) maxFreq = freq;

        group[freq].push(x);

    }

    int pop() {

        int ret = group[maxFreq].top();

        group[maxFreq].pop();

        --cntCache[ret];

        if (group[maxFreq].empty()) {

            group.erase(maxFreq);

            --maxFreq;

        }

        return ret;

    }

};

/\*\*

 \* Your FreqStack object will be instantiated and called as such:

 \* FreqStack\* obj = new FreqStack();

 \* obj->push(x);

 \* int param\_2 = obj->pop();

 \*/

### 902. Numbers At Most N Given Digit Set

Hard

Given an array of digits which is sorted in **non-decreasing** order. You can write numbers using each digits[i] as many times as we want. For example, if digits = ['1','3','5'], we may write numbers such as '13', '551', and '1351315'.

Return *the number of positive integers that can be generated*that are less than or equal to a given integer n.

**Example 1:**

**Input:** digits = ["1","3","5","7"], n = 100

**Output:** 20

**Explanation:**

The 20 numbers that can be written are:

1, 3, 5, 7, 11, 13, 15, 17, 31, 33, 35, 37, 51, 53, 55, 57, 71, 73, 75, 77.

**Example 2:**

**Input:** digits = ["1","4","9"], n = 1000000000

**Output:** 29523

**Explanation:**

We can write 3 one digit numbers, 9 two digit numbers, 27 three digit numbers,

81 four digit numbers, 243 five digit numbers, 729 six digit numbers,

2187 seven digit numbers, 6561 eight digit numbers, and 19683 nine digit numbers.

In total, this is 29523 integers that can be written using the digits array.

**Example 3:**

**Input:** digits = ["7"], n = 8

**Output:** 1

**Constraints:**

* 1 <= digits.length <= 9
* digits[i].length == 1
* digits[i] is a digit from '1' to '9'.
* All the values in digits are **unique**.
* digits is sorted in **non-decreasing** order.
* 1 <= n <= 109

class Solution {

public:

    int atMostNGivenDigitSet(vector<string>& D, int N) {

        string S = to\_string(N);

        int len = S.size(), dsize = D.size(), res = 0;

        vector<int> powD(len, 1);

        for (int i = 1 ; i < len ; ++i) {

            powD[i] = powD[i-1] \* D.size();

            res += powD[i];

        }

        for (int i = 0 ; i < len ; ++i) {

            bool hasSameNum = false;

            for (const string &d : D) {

                if (d[0] < S[i])

                    res += powD[len-i-1];

                else if (d[0] == S[i])

                    hasSameNum = true;

            }

            if (!hasSameNum) return res;

        }

        return res+1;

    }

};

### 903. Valid Permutations for DI Sequence

Hard

We are given S, a length n string of characters from the set {'D', 'I'}. (These letters stand for "decreasing" and "increasing".)

A *valid permutation* is a permutation P[0], P[1], ..., P[n] of integers {0, 1, ..., n}, such that for all i:

* If S[i] == 'D', then P[i] > P[i+1], and;
* If S[i] == 'I', then P[i] < P[i+1].

How many valid permutations are there?  Since the answer may be large, **return your answer modulo 10^9 + 7**.

**Example 1:**

**Input:** "DID"

**Output:** 5

**Explanation:**

The 5 valid permutations of (0, 1, 2, 3) are:

(1, 0, 3, 2)

(2, 0, 3, 1)

(2, 1, 3, 0)

(3, 0, 2, 1)

(3, 1, 2, 0)

**Note:**

1. 1 <= S.length <= 200
2. S consists only of characters from the set {'D', 'I'}.

### 913. Cat and Mouse

Hard

A game on an **undirected** graph is played by two players, Mouse and Cat, who alternate turns.

The graph is given as follows: graph[a] is a list of all nodes b such that ab is an edge of the graph.

The mouse starts at node 1 and goes first, the cat starts at node 2 and goes second, and there is a hole at node 0.

During each player's turn, they **must** travel along one edge of the graph that meets where they are.  For example, if the Mouse is at node 1, it **must** travel to any node in graph[1].

Additionally, it is not allowed for the Cat to travel to the Hole (node 0.)

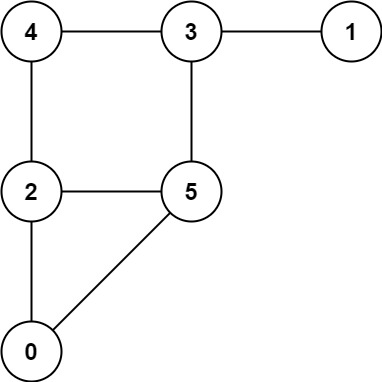
Then, the game can end in three ways:

* If ever the Cat occupies the same node as the Mouse, the Cat wins.
* If ever the Mouse reaches the Hole, the Mouse wins.
* If ever a position is repeated (i.e., the players are in the same position as a previous turn, and it is the same player's turn to move), the game is a draw.

Given a graph, and assuming both players play optimally, return

* 1 if the mouse wins the game,
* 2 if the cat wins the game, or
* 0 if the game is a draw.

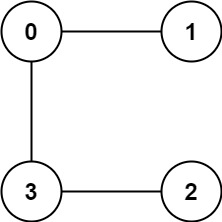
**Example 1:**



**Input:** graph = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]

**Output:** 0

**Example 2:**



**Input:** graph = [[1,3],[0],[3],[0,2]]

**Output:** 1

**Constraints:**

* 3 <= graph.length <= 50
* 1 <= graph[i].length < graph.length
* 0 <= graph[i][j] < graph.length
* graph[i][j] != i
* graph[i] is unique.
* The mouse and the cat can always move.

### 920. Number of Music Playlists

Hard

Your music player contains N different songs and she wants to listen to L(not necessarily different) songs during your trip.  You create a playlist so that:

* Every song is played at least once
* A song can only be played again only if K other songs have been played

Return the number of possible playlists.  **As the answer can be very large, return it modulo 10^9 + 7**.

**Example 1:**

**Input:** N = 3, L = 3, K = 1

**Output:** 6

**Explanation**: There are 6 possible playlists. [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1].

**Example 2:**

**Input:** N = 2, L = 3, K = 0

**Output:** 6

**Explanation**: There are 6 possible playlists. [1, 1, 2], [1, 2, 1], [2, 1, 1], [2, 2, 1], [2, 1, 2], [1, 2, 2]

**Example 3:**

**Input:** N = 2, L = 3, K = 1

**Output:** 2

**Explanation**: There are 2 possible playlists. [1, 2, 1], [2, 1, 2]

**Note:**

1. 0 <= K < N <= L <= 100

//dp(i,j) 前i首歌有j首不同的歌

class Solution {

public:

    using ll = long long;

    const static ll mod = 1e9 + 7;

    int numMusicPlaylists(int N, int L, int K) {

        vector<vector<ll>> dp(L+1, vector<ll> (N+1, 0));

        dp[0][0] = 1;

        for (int i = 1; i <= L; ++i) {

            for (int j = N; j > 0; --j) {

                dp[i][j] += dp[i-1][j-1]\*(N+1-j);

                dp[i][j] += dp[i-1][j] \* max(0, j-K);

                dp[i][j] %= mod;

            }

        }

        return static\_cast<int> (dp[L][N]);

    }

};

### 924. Minimize Malware Spread

Hard

You are given a network of n nodes represented as an n x n adjacency matrix graph, where the ith node is directly connected to the jth node if graph[i][j] == 1.

Some nodes initial are initially infected by malware. Whenever two nodes are directly connected, and at least one of those two nodes is infected by malware, both nodes will be infected by malware. This spread of malware will continue until no more nodes can be infected in this manner.

Suppose M(initial) is the final number of nodes infected with malware in the entire network after the spread of malware stops. We will remove **exactly one node** from initial.

Return the node that, if removed, would minimize M(initial). If multiple nodes could be removed to minimize M(initial), return such a node with **the smallest index**.

Note that if a node was removed from the initial list of infected nodes, it might still be infected later due to the malware spread.

**Example 1:**

**Input:** graph = [[1,1,0],[1,1,0],[0,0,1]], initial = [0,1]

**Output:** 0

**Example 2:**

**Input:** graph = [[1,0,0],[0,1,0],[0,0,1]], initial = [0,2]

**Output:** 0

**Example 3:**

**Input:** graph = [[1,1,1],[1,1,1],[1,1,1]], initial = [1,2]

**Output:** 1

**Constraints:**

* n == graph.length
* n == graph[i].length
* 2 <= n <= 300
* graph[i][j] is 0 or 1.
* graph[i][j] == graph[j][i]
* graph[i][i] == 1
* 1 <= initial.length <= n
* 0 <= initial[i] <= n - 1
* All the integers in initial are **unique**.

class Solution {

public:

    int minMalwareSpread(vector<vector<int>>& graph, vector<int>& initial) {

        if (initial.empty()) return 0;

        int n = graph.size();

        vector<int> fa(n, -1);

        for (int i = 0; i < n; ++i) {

            for (int j = 0; j < i; ++j) {

                if (graph[i][j]) Union(i, j, fa);

            }

        }

        sort(initial.begin(), initial.end());

        map<int, pair<int, int>> mp;//fa, <cnt, minid>

        for (auto x : initial) {

            int fax = find(x, fa);

            if (!mp.count(fax)) mp[fax] = {1, x};

            else {

                mp[fax] = {mp[fax].first+1, mp[fax].second};

            }

        }

        int res = -1, maxcnt = 0;

        for (auto &[fax, pii] : mp) {

            auto [cnt, id] = pii;

            if (cnt == 1 && (-fa[fax] > maxcnt || -fa[fax] == maxcnt && res > id)) {

                maxcnt = -fa[fax];

                res = id;

            }

        }

        return maxcnt == 0 ? initial[0] : res;

    }

    int find(int x, vector<int> &fa) {

        return fa[x] < 0 ? x : (fa[x] = find(fa[x], fa));

    }

    void Union(int x, int y, vector<int> &fa) {

        int fax = find(x, fa);

        int fay = find(y, fa);

        if (fax != fay) {

            fa[fay] += fa[fax];

            fa[fax] = fay;

        }

    }

};

### 927. Three Equal Parts

Hard

Given an array A of 0s and 1s, divide the array into 3 non-empty parts such that all of these parts represent the same binary value.

If it is possible, return **any** [i, j] with i+1 < j, such that:

* A[0], A[1], ..., A[i] is the first part;
* A[i+1], A[i+2], ..., A[j-1] is the second part, and
* A[j], A[j+1], ..., A[A.length - 1] is the third part.
* All three parts have equal binary value.

If it is not possible, return [-1, -1].

Note that the entire part is used when considering what binary value it represents.  For example, [1,1,0] represents 6 in decimal, not 3.  Also, leading zeros are allowed, so [0,1,1] and [1,1] represent the same value.

**Example 1:**

**Input:** [1,0,1,0,1]

**Output:** [0,3]

**Example 2:**

**Input:** [1,1,0,1,1]

**Output:** [-1,-1]

**Note:**

1. 3 <= A.length <= 30000
2. A[i] == 0 or A[i] == 1

class Solution {

public:

    vector<int> threeEqualParts(vector<int>& A) {

        vector<int> v;

        for (int i = 0 ; i < A.size(); i++)

            if (A[i]) v.push\_back(i);

        if (v.size() % 3) return {-1, -1};

        if (v.empty()) return {0, 2};

        int l1 = 0, l2 = v.size() / 3, l3 = l2 \* 2;

        int tail0 = A.size() - v.back();

        if (v[l3]-v[l3-1] < tail0 || v[l2]-v[l2-1] < tail0)

return {-1,-1};

        for (int i = 1; i < l2; i++) {

            int diff = v[i] - v[i-1];

            if (v[l2+i]-v[l2+i-1]!=diff || v[l3+i]-v[l3+i-1]!=diff) {

                return {-1, -1};

            }

        }

        return {v[l2-1] + tail0 - 1, v[l3-1] + tail0};

    }

};

### 928. Minimize Malware Spread II

Hard

(This problem is the same as *Minimize Malware Spread*, with the differences bolded.)

In a network of nodes, each node i is directly connected to another node j if and only if graph[i][j] = 1.

Some nodes initial are initially infected by malware.  Whenever two nodes are directly connected and at least one of those two nodes is infected by malware, both nodes will be infected by malware.  This spread of malware will continue until no more nodes can be infected in this manner.

Suppose M(initial) is the final number of nodes infected with malware in the entire network, after the spread of malware stops.

We will remove one node from the initial list, **completely removing it and any connections from this node to any other node**.  Return the node that if removed, would minimize M(initial).  If multiple nodes could be removed to minimize M(initial), return such a node with the smallest index.

**Example 1:**

**Input:** graph = [[1,1,0],[1,1,0],[0,0,1]], initial = [0,1]

**Output:** 0

**Example 2:**

**Input:** graph = [[1,1,0],[1,1,1],[0,1,1]], initial = [0,1]

**Output:** 1

**Example 3:**

**Input:** graph = [[1,1,0,0],[1,1,1,0],[0,1,1,1],[0,0,1,1]], initial = [0,1]

**Output:** 1

**Note:**

1. 1 < graph.length = graph[0].length <= 300
2. 0 <= graph[i][j] == graph[j][i] <= 1
3. graph[i][i] = 1
4. 1 <= initial.length < graph.length
5. 0 <= initial[i] < graph.length

class Solution {

public:

    int minMalwareSpread(vector<vector<int>>& graph, vector<int>& initial) {

        if (initial.empty()) return 0;

        int n = graph.size();

        set<int> unClean(initial.begin(), initial.end()), clean;

        for (int i = 0; i < n; ++i)

if (!unClean.count(i)) clean.insert(i);

        vector<int> fa(n, -1);

        for (auto i : clean) {

            for (auto j : clean) {

                if (graph[i][j]) Union(i, j, fa);

            }

        }

        unordered\_map<int, unordered\_set<int>> cnt; //<fa, unclean\_set>

        unordered\_map<int, unordered\_set<int>> neigbor;//<unclean, clean\_neigbor\_set>

        for (auto i : initial) {

            for (auto j : clean) {

                if (graph[i][j]) {

                    cnt[find(j, fa)].insert(i);

                    neigbor[i].insert(find(j, fa));

                }

            }

        }

        int res = -1, maxcnt = -1;

        for (auto i : unClean) {

            int sum = 0;

            for (auto j : neigbor[i]) {

                if (cnt[j].size() == 1)

                    sum -= fa[j];

            }

            if (sum > maxcnt) {

                maxcnt = sum;

                res = i;

            }

        }

        return maxcnt == -1 ? \*unClean.begin() : res;

    }

    int find(int x, vector<int> &fa) {

        return fa[x] < 0 ? x : (fa[x] = find(fa[x], fa));

    }

    void Union(int x, int y, vector<int> &fa) {

        int fax = find(x, fa);

        int fay = find(y, fa);

        if (fax != fay) {

            fa[fay] += fa[fax];

            fa[fax] = fay;

        }

    }

};

### 936. Stamping The Sequence

Hard

You want to form a target string of **lowercase letters**.

At the beginning, your sequence is target.length '?' marks.  You also have a stamp of lowercase letters.

On each turn, you may place the stamp over the sequence, and replace every letter in the sequence with the corresponding letter from the stamp.  You can make up to 10 \* target.length turns.

For example, if the initial sequence is "?????", and your stamp is "abc",  then you may make "abc??", "?abc?", "??abc" in the first turn.  (Note that the stamp must be fully contained in the boundaries of the sequence in order to stamp.)

If the sequence is possible to stamp, then return an array of the index of the left-most letter being stamped at each turn.  If the sequence is not possible to stamp, return an empty array.

For example, if the sequence is "ababc", and the stamp is "abc", then we could return the answer [0, 2], corresponding to the moves "?????" -> "abc??" -> "ababc".

Also, if the sequence is possible to stamp, it is guaranteed it is possible to stamp within 10 \* target.length moves.  Any answers specifying more than this number of moves will not be accepted.

**Example 1:**

**Input:** stamp = "abc", target = "ababc"

**Output:** [0,2]

([1,0,2] would also be accepted as an answer, as well as some other answers.)

**Example 2:**

**Input:** stamp = "abca", target = "aabcaca"

**Output:** [3,0,1]

**Note:**

1. 1 <= stamp.length <= target.length <= 1000
2. stamp and target only contain lowercase letters.

class Solution {

public:

    vector<int> movesToStamp(string stamp, string target) {

        vector<int> res;

        int cnt = 0, n = stamp.size(), m = target.size();

        while (cnt < m) {

            bool flag = false;

            for (int i = 0; i <= m-n; ++i) {

                int j = 0;

                bool ok = false;

                while (j < n) {

                    if (target[i+j] != stamp[j] && target[i+j] != '?')

break;

                    else if (target[i+j] == stamp[j]) ok = true;

                    ++j;

                }

                if (ok && j == n) {

                    for (int j = 0; j < n; ++j) {

                        if (target[i+j] != '?') {

                            target[i+j] = '?';

                            ++cnt;

                        }

                    }

                    res.push\_back(i);

                    flag = true;

                }

            }

            if (!flag) break;

        }

        if (cnt != m) return {};

        reverse(res.begin(), res.end());

        return res;

    }

};

### 940. Distinct Subsequences II

Hard

Given a string S, count the number of distinct, non-empty subsequences of S .

Since the result may be large, **return the answer modulo 10^9 + 7**.

**Example 1:**

**Input:** "abc"

**Output:** 7

**Explanation**: The 7 distinct subsequences are "a", "b", "c", "ab", "ac", "bc", and "abc".

**Example 2:**

**Input:** "aba"

**Output:** 6

**Explanation**: The 6 distinct subsequences are "a", "b", "ab", "ba", "aa" and "aba".

**Example 3:**

**Input:** "aaa"

**Output:** 3

**Explanation**: The 3 distinct subsequences are "a", "aa" and "aaa".

**Note:**

1. S contains only lowercase letters.
2. 1 <= S.length <= 2000

class Solution {

public:

    using ll = long long;

    static const ll mod = 1e9 + 7;

    int distinctSubseqII(string S) {

        ll sum = 1;

        int cnt[26] = {0};

        for (auto c : S) {

            ll old = sum;

            sum = ((2\*sum - cnt[c-'a']) % mod + mod) % mod;

            cnt[c-'a'] = old;

        }

        return static\_cast<int> ((sum - 1 + mod) % mod);

    }

};

### 943. Find the Shortest Superstring

Hard

Given an array A of strings, find any smallest string that contains each string in A as a substring.

We may assume that no string in A is substring of another string in A.

**Example 1:**

**Input:** ["alex","loves","leetcode"]

**Output:** "alexlovesleetcode"

**Explanation:** All permutations of "alex","loves","leetcode" would also be accepted.

**Example 2:**

**Input:** ["catg","ctaagt","gcta","ttca","atgcatc"]

**Output:** "gctaagttcatgcatc"

class Solution {

public:

    string shortestSuperstring(vector<string>& A) {

        int n = A.size();

        // dp[mask][i] : min superstring made by strings in mask,

        // and the last one is A[i]

        vector<vector<string>> dp(1<<n, vector<string>(n));

        vector<vector<int>> overlap(n, vector<int>(n,0));

        // calculate overlap for A[i], A[j]

        for(int i=0; i<n; ++i) for(int j=0; j<n; ++j) if (i!=j) {

            for (int k = min(A[i].size(), A[j].size()); k > 0; --k) {

                if (A[i].substr(A[i].size()-k) == A[j].substr(0,k)) {

                    overlap[i][j] = k;

                    break;

                }

            }

        }

        for(int i = 0; i < n; ++i) dp[1<<i][i] += A[i];

        for (int mask = 1; mask<(1<<n); ++mask) {

            for (int j = 0; j < n; ++j) {

                if (!(mask & (1<<j))) continue;

                for (int i = 0; i < n; ++i) {

                    if (i == j || !(mask&(1<<i))) continue;

                    string s = dp[mask^(1<<j)][i]

+ A[j].substr(overlap[i][j]);

                    if (dp[mask][j].empty() || s.size()

< dp[mask][j].size())

                        dp[mask][j] = std::move(s);

                }

            }

        }

        string res = dp.back()[0];

        for (const auto &s : dp.back()) {

            if (s.size() < res.size()) {

                res = s;

            }

        }

        return res;

    }

};

### 952. Largest Component Size by Common Factor

Hard

Given a non-empty array of unique positive integers A, consider the following graph:

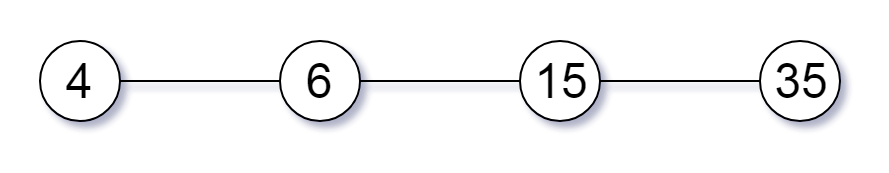
* There are A.length nodes, labelled A[0] to A[A.length - 1];
* There is an edge between A[i] and A[j] if and only if A[i] and A[j] share a common factor greater than 1.

Return the size of the largest connected component in the graph.

**Example 1:**

**Input:** [4,6,15,35]

**Output:** 4



**Example 2:**

**Input:** [20,50,9,63]

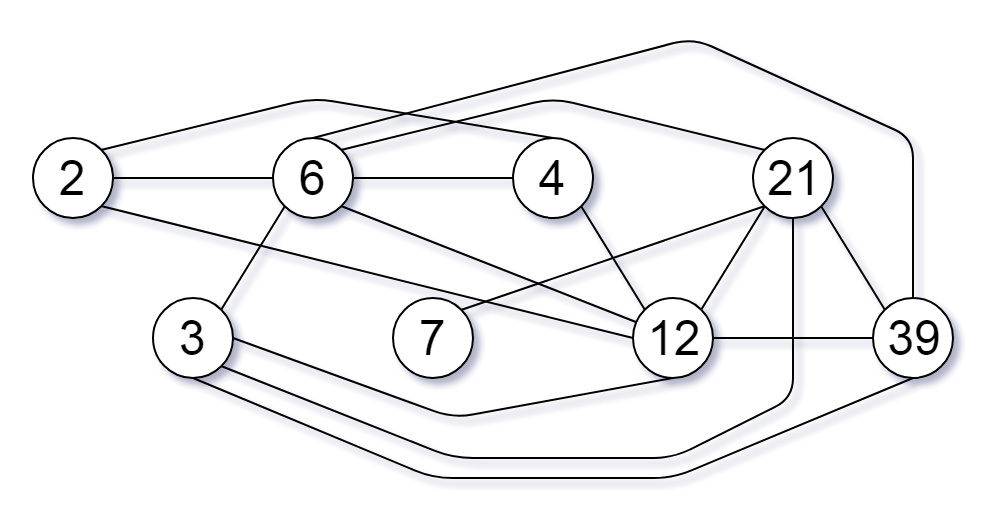
**Output:** 2



**Example 3:**

**Input:** [2,3,6,7,4,12,21,39]

**Output:** 8



**Note:**

1. 1 <= A.length <= 20000
2. 1 <= A[i] <= 100000

class Solution {

public:

    int fa[100005], cnt[100005] = {0};

    int largestComponentSize(vector<int>& A) {

        for (int i = 0; i < 100005; ++i) fa[i] = i;

        for (auto i : A) {

            int Sqrt = sqrt(i) + 0.5;

            for (int j = 2; j <= Sqrt; ++j) {

                if (i % j == 0) {

                    Union(i, j);

                    Union(i, i/j);

                }

            }

        }

        int res = 0;

        for (auto i : A) {

            res = max(res, ++cnt[find(i)]);

        }

        return res;

    }

    int find(int x) {

        return fa[x] == x ? x : (fa[x] = find(fa[x]));

    }

    void Union(int x, int y) {

        fa[find(y)] = find(x);

    }

};

### 956. Tallest Billboard

Hard

You are installing a billboard and want it to have the largest height.  The billboard will have two steel supports, one on each side.  Each steel support must be an equal height.

You have a collection of rods which can be welded together.  For example, if you have rods of lengths 1, 2, and 3, you can weld them together to make a support of length 6.

Return the largest possible height of your billboard installation.  If you cannot support the billboard, return 0.

**Example 1:**

**Input:** [1,2,3,6]

**Output:** 6

**Explanation:** We have two disjoint subsets {1,2,3} and {6}, which have the same sum = 6.

**Example 2:**

**Input:** [1,2,3,4,5,6]

**Output:** 10

**Explanation:** We have two disjoint subsets {2,3,5} and {4,6}, which have the same sum = 10.

**Example 3:**

**Input:** [1,2]

**Output:** 0

**Explanation:** The billboard cannot be supported, so we return 0.

**Note:**

1. 0 <= rods.length <= 20
2. 1 <= rods[i] <= 1000
3. The sum of rods is at most 5000.

//O(N \* Sum)

class Solution {

public:

    int tallestBillboard(vector<int>& rods) {

        int S = accumulate(rods.begin(), rods.end(), 0);

        vector<vector<int>> dp(2, vector<int> (S + 1, INT\_MIN));

        //dp[i]为高度相差i的两杆和, 转移方程由接到 短的、长的、不接 得到

        dp[0][0] = 0;

        int p = 0;

        for (auto rod : rods) {

            auto &pre = dp[p], &cur = dp[p^1];

            for (int i = 0; i <= S; ++i) {

                cur[i] = pre[i]; //不接

                cur[i] = max(cur[i], pre[abs(i-rod)] + rod);

                if (i + rod <= S)

                    cur[i] = max(cur[i], pre[i+rod] + rod);

            }

            p ^= 1;

        }

        return dp[p][0] / 2;

    }

};

//O(2^N \* N)

class Solution {

public:

    int tallestBillboard(vector<int>& rods) {

        unordered\_map<int, int> mp{{0, 0}};

        //key 差值  value 长度和

        for(auto &rod :rods) {

            unordered\_map<int, int> tmp(mp);

            for(auto &[sub, len] : tmp) {

                mp[sub + rod] = max(mp[sub + rod], len + rod);

                mp[abs(sub - rod)] = max(mp[abs(sub - rod)],len + rod);

            }

        }

        return mp[0] / 2;

    }

};

### 960. Delete Columns to Make Sorted III

Hard

We are given an array A of N lowercase letter strings, all of the same length.

Now, we may choose any set of deletion indices, and for each string, we delete all the characters in those indices.

For example, if we have an array A = ["babca","bbazb"] and deletion indices {0, 1, 4}, then the final array after deletions is ["bc","az"].

Suppose we chose a set of deletion indices D such that after deletions, the final array has **every element (row) in lexicographic** order.

For clarity, A[0] is in lexicographic order (ie. A[0][0] <= A[0][1] <= ... <= A[0][A[0].length - 1]), A[1] is in lexicographic order (ie. A[1][0] <= A[1][1] <= ... <= A[1][A[1].length - 1]), and so on.

Return the minimum possible value of D.length.

**Example 1:**

**Input:** ["babca","bbazb"]

**Output:** 3

**Explanation:** After deleting columns 0, 1, and 4, the final array is A = ["bc", "az"].

Both these rows are individually in lexicographic order (ie. A[0][0] <= A[0][1] and A[1][0] <= A[1][1]).

Note that A[0] > A[1] - the array A isn't necessarily in lexicographic order.

**Example 2:**

**Input:** ["edcba"]

**Output:** 4

**Explanation:** If we delete less than 4 columns, the only row won't be lexicographically sorted.

**Example 3:**

**Input:** ["ghi","def","abc"]

**Output:** 0

**Explanation:** All rows are already lexicographically sorted.

**Note:**

1. 1 <= A.length <= 100
2. 1 <= A[i].length <= 100

class Solution {

public:

    int minDeletionSize(vector<string>& A) {

        vector<int> dp(A[0].size(), 1);

        for (auto i = 0; i < A[0].size(); ++i) {

            for (auto j = 0; j < i; ++j) {

                bool ok = true;

                for (auto k = 0; k < A.size(); ++k) {

                    if (A[k][j] > A[k][i]) {ok = false; break;}

                }

                if (ok) dp[i] = max(dp[i], dp[j] + 1);

            }

        }

        return A[0].size() - \*max\_element(begin(dp), end(dp));

    }

};

### 964. Least Operators to Express Number

Hard

Given a single positive integer x, we will write an expression of the form x (op1) x (op2) x (op3) x ... where each operator op1, op2, etc. is either addition, subtraction, multiplication, or division (+, -, \*, or /).  For example, with x = 3, we might write 3 \* 3 / 3 + 3 - 3 which is a value of 3.

When writing such an expression, we adhere to the following conventions:

1. The division operator (/) returns rational numbers.
2. There are no parentheses placed anywhere.
3. We use the usual order of operations: multiplication and division happens before addition and subtraction.
4. It's not allowed to use the unary negation operator (-).  For example, "x - x" is a valid expression as it only uses subtraction, but "-x + x" is not because it uses negation.

We would like to write an expression with the least number of operators such that the expression equals the given target.  Return the least number of operators used.

**Example 1:**

**Input:** x = 3, target = 19 **Output:** 5

**Explanation:** 3 \* 3 + 3 \* 3 + 3 / 3. The expression contains 5 operations.

**Example 2:**

**Input:** x = 5, target = 501 **Output:** 8

5 \* 5 \* 5 \* 5 - 5 \* 5 \* 5 + 5 / 5. The expression contains 8 operations.

**Example 3:**

**Input:** x = 100, target = 100000000 **Output:** 3

**Explanation:** 100 \* 100 \* 100 \* 100. The expression contains 3 operations.

**Note:**

* 2 <= x <= 100
* 1 <= target <= 2 \* 10^8

//以加减来分割

class Solution {

public:

    unordered\_map<int, int> mp;

    int leastOpsExpressTarget(int x, int y) {

        if (mp.count(y)) return mp[y];

        if (x > y) return min(y \* 2 - 1, (x - y) \* 2);

        if (x == y) return 0;

        long long sum = x;

        int cnt = 0;

        while (sum < y) {

            ++cnt;

            sum \*= x;

        }

        if (sum == cnt)  return mp[y] = cnt;

        int a = INT\_MAX, b = INT\_MAX;

        // using subtract

        if (sum - y < y) {

            a = leastOpsExpressTarget(x, sum - y) + cnt;

        }

        // using add

        b = leastOpsExpressTarget(x, y - (sum / x)) + cnt - 1;

        return mp[y] = min(a, b) + 1;

    }

};

//转化成x进制 pos前一位加法得到  neg前一位减法得到

class Solution {

public:

    int leastOpsExpressTarget(int x, int y) {

        int k = 0, pos, neg;

        while (y > 0) {

            int cur = y % x;

            y /= x;

            if (k == 0) {

                pos = cur \* 2;

                neg = (x - cur) \* 2;

            }

            else {

                int pos2 = min(cur \* k + pos, (cur + 1) \* k + neg);

                int neg2 = min((x - cur)\*k+pos, (x-cur-1) \* k + neg);

                pos = pos2, neg = neg2;

            }

            ++k;

        }

        return min(pos, k + neg) - 1;

    }

};

### 968. Binary Tree Cameras

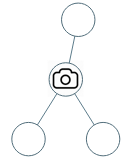
Hard

Given a binary tree, we install cameras on the nodes of the tree.

Each camera at a node can monitor **its parent, itself, and its immediate children**.

Calculate the minimum number of cameras needed to monitor all nodes of the tree.

**Example 1:**

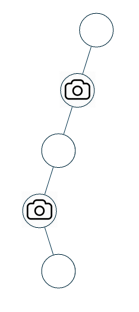


**Input:** [0,0,null,0,0]

**Output:** 1

**Explanation:** One camera is enough to monitor all nodes if placed as shown.

**Example 2:**



**Input:** [0,0,null,0,null,0,null,null,0]

**Output:** 2

**Explanation:** At least two cameras are needed to monitor all nodes of the tree. The above image shows one of the valid configurations of camera placement.  
**Note:**

1. The number of nodes in the given tree will be in the range [1, 1000].
2. **Every** node has value 0.

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     TreeNode \*left;

 \*     TreeNode \*right;

 \*     TreeNode(int x) : val(x), left(NULL), right(NULL) {}

 \* };

 \*/

class Solution {

public:

    int minCameraCover(TreeNode\* root) {

        auto p = dfs(root);

        return min(p[1], p[2]);

    }

    // 0: Strict ST; All nodes below this are covered, but not this one

    // 1: Normal ST; All nodes below and incl this are covered - no camera

    // 2: Placed camera; All nodes below this are covered, plus camera here

    array<int, 3> dfs(TreeNode\* p) {

        array<int, 3> res{0, 0, 1000};

        if (!p) return res;

        auto l = dfs(p->left), r = dfs(p->right);

        res[0] = l[1]+r[1];

        res[1] = min({l[2]+r[1], l[1]+r[2], l[2]+r[2]});

        res[2] = 1 + min({l[0], l[1], l[2]}) + min({r[0], r[1], r[2]});

        return res;

    }

};

class Solution {

public:

#define NO\_CAMERA       0

#define NOT\_NEEDED      1

#define HAS\_CAMERA      2

    int res = 0;

    int minCameraCover(TreeNode\* root) {

        return  (dfs(root) == NO\_CAMERA ? 1 : 0) + res;

    }

    int dfs(TreeNode \*p) {

        if (!p) return NOT\_NEEDED;

        int l = dfs(p->left), r = dfs(p->right);

        if (l == NO\_CAMERA || r == NO\_CAMERA) {

            ++res;

            return HAS\_CAMERA;

        }

        else if (l == HAS\_CAMERA || r == HAS\_CAMERA) {

            return NOT\_NEEDED;

        }

        else return NO\_CAMERA;

    }

};

### 975. Odd Even Jump

Hard

You are given an integer array A. From some starting index, you can make a series of jumps. The (1st, 3rd, 5th, ...) jumps in the series are called **odd-numbered jumps**, and the (2nd, 4th, 6th, ...) jumps in the series are called **even-numbered jumps**. Note that the **jumps** are numbered, not the indices.

You may jump forward from index i to index j (with i < j) in the following way:

* During **odd-numbered jumps** (i.e., jumps 1, 3, 5, ...), you jump to the index j such that A[i] <= A[j] and A[j] is the smallest possible value. If there are multiple such indices j, you can only jump to the **smallest** such index j.
* During **even-numbered jumps** (i.e., jumps 2, 4, 6, ...), you jump to the index j such that A[i] >= A[j] and A[j] is the largest possible value. If there are multiple such indices j, you can only jump to the **smallest** such index j.
* It may be the case that for some index i, there are no legal jumps.

A starting index is **good** if, starting from that index, you can reach the end of the array (index A.length - 1) by jumping some number of times (possibly 0 or more than once).

Return *the number of****good****starting indices*.

**Example 1:**

**Input:** A = [10,13,12,14,15]

**Output:** 2

**Explanation:**

From starting index i = 0, we can make our 1st jump to i = 2 (since A[2] is the smallest among A[1], A[2], A[3],

A[4] that is greater or equal to A[0]), then we cannot jump any more.

From starting index i = 1 and i = 2, we can make our 1st jump to i = 3, then we cannot jump any more.

From starting index i = 3, we can make our 1st jump to i = 4, so we have reached the end.

From starting index i = 4, we have reached the end already.

In total, there are 2 different starting indices i = 3 and i = 4, where we can reach the end with some number of

jumps.

**Example 2:**

**Input:** A = [2,3,1,1,4]

**Output:** 3

**Explanation:**

From starting index i = 0, we make jumps to i = 1, i = 2, i = 3:

During our 1st jump (odd-numbered), we first jump to i = 1 because A[1] is the smallest value in [A[1], A[2],

A[3], A[4]] that is greater than or equal to A[0].

During our 2nd jump (even-numbered), we jump from i = 1 to i = 2 because A[2] is the largest value in [A[2], A[3],

A[4]] that is less than or equal to A[1]. A[3] is also the largest value, but 2 is a smaller index, so we can

only jump to i = 2 and not i = 3

During our 3rd jump (odd-numbered), we jump from i = 2 to i = 3 because A[3] is the smallest value in [A[3], A[4]]

that is greater than or equal to A[2].

We can't jump from i = 3 to i = 4, so the starting index i = 0 is not good.

In a similar manner, we can deduce that:

From starting index i = 1, we jump to i = 4, so we reach the end.

From starting index i = 2, we jump to i = 3, and then we can't jump anymore.

From starting index i = 3, we jump to i = 4, so we reach the end.

From starting index i = 4, we are already at the end.

In total, there are 3 different starting indices i = 1, i = 3, and i = 4, where we can reach the end with some

number of jumps.

**Example 3:**

**Input:** A = [5,1,3,4,2]

**Output:** 3

**Explanation:**

We can reach the end from starting indices 1, 2, and 4.

**Constraints:**

* 1 <= A.length <= 2 \* 104
* 0 <= A[i] < 105

class Solution {

public:

    int oddEvenJumps(vector<int>& A) {

        int n = A.size(), res = 1;

        vector<int> index(n);

        iota(index.begin(), index.end(), 0);

        auto cmp\_odd = [&A](const int x, const int y){

            return A[x] < A[y] || A[x] == A[y] && x < y;

        };

        auto cmp\_even = [&A](const int x, const int y){

            return A[x] > A[y] || A[x] == A[y] && x < y;

        };

        auto find\_next = [n](vector<int>& v) -> vector<int> {

            vector<int> next(n);

            stack<int> stk;

            for (auto i: v) {

                while (!stk.empty() && stk.top() < i) {

                    next[stk.top()] = i;

                    stk.pop();

                }

                stk.push(i);

            }

            return next;

        };

        sort(index.begin(), index.end(), cmp\_odd);

        vector<int> odd\_next = find\_next(index);

        sort(index.begin(), index.end(), cmp\_even);

        vector<int> even\_next = find\_next(index);

        vector<int> odd(n), even(n);

        even[n-1] = odd[n-1] = true;

        for (int i = n-2; i >= 0; i--) {

            if (odd\_next[i]) odd[i] = even[odd\_next[i]];

            if (even\_next[i]) even[i] = odd[even\_next[i]];

            if (odd[i]) ++res;

        }

        return res;

    }

};

class Solution {

public:

    int oddEvenJumps(vector<int>& A) {

        int n  = A.size(), res = 1;

        vector<bool> higher(n), lower(n);

        higher[n - 1] = lower[n - 1] = true;

        map<int, int> mp;

        mp[A[n - 1]] = n - 1;

        for (int i = n - 2; i >= 0; --i) {

            auto [hi, lo] = mp.equal\_range(A[i]);

            if (hi != mp.end()) higher[i] = lower[hi->second];

            if (lo != mp.begin()) lower[i] = higher[prev(lo)->second];

            if (higher[i]) res++;

            mp[A[i]] = i;

        }

        return res;

    }

};

### 980. Unique Paths III

Hard

On a 2-dimensional grid, there are 4 types of squares:

* 1 represents the starting square.  There is exactly one starting square.
* 2 represents the ending square.  There is exactly one ending square.
* 0 represents empty squares we can walk over.
* -1 represents obstacles that we cannot walk over.

Return the number of 4-directional walks from the starting square to the ending square, that **walk over every non-obstacle square exactly once**.

**Example 1:**

**Input:** [[1,0,0,0],[0,0,0,0],[0,0,2,-1]]

**Output:** 2

**Explanation:** We have the following two paths:

1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2)

2. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2)

**Example 2:**

**Input:** [[1,0,0,0],[0,0,0,0],[0,0,0,2]]

**Output:** 4

**Explanation:** We have the following four paths:

1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2),(2,3)

2. (0,0),(0,1),(1,1),(1,0),(2,0),(2,1),(2,2),(1,2),(0,2),(0,3),(1,3),(2,3)

3. (0,0),(1,0),(2,0),(2,1),(2,2),(1,2),(1,1),(0,1),(0,2),(0,3),(1,3),(2,3)

4. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2),(2,3)

**Example 3:**

**Input:** [[0,1],[2,0]]

**Output:** 0

**Explanation:**

There is no path that walks over every empty square exactly once.

Note that the starting and ending square can be anywhere in the grid.

**Note:**

1. 1 <= grid.length \* grid[0].length <= 20

class Solution {

public:

    const int dirs[5] = {1, 0, -1, 0, 1};

    int n, m;

    map<pair<int, int>, int> dp;

    bool inside(int x, int y) {

        return x >= 0 && y >= 0 && x < n && y < m;

    }

    int dfs(vector<vector<int>>& g, int pos, int steps, int state) {

        int x = pos/m, y = pos%m, ret = 0;

        if (g[x][y] == 2) return 1 == steps;

        if (dp.count({pos, state})) return dp[{pos, state}];

        g[x][y] = -1;

        for (int k = 0; k < 4; ++k) {

            int xx = x + dirs[k], yy = y + dirs[k+1];

            if (!inside(xx, yy) || g[xx][yy] == -1) continue;

            int newPos = xx\*m+yy;

            ret += dfs(g, newPos, steps-1, state|(1<<newPos));

        }

        g[x][y] = 0;

        return dp[{pos, state}] = ret;

    }

    int uniquePathsIII(vector<vector<int>>& g) {

        n = g.size(), m = g[0].size();

        int pos, steps = 0;

        for (auto i = 0; i < n; ++i) {

            for (auto j = 0; j < m; ++j) {

                if (g[i][j] == 1) pos = i\*m + j;

                if (g[i][j] != -1) ++steps;

            }

        }

        return dfs(g, pos, steps, 0);

    }

};

### 992. Subarrays with K Different Integers

Hard

Given an array A of positive integers, call a (contiguous, not necessarily distinct) subarray of A *good* if the number of different integers in that subarray is exactly K.

(For example, [1,2,3,1,2] has 3 different integers: 1, 2, and 3.)

Return the number of good subarrays of A.

**Example 1:**

**Input:** A = [1,2,1,2,3], K = 2

**Output:** 7

**Explanation:** Subarrays formed with exactly 2 different integers: [1,2], [2,1], [1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2].

**Example 2:**

**Input:** A = [1,2,1,3,4], K = 3

**Output:** 3

**Explanation:** Subarrays formed with exactly 3 different integers: [1,2,1,3], [2,1,3], [1,3,4].

**Note:**

1. 1 <= A.length <= 20000
2. 1 <= A[i] <= A.length
3. 1 <= K <= A.length

class Solution {

public:

    int subarraysWithKDistinct(vector<int>& A, int K) {

        return atMostK(A, K) - atMostK(A, K-1);

    }

    int atMostK(vector<int> &A, int K) {

        if (K == 0) return 0;

        int i = 0, j = 0;

        int n = A.size(), ret = 0;

        unordered\_map<int, int> mp;

        while (j < n) {

            while (i < n && mp.size() + !(mp.count(A[i])) <= K) {

                ++mp[A[i++]];

            }

            ret += i-j;

            if (!--mp[A[j]]) mp.erase(A[j]);

            ++j;

        }

        return ret;

    }

};

### 995. Minimum Number of K Consecutive Bit Flips

Hard

In an array A containing only 0s and 1s, a *K-bit flip*consists of choosing a (contiguous) subarray of length K and simultaneously changing every 0 in the subarray to 1, and every 1 in the subarray to 0.

Return the minimum number of K-bit flips required so that there is no 0 in the array.  If it is not possible, return -1.

**Example 1:**

**Input:** A = [0,1,0], K = 1

**Output:** 2

**Explanation:** Flip A[0], then flip A[2].

**Example 2:**

**Input:** A = [1,1,0], K = 2

**Output:** -1

**Explanation:** No matter how we flip subarrays of size 2, we can't make the array become [1,1,1].

**Example 3:**

**Input:** A = [0,0,0,1,0,1,1,0], K = 3

**Output:** 3

**Explanation:**

Flip A[0],A[1],A[2]: A becomes [1,1,1,1,0,1,1,0]

Flip A[4],A[5],A[6]: A becomes [1,1,1,1,1,0,0,0]

Flip A[5],A[6],A[7]: A becomes [1,1,1,1,1,1,1,1]

**Note:**

1. 1 <= A.length <= 30000
2. 1 <= K <= A.length

class Solution {

public:

    int minKBitFlips(vector<int>& A, int K) {

        deque<int> deq;

        int res = 0, n = A.size();

        for (int i = 0; i < n; ++i) {

            if (!deq.empty() && i-deq.front() >= K) {

                deq.pop\_front();

            }

            if (!(A[i] ^ (deq.size() % 2))) {

                if (n-i < K) return -1;

                deq.push\_back(i);

                ++res;

            }

        }

        return res;

    }

};

### 996. Number of Squareful Arrays

Hard

Given an array A of non-negative integers, the array is *squareful* if for every pair of adjacent elements, their sum is a perfect square.

Return the number of permutations of A that are squareful.  Two permutations A1 and A2 differ if and only if there is some index i such that A1[i] != A2[i].

**Example 1:**

**Input:** [1,17,8]

**Output:** 2

**Explanation:**

[1,8,17] and [17,8,1] are the valid permutations.

**Example 2:**

**Input:** [2,2,2]

**Output:** 1

**Note:**

1. 1 <= A.length <= 12
2. 0 <= A[i] <= 1e9

class Solution {

public:

    int f(int x) {return x == 1 ? 1 : x\*f(x-1);}

    int numSquarefulPerms(vector<int>& A) {

        int n = A.size(), m = 1 << n, res = 0;

        vector<vector<int>> dp(n, vector<int> (m, 0));

        for (int i = 0; i < n; ++i) dp[i][1<<i] = 1;

        vector<vector<int>> g(n);

        for (int i = 0; i < n; ++i) {

            for (int j = 0; j < i; ++j) {

                int Sqrt = sqrt(A[i] + A[j]);

                if (Sqrt\*Sqrt == A[i] + A[j]) {

                    g[i].push\_back(j);

                    g[j].push\_back(i);

                }

            }

        }

         for (int s = 0; s < m; ++s) {

            for (int i = 0; i < n; ++i) if (s&(1<<i)) {

                for (int j : g[i]) if (s&(1<<j)) {

                    dp[i][s] += dp[j][s^(1<<i)];

                }

            }

        }

        for (int i = 0; i < n; ++i) res += dp[i].back();

        map<int, int> mp;

        for (auto i : A) ++mp[i];

        for (auto i : mp) res /= f(i.second);

        return res;

    }

};

### 1000. Minimum Cost to Merge Stones

Hard

There are N piles of stones arranged in a row.  The i-th pile has stones[i] stones.

A *move* consists of merging **exactly K consecutive** piles into one pile, and the cost of this move is equal to the total number of stones in these K piles.

Find the minimum cost to merge all piles of stones into one pile.  If it is impossible, return -1.

**Example 1:**

**Input:** stones = [3,2,4,1], K = 2

**Output:** 20

**Explanation:**

We start with [3, 2, 4, 1].

We merge [3, 2] for a cost of 5, and we are left with [5, 4, 1].

We merge [4, 1] for a cost of 5, and we are left with [5, 5].

We merge [5, 5] for a cost of 10, and we are left with [10].

The total cost was 20, and this is the minimum possible.

**Example 2:**

**Input:** stones = [3,2,4,1], K = 3

**Output:** -1

**Explanation:** After any merge operation, there are 2 piles left, and we can't merge anymore. So the task is impossible.

**Example 3:**

**Input:** stones = [3,5,1,2,6], K = 3

**Output:** 25

**Explanation:**

We start with [3, 5, 1, 2, 6].

We merge [5, 1, 2] for a cost of 8, and we are left with [3, 8, 6].

We merge [3, 8, 6] for a cost of 17, and we are left with [17].

The total cost was 25, and this is the minimum possible.

**Note:**

* 1 <= stones.length <= 30
* 2 <= K <= 30
* 1 <= stones[i] <= 100

class Solution {

public:

    int mergeStones(vector<int>& stones, int K) {

        int n = stones.size();

        if ((n - 1) % (K - 1)) return -1;

        vector<int> preS{0};

        partial\_sum(stones.begin(), stones.end(), back\_inserter(preS));

        vector<vector<int> > dp(n, vector<int>(n));

        for (int len = K; len <= n; ++len) {

            for (int i = 0, j = len-1; j < n; ++i, ++j) {

                dp[i][j] = INT\_MAX;

                for (int mid = i; mid < j; mid += K - 1)

                    dp[i][j] = min(dp[i][mid]+dp[mid+1][j], dp[i][j]);

                if ((j - i) % (K - 1) == 0)

                    dp[i][j] += preS[j + 1] - preS[i];

            }

        }

        return dp[0][n - 1];

    }

};

### 1001. Grid Illumination

Hard

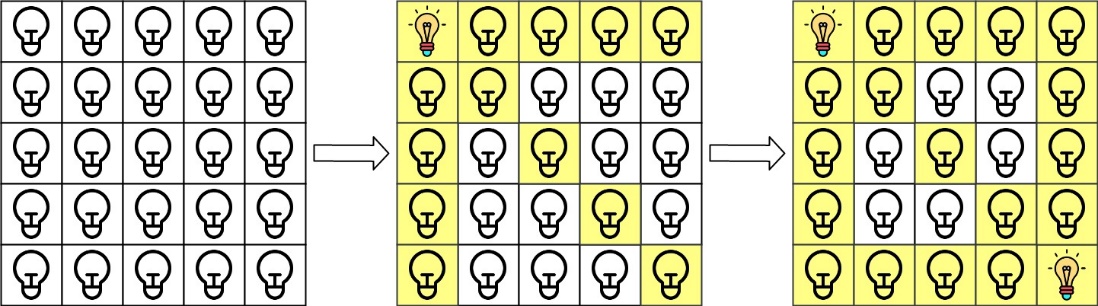
You are given a grid of size N x N, and each cell of this grid has a lamp that is initially **turned off**.

You are also given an array of lamp positions lamps, where lamps[i] = [rowi, coli] indicates that the lamp at grid[rowi][coli] is **turned on**. When a lamp is turned on, it **illuminates its cell** and **all other cells** in the same **row, column, or diagonal**.

Finally, you are given a query array queries, where queries[i] = [rowi, coli]. For the ith query, determine whether grid[rowi][coli] is illuminated or not. After answering the ith query, **turn off** the lamp at grid[rowi][coli] and its **8 adjacent lamps** if they exist. A lamp is adjacent if its cell shares either a side or corner with grid[rowi][coli].

Return *an array of integers*ans*, where*ans[i]*should be*1*if the lamp in the*ith*query was illuminated, or*0*if the lamp was not.*

**Example 1:**

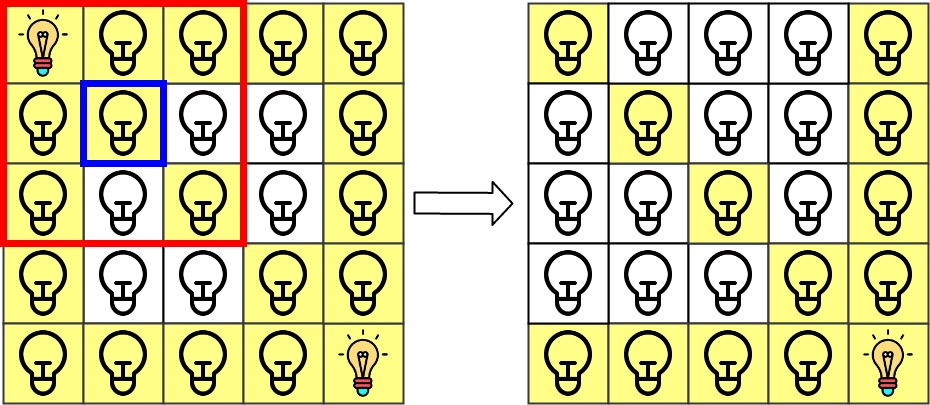


**Input:** N = 5, lamps = [[0,0],[4,4]], queries = [[1,1],[1,0]]

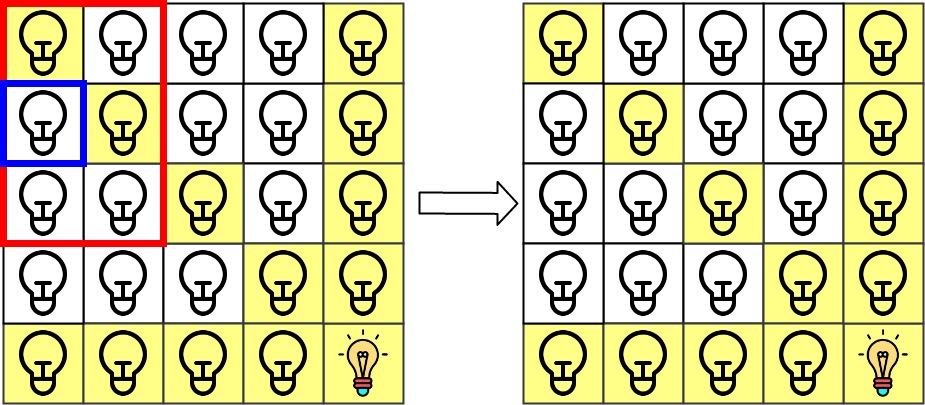
**Output:** [1,0]

**Explanation:** We have the initial grid with all lamps turned off. In the above picture we see the grid after turning on the lamp at grid[0][0] then turning on the lamp at grid[4][4].

The 0th query asks if the lamp at grid[1][1] is illuminated or not (the blue square). It is illuminated, so set ans[0] = 1. Then, we turn off all lamps in the red square.



The 1st query asks if the lamp at grid[1][0] is illuminated or not (the blue square). It is not illuminated, so set ans[1] = 1. Then, we turn off all lamps in the red rectangle.



**Example 2:**

**Input:** N = 5, lamps = [[0,0],[4,4]], queries = [[1,1],[1,1]]

**Output:** [1,1]

**Example 3:**

**Input:** N = 5, lamps = [[0,0],[0,4]], queries = [[0,4],[0,1],[1,4]]

**Output:** [1,1,0]

**Constraints:**

* 1 <= N <= 109
* 0 <= lamps.length <= 20000
* lamps[i].length == 2
* 0 <= lamps[i][j] < N
* 0 <= queries.length <= 20000
* queries[i].length == 2
* 0 <= queries[i][j] < N

class Solution {

public:

    const int dirs[10] = {0,0,1,0,-1,-1,1,1,-1,0};

    vector<int> gridIllumination(int N, vector<vector<int>>& lamps, vector<vector<int>>& queries) {

        map<int, int> C, R, D0, D1;

        auto lightOpt = [&C, &R, &D0, &D1, &N](int x, int y, int on) {

            C[x] += on;

            R[y] += on;

            D0[x+y] += on;

            D1[x-y] += on;

        };

        unordered\_set<long long> st;

        for (auto &i : lamps) {

            long long id = 1ll\*i[0]\*N+i[1];

            st.insert(id);

            lightOpt(i[0], i[1], 1);

        }

        vector<int> res;

        for (auto &i : queries) {

            if (C[i[0]] || R[i[1]] || D0[i[0]+i[1]] || D1[i[0]-i[1]])

                res.push\_back(1);

            else {

                res.push\_back(0);

            }

            for (int k = 0; k < 9; ++k) {

                int x = i[0] + dirs[k];

                int y = i[1] + dirs[k+1];

                if (x < 0 || y < 0 || x >= N || y >= N) continue;

                long long id = 1ll\*x\*N+y;

                if (!st.count(id)) continue;

                st.erase(id);

                lightOpt(x, y, -1);

            }

        }

        return res;

    }

};

### 1012. Numbers With Repeated Digits

Hard

Given a positive integer N, return the number of positive integers less than or equal to N that have at least 1 repeated digit.

**Example 1:**

**Input:** 20

**Output:** 1

**Explanation:** The only positive number (<= 20) with at least 1 repeated digit is 11.

**Example 2:**

**Input:** 100

**Output:** 10

**Explanation:** The positive numbers (<= 100) with atleast 1 repeated digit are 11, 22, 33, 44, 55, 66, 77, 88, 99, and 100.

**Example 3:**

**Input:** 1000

**Output:** 262

**Note:**

1. 1 <= N <= 10^9

class Solution {

public:

    int P(int n, int m) {

        int res = 1;

        while (m--) res \*= n--;

        return res;

    }

    int numDupDigitsAtMostN(int N) {

        if(N < 10) return 0;

        string str = to\_string(N);

        int k = str.size();

        vector<int> noDupRes(k, 0);

        int noDupBaseSum = 0;

        for(int i = 0; i < k - 1; i++) {

            noDupBaseSum += 9\*P(9, i);

        }

        int count[10] = {0};

        bool duplicate = false;

        for (int i = 0; i < k; i++) {

            int c = str[i]-'0';

            noDupRes[i] = i == 0 ? 9 : noDupRes[i - 1] \* (10 - i);

            if (!duplicate) {

                int diff = 0;

                for (int j = c + 1; j < 10; j++) diff += count[j] == 0;

                noDupRes[i] -= diff;

                count[c]++;

                if (count[c] > 1) duplicate = true;

            }

        }

        return N - (noDupBaseSum + noDupRes[k - 1]);

    }

};

### 1028. Recover a Tree From Preorder Traversal

Hard

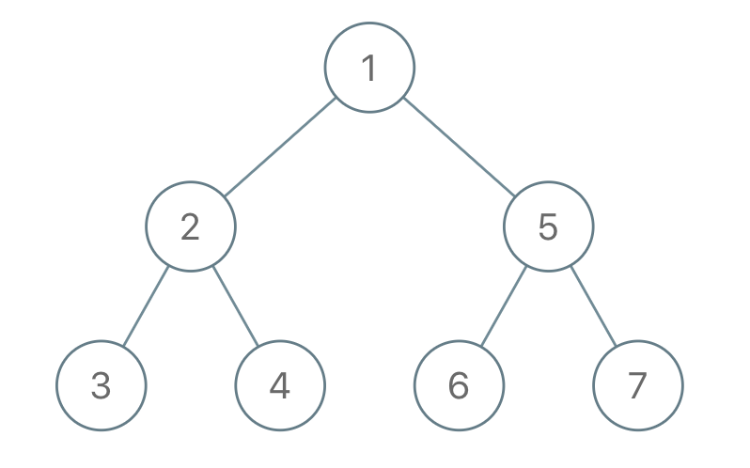
We run a preorder depth-first search (DFS) on the root of a binary tree.

At each node in this traversal, we output D dashes (where D is the depth of this node), then we output the value of this node.  If the depth of a node is D, the depth of its immediate child is D + 1.  The depth of the root node is 0.

If a node has only one child, that child is guaranteed to be **the left child**.

Given the output S of this traversal, recover the tree and return *its* root.

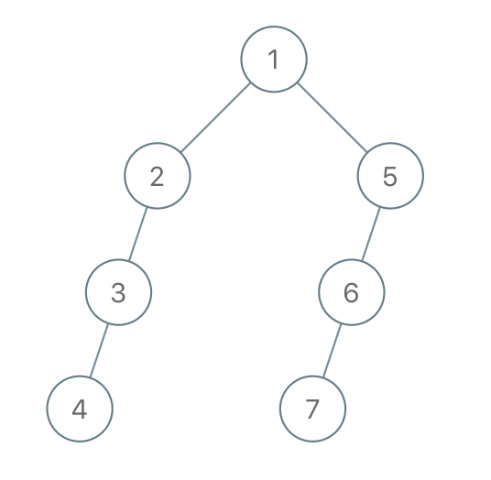
**Example 1:**



**Input:** S = "1-2--3--4-5--6--7"

**Output:** [1,2,5,3,4,6,7]

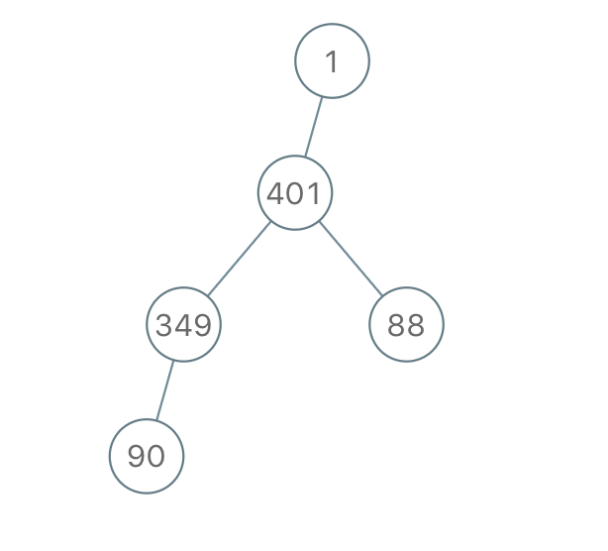
**Example 2:**



**Input:** S = "1-2--3---4-5--6---7"

**Output:** [1,2,5,3,null,6,null,4,null,7]

**Example 3:**



**Input:** S = "1-401--349---90--88"

**Output:** [1,401,null,349,88,90]

**Constraints:**

* The number of nodes in the original tree is in the range [1, 1000].
* 1 <= Node.val <= 109

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     TreeNode \*left;

 \*     TreeNode \*right;

 \*     TreeNode() : val(0), left(nullptr), right(nullptr) {}

 \*     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

 \*     TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

 \* };

 \*/

class Solution {

public:

    TreeNode\* recoverFromPreorder(string s) {

        stringstream ss(s);

        vector<pair<TreeNode\*, int>> stk{{new TreeNode(-1), -1}};

        while (ss.peek() != EOF) {

            int cnt = 0, val;

            while (ss.peek() == '-') {

                ++cnt;

                ss.get();

            }

            ss >> val;

            while (stk.back().second >= cnt) stk.pop\_back();

            auto &p = stk.back().first->left ? stk.back().first->right

: stk.back().first->left;

            stk.emplace\_back(p = new TreeNode(val), cnt);

        }

        return stk[1].first;

    }

};

### 1032. Stream of Characters

Hard

Implement the StreamChecker class as follows:

* StreamChecker(words): Constructor, init the data structure with the given words.
* query(letter): returns true if and only if for some k >= 1, the last k characters queried (in order from oldest to newest, including this letter just queried) spell one of the words in the given list.

**Example:**

StreamChecker streamChecker = new StreamChecker(["cd","f","kl"]); // init the dictionary.

streamChecker.query('a'); // return false

streamChecker.query('b'); // return false

streamChecker.query('c'); // return false

streamChecker.query('d'); // return true, because 'cd' is in the wordlist

streamChecker.query('e'); // return false

streamChecker.query('f'); // return true, because 'f' is in the wordlist

streamChecker.query('g'); // return false

streamChecker.query('h'); // return false

streamChecker.query('i'); // return false

streamChecker.query('j'); // return false

streamChecker.query('k'); // return false

streamChecker.query('l'); // return true, because 'kl' is in the wordlist

**Note:**

* 1 <= words.length <= 2000
* 1 <= words[i].length <= 2000
* Words will only consist of lowercase English letters.
* Queries will only consist of lowercase English letters.
* The number of queries is at most 40000.

/\*\*

 \* Your StreamChecker object will be instantiated and called as such:

 \* StreamChecker\* obj = new StreamChecker(words);

 \* bool param\_1 = obj->query(letter);

 \*/

class Trie {

public:

    void insert(const string &s) {

        auto p = this;

        for (int i = s.size()-1; i >= 0; --i) {

            if (!p->mp.count(s[i])) p->mp[s[i]] = new Trie();

            p = p->mp[s[i]];

        }

        p->isWord = true;

    }

    bool search(const string &s) {

        auto p = this;

        for (int i = s.size()-1; i >= 0; --i) {

            if (p->isWord) return true;

            if (!p->mp.count(s[i])) return false;

            p = p->mp[s[i]];

        }

        return p->isWord;

    }

    unordered\_map<char, Trie\*> mp;

    bool isWord = false;

};

class StreamChecker {

public:

    Trie\* trie = new Trie();

    string s;

    StreamChecker(vector<string>& words) {

        for (auto &s : words)

            trie->insert(s);

    }

    bool query(char letter) {

        s += letter;

        return trie->search(s);

    }

};

### 1036. Escape a Large Maze

Hard

In a 1 million by 1 million grid, the coordinates of each grid square are (x, y).

We start at the source square and want to reach the target square.  Each move, we can walk to a 4-directionally adjacent square in the grid that isn't in the given list of blocked squares.

Return true if and only if it is possible to reach the target square through a sequence of moves.

**Example 1:**

**Input:** blocked = [[0,1],[1,0]], source = [0,0], target = [0,2]

**Output:** false

**Explanation:** The target square is inaccessible starting from the source square, because we can't walk outside the grid.

**Example 2:**

**Input:** blocked = [], source = [0,0], target = [999999,999999]

**Output:** true

**Explanation:** Because there are no blocked cells, it's possible to reach the target square.

**Constraints:**

* 0 <= blocked.length <= 200
* blocked[i].length == 2
* 0 <= blocked[i][j] < 10^6
* source.length == target.length == 2
* 0 <= source[i][j], target[i][j] < 10^6
* source != target

### 1044. Longest Duplicate Substring

Hard

Given a string s, consider all *duplicated substrings*: (contiguous) substrings of s that occur 2 or more times. The occurrences may overlap.

Return **any** duplicated substring that has the longest possible length. If s does not have a duplicated substring, the answer is "".

**Example 1:**

**Input:** s = "banana"

**Output:** "ana"

**Example 2:**

**Input:** s = "abcd"

**Output:** ""

**Constraints:**

* 2 <= s.length <= 3 \* 104
* s consists of lowercase English letters.

class RollingHash {

    using ul = unsigned long;

    ul p = 31,  m = 1e9 + 7;

    vector<ul> hashMap, pPower;

public:

    RollingHash(const string &s) {

        pPower.resize(s.size()+1, 1);

        hashMap.resize(s.size()+1);

        for (int i = 1; i < hashMap.size(); ++i) {

            pPower[i] = (pPower[i-1] \* p) % m;

            hashMap[i]= (hashMap[i-1]\*p + s[i-1]) % m;

        }

    }

    ul getHashCode(int i, int j) {

        if (i == 0) return hashMap[j];

        return (hashMap[j] + m - (hashMap[i]\*pPower[j-i])%m) % m;

    }

};

class Solution {

    using ul = unsigned long;

    int dupSubstrK(RollingHash &rs, string &s, int k) {

        unordered\_multimap<ul, int> mp;

        for (int i = 0; i + k <= s.size(); ++i) {

            ul key = rs.getHashCode(i, i+k);

            auto range = mp.equal\_range(key);

            for (auto it = range.first; it != range.second; ++it) {

                if (s.substr(it->second, k) == s.substr(i, k)) {

                    return i;

                }

            }

            mp.emplace(key, i);

        }

        return -1;

    }

public:

    string longestDupSubstring(string S) {

        RollingHash rs(S);

        int start = 0;

        int left = 0, right = S.size();

        while (left < right) {

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

            int tmp = dupSubstrK(rs, S, mid);

            if (tmp != -1) {

                left = mid+1;

                start = tmp;

            }

            else right = mid;

        }

        return S.substr(start, left-1);

    }

};

### 1074. Number of Submatrices That Sum to Target

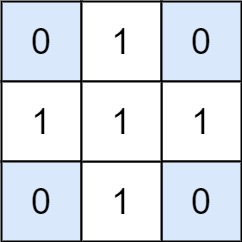
Hard

Given a matrix and a target, return the number of non-empty submatrices that sum to target.

A submatrix x1, y1, x2, y2 is the set of all cells matrix[x][y] with x1 <= x <= x2 and y1 <= y <= y2.

Two submatrices (x1, y1, x2, y2) and (x1', y1', x2', y2') are different if they have some coordinate that is different: for example, if x1 != x1'.

**Example 1:**



**Input:** matrix = [[0,1,0],[1,1,1],[0,1,0]], target = 0

**Output:** 4

**Explanation:** The four 1x1 submatrices that only contain 0.

**Example 2:**

**Input:** matrix = [[1,-1],[-1,1]], target = 0

**Output:** 5

**Explanation:** The two 1x2 submatrices, plus the two 2x1 submatrices, plus the 2x2 submatrix.

**Example 3:**

**Input:** matrix = [[904]], target = 0

**Output:** 0

**Constraints:**

* 1 <= matrix.length <= 100
* 1 <= matrix[0].length <= 100
* -1000 <= matrix[i] <= 1000
* -10^8 <= target <= 10^8

class Solution {

public:

    int numSubmatrixSumTarget(vector<vector<int>>& matrix, int target){

        int n = matrix.size(), m = matrix[0].size(), res = 0;

        vector<vector<int>> v{vector<int> (m, 0)};

        for (auto &t : matrix) {

            v.push\_back(v.back());

            for (int i = 0; i < m; ++i) {

                v.back()[i] += t[i];

            }

        }

        for (int i = 0; i <= n; ++i) {

            for (int j = 0; j < i; ++j) {

                int sum = 0;

                unordered\_map<int, int> mp{{0, 1}};

                for (int k = 0; k < m; ++k) {

                    sum += v[i][k] - v[j][k];

                    if (mp.count(sum-target)) res += mp[sum-target];

                    ++mp[sum];

                }

            }

        }

        return res;

    }

};

### 1092. Shortest Common Supersequence

Hard

Given two strings str1 and str2, return the shortest string that has both str1 and str2 as subsequences.  If multiple answers exist, you may return any of them.

*(A string S is a subsequence of string T if deleting some number of characters from T (possibly 0, and the characters are chosen anywhere from T) results in the string S.)*

**Example 1:**

**Input:** str1 = "abac", str2 = "cab"

**Output:** "cabac"

**Explanation:**

str1 = "abac" is a subsequence of "cabac" because we can delete the first "c".

str2 = "cab" is a subsequence of "cabac" because we can delete the last "ac".

The answer provided is the shortest such string that satisfies these properties.

**Note:**

1. 1 <= str1.length, str2.length <= 1000
2. str1 and str2 consist of lowercase English letters.

class Solution {

public:

    struct node {

        node(int len, int x, int y, char c) : len(len), x(x), y(y), c(c) {}

        node() {}

        int len, x, y;

        char c;

        bool operator < (const node& rhs) const {

            return len < rhs.len;

        }

    };

    string shortestCommonSupersequence(string a, string b) {

        int n = a.size(), m = b.size();

        vector<vector<node>> dp(n+1, vector<node> (m+1));

        for (int i = 1; i <= n; ++i)

 dp[i][0] = node(i, i-1, 0, a[i-1]);

        for (int j = 1; j <= m; ++j)

 dp[0][j] = node(j, 0, j-1, b[j-1]);

        for (int i = 1; i <= n; ++i) {

            for (int j = 1; j <= m; ++j) {

                if (a[i-1] == b[j-1])

                    dp[i][j] = {dp[i-1][j-1].len + 1, i-1, j-1,a[i-1]};

                else if (dp[i][j-1] < dp[i-1][j])

                    dp[i][j] = {dp[i][j-1].len + 1, i, j-1, b[j-1]};

                else

                    dp[i][j] = {dp[i-1][j].len + 1, i-1, j, a[i-1]};

            }

        }

        string res;

        int x = n, y = m;

        while (x || y) {

            res += dp[x][y].c;

            int xx = dp[x][y].x;

            y = dp[x][y].y;

            x = xx;

        }

        reverse(res.begin(), res.end());

        return res;

    }

};

### 1095. Find in Mountain Array

Hard

*(This problem is an****interactive problem****.)*

You may recall that an array A is a *mountain array* if and only if:

* A.length >= 3
* There exists some i with 0 < i < A.length - 1 such that:
  + A[0] < A[1] < ... A[i-1] < A[i]
  + A[i] > A[i+1] > ... > A[A.length - 1]

Given a mountain array mountainArr, return the **minimum** index such that mountainArr.get(index) == target.  If such an index doesn't exist, return -1.

**You can't access the mountain array directly.**  You may only access the array using a MountainArray interface:

* MountainArray.get(k) returns the element of the array at index k (0-indexed).
* MountainArray.length() returns the length of the array.

Submissions making more than 100 calls to MountainArray.get will be judged *Wrong Answer*.  Also, any solutions that attempt to circumvent the judge will result in disqualification.

**Example 1:**

**Input:** array = [1,2,3,4,5,3,1], target = 3

**Output:** 2

**Explanation:** 3 exists in the array, at index=2 and index=5. Return the minimum index, which is 2.

**Example 2:**

**Input:** array = [0,1,2,4,2,1], target = 3

**Output:** -1

**Explanation:** 3 does not exist in the array, so we return -1.

**Constraints:**

* 3 <= mountain\_arr.length() <= 10000
* 0 <= target <= 10^9
* 0 <= mountain\_arr.get(index) <= 10^9

class Solution {

public:

    int n;

    int findPeak(MountainArray &mountainArr) {

        int low = 0, high = n-1;

        while (low < high) {

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

            int val = mountainArr.get(mid);

            int next = mountainArr.get(mid+1);

            if (next > val) low = mid+1;

            else high = mid;

        }

        return low;

    }

    int findVal(int target, int low, int high, MountainArray &mountainArr, bool up) {

        while (low < high) {

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

            int midVal = mountainArr.get(mid);

            if (midVal == target) return mid;

            else if (!up ^ (midVal > target)) high = mid;

            else low = mid+1;

        }

        return -1;

    }

    int findInMountainArray(int target, MountainArray &mountainArr) {

        n = mountainArr.length();

        int peak = findPeak(mountainArr);

        int pos = findVal(target, 0, peak, mountainArr, true);

        if (pos != -1) return pos;

        return findVal(target, peak, n, mountainArr, false);

    }

};

### 1096. Brace Expansion II

Hard

Under a grammar given below, strings can represent a set of lowercase words.  Let's use R(expr) to denote the **set** of words the expression represents.

Grammar can best be understood through simple examples:

* Single letters represent a singleton set containing that word.
  + R("a") = {"a"}
  + R("w") = {"w"}
* When we take a comma delimited list of 2 or more expressions, we take the union of possibilities.
  + R("{a,b,c}") = {"a","b","c"}
  + R("{{a,b},{b,c}}") = {"a","b","c"} (notice the final set only contains each word at most once)
* When we concatenate two expressions, we take the set of possible concatenations between two words where the first word comes from the first expression and the second word comes from the second expression.
  + R("{a,b}{c,d}") = {"ac","ad","bc","bd"}
  + R("a{b,c}{d,e}f{g,h}") = {"abdfg", "abdfh", "abefg", "abefh", "acdfg", "acdfh", "acefg", "acefh"}

Formally, the 3 rules for our grammar:

* For every lowercase letter x, we have R(x) = {x}
* For expressions e\_1, e\_2, ... , e\_k with k >= 2, we have R({e\_1,e\_2,...}) = R(e\_1) ∪ R(e\_2) ∪ ...
* For expressions e\_1 and e\_2, we have R(e\_1 + e\_2) = {a + b for (a, b) in R(e\_1) × R(e\_2)}, where + denotes concatenation, and × denotes the cartesian product.

Given an expression representing a set of words under the given grammar, return the sorted list of words that the expression represents.

**Example 1:**

**Input:** "{a,b}{c,{d,e}}"

**Output:** ["ac","ad","ae","bc","bd","be"]

**Example 2:**

**Input:** "{{a,z},a{b,c},{ab,z}}"

**Output:** ["a","ab","ac","z"]

**Explanation:** Each distinct word is written only once in the final answer.

**Constraints:**

1. 1 <= expression.length <= 60
2. expression[i] consists of '{', '}', ','or lowercase English letters.
3. The given expression represents a set of words based on the grammar given in the description.

class Solution {

public:

    string getWord(const string& expr, int& i) {

        string res;

        while ('a' <= expr[i] && expr[i] <= 'z')

            res += expr[i++];

        return res;

    }

    unordered\_set<string> dfs(const string& expr, int& i) {

        unordered\_set<string> res, S;

        while (i < expr.length() && expr[i] != '}') {

            if (expr[i] == ',') {

                for (auto& i : S) res.insert(move(i));

                ++i;

                S.clear();

            } else {

                unordered\_set<string> t;

                if (expr[i] == '{') t = dfs(expr, ++i);

                else t.insert(getWord(expr, i));

                if (S.empty()) S = std::move(t);

                else {

                    unordered\_set<string> S2;

                    for (auto& i : S) {

                        for (auto& j : t)

                            S2.insert(i + j);

                    }

                    S = std::move(S2);

                }

            }

        }

        for (auto& i : S) res.insert(move(i));

        ++i;

        return res;

    }

    vector<string> braceExpansionII(string expression) {

        int i = 0;

        auto S = dfs(expression, i);

        vector<string> res(S.begin(), S.end());

        sort(res.begin(), res.end());

        return res;

    }

};

### 1106. Parsing A Boolean Expression

Hard

Return the result of evaluating a given boolean expression, represented as a string.

An expression can either be:

* "t", evaluating to True;
* "f", evaluating to False;
* "!(expr)", evaluating to the logical NOT of the inner expression expr;
* "&(expr1,expr2,...)", evaluating to the logical AND of 2 or more inner expressions expr1, expr2, ...;
* "|(expr1,expr2,...)", evaluating to the logical OR of 2 or more inner expressions expr1, expr2, ...

**Example 1:**

**Input:** expression = "!(f)"

**Output:** true

**Example 2:**

**Input:** expression = "|(f,t)"

**Output:** true

**Example 3:**

**Input:** expression = "&(t,f)"

**Output:** false

**Example 4:**

**Input:** expression = "|(&(t,f,t),!(t))"

**Output:** false

**Constraints:**

* 1 <= expression.length <= 20000
* expression[i] consists of characters in {'(', ')', '&', '|', '!', 't', 'f', ','}.
* expression is a valid expression representing a boolean, as given in the description.

class Solution {

public:

    bool parseBoolExpr(string s) {

        stack<char> stk;

        for (const auto &c : s) {

            if (c == ')') {

                auto hasT = false, hasF = false;

                while (stk.top() == 't' || stk.top() == 'f') {

                    hasT |= stk.top() == 't';

                    hasF |= stk.top() == 'f';

                    stk.pop();

                }

                auto op = stk.top();

                stk.pop();

                stk.push(op == '!' ? hasF ? 't' : 'f'

: op == '&' ? !hasF ? 't' : 'f'

: hasT ? 't' : 'f');

            }

            else if (c != ',' && c != '(') stk.push(c);

        }

        return stk.top() == 't';

    }

};

### 1125. Smallest Sufficient Team

Hard

In a project, you have a list of required skills req\_skills, and a list of people.  The i-th person people[i] contains a list of skills that person has.

Consider a *sufficient team*: a set of people such that for every required skill in req\_skills, there is at least one person in the team who has that skill.  We can represent these teams by the index of each person: for example, team = [0, 1, 3] represents the people with skills people[0], people[1], and people[3].

Return **any** sufficient team of the smallest possible size, represented by the index of each person.

You may return the answer in any order.  It is guaranteed an answer exists.

**Example 1:**

**Input:** req\_skills = ["java","nodejs","reactjs"], people = [["java"],["nodejs"],["nodejs","reactjs"]]

**Output:** [0,2]

**Example 2:**

**Input:** req\_skills = ["algorithms","math","java","reactjs","csharp","aws"], people = [["algorithms","math","java"],["algorithms","math","reactjs"],["java","csharp","aws"],["reactjs","csharp"],["csharp","math"],["aws","java"]]

**Output:** [1,2]

**Constraints:**

* 1 <= req\_skills.length <= 16
* 1 <= people.length <= 60
* 1 <= people[i].length, req\_skills[i].length, people[i][j].length <= 16
* Elements of req\_skills and people[i] are (respectively) distinct.
* req\_skills[i][j], people[i][j][k] are lowercase English letters.
* Every skill in people[i] is a skill in req\_skills.
* It is guaranteed a sufficient team exists.

class Solution {

public:

    vector<int> smallestSufficientTeam(vector<string>& req\_skills, vector<vector<string>>& people) {

        int n = req\_skills.size(), m = 1 << n;

        unordered\_map<string, int> mp;

        for (int i = 0; i < n; ++i) mp[req\_skills[i]] = i;

        vector<vector<int>> dp(m);

        for (int p = 0; p < people.size(); ++p) {

            int mask = 0;

            for (auto &i : people[p]) mask |= (1 << mp[i]);

            for (int i = 0; i < m; ++i) {

                if ((i|mask) == i) continue;

                if (i && dp[i].empty()) continue;

                if (dp[i|mask].empty() || dp[i|mask].size() > dp[i].size() + 1) {

                    dp[i|mask] = dp[i];

                    dp[i|mask].push\_back(p);

                }

            }

        }

        return dp.back();

    }

};

### 1147. Longest Chunked Palindrome Decomposition

Hard

Return the largest possible k such that there exists a\_1, a\_2, ..., a\_k such that:

* Each a\_i is a non-empty string;
* Their concatenation a\_1 + a\_2 + ... + a\_k is equal to text;
* For all 1 <= i <= k,  a\_i = a\_{k+1 - i}.

**Example 1:**

**Input:** text = "ghiabcdefhelloadamhelloabcdefghi"

**Output:** 7

**Explanation:** We can split the string on "(ghi)(abcdef)(hello)(adam)(hello)(abcdef)(ghi)".

**Example 2:**

**Input:** text = "merchant"

**Output:** 1

**Explanation:** We can split the string on "(merchant)".

**Example 3:**

**Input:** text = "antaprezatepzapreanta"

**Output:** 11

**Explanation:** We can split the string on "(a)(nt)(a)(pre)(za)(tpe)(za)(pre)(a)(nt)(a)".

**Example 4:**

**Input:** text = "aaa"

**Output:** 3

**Explanation:** We can split the string on "(a)(a)(a)".

**Constraints:**

* text consists only of lowercase English characters.
* 1 <= text.length <= 1000

class Solution {

    using ul = unsigned long;

    ul p = 31,  m = 1e9 + 7;

    int helper(const string\_view& s, int l, int r) {

        if (l >= r) return 0;

        ul prefix\_hash = 0, suffix\_hash = 0, pow = 1;

        for (int i = 0; i < (r - l) / 2; ++i) {

            prefix\_hash = (prefix\_hash + s[l + i] \* pow) % m;

            suffix\_hash = (suffix\_hash \* p + s[r - i - 1]) % m;

            pow = (pow \* p) % m;

            if (prefix\_hash == suffix\_hash) {

                if (s.substr(l, i + 1) == s.substr(r - i - 1, i + 1)) {

                    return 2 + helper(s, l + i + 1, r - i - 1);

                }

            }

        }

        return 1;

    }

public:

    int longestDecomposition(string text) {

        return helper(text, 0, text.size());

    }

};

### 1157. Online Majority Element In Subarray

Hard

Implementing the class MajorityChecker, which has the following API:

* MajorityChecker(int[] arr) constructs an instance of MajorityChecker with the given array arr;
* int query(int left, int right, int threshold) has arguments such that:
  + 0 <= left <= right < arr.length representing a subarray of arr;
  + 2 \* threshold > right - left + 1, ie. the threshold is always a strict majority of the length of the subarray

Each query(...) returns the element in arr[left], arr[left+1], ..., arr[right] that occurs at least threshold times, or -1 if no such element exists.

**Example:**

MajorityChecker majorityChecker = new MajorityChecker([1,1,2,2,1,1]);

majorityChecker.query(0,5,4); // returns 1

majorityChecker.query(0,3,3); // returns -1

majorityChecker.query(2,3,2); // returns 2

**Constraints:**

* 1 <= arr.length <= 20000
* 1 <= arr[i] <= 20000
* For each query, 0 <= left <= right < len(arr)
* For each query, 2 \* threshold > right - left + 1
* The number of queries is at most 10000

### 1172. Dinner Plate Stacks

Hard

You have an infinite number of stacks arranged in a row and numbered (left to right) from 0, each of the stacks has the same maximum capacity.

Implement the DinnerPlates class:

* DinnerPlates(int capacity) Initializes the object with the maximum capacity of the stacks.
* void push(int val) Pushes the given positive integer val into the leftmost stack with size less than capacity.
* int pop() Returns the value at the top of the rightmost non-empty stack and removes it from that stack, and returns -1 if all stacks are empty.
* int popAtStack(int index) Returns the value at the top of the stack with the given index and removes it from that stack, and returns -1 if the stack with that given index is empty.

**Example:**

**Input:**

["DinnerPlates","push","push","push","push","push","popAtStack","push","push","popAtStack","popAtStack","pop","pop","pop","pop","pop"]

[[2],[1],[2],[3],[4],[5],[0],[20],[21],[0],[2],[],[],[],[],[]]

**Output:**

[null,null,null,null,null,null,2,null,null,20,21,5,4,3,1,-1]

**Explanation:**

DinnerPlates D = DinnerPlates(2); // Initialize with capacity = 2

D.push(1);

D.push(2);

D.push(3);

D.push(4);

D.push(5); // The stacks are now: 2  4

  1  3  5

﹈ ﹈ ﹈

D.popAtStack(0); // Returns 2. The stacks are now:  4

  1  3  5

﹈ ﹈ ﹈

D.push(20); // The stacks are now: 20 4

  1  3  5

﹈ ﹈ ﹈

D.push(21); // The stacks are now: 20 4 21

  1  3  5

﹈ ﹈ ﹈

D.popAtStack(0); // Returns 20. The stacks are now: 4 21

  1  3  5

﹈ ﹈ ﹈

D.popAtStack(2); // Returns 21. The stacks are now: 4

  1  3  5

﹈ ﹈ ﹈

D.pop() // Returns 5. The stacks are now: 4

  1  3

﹈ ﹈

D.pop() // Returns 4. The stacks are now: 1  3

﹈ ﹈

D.pop() // Returns 3. The stacks are now: 1

﹈

D.pop() // Returns 1. There are no stacks.

D.pop() // Returns -1. There are still no stacks.

**Constraints:**

* 1 <= capacity <= 20000
* 1 <= val <= 20000
* 0 <= index <= 100000
* At most 200000 calls will be made to push, pop, and popAtStack.

class DinnerPlates {

public:

    int c;

    map<int, vector<int>> notEmpty;

    set<int> notFull;

    DinnerPlates(int capacity) {c = capacity;}

    void push(int val) {

        if (notFull.empty())

            notFull.insert(notEmpty.size());

        notEmpty[\*notFull.begin()].push\_back(val);

        if (notEmpty[\*notFull.begin()].size() == c)

            notFull.erase(notFull.begin());

    }

    int pop() {

        if (notEmpty.empty()) return -1;

        return popAtStack(notEmpty.rbegin()->first);

    }

    int popAtStack(int index) {

        if (!notEmpty.count(index)) return -1;

        int val = notEmpty[index].back();

        notEmpty[index].pop\_back();

        notFull.insert(index);

        if (notEmpty[index].empty()) notEmpty.erase(index);

        return val;

    }

};

/\*\*

 \* Your DinnerPlates object will be instantiated and called as such:

 \* DinnerPlates\* obj = new DinnerPlates(capacity);

 \* obj->push(val);

 \* int param\_2 = obj->pop();

 \* int param\_3 = obj->popAtStack(index);

 \*/

### 1178. Number of Valid Words for Each Puzzle

Hard

With respect to a given puzzle string, a word is *valid* if both the following conditions are satisfied:

* word contains the first letter of puzzle.
* For each letter in word, that letter is in puzzle.  
  For example, if the puzzle is "abcdefg", then valid words are "faced", "cabbage", and "baggage"; while invalid words are "beefed" (doesn't include "a") and "based" (includes "s" which isn't in the puzzle).

Return an array answer, where answer[i] is the number of words in the given word list words that are valid with respect to the puzzle puzzles[i].

**Example :**

**Input:**

words = ["aaaa","asas","able","ability","actt","actor","access"],

puzzles = ["aboveyz","abrodyz","abslute","absoryz","actresz","gaswxyz"]

**Output:** [1,1,3,2,4,0]

**Explanation:**

1 valid word for "aboveyz" : "aaaa"

1 valid word for "abrodyz" : "aaaa"

3 valid words for "abslute" : "aaaa", "asas", "able"

2 valid words for "absoryz" : "aaaa", "asas"

4 valid words for "actresz" : "aaaa", "asas", "actt", "access"

There're no valid words for "gaswxyz" cause none of the words in the list contains letter 'g'.

**Constraints:**

* 1 <= words.length <= 10^5
* 4 <= words[i].length <= 50
* 1 <= puzzles.length <= 10^4
* puzzles[i].length == 7
* words[i][j], puzzles[i][j] are English lowercase letters.
* Each puzzles[i] doesn't contain repeated characters.

class Solution {

public:

    vector<int> findNumOfValidWords(vector<string>& words, vector<string>& puzzles) {

        unordered\_map<int, int> mp;

        for (const auto &s : words) {

            int mask = 0;

            for (const auto &c : s) {

                mask |= 1 << (c - 'a');

            }

            ++mp[mask];

        }

        vector<int> res;

        for (const auto &s : puzzles) {

            int mask = 0, cnt = 0, firstLetter = 1 << (s[0]-'a');

            for (const auto &c : s) {

                mask |= 1 << (c - 'a');

            }

            for (int j = mask; j != 0; j = (j-1)&mask) {

                if ((firstLetter & j) && mp.count(j))

                    cnt += mp[j];

            }

            res.push\_back(cnt);

        }

        return res;

    }

};

### 1187. Make Array Strictly Increasing

Hard

Given two integer arrays arr1 and arr2, return the minimum number of operations (possibly zero) needed to make arr1 strictly increasing.

In one operation, you can choose two indices 0 <= i < arr1.length and 0 <= j < arr2.length and do the assignment arr1[i] = arr2[j].

If there is no way to make arr1 strictly increasing, return -1.

**Example 1:**

**Input:** arr1 = [1,5,3,6,7], arr2 = [1,3,2,4]

**Output:** 1

**Explanation:** Replace 5 with 2, then arr1 = [1, 2, 3, 6, 7].

**Example 2:**

**Input:** arr1 = [1,5,3,6,7], arr2 = [4,3,1]

**Output:** 2

**Explanation:** Replace 5 with 3 and then replace 3 with 4. arr1 = [1, 3, 4, 6, 7].

**Example 3:**

**Input:** arr1 = [1,5,3,6,7], arr2 = [1,6,3,3]

**Output:** -1

**Explanation:** You can't make arr1 strictly increasing.

**Constraints:**

* 1 <= arr1.length, arr2.length <= 2000
* 0 <= arr1[i], arr2[i] <= 10^9

class Solution {

public:

    //dp[i][k]表示当前进行到第i个数，且最后一个数不超过k时最少需要的操作次数

    const static int INF = 0x3f3f3f3f;

    int makeArrayIncreasing(vector<int> &arr1, vector<int> &arr2) {

        set<int> st(arr1.begin(), arr1.end());

        for (int i : arr2) st.insert(i);

        // 建立索引关系，离散化

        unordered\_map<int, int> mp;

        int idx = 0;

        for (int i : st) mp[i] = ++idx;

        for (int &i : arr1) i = mp[i];

        for (int &i : arr2) i = mp[i];

        unordered\_set<int> b(arr2.begin(), arr2.end());

        int n = arr1.size(), m = st.size();

        vector<int> dp(m + 1, 0);

        for (int i = 0; i < n; ++i) {

            for (int j = m; j >= 0; --j) {

                dp[j] = INF;

                if (j == arr1[i])

                    dp[j] = min(dp[j], dp[j-1]);

                else if (b.find(j) != b.end())

                    dp[j] = min(dp[j], dp[j-1] + 1);

            }

            for (int j = 1; j <= m; ++j)

                dp[j] = min(dp[j], dp[j-1]);

        }

        return dp[m] == INF ? -1 : dp[m];

    }

};

### 1192. Critical Connections in a Network

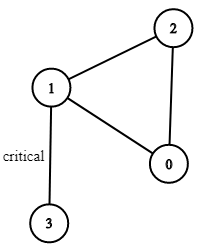
Hard

There are n servers numbered from 0 to n-1 connected by undirected server-to-server connections forming a network where connections[i] = [a, b] represents a connection between servers a and b. Any server can reach any other server directly or indirectly through the network.

A *critical connection* is a connection that, if removed, will make some server unable to reach some other server.

Return all critical connections in the network in any order.

**Example 1:**

****

**Input:** n = 4, connections = [[0,1],[1,2],[2,0],[1,3]]

**Output:** [[1,3]]

**Explanation:** [[3,1]] is also accepted.

**Constraints:**

* 1 <= n <= 10^5
* n-1 <= connections.length <= 10^5
* connections[i][0] != connections[i][1]
* There are no repeated connections.

### 1203. Sort Items by Groups Respecting Dependencies

Hard

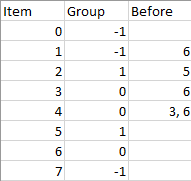
There are n items each belonging to zero or one of m groups where group[i] is the group that the i-th item belongs to and it's equal to -1 if the i-th item belongs to no group. The items and the groups are zero indexed. A group can have no item belonging to it.

Return a sorted list of the items such that:

* The items that belong to the same group are next to each other in the sorted list.
* There are some relations between these items where beforeItems[i] is a list containing all the items that should come before the i-th item in the sorted array (to the left of the i-th item).

Return any solution if there is more than one solution and return an **empty list** if there is no solution.

**Example 1:**

****

**Input:** n = 8, m = 2, group = [-1,-1,1,0,0,1,0,-1], beforeItems = [[],[6],[5],[6],[3,6],[],[],[]]

**Output:** [6,3,4,1,5,2,0,7]

**Example 2:**

**Input:** n = 8, m = 2, group = [-1,-1,1,0,0,1,0,-1], beforeItems = [[],[6],[5],[6],[3],[],[4],[]]

**Output:** []

**Explanation:** This is the same as example 1 except that 4 needs to be before 6 in the sorted list.

**Constraints:**

* 1 <= m <= n <= 3\*10^4
* group.length == beforeItems.length == n
* -1 <= group[i] <= m-1
* 0 <= beforeItems[i].length <= n-1
* 0 <= beforeItems[i][j] <= n-1
* i != beforeItems[i][j]
* beforeItems[i] does not contain duplicates elements.

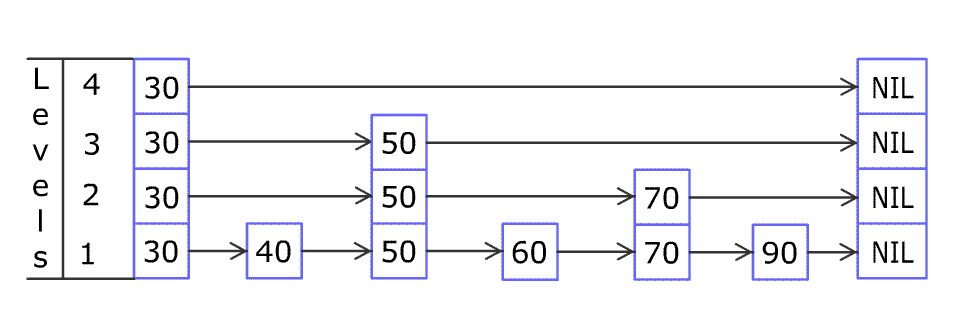
### 1206. Design Skiplist

Hard

Design a Skiplist without using any built-in libraries.

A Skiplist is a data structure that takes O(log(n)) time to *add*, *erase* and *search*. Comparing with treap and red-black tree which has the same function and performance, the code length of Skiplist can be comparatively short and the idea behind Skiplists are just simple linked lists.

For example: we have a Skiplist containing *[30,40,50,60,70,90]* and we want to add *80* and *45* into it. The Skiplist works this way:

  
Artyom Kalinin [CC BY-SA 3.0], via [Wikimedia Commons](https://commons.wikimedia.org/wiki/File:Skip_list_add_element-en.gif)

You can see there are many layers in the Skiplist. Each layer is a sorted linked list. With the help of the top layers, *add* , *erase* and *search*can be faster than O(n). It can be proven that the average time complexity for each operation is O(log(n)) and space complexity is O(n).

To be specific, your design should include these functions:

* bool search(int target) : Return whether the target exists in the Skiplist or not.
* void add(int num): Insert a value into the SkipList.
* bool erase(int num): Remove a value in the Skiplist. If num does not exist in the Skiplist, do nothing and return false. If there exists multiple num values, removing any one of them is fine.

See more about Skiplist : <https://en.wikipedia.org/wiki/Skip_list>

Note that duplicates may exist in the Skiplist, your code needs to handle this situation.

**Example:**

Skiplist skiplist = new Skiplist();

skiplist.add(1);

skiplist.add(2);

skiplist.add(3);

skiplist.search(0); // return false.

skiplist.add(4);

skiplist.search(1); // return true.

skiplist.erase(0); // return false, 0 is not in skiplist.

skiplist.erase(1); // return true.

skiplist.search(1); // return false, 1 has already been erased.

**Constraints:**

* 0 <= num, target <= 20000
* At most 50000 calls will be made to search, add, and erase.

class Skiplist {

private:

    struct Node {

        int val;

        Node \*right, \*down;

        Node(int v, Node \*r, Node \*d): val(v), right(r), down(d) {}

    };

    Node \*head;

    int level = 0;

public:

    Skiplist() : head(new Node(-1, nullptr, nullptr)) {}

    bool search(int target) {

        return find(target) != nullptr;

    }

    void add(int num) {

        int h = 1;

        //随机找到一高度h, 在h层以下插入num

        while (h <= level && rand() & 1) ++h;

        if (h == level+1) {

            ++level;

            head = new Node(-1, nullptr, head);

        }

        Node\* p = head, \*pre = nullptr;

        for (int i = level; i > 0; --i) {

            while (p->right && p->right->val < num) p = p->right;

            if (i <= h) {

                p->right = new Node(num, p->right, nullptr);

                if (pre) pre->down = p->right;

                pre = p->right;

            }

            p = p->down;

        }

    }

    bool erase(int num) {

        Node\* p = find(num);

        if (!p) return false;

        Node\* del = p->right;

        while (del) {

            while (p->right != del) p = p->right;

            auto tmp = del;

            p->right = del->right;

            del = del->down;

            p = p->down;

            delete tmp;

        }

        return true;

    }

    Node\* find(int target) {

        Node\* p = head;

        while (p) {

            while (p->right && p->right->val < target) p = p->right;

            if (p->right && p->right->val == target) return p;

            p = p->down;

        }

        return nullptr;

    }

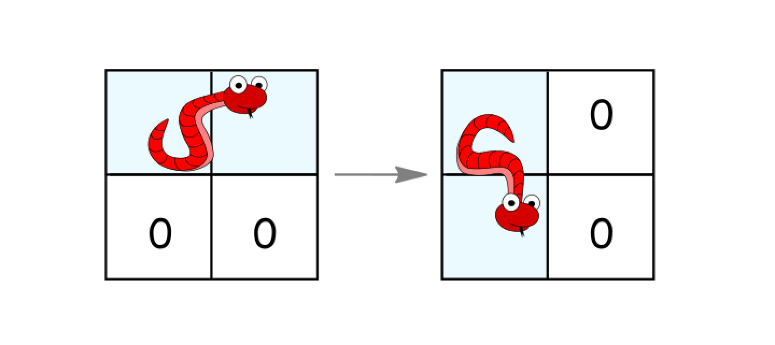
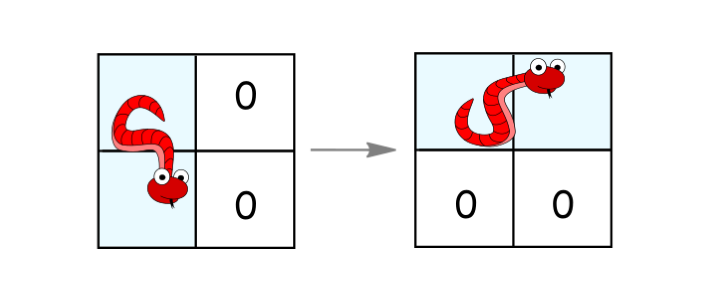
};

### 1210. Minimum Moves to Reach Target with Rotations

Hard

In an n\*n grid, there is a snake that spans 2 cells and starts moving from the top left corner at (0, 0) and (0, 1). The grid has empty cells represented by zeros and blocked cells represented by ones. The snake wants to reach the lower right corner at (n-1, n-2) and (n-1, n-1).

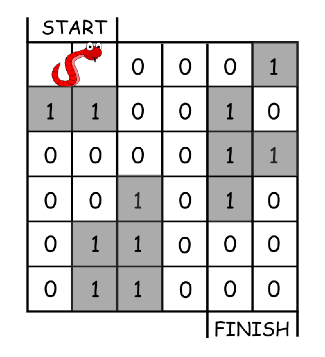
In one move the snake can:

* Move one cell to the right if there are no blocked cells there. This move keeps the horizontal/vertical position of the snake as it is.
* Move down one cell if there are no blocked cells there. This move keeps the horizontal/vertical position of the snake as it is.
* Rotate clockwise if it's in a horizontal position and the two cells under it are both empty. In that case the snake moves from (r, c) and (r, c+1) to (r, c) and (r+1, c).  
  
* Rotate counterclockwise if it's in a vertical position and the two cells to its right are both empty. In that case the snake moves from (r, c) and (r+1, c) to (r, c) and (r, c+1).  
  

Return the minimum number of moves to reach the target.

If there is no way to reach the target, return -1.

**Example 1:**

****

**Input:** grid = [[0,0,0,0,0,1],

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

  [0,0,0,0,1,1],

  [0,0,1,0,1,0],

  [0,1,1,0,0,0],

  [0,1,1,0,0,0]]

**Output:** 11

**Explanation:**

One possible solution is [right, right, rotate clockwise, right, down, down, down, down, rotate counterclockwise, right, down].

**Example 2:**

**Input:** grid = [[0,0,1,1,1,1],

  [0,0,0,0,1,1],

  [1,1,0,0,0,1],

  [1,1,1,0,0,1],

  [1,1,1,0,0,1],

  [1,1,1,0,0,0]]

**Output:** 9

**Constraints:**

* 2 <= n <= 100
* 0 <= grid[i][j] <= 1
* It is guaranteed that the snake starts at empty cells.

class Solution {

public:

    int n;

    bool canRotate(vector<vector<int>>& g, int i, int j) {

        return i < n-1 && j < n-1

            && !g[i+1][j] && !g[i][j+1] && !g[i+1][j+1];

    }

    bool canGoDown(vector<vector<int>>& g,int i,int j,bool vertical) {

        if (vertical)  return i < n-2 && !g[i+2][j];

        else return i < n-1 && !g[i+1][j] && !g[i+1][j+1];

    }

    bool canGoRight(vector<vector<int>>& g,int i,int j,bool vertical) {

        if (!vertical) return j < n-2 && !g[i][j+2];

        else return j < n-1 && !g[i][j+1] && !g[i+1][j+1];

    }

    int minimumMoves(vector<vector<int>>& grid, int steps = 0) {

        n = grid.size();

        queue<array<int, 3>> q;

        q.push({ 0, 0, false });

        while (!q.empty()) {

            int sz = q.size();

            while (sz--) {

                auto& a = q.front();

                if (a[0] == n-1 && a[1] == n-2) return steps;

                if ((grid[a[0]][a[1]] & (a[2] ? 2 : 4)) == 0) {

                    grid[a[0]][a[1]] = grid[a[0]][a[1]] | (a[2]?2:4);

                    if (canGoDown(grid, a[0], a[1], a[2]))

q.push({ a[0] + 1, a[1], a[2] });

                    if (canGoRight(grid, a[0], a[1], a[2]))

q.push({ a[0], a[1] + 1, a[2] });

                    if (canRotate(grid, a[0], a[1]))

q.push({ a[0], a[1], a[2] ? false : true });

                }

                q.pop();

            }

            ++steps;

        }

        return -1;

    }

};

### 1220. Count Vowels Permutation

Hard

Given an integer n, your task is to count how many strings of length n can be formed under the following rules:

* Each character is a lower case vowel ('a', 'e', 'i', 'o', 'u')
* Each vowel 'a' may only be followed by an 'e'.
* Each vowel 'e' may only be followed by an 'a' or an 'i'.
* Each vowel 'i' **may not** be followed by another 'i'.
* Each vowel 'o' may only be followed by an 'i' or a 'u'.
* Each vowel 'u' may only be followed by an 'a'.

Since the answer may be too large, return it modulo 10^9 + 7.

**Example 1:**

**Input:** n = 1

**Output:** 5

**Explanation:** All possible strings are: "a", "e", "i" , "o" and "u".

**Example 2:**

**Input:** n = 2

**Output:** 10

**Explanation:** All possible strings are: "ae", "ea", "ei", "ia", "ie", "io", "iu", "oi", "ou" and "ua".

**Example 3:**

**Input:** n = 5

**Output:** 68

**Constraints:**

* 1 <= n <= 2 \* 10^4

class Solution {

public:

    int countVowelPermutation(int n) {

        long long cnt = 5, MOD = 1e9 + 7;

        vector<long long> pre(5, 1); //a, e, i, o, u

        while (--n) {

            vector<long long> cur(5, 0);

            cur[0] = (pre[1] + pre[2] + pre[4]) % MOD;

            cur[1] = (pre[0] + pre[2]) % MOD;

            cur[2] = (pre[1] + pre[3]) % MOD;

            cur[3] = (pre[2]) % MOD;

            cur[4] = (pre[2] + pre[3]) % MOD;

            cnt = 0;

            for (auto i : cur) cnt = (cnt + i) % MOD;

            pre = cur;

        }

        return cnt;

    }

};

### 1224. Maximum Equal Frequency

Hard

Given an array nums of positive integers, return the longest possible length of an array prefix of nums, such that it is possible to remove **exactly one** element from this prefix so that every number that has appeared in it will have the same number of occurrences.

If after removing one element there are no remaining elements, it's still considered that every appeared number has the same number of ocurrences (0).

**Example 1:**

**Input:** nums = [2,2,1,1,5,3,3,5]

**Output:** 7

**Explanation:** For the subarray [2,2,1,1,5,3,3] of length 7, if we remove nums[4]=5, we will get [2,2,1,1,3,3], so that each number will appear exactly twice.

**Example 2:**

**Input:** nums = [1,1,1,2,2,2,3,3,3,4,4,4,5]

**Output:** 13

**Example 3:**

**Input:** nums = [1,1,1,2,2,2]

**Output:** 5

**Example 4:**

**Input:** nums = [10,2,8,9,3,8,1,5,2,3,7,6]

**Output:** 8

**Constraints:**

* 2 <= nums.length <= 10^5
* 1 <= nums[i] <= 10^5

class Solution {

public:

    int maxEqualFreq(vector<int>& A) {

        vector<int> count(100001), freq(100001);

        int res = 0, N = A.size(), a,c,d;

        for (int n = 1; n <= N; ++n) {

            a = A[n - 1];

            --freq[count[a]];

            c = ++count[a];

            ++freq[count[a]];

            if (freq[c] \* c == n && n < N)

                res = n + 1;

            d = n - freq[c] \* c;

            if ((d == c + 1 || d == 1) && freq[d] == 1)

                res = n;

        }

        return res;

    }

};

### 1235. Maximum Profit in Job Scheduling

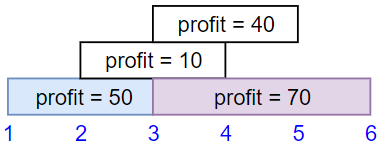
Hard

We have n jobs, where every job is scheduled to be done from startTime[i] to endTime[i], obtaining a profit of profit[i].

You're given the startTime , endTime and profit arrays, you need to output the maximum profit you can take such that there are no 2 jobs in the subset with overlapping time range.

If you choose a job that ends at time X you will be able to start another job that starts at time X.

**Example 1:**

****

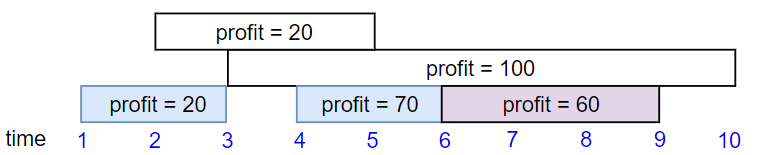
**Input:** startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]

**Output:** 120

**Explanation:** The subset chosen is the first and fourth job.

Time range [1-3]+[3-6] , we get profit of 120 = 50 + 70.

**Example 2:**

****

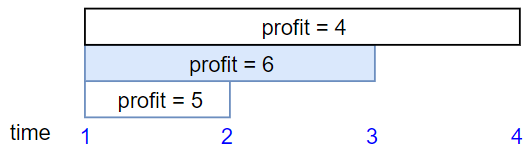
**Input:** startTime = [1,2,3,4,6], endTime = [3,5,10,6,9], profit = [20,20,100,70,60]

**Output:** 150

**Explanation:** The subset chosen is the first, fourth and fifth job.

Profit obtained 150 = 20 + 70 + 60.

**Example 3:**

****

**Input:** startTime = [1,1,1], endTime = [2,3,4], profit = [5,6,4]

**Output:** 6

**Constraints:**

* 1 <= startTime.length == endTime.length == profit.length <= 5 \* 10^4
* 1 <= startTime[i] < endTime[i] <= 10^9
* 1 <= profit[i] <= 10^4

class Solution {

public:

    int jobScheduling(vector<int>& startTime, vector<int>& endTime, vector<int>& profit) {

        int n = startTime.size();

        vector<vector<int>> jobs;

        for (int i = 0; i < n; ++i) {

            jobs.push\_back({endTime[i], startTime[i], profit[i]});

        }

        sort(jobs.begin(), jobs.end());

        map<int, int> dp = {{0, 0}};

        for (auto& job : jobs) {

            int cur = prev(dp.upper\_bound(job[1]))->second + job[2];

            if (cur > dp.rbegin()->second)

                dp[job[0]] = cur;

        }

        return dp.rbegin()->second;

    }

};