目录

[1283. Find the Smallest Divisor Given a Threshold 4](#_Toc57555202)

[1284. Minimum Number of Flips to Convert Binary Matrix to Zero Matrix★★★ 6](#_Toc57555203)

[1286. Iterator for Combination 10](#_Toc57555204)

[1288. Remove Covered Intervals 13](#_Toc57555205)

[1292. Maximum Side Length of a Square with Sum Less than or Equal to Threshold★ 15](#_Toc57555206)

[1293. Shortest Path in a Grid with Obstacles Elimination★ 18](#_Toc57555207)

[1296. Divide Array in Sets of K Consecutive Numbers★★ 21](#_Toc57555208)

[1298. Maximum Candies You Can Get from Boxes 24](#_Toc57555209)

[1300. Sum of Mutated Array Closest to Target 27](#_Toc57555210)

[1301. Number of Paths with Max Score 30](#_Toc57555211)

[1307. Verbal Arithmetic Puzzle★★ 33](#_Toc57555212)

[1312. Minimum Insertion Steps to Make a String Palindrome 37](#_Toc57555213)

[1314. Matrix Block Sum 41](#_Toc57555214)

[1315. Sum of Nodes with Even-Valued Grandparent 43](#_Toc57555215)

[1316. Distinct Echo Substrings★★ 45](#_Toc57555216)

[1318. Minimum Flips to Make a OR b Equal to c 47](#_Toc57555217)

[1319. Number of Operations to Make Network Connected 49](#_Toc57555218)

[1320. Minimum Distance to Type a Word Using Two Fingers★★ 52](#_Toc57555219)

[1326. Minimum Number of Taps to Open to Water a Garden★★ 55](#_Toc57555220)

[1330. Reverse Subarray To Maximize Array Value★★ 58](#_Toc57555221)

[1334. Find the City With the Smallest Number of Neighbor at a Threshold Distance 60](#_Toc57555222)

[1335. Minimum Difficulty of a Job Schedule 63](#_Toc57555223)

[1340. Jump Game V 66](#_Toc57555224)

[1345. Jump Game IV 71](#_Toc57555225)

[1349. Maximum Students Taking Exam★★ 74](#_Toc57555226)

[1353. Maximum Number of Events That Can Be Attended★★ 77](#_Toc57555227)

[1354. Construct Target Array With Multiple Sums★★ 80](#_Toc57555228)

[1363. Largest Multiple of Three★ 82](#_Toc57555229)

[1368. Minimum Cost to Make at Least One Valid Path in a Grid★★ 84](#_Toc57555230)

[1371. Find the Longest Substring Containing Vowels in Even Counts★★ 89](#_Toc57555231)

[1373. Maximum Sum BST in Binary Tree 91](#_Toc57555232)

[1375. Bulb Switcher III★ 94](#_Toc57555233)

[1383. Maximum Performance of a Team★★ 97](#_Toc57555234)

[1388. Pizza With 3n Slices★★ 99](#_Toc57555235)

[1395. Count Number of Teams 102](#_Toc57555236)

[1397. Find All Good Strings 106](#_Toc57555237)

[1402. Reducing Dishes★★ 108](#_Toc57555238)

[1405. Longest Happy String 110](#_Toc57555239)

[1406. Stone Game III★ 112](#_Toc57555240)

[1411. Number of Ways to Paint N × 3 Grid 115](#_Toc57555241)

[1414. Find the Minimum Number of Fibonacci Numbers Whose Sum Is K★ 118](#_Toc57555242)

[1415. The k-th Lexicographical String of All Happy Strings of Length n★ 120](#_Toc57555243)

[1416. Restore The Array 123](#_Toc57555244)

[1420. Build Array Where You Can Find The Maximum Exactly K Comparisons★★ 126](#_Toc57555245)

[1423. Maximum Points You Can Obtain from Cards 129](#_Toc57555246)

[1425. Constrained Subsequence Sum★★ 132](#_Toc57555247)

[1434. Number of Ways to Wear Different Hats to Each Other★★ 135](#_Toc57555248)

[1438. Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit★★ 138](#_Toc57555249)

[1439. Find the Kth Smallest Sum of a Matrix With Sorted Rows★ 142](#_Toc57555250)

[1442. Count Triplets That Can Form Two Arrays of Equal XOR★★ 144](#_Toc57555251)

[1444. Number of Ways of Cutting a Pizza★★ 146](#_Toc57555252)

[1449. Form Largest Integer With Digits That Add up to Target★★ 149](#_Toc57555253)

[1453. Maximum Number of Darts Inside of a Circular Dartboard★★ 152](#_Toc57555254)

[1458. Max Dot Product of Two Subsequences 156](#_Toc57555255)

[1462. Course Schedule IV 157](#_Toc57555256)

[1463. Cherry Pickup II 160](#_Toc57555257)

[1467. Probability of a Two Boxes Having The Same Number of Distinct Balls 164](#_Toc57555258)

[1477. Find Two Non-overlapping Sub-arrays Each With Target Sum★★ 167](#_Toc57555259)

[1478. Allocate Mailboxes★★ 170](#_Toc57555260)

[1482. Minimum Number of Days to Make m Bouquets 173](#_Toc57555261)

[1483. Kth Ancestor of a Tree Node★★ 176](#_Toc57555262)

[1488. Avoid Flood in The City 179](#_Toc57555263)

[1489. Find Critical and Pseudo-Critical Edges in Minimum Spanning Tree 182](#_Toc57555264)

[1494. Parallel Courses II★★ 186](#_Toc57555265)

[1498. Number of Subsequences That Satisfy the Given Sum Condition 189](#_Toc57555266)

[1499. Max Value of Equation★★ 191](#_Toc57555267)

[1504. Count Submatrices With All Ones★★ 193](#_Toc57555268)

[1505. Minimum Possible Integer After at Most K Adjacent Swaps On Digits 197](#_Toc57555269)

[1514. Path with Maximum Probability 201](#_Toc57555270)

[1515. Best Position for a Service Centre 204](#_Toc57555271)

[1520. Maximum Number of Non-Overlapping Substrings 207](#_Toc57555272)

[1521. Find a Value of a Mysterious Function Closest to Target 211](#_Toc57555273)

[1526. Minimum Number of Increments on Subarrays to Form a Target Array 215](#_Toc57555274)

[1531. String Compression II★★ 217](#_Toc57555275)

[1537. Get the Maximum Score 221](#_Toc57555276)

[1541. Minimum Insertions to Balance a Parentheses String 224](#_Toc57555277)

[1542. Find Longest Awesome Substring★★ 227](#_Toc57555278)

[1546. Maximum Number of NonOverlapping Subarrays With Sum Equals Target★★ 229](#_Toc57555279)

[1547. Minimum Cost to Cut a Stick 232](#_Toc57555280)

[1553. Minimum Number of Days to Eat N Oranges★ 235](#_Toc57555281)

[1559. Detect Cycles in 2D Grid 240](#_Toc57555282)

[1562. Find Latest Group of Size M★★ 243](#_Toc57555283)

[1563. Stone Game V 246](#_Toc57555284)

[1568. Minimum Number of Days to Disconnect Island 249](#_Toc57555285)

[1569. Number of Ways to Reorder Array to Get Same BST★★ 252](#_Toc57555286)

[1574. Shortest Subarray to be Removed to Make Array Sorted★★ 256](#_Toc57555287)

[1575. Count All Possible Routes 258](#_Toc57555288)

[1579. Remove Max Number of Edges to Keep Graph Fully Traversable★★ 261](#_Toc57555289)

[1589. Maximum Sum Obtained of Any Permutation★★ 264](#_Toc57555290)

[1591. Strange Printer II★★ 266](#_Toc57555291)

[1595. Minimum Cost to Connect Two Groups of Points 269](#_Toc57555292)

[1601. Maximum Number of Achievable Transfer Requests 272](#_Toc57555293)

[1610. Maximum Number of Visible Points 275](#_Toc57555294)

[1617. Count Subtrees With Max Distance Between Cities 278](#_Toc57555295)

[1627. Graph Connectivity With Threshold 285](#_Toc57555296)

[1621. Number of Sets of K Non-Overlapping Line Segments 289](#_Toc57555297)

[1622. Fancy Sequence 294](#_Toc57555298)

[1626. Best Team With No Conflicts 297](#_Toc57555299)

[1631. Path With Minimum Effort 299](#_Toc57555300)

[1632. Rank Transform of a Matrix 303](#_Toc57555301)

[毕业旅行问题 306](#_Toc57555302)

[5614. Find the Most Competitive Subsequence★ 308](#_Toc57555303)

[5615. Minimum Moves to Make Array Complementary★★ 310](#_Toc57555304)

### 1283. Find the Smallest Divisor Given a Threshold

Medium

Given an array of integers nums and an integer threshold, we will choose a positive integer divisor and divide all the array by it and sum the result of the division. Find the **smallest** divisor such that the result mentioned above is less than or equal to threshold.

Each result of division is rounded to the nearest integer greater than or equal to that element. (For example: 7/3 = 3 and 10/2 = 5).

It is guaranteed that there will be an answer.

**Example 1:**

**Input:** nums = [1,2,5,9], threshold = 6

**Output:** 5

**Explanation:** We can get a sum to 17 (1+2+5+9) if the divisor is 1.

If the divisor is 4 we can get a sum to 7 (1+1+2+3) and if the divisor is 5 the sum will be 5 (1+1+1+2).

**Example 2:**

**Input:** nums = [2,3,5,7,11], threshold = 11

**Output:** 3

**Example 3:**

**Input:** nums = [19], threshold = 5

**Output:** 4

**Constraints:**

* 1 <= nums.length <= 5 \* 10^4
* 1 <= nums[i] <= 10^6
* nums.length <= threshold <= 10^6

class Solution {

public:

    int smallestDivisor(vector<int>& nums, int threshold) {

        int l = 1, r = INT\_MAX;

        while (l < r) {

            int mid = l + (r-l)/2, sum = 0;

            for (double i : nums) sum += ceil(i / mid);

            if (sum > threshold) l = mid+1;

            else r = mid;

        }

        return l;

    }

};

### 1284. Minimum Number of Flips to Convert Binary Matrix to Zero Matrix★★★

Hard

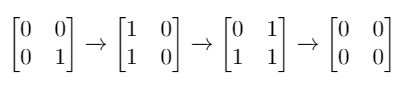
Given a m x n binary matrix mat. In one step, you can choose one cell and flip it and all the four neighbours of it if they exist (Flip is changing 1 to 0 and 0 to 1). A pair of cells are called neighboors if they share one edge.

Return the *minimum number of steps* required to convert mat to a zero matrix or **-1** if you cannot.

Binary matrix is a matrix with all cells equal to 0 or 1 only.

Zero matrix is a matrix with all cells equal to 0.

**Example 1:**



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

**Output:** 3

**Explanation:** One possible solution is to flip (1, 0) then (0, 1) and finally (1, 1) as shown.

**Example 2:**

**Input:** mat = [[0]]

**Output:** 0

**Explanation:** Given matrix is a zero matrix. We don't need to change it.

**Example 3:**

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

**Output:** 6

**Example 4:**

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

**Output:** -1

**Explanation:** Given matrix can't be a zero matrix

**Constraints:**

* m == mat.length
* n == mat[0].length
* 1 <= m <= 3
* 1 <= n <= 3
* mat[i][j] is 0 or 1.

class Solution {

public:

    int m, n;

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

    int minFlips(vector<vector<int>>& mat) {

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

        int bitvec = createBitVec(mat), cnt = 0;

        if (!bitvec) return 0;

        queue<int> q;

        unordered\_set<int> vis;

        q.push(bitvec);

        while (!q.empty()) {

            int sz = q.size();

            ++cnt;

            while (sz--) {

                auto u = q.front();

                q.pop();

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

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

                        int t = getFlip(i, j, u);

                        if (!t) return cnt;

                        if (!vis.count(t)) {

                            q.push(t);

                            vis.insert(t);

                        }

                    }

                }

            }

        }

        return -1;

}

int createBitVec(vector<vector<int>>& mat) {

        int bitvec = 0;

        for(int i = 0;i < n; i++) {

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

                bitvec <<= 1;

                bitvec |= mat[i][j];

            }

        }

        return bitvec;

    }

    int getFlip(int i, int j, int bitVec) {

        for (int d = 0; d < 5; d++) {

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

            if (x >= 0 && y >= 0 && x < n && y < m) {

                bitVec ^= 1<<(x\*m+y);

            }

        }

        return bitVec;

    }

};

### 1286. Iterator for Combination

Medium

Design an Iterator class, which has:

* A constructor that takes a string characters of **sorted distinct** lowercase English letters and a number combinationLength as arguments.
* A function *next()* that returns the next combination of length combinationLength in **lexicographical order**.
* A function *hasNext()* that returns True if and only if there exists a next combination.

**Example:**

CombinationIterator iterator = new CombinationIterator("abc", 2); // creates the iterator.

iterator.next(); // returns "ab"

iterator.hasNext(); // returns true

iterator.next(); // returns "ac"

iterator.hasNext(); // returns true

iterator.next(); // returns "bc"

iterator.hasNext(); // returns false

**Constraints:**

* 1 <= combinationLength <= characters.length <= 15
* There will be at most 10^4 function calls per test.
* It's guaranteed that all calls of the function next are valid.

class CombinationIterator {

public:

    int len, mask;

    string s;

    CombinationIterator(string characters, int combinationLength) {

        s=characters;

        len = combinationLength;

        mask = (1 << characters.length()) - 1;

    }

    string next() {

        while(mask && \_\_builtin\_popcount(mask) != len) mask--;

        string out;

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

            if (mask & (1 << (s.length() - i - 1)))

                out += s[i];

        }

        mask--;

        return out;

    }

    bool hasNext() {

        while(mask && \_\_builtin\_popcount(mask) != len) mask--;

        if (!mask)

            return false;

        return true;

    }

};

class CombinationIterator {

public:

    int len, mask;

    string s, characters;

    bool ok = true, end = false;

    CombinationIterator(string a, int n) {

        s = string(n, '1') + string(a.size() - n, '0');

        characters = a;

    }

    string next() {

        hasNext();

        ok = false;

        string res;

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

            if (s[i] == '1')

                res += characters[i];

        }

        return res;

    }

    bool hasNext() {

       if (end) return false;

       if (ok) return true;

       if (!prev\_permutation(s.begin(), s.end())) {

           end = true;

           return false;

       }

       return ok = true;

    }

};

### 1288. Remove Covered Intervals

Medium

Given a list of intervals, remove all intervals that are covered by another interval in the list. Interval [a,b) is covered by interval [c,d) if and only if c <= a and b <= d.

After doing so, return the number of remaining intervals.

**Example 1:**

**Input:** intervals = [[1,4],[3,6],[2,8]]

**Output:** 2

**Explanation:** Interval [3,6] is covered by [2,8], therefore it is removed.

**Constraints:**

* 1 <= intervals.length <= 1000
* 0 <= intervals[i][0] < intervals[i][1] <= 10^5
* intervals[i] != intervals[j] for all i != j

class Solution {

public:

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

        auto cmp = [](const vector<int> &lhs, const vector<int> &rhs) {

            if (lhs[0] != rhs[0]) return lhs[0] < rhs[0];

            else return lhs[1] > rhs[1];

        };

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

        int pre = intervals[0][0], res = intervals.size();

        for (auto &v : intervals) {

            if (pre >= v[1]) --res;

            else pre = v[1];

        }

        return res;

    }

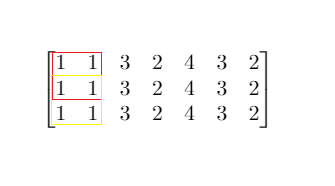
};

### 1292. Maximum Side Length of a Square with Sum Less than or Equal to Threshold★

Medium

Given a m x n matrix mat and an integer threshold. Return the maximum side-length of a square with a sum less than or equal to threshold or return **0** if there is no such square.

**Example 1:**



**Input:** mat = [[1,1,3,2,4,3,2],[1,1,3,2,4,3,2],[1,1,3,2,4,3,2]], threshold = 4

**Output:** 2

**Explanation:** The maximum side length of square with sum less than 4 is 2 as shown.

**Example 2:**

**Input:** mat = [[2,2,2,2,2],[2,2,2,2,2],[2,2,2,2,2],[2,2,2,2,2],[2,2,2,2,2]], threshold = 1

**Output:** 0

**Example 3:**

**Input:** mat = [[1,1,1,1],[1,0,0,0],[1,0,0,0],[1,0,0,0]], threshold = 6

**Output:** 3

**Example 4:**

**Input:** mat = [[18,70],[61,1],[25,85],[14,40],[11,96],[97,96],[63,45]], threshold = 40184

**Output:** 2

**Constraints:**

* 1 <= m, n <= 300
* m == mat.length
* n == mat[i].length
* 0 <= mat[i][j] <= 10000
* 0 <= threshold <= 10^5

class Solution {

    vector<vector<int>> preS;

    int squareSum(int x1, int y1, int x2, int y2) {

        return preS[x2][y2] - preS[x1][y2]

                - preS[x2][y1] + preS[x1][y1];

    }

public:

    int maxSideLength(vector<vector<int>>& mat, int t) {

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

        preS.resize(m+1, vector<int>(n+1));

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

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

                preS[i][j] = preS[i-1][j] + preS[i][j-1]

                            - preS[i-1][j-1] + mat[i-1][j-1];

            }

        }

        int res = 0;

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

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

                int len = res+1;

                while (i+len <= m && j+len <= n

&& squareSum(i, j, i+len, j+len) <= t) {

                    res = len++;

                }

            }

        }

        return res;

    }

};

### 1293. Shortest Path in a Grid with Obstacles Elimination★

Hard

Given a m \* n grid, where each cell is either 0 (empty) or 1 (obstacle). In one step, you can move up, down, left or right from and to an empty cell.

Return the minimum number of steps to walk from the upper left corner (0, 0) to the lower right corner (m-1, n-1) given that you can eliminate **at most** k obstacles. If it is not possible to find such walk return -1.

**Example 1:**

**Input:**

grid =

[[0,0,0],

 [1,1,0],

[0,0,0],

 [0,1,1],

[0,0,0]],

k = 1

**Output:** 6

**Explanation:**

The shortest path without eliminating any obstacle is 10.

The shortest path with one obstacle elimination at position (3,2) is 6. Such path is (0,0) -> (0,1) -> (0,2) -> (1,2) -> (2,2) -> **(3,2)** -> (4,2).

**Example 2:**

**Input:**

grid =

[[0,1,1],

 [1,1,1],

 [1,0,0]],

k = 1

**Output:** -1

**Explanation:**

We need to eliminate at least two obstacles to find such a walk.

**Constraints:**

* grid.length == m
* grid[0].length == n
* 1 <= m, n <= 40
* 1 <= k <= m\*n
* grid[i][j] == 0 **or** 1
* grid[0][0] == grid[m-1][n-1] == 0

class Solution {

public:

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

    int shortestPath(vector<vector<int>>& grid, int k) {

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

        if (max(0, n + m - 2) <= k-1) return max(0, n + m - 2);

        queue<tuple<int, int, int>> q;

        q.emplace(0,0,k);

        vector<int> visited(n\*m, -1);

        visited[0] = k;

        int steps = 0, res = n\*m;

        while (!q.empty()) {

            steps++;

            int sz = q.size();

            while (sz--) {

                //int x = get<0>(q.front())

                auto [x, y, k] = q.front();

                q.pop();

                for (const auto& d : dirs) {

                    int xx = x + d[0], yy = y + d[1];

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

                    int kk = k - grid[xx][yy];

                    if (visited[xx\*m+yy] >= kk) continue;

                    int stepsToTarget = n-1-xx+m-1-yy;

                    if (stepsToTarget-1 <= kk)

                        res = min(res, steps+stepsToTarget);

                    if (stepsToTarget == 0) continue;

                    q.emplace(xx,yy,kk);

                    visited[xx\*m+yy] = kk;

                }

            }

        }

        return res != n\*m ? res : -1;

    }

};

### 1296. Divide Array in Sets of K Consecutive Numbers★★

Medium

Given an array of integers nums and a positive integer k, find whether it's possible to divide this array into sets of k consecutive numbers  
Return True if its possibleotherwise return False.

**Example 1:**

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

**Output:** true

**Explanation:** Array can be divided into [1,2,3,4] and [3,4,5,6].

**Example 2:**

**Input:** nums = [3,2,1,2,3,4,3,4,5,9,10,11], k = 3

**Output:** true

**Explanation:** Array can be divided into [1,2,3] , [2,3,4] , [3,4,5] and [9,10,11].

**Example 3:**

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

**Output:** true

**Example 4:**

**Input:** nums = [1,2,3,4], k = 3

**Output:** false

**Explanation:** Each array should be divided in subarrays of size 3.

**Constraints:**

* 1 <= nums.length <= 10^5
* 1 <= nums[i] <= 10^9
* 1 <= k <= nums.length

//O(nlogn)

class Solution {

public:

    struct node {

        int val, sz;

        node (int a, int b) : val(a), sz(b) {}

        bool operator < (const node &rhs) const {

            if (val != rhs.val) return val > rhs.val;

            else return sz < rhs.sz;

        }

    };

    bool isPossibleDivide(vector<int>& nums, int k) {

        if (k == 1) return true;

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

        priority\_queue<node> pq;

        for (auto i : nums) {

            if (pq.empty()) pq.emplace(i, 1);

            else if (pq.top().val == i-1) {

                if (pq.top().sz == k-1) pq.pop();

                else {

                    node t(pq.top().val+1, pq.top().sz+1);

                    pq.pop();

                    pq.push(t);

                }

            }

            else pq.emplace(i, 1);

        }

        return pq.empty();

    }

};

//O(n) 每个节点访问最多删2次

class Solution {

public:

    bool isPossibleDivide(vector<int>& nums, int k) {

        unordered\_map<int, int> mp;

        for (int i : nums) mp[i]++;

        for (int i : nums) {

            int start = i;

            while (mp[start - 1]) start--;

            for (; start <= i; start++) {

                int cnt = mp[start];

                if (cnt) {

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

                        if ((mp[j] -= cnt) < 0)

                            return false;

                    }

                }

            }

        }

        return true;

    }

};

### 1298. Maximum Candies You Can Get from Boxes

Hard

Given n boxes, each box is given in the format [status, candies, keys, containedBoxes] where:

* status[i]: an integer which is **1** if box[i] is open and **0** if box[i] is closed.
* candies[i]: an integer representing the number of candies in box[i].
* keys[i]: an array contains the indices of the boxes you can open with the key in box[i].
* containedBoxes[i]: an array contains the indices of the boxes found in box[i].

You will start with some boxes given in initialBoxes array. You can take all the candies in any open box and you can use the keys in it to open new boxes and you also can use the boxes you find in it.

Return *the maximum number of candies* you can get following the rules above.

**Example 1:**

**Input:** status = [1,0,1,0], candies = [7,5,4,100], keys = [[],[],[1],[]], containedBoxes = [[1,2],[3],[],[]], initialBoxes = [0]

**Output:** 16

**Explanation:** You will be initially given box 0. You will find 7 candies in it and boxes 1 and 2. Box 1 is closed and you don't have a key for it so you will open box 2. You will find 4 candies and a key to box 1 in box 2.

In box 1, you will find 5 candies and box 3 but you will not find a key to box 3 so box 3 will remain closed.

Total number of candies collected = 7 + 4 + 5 = 16 candy.

**Example 2:**

**Input:** status = [1,0,0,0,0,0], candies = [1,1,1,1,1,1], keys = [[1,2,3,4,5],[],[],[],[],[]], containedBoxes = [[1,2,3,4,5],[],[],[],[],[]], initialBoxes = [0]

**Output:** 6

**Explanation:** You have initially box 0. Opening it you can find boxes 1,2,3,4 and 5 and their keys. The total number of candies will be 6.

**Example 3:**

**Input:** status = [1,1,1], candies = [100,1,100], keys = [[],[0,2],[]], containedBoxes = [[],[],[]], initialBoxes = [1]

**Output:** 1

**Example 4:**

**Input:** status = [1], candies = [100], keys = [[]], containedBoxes = [[]], initialBoxes = []

**Output:** 0

**Example 5:**

**Input:** status = [1,1,1], candies = [2,3,2], keys = [[],[],[]], containedBoxes = [[],[],[]], initialBoxes = [2,1,0]

**Output:** 7

**Constraints:**

* 1 <= status.length <= 1000
* status.length == candies.length == keys.length == containedBoxes.length == n
* status[i] is 0 or 1.
* 1 <= candies[i] <= 1000
* 0 <= keys[i].length <= status.length
* 0 <= keys[i][j] < status.length
* All values in keys[i] are unique.
* 0 <= containedBoxes[i].length <= status.length
* 0 <= containedBoxes[i][j] < status.length
* All values in containedBoxes[i] are unique.
* Each box is contained in one box at most.
* 0 <= initialBoxes.length <= status.length
* 0 <= initialBoxes[i] < status.length

class Solution {

public:

    int maxCandies(vector<int>& status, vector<int>& candies, vector<vector<int>>& keys, vector<vector<int>>& containedBoxes, vector<int>& initialBoxes) {

        int res = 0;

        for(auto i : initialBoxes) {

            res += dfs(i, status, candies, keys, containedBoxes);

        }

        return res;

    }

private:

    unordered\_set<int> locked;

    int dfs(int i, vector<int>& status, vector<int>& candies, vector<vector<int>>& keys, vector<vector<int>>& containedBoxes) {

        if (status[i] == 0) locked.insert(i);

        if (status[i] <= 0) return 0;

        int res = candies[i];

        status[i] = -1;

        vector<int> &t = containedBoxes[i];

        for (auto j : keys[i]) {

            if (status[j] == 0) {

                status[j] = 1;

                if (locked.count(j)) {

                    locked.erase(j);

                    t.push\_back(j);

                }

            }

        }

        return  res+maxCandies(status,candies, keys,containedBoxes,t);

    }

};

### 1300. Sum of Mutated Array Closest to Target

Medium

Given an integer array arr and a target value target, return the integer value such that when we change all the integers larger than value in the given array to be equal to value, the sum of the array gets as close as possible (in absolute difference) to target.

In case of a tie, return the minimum such integer.

Notice that the answer is not neccesarilly a number from arr.

**Example 1:**

**Input:** arr = [4,9,3], target = 10

**Output:** 3

**Explanation:** When using 3 arr converts to [3, 3, 3] which sums 9 and that's the optimal answer.

**Example 2:**

**Input:** arr = [2,3,5], target = 10

**Output:** 5

**Example 3:**

**Input:** arr = [60864,25176,27249,21296,20204], target = 56803

**Output:** 11361

**Constraints:**

* 1 <= arr.length <= 10^4
* 1 <= arr[i], target <= 10^5

class Solution {

public:

    int findBestValue(vector<int>& arr, int target) {

        int n = arr.size();

        sort(arr.begin(), arr.end());

        vector<int> v(n, 0);

        v[0] = arr[0];

        for (int i = 1; i < n; ++i) v[i] += v[i-1] + arr[i];

        int sum = v.back(), l = min(arr[0], target/n), r = arr.back();

        int sub = INT\_MAX, res = INT\_MAX;

        while (l <= r) {

            int mid = l + (r-l)/2;

            int pos = lower\_bound(arr.begin(), arr.end(), mid)

- arr.begin();

            int cur\_sum = (pos == 0 ? 0 : v[pos-1]) + (n-pos)\*mid;

            if (abs(target-cur\_sum) < sub

|| abs(target-cur\_sum) == sub && mid < res) {

                res = mid;

                sub = abs(target-cur\_sum);

            }

            if (cur\_sum == target) return mid;

            else if (cur\_sum > target) r = mid-1;

            else l = mid+1;

        }

        return res;

    }

};

class Solution {

public:

    int findBestValue(vector<int>& A, int target) {

        sort(A.begin(), A.end());

        int n = A.size(), i = 0;

        while (i < n && target > A[i] \* (n - i))

            target -= A[i++];

        return i == n ? A[n - 1]

: int(round((target - 0.0001) / (n - i)));

    }

};

### 1301. Number of Paths with Max Score

Hard

You are given a square board of characters. You can move on the board starting at the bottom right square marked with the character 'S'.

You need to reach the top left square marked with the character 'E'. The rest of the squares are labeled either with a numeric character 1, 2, ..., 9 or with an obstacle 'X'. In one move you can go up, left or up-left (diagonally) only if there is no obstacle there.

Return a list of two integers: the first integer is the maximum sum of numeric characters you can collect, and the second is the number of such paths that you can take to get that maximum sum, **taken modulo 10^9 + 7**.

In case there is no path, return [0, 0].

**Example 1:**

**Input:** board = ["E23","2X2","12S"]

**Output:** [7,1]

**Example 2:**

**Input:** board = ["E12","1X1","21S"]

**Output:** [4,2]

**Example 3:**

**Input:** board = ["E11","XXX","11S"]

**Output:** [0,0]

**Constraints:**

* 2 <= board.length == board[i].length <= 100

class Solution {

public:

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

    vector<int> pathsWithMaxScore(vector<string>& board) {

        auto n = board.size();

        vector<vector<int>> score(n+1, vector<int>(n+1));

vector<vector<int>> paths(n+1, vector<int>(n+1));

        board[0][0] = board[n-1][n-1] = '0';

        paths[0][0] = 1;

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

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

                if (board[i-1][j-1] == 'X') continue;

                for (auto d : dirs) {

                    auto x = i - d[0], y = j - d[1];

                    auto val = score[x][y]+(board[i-1][j-1]-'0');

                    if (score[i][j] <= val && paths[x][y] > 0) {

                        paths[i][j] = ((score[i][j]==val?paths[i][j]:0)  + paths[x][y]) % 1000000007;

                        score[i][j] = val;

                    }

                }

            }

        }

        return {paths[n][n] ? score[n][n] : 0, paths[n][n]};

    }

};

class Solution {

public:

    long MOD = 1e9 + 7;

    vector<vector<pair<int, long>>> dp;

    int n, m;

    const int dir[4]{1, 0, 1, 1};

    pair<int, long> f(vector<string>& A, int i, int j) {

        if (i >= n || j >= m || A[i][j] == 'X') return {-1, 0};

        if (i == n-1 && j == m-1) return {0, 1};

        if (dp[i][j].second != -1) return dp[i][j];

        int val = A[i][j] == 'E' ? 0 : (A[i][j] - '0');

        vector<pair<int, long>> v;

        int pnt = -1;

        long cnt = 0;

        for (int k = 1; k < 4; ++k) {

            auto t = f(A, i+dir[k-1], j+dir[k]);

            pnt = max(pnt, t.first);

            v.emplace\_back(t);

        }

        if (pnt < 0) return dp[i][j] = {-1, 0};

        for (auto &i : v) {

            if (pnt == i.first) {

                cnt += i.second;

            }

        }

        return dp[i][j] = {pnt+val, cnt % MOD};

    }

    vector<int> pathsWithMaxScore(vector<string>& A) {

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

        dp.resize(n, vector<pair<int, long>> (m, {-1, -1ll}));

        auto p = f(A, 0, 0);

        if (p.first < 0) p.first = 0;

        return {p.first, static\_cast<int>(p.second)};

    }

};

### 1307. Verbal Arithmetic Puzzle★★

Hard

Given an equation, represented by words on left side and the result on right side.

You need to check if the equation is solvable under the following rules:

* Each character is decoded as one digit (0 - 9).
* Every pair of different characters they must map to different digits.
* Each words[i] and result are decoded as one number **without** leading zeros.
* Sum of numbers on left side (words) will equal to the number on right side (result).

Return True if the equation is solvable otherwise return False.

**Example 1:**

**Input:** words = ["SEND","MORE"], result = "MONEY"

**Output:** true

**Explanation:** Map 'S'-> 9, 'E'->5, 'N'->6, 'D'->7, 'M'->1, 'O'->0, 'R'->8, 'Y'->'2'

Such that: "SEND" + "MORE" = "MONEY" , 9567 + 1085 = 10652

**Example 2:**

**Input:** words = ["SIX","SEVEN","SEVEN"], result = "TWENTY"

**Output:** true

**Explanation:** Map 'S'-> 6, 'I'->5, 'X'->0, 'E'->8, 'V'->7, 'N'->2, 'T'->1, 'W'->'3', 'Y'->4

Such that: "SIX" + "SEVEN" + "SEVEN" = "TWENTY" , 650 + 68782 + 68782 = 138214

**Example 3:**

**Input:** words = ["THIS","IS","TOO"], result = "FUNNY"

**Output:** true

**Example 4:**

**Input:** words = ["LEET","CODE"], result = "POINT"

**Output:** false

**Constraints:**

* 2 <= words.length <= 5
* 1 <= words[i].length, result.length <= 7
* words[i], result contains only upper case English letters.
* Number of different characters used on the expression is at most 10.

class Solution {

public:

    int c2i[26], i2c[10];

    bool dfs(int index, int l, int sum, vector<string>& w, string &r) {

        if (l == r.length()) return sum == 0;

        if (index == w.size()) {

            if (c2i[r[l]-'A'] != -1) {

                if (c2i[r[l]-'A'] != sum%10) return false;

                else return dfs(0, l+1, sum/10, w, r);

            }

            else if (i2c[sum%10] == -1) {

                if (l+1 == r.length() && sum%10 == 0)

                    return false;

                c2i[r[l]-'A'] = sum%10;

                i2c[sum%10] = r[l]-'A';

                if (dfs(0, l+1, sum/10, w, r)) return true;

                c2i[r[l]-'A'] = i2c[sum%10] = -1;

                return false;

            }

            else return false;

        }

        if (l >= w[index].length()) return dfs(index+1, l, sum, w, r);

        if (c2i[w[index][l]-'A'] != -1)

            if (l+1 != w[index].length() || c2i[w[index][l]-'A'] != 0)

                return dfs(index+1, l, sum+c2i[w[index][l]-'A'], w, r);

        for(int i = 0; i < 10; i++) {

            if (i2c[i] != -1) continue;

            if (i == 0&&l+1==w[index].length()&&w[index].length() != 1)

             continue;

            i2c[i] = w[index][l]-'A';

            c2i[w[index][l]-'A'] = i;

            if (dfs(index+1, l, sum+i, w, r)) return true;

            i2c[i] = c2i[w[index][l]-'A'] = -1;

        }

        return false;

    }

    bool isSolvable(vector<string>& words, string result) {

        for(auto &w : words) {

            if (w.length() > result.length())

                return false;

        }

        memset(c2i, -1, sizeof(c2i));

        memset(i2c, -1, sizeof(i2c));

        for(auto &s : words) reverse(s.begin(), s.end());

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

        return dfs(0, 0, 0, words, result);

    }

};

### 1312. Minimum Insertion Steps to Make a String Palindrome

Hard

Given a string s. In one step you can insert any character at any index of the string.

Return *the minimum number of steps* to make s palindrome.

A **Palindrome String** is one that reads the same backward as well as forward.

**Example 1:**

**Input:** s = "zzazz"

**Output:** 0

**Explanation:** The string "zzazz" is already palindrome we don't need any insertions.

**Example 2:**

**Input:** s = "mbadm"

**Output:** 2

**Explanation:** String can be "mbdadbm" or "mdbabdm".

**Example 3:**

**Input:** s = "leetcode"

**Output:** 5

**Explanation:** Inserting 5 characters the string becomes "leetcodocteel".

**Example 4:**

**Input:** s = "g"

**Output:** 0

**Example 5:**

**Input:** s = "no"

**Output:** 1

**Constraints:**

* 1 <= s.length <= 500
* All characters of s are lower case English letters.

class Solution {

public:

    int minInsertions(string s) {

        dp.resize(s.length(), vector<int> (s.length(), -1));

        return dfs(0, s.length()-1, s);

    }

private:

    vector<vector<int>> dp;

    int dfs(int i, int j, string &s) {

        if (i >= j) return 0;

        else if (dp[i][j] != -1) return dp[i][j];

        int ret;

        if (s[i] == s[j]) ret = dfs(i+1, j-1, s);

        else ret = 1 + min(dfs(i+1, j, s), dfs(i, j-1, s));

        return dp[i][j] = ret;

    }

};

class Solution {

public:

    int minInsertions(string s) {

        int n = s.length();

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

        for (int i = 0; i < n; ++i)

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

                dp[i + 1][j + 1] = s[i] == s[n - 1 - j] ? dp[i][j] + 1

: max(dp[i][j + 1], dp[i + 1][j]);

        return n - dp[n][n];

    }

};

### 1314. Matrix Block Sum

Medium

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

**Example 1:**

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

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

**Example 2:**

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

**Output:** [[45,45,45],[45,45,45],[45,45,45]]

**Constraints:**

* m == mat.length
* n == mat[i].length
* 1 <= m, n, K <= 100
* 1 <= mat[i][j] <= 100

class Solution {

public:

    int sizeX, sizeY;

    int extractSum(int i, int j, const vector<vector<int> >& sum) {

        if (i < 0 || j < 0) return 0;

        if (i >= sizeX) i = sizeX - 1;

        if (j >= sizeY) j = sizeY - 1;

        return sum[i][j];

    }

    vector<vector<int>> matrixBlockSum(vector<vector<int>>& mat,int K){

        sizeX = mat.size();

        sizeY = mat[0].size();

        vector<vector<int>> sum(sizeX, vector<int>(sizeY, 0));

        // Calculate prefix matrix

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

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

                sum[i][j] = mat[i][j] + extractSum(i-1, j, sum)

+extractSum(i, j-1, sum)-extractSum(i-1, j-1, sum);

            }

        }

        // Use prefix matrix to calculate our sum

        vector<vector<int>> ans(sizeX, vector<int>(sizeY, 0));

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

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

                ans[i][j] = extractSum(i+K,j+K,sum)

-extractSum(i+K, j-K-1,sum)-extractSum (i-K-1, j+K, sum) + extractSum(i-K-1, j-K-1, sum);

            }

        }

        return ans;

    }

};

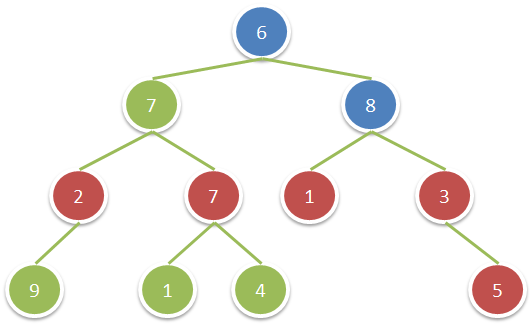
### 1315. Sum of Nodes with Even-Valued Grandparent

Medium

Given a binary tree, return the sum of values of nodes with even-valued grandparent.  (A *grandparent* of a node is the parent of its parent, if it exists.)

If there are no nodes with an even-valued grandparent, return 0.

**Example 1:**

****

**Input:** root = [6,7,8,2,7,1,3,9,null,1,4,null,null,null,5]

**Output:** 18

**Explanation:** The red nodes are the nodes with even-value grandparent while the blue nodes are the even-value grandparents.

**Constraints:**

* The number of nodes in the tree is between 1 and 10^4.
* The value of nodes is between 1 and 100.

/\*\*

 \* 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 sumEvenGrandparent(TreeNode\* root) {

        dfs(root, false, false, 0);

        return res;

    }

private:

    int res = 0;

    void dfs(TreeNode \*p, bool f0, bool f1, int depth) {

        bool f2 = p->val % 2 == 0;

        if (f0) res += p->val;

        if (p->left) dfs(p->left, f1, f2, depth+1);

        if (p->right) dfs(p->right, f1, f2, depth+1);

    }

};

class Solution {

public:

    int sumEvenGrandparent(TreeNode\* root, int p = 1, int gp = 1) {

        return root ? sumEvenGrandparent(root->left, root->val, p)

               + sumEvenGrandparent(root->right, root->val, p)

               + (gp % 2 ? 0 : root->val)  : 0;

    }

};

### 1316. Distinct Echo Substrings★★

Hard

Return the number of **distinct** non-empty substrings of text that can be written as the concatenation of some string with itself (i.e. it can be written as a + a where a is some string).

**Example 1:**

**Input:** text = "abcabcabc"

**Output:** 3

**Explanation:** The 3 substrings are "abcabc", "bcabca" and "cabcab".

**Example 2:**

**Input:** text = "leetcodeleetcode"

**Output:** 2

**Explanation:** The 2 substrings are "ee" and "leetcodeleetcode".

**Constraints:**

* 1 <= text.length <= 2000
* text has only lowercase English letters.

class Solution {

public:

    typedef long long ll;

    static constexpr int A = 1000000007, B = 1000000037;

    class Hash {

    public:

        Hash(const string &s) {

            h[0] = s[0] % B;

            p[0] = 1;

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

                h[i] = (h[i-1]\*A + s[i]) % B;

                p[i] = (p[i-1]\*A) % B;

            }

        }

        // a and b are inclusive

        ll hash(const int a, const int b) {

            if (a) {

                const auto res = (h[b] - h[a-1] \* p[b-a+1]) % B;

                return res < 0 ? B + res : res;

            }   else {

                return h[b];

            }

        }

    private:

        ll h[2000], p[2000];

    };

    int distinctEchoSubstrings(string text) {

        Hash h(text);

        unordered\_set<string\_view> res;

        const char \*p = text.c\_str();

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

            for (int len=1; i+len+len<=text.size(); ++len) {

                if (h.hash(i, i+len-1) == h.hash(i+len, i+len+len-1)) {

                    res.insert(string\_view(p+i, len));

                }

            }

        }

        return res.size();

    }

};

### 1318. Minimum Flips to Make a OR b Equal to c

Medium

Given 3 positives numbers a, b and c. Return the minimum flips required in some bits of a and b to make ( a OR b == c ). (bitwise OR operation).  
Flip operation consists of change **any** single bit 1 to 0 or change the bit 0 to 1 in their binary representation.

**Example 1:**



**Input:** a = 2, b = 6, c = 5

**Output:** 3

**Explanation:** After flips a = 1 , b = 4 , c = 5 such that (a OR b == c)

**Example 2:**

**Input:** a = 4, b = 2, c = 7

**Output:** 1

**Example 3:**

**Input:** a = 1, b = 2, c = 3

**Output:** 0

**Constraints:**

* 1 <= a <= 10^9
* 1 <= b <= 10^9
* 1 <= c <= 10^9

//popcount c++20  == \_\_builtin\_popcount

class Solution {

public:

    int minFlips(int a, int b, int c) {

        return \_\_builtin\_popcount((a | b) ^ c)

+ popcount(a & b & ((a | b) ^ c));

    }

};

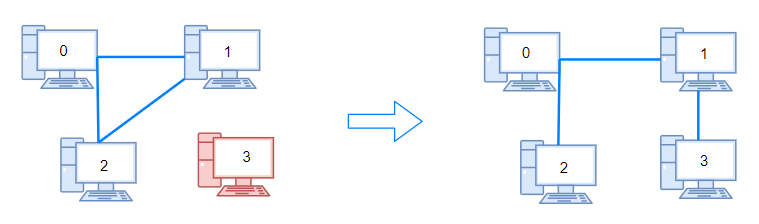
### 1319. Number of Operations to Make Network Connected

Medium

There are n computers numbered from 0 to n-1 connected by ethernet cables connections forming a network where connections[i] = [a, b] represents a connection between computers a and b. Any computer can reach any other computer directly or indirectly through the network.

Given an initial computer network connections. You can extract certain cables between two directly connected computers, and place them between any pair of disconnected computers to make them directly connected. Return the *minimum number of times* you need to do this in order to make all the computers connected. If it's not possible, return -1.

**Example 1:**

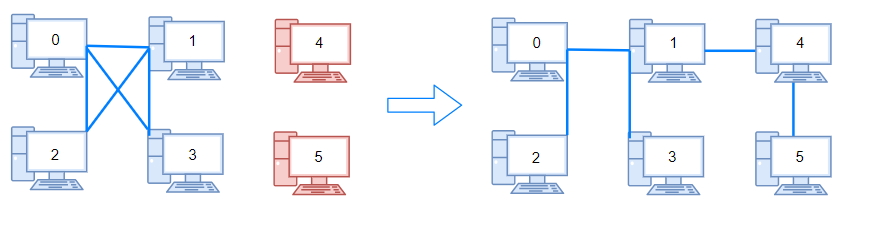
****

**Input:** n = 4, connections = [[0,1],[0,2],[1,2]]

**Output:** 1

**Explanation:** Remove cable between computer 1 and 2 and place between computers 1 and 3.

**Example 2:**

****

**Input:** n = 6, connections = [[0,1],[0,2],[0,3],[1,2],[1,3]]

**Output:** 2

**Example 3:**

**Input:** n = 6, connections = [[0,1],[0,2],[0,3],[1,2]]

**Output:** -1

**Explanation:** There are not enough cables.

**Example 4:**

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

**Output:** 0

**Constraints:**

* 1 <= n <= 10^5
* 1 <= connections.length <= min(n\*(n-1)/2, 10^5)
* connections[i].length == 2
* 0 <= connections[i][0], connections[i][1] < n
* connections[i][0] != connections[i][1]
* There are no repeated connections.
* No two computers are connected by more than one cable.

class Solution {

public:

    int find(vector<int> &ds, int i) {

        return ds[i] < 0 ? i : ds[i] = find(ds, ds[i]);

    }

    int makeConnected(int n, vector<vector<int>>& connections) {

        if (connections.size() < n - 1) return -1;

        vector<int> ds(n, -1);

        for (auto &c : connections) {

            auto i = find(ds, c[0]), j = find(ds, c[1]);

            if (i != j) {

                if (ds[j] < ds[i])

                    swap(i, j);

                ds[i] += ds[j];

                ds[j] = i;

                --n;

            }

        }

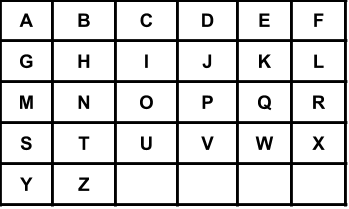
        return n - 1;

    }

};

### 1320. Minimum Distance to Type a Word Using Two Fingers★★

Hard



You have a keyboard layout as shown above in the XY plane, where each English uppercase letter is located at some coordinate, for example, the letter **A** is located at coordinate **(0,0)**, the letter **B** is located at coordinate **(0,1)**, the letter **P** is located at coordinate **(2,3)** and the letter **Z** is located at coordinate **(4,1)**.

Given the string word, return the minimum total distance to type such string using only two fingers. The distance between coordinates **(x1,y1)** and **(x2,y2)** is **|x1 - x2| + |y1 - y2|**.

Note that the initial positions of your two fingers are considered free so don't count towards your total distance, also your two fingers do not have to start at the first letter or the first two letters.

**Example 1:**

**Input:** word = "CAKE"

**Output:** 3

**Explanation:**

Using two fingers, one optimal way to type "CAKE" is:

Finger 1 on letter 'C' -> cost = 0

Finger 1 on letter 'A' -> cost = Distance from letter 'C' to letter 'A' = 2

Finger 2 on letter 'K' -> cost = 0

Finger 2 on letter 'E' -> cost = Distance from letter 'K' to letter 'E' = 1

Total distance = 3

**Example 2:**

**Input:** word = "HAPPY"

**Output:** 6

**Explanation:**

Using two fingers, one optimal way to type "HAPPY" is:

Finger 1 on letter 'H' -> cost = 0

Finger 1 on letter 'A' -> cost = Distance from letter 'H' to letter 'A' = 2

Finger 2 on letter 'P' -> cost = 0

Finger 2 on letter 'P' -> cost = Distance from letter 'P' to letter 'P' = 0

Finger 1 on letter 'Y' -> cost = Distance from letter 'A' to letter 'Y' = 4

Total distance = 6

**Example 3:**

**Input:** word = "NEW"

**Output:** 3

**Example 4:**

**Input:** word = "YEAR"

**Output:** 7

**Constraints:**

* 2 <= word.length <= 300
* Each word[i] is an English uppercase letter.

class Solution {

public:

    int minimumDistance(string word) {

        vector<int> dp(26);

        //dp[the postiopn of the second finger] = save dist

        int res = 0, n = word.size();

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

            int b = word[i] - 'A', c = word[i + 1] - 'A';

            for (int a = 0; a < 26; ++a)

                dp[b] = max(dp[b], dp[a] + d(b, c) - d(a, c));

            res += d(b, c);

        }

        return res - \*max\_element(dp.begin(), dp.end());

    }

    int d(int a, int b) {

        return abs(a / 6 - b / 6) + abs(a % 6 - b % 6);

    }

};

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

Hard

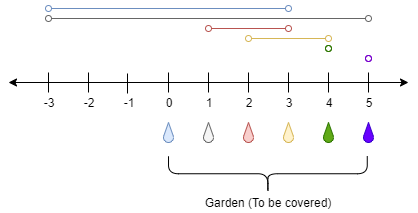
There is a one-dimensional garden on the x-axis. The garden starts at the point 0 and ends at the point n. (i.e The length of the garden is n).

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

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

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

**Example 1:**



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

**Output:** 1

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

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

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

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

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

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

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

**Example 2:**

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

**Output:** -1

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

**Example 3:**

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

**Output:** 3

**Example 4:**

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

**Output:** 2

**Example 5:**

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

**Output:** 1

**Constraints:**

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

class Solution {

public:

    int minTaps(int n, vector<int>& nums) {

        vector<int> maxRight(n+1);

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

            int L = max(0, i-nums[i]);

            int R = min(n, i+nums[i]);

            maxRight[L] = max(maxRight[L], R);

        }

        int cnt = 0, cur\_right\_most = 0;

        for(int i = 0; i < n && cur\_right\_most != n;){

            if (i > cur\_right\_most){

               return -1;

            }

            ++cnt;

            int far = cur\_right\_most;

            while (i <= cur\_right\_most) {

                far = max(far, maxRight[i++]);

            }

            cur\_right\_most = far;

        }

        return cnt;

    }

};

### 1330. Reverse Subarray To Maximize Array Value★★

Hard

You are given an integer array nums. The *value* of this array is defined as the sum of |nums[i]-nums[i+1]| for all 0 <= i < nums.length-1.

You are allowed to select any subarray of the given array and reverse it. You can perform this operation **only once**.

Find maximum possible value of the final array.

**Example 1:**

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

**Output:** 10

**Explanation:** By reversing the subarray [3,1,5] the array becomes [2,5,1,3,4] whose value is 10.

**Example 2:**

**Input:** nums = [2,4,9,24,2,1,10]

**Output:** 68

**Constraints:**

* 1 <= nums.length <= 3\*10^4
* -10^5 <= nums[i] <= 10^5

class Solution {

public:

    int maxValueAfterReverse(vector<int>& A) {

        int n = A.size();

        //a [b...c] d => a [c...b] d

        //max(|a-c|+|b-d|-|a-b|-|c-d|)

        //max(a-c+b-d, c-a+b-d, a-c+d-b, c-a+d-b) - |a-b|-|c-d|

        //max(t0, t1, t2, t3)

        //t0 = c-d-|c-d| + max(b-a-|a-b|) = c-d-|c-d| +t0

        int t0 = -1e9, t1 = -1e9, t2 = -1e9;

        int t3 = -1e9, diff = 0, res = 0;

        for (int i = 1; i < n; i++) {

            diff = max(diff, -abs(A[i]-A[i-1]) + abs(A[n-1]-A[i-1]));

        }

        for (int i = n-2; i >= 0; i--) {

            diff = max(diff, -abs(A[i]-A[i+1]) + abs(A[0]-A[i+1]));

        }

        for (int i = 1; i < n; ++i) {

            res += abs(A[i] - A[i-1]);

            int tt0 = A[i-1] - A[i] - abs(A[i-1] - A[i]);

            int tt1 = A[i-1] + A[i] - abs(A[i-1] - A[i]);

            int tt2 = -A[i-1] - A[i] - abs(A[i-1] - A[i]);

            int tt3 = -A[i-1] + A[i] - abs(A[i-1] - A[i]);

            diff = max({diff, t0+tt3, t1+tt2, t2+tt1, t3+tt0});

            t0 = max(tt0, t0);

            t1 = max(tt1, t1);

            t2 = max(tt2, t2);

            t3 = max(tt3, t3);

        }

        return res + diff;

    }

};

### 1334. Find the City With the Smallest Number of Neighbor at a Threshold Distance

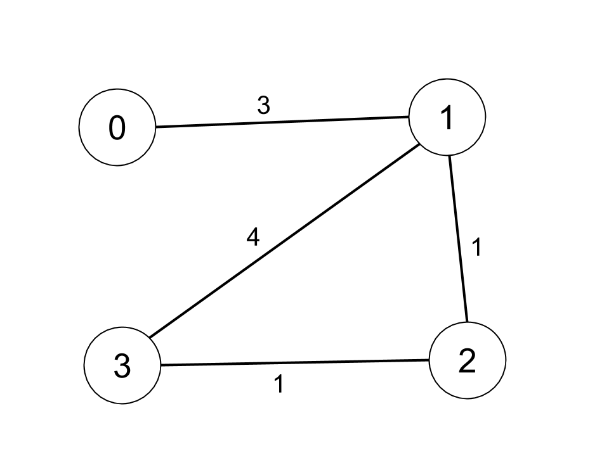
Medium

There are n cities numbered from 0 to n-1. Given the array edges where edges[i] = [fromi, toi, weighti] represents a bidirectional and weighted edge between cities fromi and toi, and given the integer distanceThreshold.

Return the city with the smallest numberof cities that are reachable through some path and whose distance is **at most** distanceThreshold, If there are multiple such cities, return the city with the greatest number.

Notice that the distance of a path connecting cities ***i*** and ***j*** is equal to the sum of the edges' weights along that path.

**Example 1:**



**Input:** n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], distanceThreshold = 4

**Output:** 3

**Explanation:** The figure above describes the graph.

The neighboring cities at a distanceThreshold = 4 for each city are:

City 0 -> [City 1, City 2]

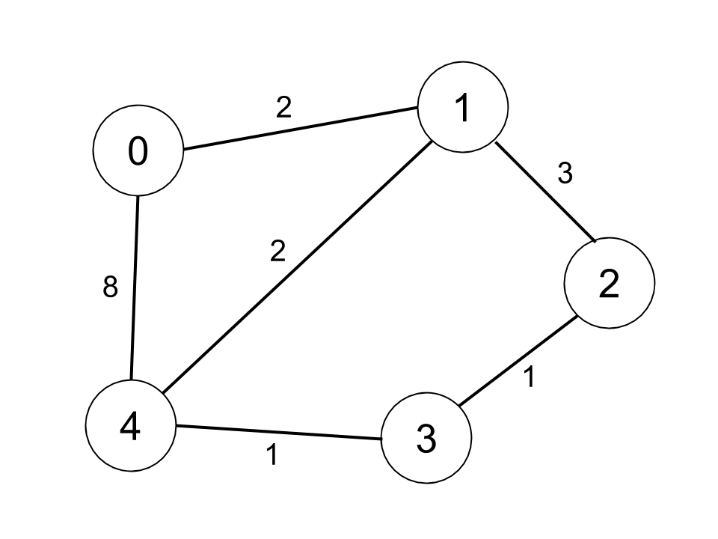
City 1 -> [City 0, City 2, City 3]

City 2 -> [City 0, City 1, City 3]

City 3 -> [City 1, City 2]

Cities 0 and 3 have 2 neighboring cities at a distanceThreshold = 4, but we have to return city 3 since it has the greatest number.

**Example 2:**

****

**Input:** n = 5, edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], distanceThreshold = 2

**Output:** 0

**Explanation:** The figure above describes the graph.

The neighboring cities at a distanceThreshold = 2 for each city are:

City 0 -> [City 1]

City 1 -> [City 0, City 4]

City 2 -> [City 3, City 4]

City 3 -> [City 2, City 4]

City 4 -> [City 1, City 2, City 3]

The city 0 has 1 neighboring city at a distanceThreshold = 2.

**Constraints:**

* 2 <= n <= 100
* 1 <= edges.length <= n \* (n - 1) / 2
* edges[i].length == 3
* 0 <= fromi < toi < n
* 1 <= weighti, distanceThreshold <= 10^4
* All pairs (fromi, toi) are distinct.

class Solution {

public:

    int findTheCity(int n, vector<vector<int>>& edges, int distanceThreshold) {

        vector<vector<int>> dis(n, vector(n, 10001));

        int res = 0, smallest = n;

        for (auto& e : edges)

            dis[e[0]][e[1]] = dis[e[1]][e[0]] = e[2];

        for (int i = 0; i < n; ++i)

            dis[i][i] = 0;

        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]);

        for (int i = 0; i < n; i++) {

            int count = 0;

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

                if (dis[i][j] <= distanceThreshold)

                    ++count;

            if (count <= smallest) {

                res = i;

                smallest = count;

            }

        }

        return res;

    }

};

### 1335. Minimum Difficulty of a Job Schedule

Hard

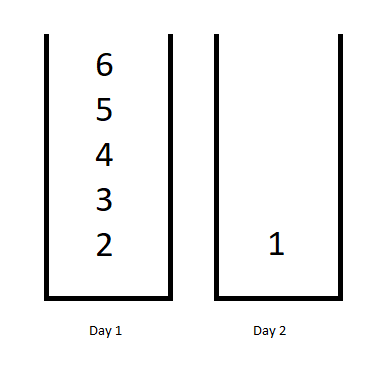
You want to schedule a list of jobs in d days. Jobs are dependent (i.e To work on the i-th job, you have to finish all the jobs j where 0 <= j < i).

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

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

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

**Example 1:**



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

**Output:** 7

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

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

The difficulty of the schedule = 6 + 1 = 7

**Example 2:**

**Input:** jobDifficulty = [9,9,9], d = 4

**Output:** -1

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

**Example 3:**

**Input:** jobDifficulty = [1,1,1], d = 3

**Output:** 3

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

**Example 4:**

**Input:** jobDifficulty = [7,1,7,1,7,1], d = 3

**Output:** 15

**Example 5:**

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

**Output:** 843

**Constraints:**

* 1 <= jobDifficulty.length <= 300
* 0 <= jobDifficulty[i] <= 1000
* 1 <= d <= 10

class Solution {

public:

    int minDifficulty(vector<int>& jobs, int days) {

        const int n = (int)jobs.size();

        if (n < days) return -1;

        vector<int> dp(n), old(n); // new and old dp

        // fill dp for d = 1

        dp[0] = jobs[0];

        for (int i = 1; i < n; i++)

            dp[i] = max(dp[i-1], jobs[i]);

        // fill dp for remaining days

        for (int d = 1; d < days; d++) {

            swap(dp, old);

            // monotonic and minimum stack {oldBest, curMax, bestSoFar}

            vector<array<int,3>> stk = {{1<<30,1<<30,1<<30}};

            for (int i = d; i < n; i++) {

                int oldBest = old[i-1];

                while (stk.back()[1] <= jobs[i]) {

                    oldBest = min(oldBest, stk.back()[0]);

                    stk.pop\_back();

                }

                stk.push\_back({oldBest, jobs[i], min(oldBest + jobs[i], stk.back()[2])});

                dp[i] = stk.back()[2];

            }

        }

        return dp[n-1];

    }

};

### 1340. Jump Game V

Hard

Given an array of integers arr and an integer d. In one step you can jump from index i to index:

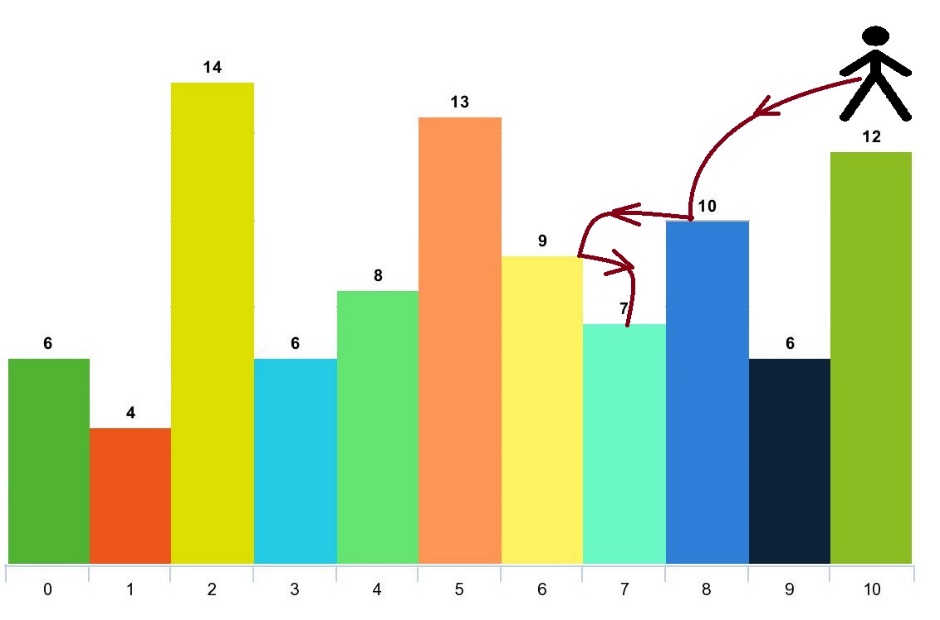
* i + x where: i + x < arr.length and 0 < x <= d.
* i - x where: i - x >= 0 and 0 < x <= d.

In addition, you can only jump from index i to index j if arr[i] > arr[j] and arr[i] > arr[k] for all indices k between i and j (More formally min(i, j) < k < max(i, j)).

You can choose any index of the array and start jumping. Return *the maximum number of indices* you can visit.

Notice that you can not jump outside of the array at any time.

**Example 1:**



**Input:** arr = [6,4,14,6,8,13,9,7,10,6,12], d = 2

**Output:** 4

**Explanation:** You can start at index 10. You can jump 10 --> 8 --> 6 --> 7 as shown.

Note that if you start at index 6 you can only jump to index 7. You cannot jump to index 5 because 13 > 9. You cannot jump to index 4 because index 5 is between index 4 and 6 and 13 > 9.

Similarly You cannot jump from index 3 to index 2 or index 1.

**Example 2:**

**Input:** arr = [3,3,3,3,3], d = 3

**Output:** 1

**Explanation:** You can start at any index. You always cannot jump to any index.

**Example 3:**

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

**Output:** 7

**Explanation:** Start at index 0. You can visit all the indicies.

**Example 4:**

**Input:** arr = [7,1,7,1,7,1], d = 2

**Output:** 2

**Example 5:**

**Input:** arr = [66], d = 1

**Output:** 1

**Constraints:**

* 1 <= arr.length <= 1000
* 1 <= arr[i] <= 10^5
* 1 <= d <= arr.length

class Solution {

public:

        int dp[1001] = {};

    int dfs(vector<int>& arr, int i, int d, int res = 1) {

        if (dp[i]) return dp[i];

        for (auto j = i + 1; j <= min(i + d, (int)arr.size() - 1) && arr[j] < arr[i]; ++j)

            res = max(res, 1 + dfs(arr, j, d));

        for (auto j = i - 1; j >= max(0, i - d) && arr[j] < arr[i]; --j)

            res = max(res, 1 + dfs(arr, j, d));

        return dp[i] = res;

    }

    int maxJumps(vector<int>& arr, int d, int res = 1) {

        for (auto i = 0; i < arr.size(); ++i)

            res = max(res, dfs(arr, i, d));

        return res;

    }

};

class Solution {

public:

    struct A {

        int h, i, j = 1;

    };

    int maxJumps(vector<int>& arr, int d) {

        vector<A> V;

        int ret = 1, i, n = arr.size(), j, h, vsiz, k, l;

        for (i = 0; i < n; ++i) {

            h = arr[i];

            vsiz = V.size();

            if (V.empty() || V[vsiz-1].h >= h) {

                V.push\_back({h, i, 1});

            } else {

                j = 1;

                while (!V.empty() && V[vsiz-1].h < h) {

                    for (k = vsiz-2; k >= 0 && V[k].h == V[vsiz-1].h; --k);

                    // populate V[l] to left and right

                    for (l = vsiz-1; l > k; --l) {

                        // populate left

                        if (k >= 0 && V[l].i - V[k].i <= d)

                            V[k].j = max(V[k].j, V[l].j + 1);

                        // populate right

                        if (i - V[l].i <= d)

                            j = max(j, V[l].j + 1);

                        // retire

                        ret = max(ret, V[l].j);

                        V.pop\_back();

                    }

                    vsiz = V.size();

                }

                V.push\_back({h, i, j});

            }

        }

        // retire the rest

        while (!V.empty()) {

            vsiz = V.size();

            for (k = vsiz-2; k >= 0 && V[k].h == V[vsiz-1].h; --k);

            // populate V[l] to left and right

            for (l = vsiz-1; l > k; --l) {

                // populate left

                if (k >= 0 && V[l].i - V[k].i <= d)

                    V[k].j = max(V[k].j, V[l].j + 1);

                // retire

                ret = max(ret, V[l].j);

                V.pop\_back();

            }

        }

        return ret;

    }

};

### 1345. Jump Game IV

Hard

Given an array of integers arr, you are initially positioned at the first index of the array.

In one step you can jump from index i to index:

* i + 1 where: i + 1 < arr.length.
* i - 1 where: i - 1 >= 0.
* j where: arr[i] == arr[j] and i != j.

Return *the minimum number of steps* to reach the **last index** of the array.

Notice that you can not jump outside of the array at any time.

**Example 1:**

**Input:** arr = [100,-23,-23,404,100,23,23,23,3,404]

**Output:** 3

**Explanation:** You need three jumps from index 0 --> 4 --> 3 --> 9. Note that index 9 is the last index of the array.

**Example 2:**

**Input:** arr = [7]

**Output:** 0

**Explanation:** Start index is the last index. You don't need to jump.

**Example 3:**

**Input:** arr = [7,6,9,6,9,6,9,7]

**Output:** 1

**Explanation:** You can jump directly from index 0 to index 7 which is last index of the array.

**Example 4:**

**Input:** arr = [6,1,9]

**Output:** 2

**Example 5:**

**Input:** arr = [11,22,7,7,7,7,7,7,7,22,13]

**Output:** 3

**Constraints:**

* 1 <= arr.length <= 5 \* 10^4
* -10^8 <= arr[i] <= 10^8

class Solution {

public:

    int minJumps(vector<int>& arr) {

        int n = arr.size();

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

        for (int i = 0; i < n; i++)

            mp[arr[i]].push\_back(i);

        vector<bool> visited(n);

        visited[0] = true;

        queue<int> q;

        q.push(0);

        int step = 0;

        while (!q.empty()) {

            for (int sz = q.size(); sz > 0; --sz) {

                int i = q.front();

                q.pop();

                if (i == n - 1) return step;

                vector<int>& next = mp[arr[i]];

                next.push\_back(i - 1);

                next.push\_back(i + 1);

                for (int j : next) {

                    if (j >= 0 && j < n && !visited[j]) {

                        visited[j] = true;

                        q.push(j);

                    }

                }

                next.clear();

            }

            step++;

        }

        return 0;

    }

};

### 1349. Maximum Students Taking Exam★★

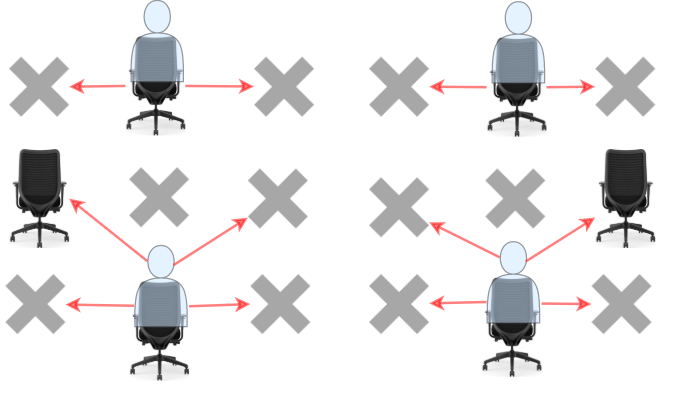
Hard

Given a m \* n matrix seats  that represent seats distributions in a classroom. If a seat is broken, it is denoted by '#' character otherwise it is denoted by a '.' character.

Students can see the answers of those sitting next to the left, right, upper left and upper right, but he cannot see the answers of the student sitting directly in front or behind him. Return the **maximum**number of students that can take the exam together without any cheating being possible..

Students must be placed in seats in good condition.

**Example 1:**



**Input:** seats = [["#",".","#","#",".","#"],

  [".","#","#","#","#","."],

  ["#",".","#","#",".","#"]]

**Output:** 4

**Explanation:** Teacher can place 4 students in available seats so they don't cheat on the exam.

**Example 2:**

**Input:** seats = [[".","#"],

  ["#","#"],

  ["#","."],

  ["#","#"],

  [".","#"]]

**Output:** 3

**Explanation:** Place all students in available seats.

**Example 3:**

**Input:** seats = [["#",".","**.**",".","#"],

  ["**.**","#","**.**","#","**.**"],

  ["**.**",".","#",".","**.**"],

  ["**.**","#","**.**","#","**.**"],

  ["#",".","**.**",".","#"]]

**Output:** 10

**Explanation:** Place students in available seats in column 1, 3 and 5.

**Constraints:**

* seats contains only characters '.' and'#'.
* m == seats.length
* n == seats[i].length
* 1 <= m <= 8
* 1 <= n <= 8

class Solution {

public:

    int maxStudents(vector<vector<char>>& seats) {

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

        vector<vector<int>> f(2, vector<int>(1 << n, -1));

        f[0][0] = 0;

        for (int i = 0; i < m; ++i) {

            auto &pre = f[k], &cur = f[k^1];

            int valid = 0;

            for (auto c : seats[i]) valid = valid \* 2 + (c == '.');

            for (int j = 0; j < (1 << n); ++j) {

                if ((j & valid) == j && !(j & (j >> 1))) {

                    for (int k = 0; k < (1 << n); ++k) {

                       if (j&(k>>1) || (j>>1)&k || pre[k]==-1)

                            continue;

                       cur[j]=max(cur[j],pre[k]+\_\_builtin\_popcount(j));

                    }

                }

            }

            k ^= 1;

        }

        return \*max\_element(f[k].begin(), f[k].end());

    }

};

### 1353. Maximum Number of Events That Can Be Attended★★

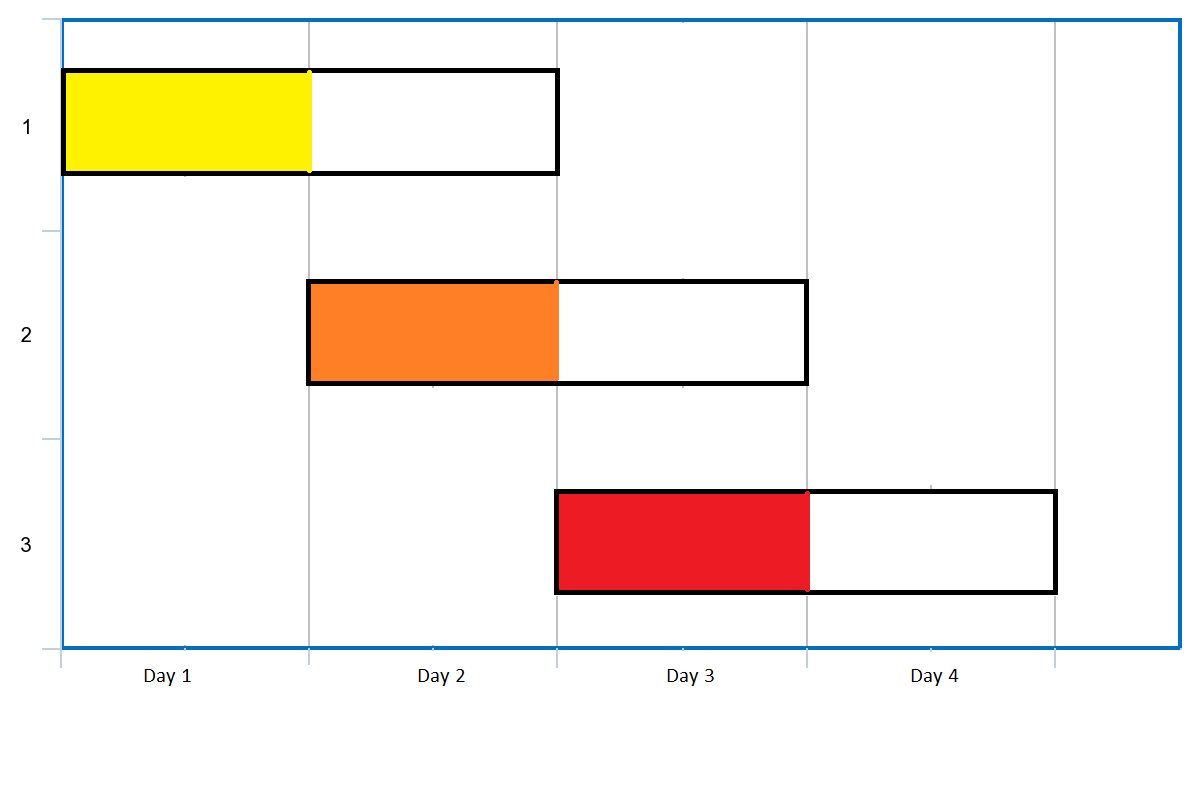
Medium

Given an array of events where events[i] = [startDayi, endDayi]. Every event i starts at startDayiand ends at endDayi.

You can attend an event i at any day d where startTimei <= d <= endTimei. Notice that you can only attend one event at any time d.

Return *the maximum number of events*you can attend.

**Example 1:**



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

**Output:** 3

**Explanation:** You can attend all the three events.

One way to attend them all is as shown.

Attend the first event on day 1.

Attend the second event on day 2.

Attend the third event on day 3.

**Example 2:**

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

**Output:** 4

**Example 3:**

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

**Output:** 4

**Example 4:**

**Input:** events = [[1,100000]]

**Output:** 1

**Example 5:**

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

**Output:** 7

**Constraints:**

* 1 <= events.length <= 10^5
* events[i].length == 2
* 1 <= events[i][0] <= events[i][1] <= 10^5

class Solution {

public:

    int maxEvents(vector<vector<int>>& A) {

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

        sort(A.begin(), A.end());

        int i = 0, res = 0, d = 0, n = A.size();

        while (!pq.empty() || i < n) {

            if (pq.empty()) d = A[i][0];

            while (i < n && A[i][0] <= d)

                pq.push(A[i++][1]);

            pq.pop();

            ++res, ++d;

            while (!pq.empty() && pq.top() < d) pq.pop();

        }

        return res;

    }

};

### 1354. Construct Target Array With Multiple Sums★★

Hard

Given an array of integers target. From a starting array, A consisting of all 1's, you may perform the following procedure :

* let x be the sum of all elements currently in your array.
* choose index i, such that 0 <= i < target.size and set the value of A at index i to x.
* You may repeat this procedure as many times as needed.

Return True if it is possible to construct the target array from A otherwise return False.

**Example 1:**

**Input:** target = [9,3,5]

**Output:** true

**Explanation:** Start with [1, 1, 1]

[1, 1, 1], sum = 3 choose index 1

[1, 3, 1], sum = 5 choose index 2

[1, 3, 5], sum = 9 choose index 0

[9, 3, 5] Done

**Example 2:**

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

**Output:** false

**Explanation:** Impossible to create target array from [1,1,1,1].

**Example 3:**

**Input:** target = [8,5]

**Output:** true

* N == target.length
* 1 <= target.length <= 5 \* 10^4
* 1 <= target[i] <= 10^9

//逆推

//设变化后的数组为 s X X X X ,其中s为最大值,和为sum

//则变化而来的数字一定为s,则其他数字之和为sum-s

//又已知变化之前的数字和为s,变化前的数字为s-(sum-s)

//即变化之前的数组为s-(sum-s) X X X X

//即只是将最大的数字减去sum-s即可

//又因为sum-s为其他数字之和,若变化后的数字仍为最大值,只需再减去sum-s即可

//即s = s % (sum-s) 这一步将多次变化合一

//最后sum更新俄为现在的数组和

class Solution {

public:

    bool isPossible(vector<int>& A) {

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

        priority\_queue<int> pq(A.begin(), A.end());

        while (1) {

            long s = pq.top(), other = sum-s;

            pq.pop();

            if (s == 1 || other == 1) return true;

            if (s < other || other == 0 || s % other == 0)return false;

            s %= other;

            sum = other + s;

            pq.push(static\_cast<int>(s));

        }

    }

};

### 1363. Largest Multiple of Three★

Hard

Given an integer array of digits, return the largest multiple of **three** that can be formed by concatenating some of the given digits in any order.

Since the answer may not fit in an integer data type, return the answer as a string.

If there is no answer return an empty string.

**Example 1:**

**Input:** digits = [8,1,9]

**Output:** "981"

**Example 2:**

**Input:** digits = [8,6,7,1,0]

**Output:** "8760"

**Example 3:**

**Input:** digits = [1]

**Output:** ""

**Example 4:**

**Input:** digits = [0,0,0,0,0,0]

**Output:** "0"

**Constraints:**

* 1 <= digits.length <= 10^4
* 0 <= digits[i] <= 9
* The returning answer must not contain unnecessary leading zeros.

class Solution {

public:

    string largestMultipleOfThree(vector<int>& digits,string res = ""){

      int m1[] = {1, 4, 7, 2, 5, 8}, m2[] = {2, 5, 8, 1, 4, 7};

      int sum = 0, ds[10] = {};

      for (auto d : digits) {

          ++ds[d];

          sum += d;

      }

      while (sum % 3 != 0) {

        for (auto i : sum % 3 == 1 ? m1 : m2) {

          if (ds[i]) {

            --ds[i];

            sum -= i;

            break;

          }

        }

      }

      for (int i = 9; i >= 0; --i)

         res += string(ds[i], '0' + i);

      return res.size() && res[0] == '0' ? "0" : res;

    }

};

### 1368. Minimum Cost to Make at Least One Valid Path in a Grid★★

Hard

Given a *m* x *n* grid. Each cell of the grid has a sign pointing to the next cell you should visit if you are currently in this cell. The sign of grid[i][j] can be:

* **1** which means go to the cell to the right. (i.e go from grid[i][j] to grid[i][j + 1])
* **2** which means go to the cell to the left. (i.e go from grid[i][j] to grid[i][j - 1])
* **3** which means go to the lower cell. (i.e go from grid[i][j] to grid[i + 1][j])
* **4** which means go to the upper cell. (i.e go from grid[i][j] to grid[i - 1][j])

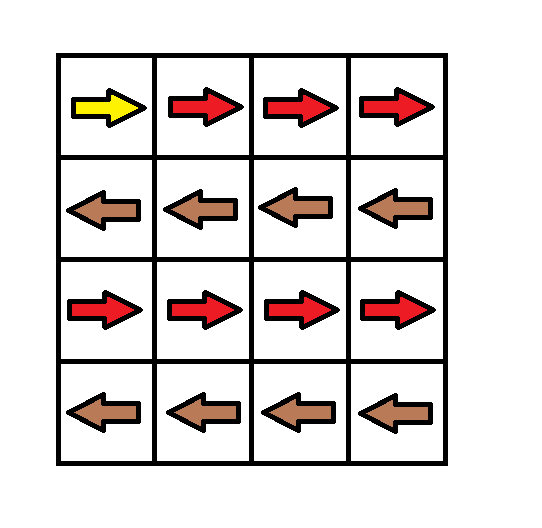
Notice that there could be some **invalid signs** on the cells of the grid which points outside the grid.

You will initially start at the upper left cell (0,0). A valid path in the grid is a path which starts from the upper left cell (0,0) and ends at the bottom-right cell (m - 1, n - 1) following the signs on the grid. The valid path **doesn't have to be the shortest**.

You can modify the sign on a cell with cost = 1. You can modify the sign on a cell **one time only**.

Return *the minimum cost* to make the grid have at least one valid path.

**Example 1:**



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

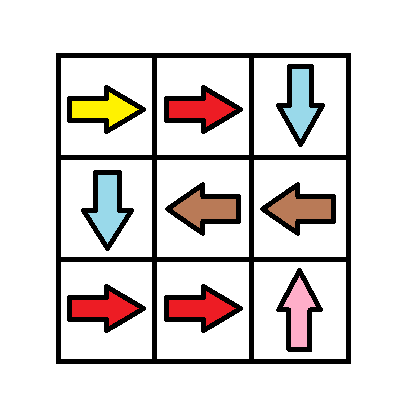
**Output:** 3

**Explanation:** You will start at point (0, 0).

The path to (3, 3) is as follows. (0, 0) --> (0, 1) --> (0, 2) --> (0, 3) change the arrow to down with cost = 1 --> (1, 3) --> (1, 2) --> (1, 1) --> (1, 0) change the arrow to down with cost = 1 --> (2, 0) --> (2, 1) --> (2, 2) --> (2, 3) change the arrow to down with cost = 1 --> (3, 3)

The total cost = 3.

**Example 2:**

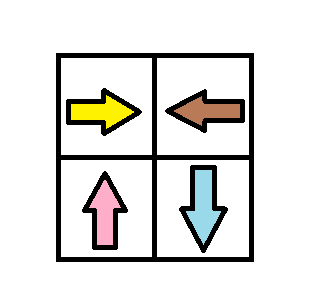


**Input:** grid = [[1,1,3],[3,2,2],[1,1,4]]

**Output:** 0

**Explanation:** You can follow the path from (0, 0) to (2, 2).

**Example 3:**



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

**Output:** 1

**Example 4:**

**Input:** grid = [[2,2,2],[2,2,2]]

**Output:** 3

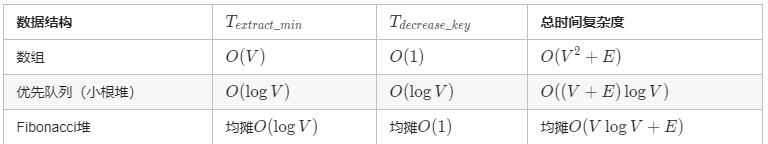
**Example 5:**

**Input:** grid = [[4]]

**Output:** 0

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 100



class Solution {

public:

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

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

    using pii = pair<int, int>;

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

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

        vector<vector<int>> dist(n, vector<int>(m, 0x3f3f3f3f));

        dist[0][0] = 0;

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

        pq.emplace(0, 0);

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

        while (!pq.empty()) {

            auto [cnt, pos] = pq.top();

            pq.pop();

            int y = pos % m, x = pos / m;

            if (vis[x][y]) continue;

            vis[x][y] = true;

            for (int k = 1; k <= 4; ++k) {

                int xx = x + dx[k], yy = y + dy[k];

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

                int nd = cnt + (grid[x][y] == k ? 0 : 1);

                if (nd >= dist[xx][yy]) continue;

                dist[xx][yy] = nd;

                pq.emplace(dist[xx][yy], xx \* m + yy);

            }

        }

        return dist[n - 1][m - 1];

    }

};

class Solution {

public:

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

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

    using pii = pair<int, int>;

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

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

        deque<pii> pq;

        pq.emplace\_back(0, 0);

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

        while (!pq.empty()) {

            auto [cnt, pos] = pq.front();

            pq.pop\_front();

            int x = pos / m, y = pos % m;

            if (vis[x][y]) continue;

            vis[x][y] = true;

            if (x == n - 1 && y == m - 1) return cnt;

            for (int k = 1; k <= 4; ++k) {

                int xx = x + dx[k], yy = y + dy[k];

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

                if (grid[x][y] == k)

                    pq.emplace\_front(cnt, xx \* m + yy);

                else

                    pq.emplace\_back(cnt + 1, xx \* m + yy);

            }

        }

        return 0;

    }

};

### 1371. Find the Longest Substring Containing Vowels in Even Counts★★

Medium

Given the string s, return the size of the longest substring containing each vowel an even number of times. That is, 'a', 'e', 'i', 'o', and 'u' must appear an even number of times.

**Example 1:**

**Input:** s = "eleetminicoworoep"

**Output:** 13

**Explanation:** The longest substring is "leetminicowor" which contains two each of the vowels: **e**, **i** and **o** and zero of the vowels: **a** and **u**.

**Example 2:**

**Input:** s = "leetcodeisgreat"

**Output:** 5

**Explanation:** The longest substring is "leetc" which contains two e's.

**Example 3:**

**Input:** s = "bcbcbc"

**Output:** 6

**Explanation:** In this case, the given string "bcbcbc" is the longest because all vowels: **a**, **e**, **i**, **o** and **u** appear zero times.

**Constraints:**

* 1 <= s.length <= 5 x 10^5
* s contains only lowercase English letters.

class Solution {

public:

    int findTheLongestSubstring(string s) {

        unordered\_map<int, int> m{{0, -1}};

        int res = 0, n = s.length(), cur = 0;

        for (int i = 0; i < n; i++) {

            cur ^= 1 << string("aeiou").find(s[i]) + 1 >> 1;

            if (!m.count(cur)) m[cur] = i;

            res = max(res, i - m[cur]);

        }

        return res;

    }

};

### 1373. Maximum Sum BST in Binary Tree

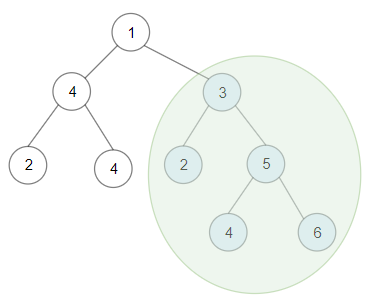
Hard

Given a **binary tree** root, the task is to return the maximum sum of all keys of **any** sub-tree which is also a Binary Search Tree (BST).

Assume a BST is defined as follows:

* The left subtree of a node contains only nodes with keys **less than** the node's key.
* The right subtree of a node contains only nodes with keys **greater than** the node's key.
* Both the left and right subtrees must also be binary search trees.

**Example 1:**

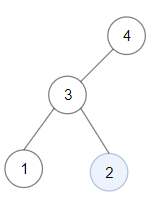


**Input:** root = [1,4,3,2,4,2,5,null,null,null,null,null,null,4,6]

**Output:** 20

**Explanation:** Maximum sum in a valid Binary search tree is obtained in root node with key equal to 3.

**Example 2:**



**Input:** root = [4,3,null,1,2]

**Output:** 2

**Explanation:** Maximum sum in a valid Binary search tree is obtained in a single root node with key equal to 2.

**Example 3:**

**Input:** root = [-4,-2,-5]

**Output:** 0

**Explanation:** All values are negatives. Return an empty BST.

**Example 4:**

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

**Output:** 6

**Example 5:**

**Input:** root = [5,4,8,3,null,6,3]

**Output:** 7

**Constraints:**

* The given binary tree will have between 1 and 40000 nodes.
* Each node's value is between [-4 \* 10^4 , 4 \* 10^4].

/\*\*

 \* 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 res = 0;

    int maxSumBST(TreeNode\* root) {

        findMaxBST(root);

        return res;

    }

    struct BSTNode {

        bool isBST;

        int mn, mx, sum;

    };

    BSTNode findMaxBST(TreeNode\* p) {

        if (p == nullptr) return {true, INT\_MAX, INT\_MIN, 0};

        BSTNode left = findMaxBST(p->left);

        BSTNode right = findMaxBST(p->right);

        bool isBST = left.isBST && right.isBST && left.mx < p->val

 && p->val < right.mn;

        int sum = p->val + left.sum + right.sum;

        if (isBST) res = max(res, sum);

        return {isBST,min(p->val, left.mn),max(p->val, right.mx),sum};

    }

};

### 1375. Bulb Switcher III★

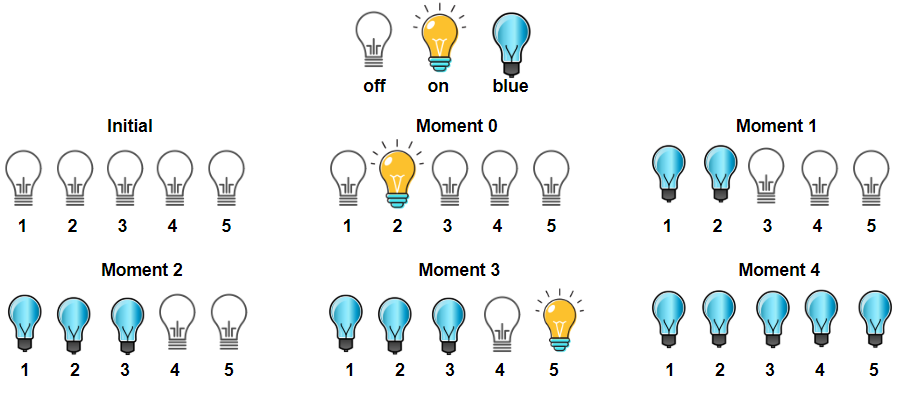
Medium

There is a room with n bulbs, numbered from 1 to n, arranged in a row from left to right. Initially, all the bulbs are turned off.

At moment *k* (for *k* from 0 to n - 1), we turn on the light[k] bulb. A bulb **change color to blue** only if it is on and all the previous bulbs (to the left) are turned on too.

Return the number of moments in which **all turned on** bulbs **are blue.**

**Example 1:**



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

**Output:** 3

**Explanation:** All bulbs turned on, are blue at the moment 1, 2 and 4.

**Example 2:**

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

**Output:** 2

**Explanation:** All bulbs turned on, are blue at the moment 3, and 4 (index-0).

**Example 3:**

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

**Output:** 1

**Explanation:** All bulbs turned on, are blue at the moment 3 (index-0).

Bulb 4th changes to blue at the moment 3.

**Example 4:**

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

**Output:** 3

**Example 5:**

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

**Output:** 6

**Constraints:**

* n == light.length
* 1 <= n <= 5 \* 10^4
* light is a permutation of  [1, 2, ..., n]

class Solution {

public:

    int numTimesAllBlue(vector<int>& A) {

        int right = 0, res = 0, n = A.size();

        for (int i = 0; i < n; ++i)

            res += (right = max(right, A[i])) == i + 1;

        return res;

    }

};

class Solution {

public:

    int numTimesAllBlue(vector<int>& light) {

        set<int> off;

        int n = light.size();

        for (int i = 1; i <= n; ++i) off.insert(i);

        int res = 0;

        for (int i = 0; i < n; ++i) {

            off.erase(light[i]);

            if (off.empty() || \*off.begin() == i+2) ++res;

        }

        return res;

    }

};

### 1383. Maximum Performance of a Team★★

Hard

There are n engineers numbered from 1 to n and two arrays: speed and efficiency, where speed[i] and efficiency[i] represent the speed and efficiency for the i-th engineer respectively. *Return the maximum****performance****of a team composed of at most k engineers, since the answer can be a huge number, return this modulo 10^9 + 7.*

The **performance** of a team is the sum of their engineers' speeds multiplied by the minimum efficiency among their engineers.

**Example 1:**

**Input:** n = 6, speed = [2,10,3,1,5,8], efficiency = [5,4,3,9,7,2], k = 2

**Output:** 60

**Explanation:**

We have the maximum performance of the team by selecting engineer 2 (with speed=10 and efficiency=4) and engineer 5 (with speed=5 and efficiency=7). That is, performance = (10 + 5) \* min(4, 7) = 60.

**Example 2:**

**Input:** n = 6, speed = [2,10,3,1,5,8], efficiency = [5,4,3,9,7,2], k = 3

**Output:** 68

**Explanation:**

This is the same example as the first but k = 3. We can select engineer 1, engineer 2 and engineer 5 to get the maximum performance of the team. That is, performance = (2 + 10 + 5) \* min(5, 4, 7) = 68.

**Example 3:**

**Input:** n = 6, speed = [2,10,3,1,5,8], efficiency = [5,4,3,9,7,2], k = 4

**Output:** 72

* 1 <= n <= 10^5
* speed.length == n
* efficiency.length == n
* 1 <= speed[i] <= 10^5
* 1 <= efficiency[i] <= 10^8
* 1 <= k <= n

class Solution {

public:

    int maxPerformance(int n, vector<int>& speed, vector<int>& efficiency, int k) {

        vector<pair<int, int>> ess;

        for (int i = 0; i < n; ++i)

            ess.emplace\_back(efficiency[i], speed[i]);

        sort(rbegin(ess), rend(ess));

        long sumS = 0, res = 0;

        priority\_queue <int, vector<int>, greater<int>> pq; //min heap

        for(auto& [e, s]: ess){

            pq.emplace(s);

            sumS += s;

            if (pq.size() > k) {

                sumS -= pq.top();

                pq.pop();

            }

            res = max(res, sumS \* e);

        }

        return res % (int)(1e9+7);

    }

};

### 1388. Pizza With 3n Slices★★

Hard

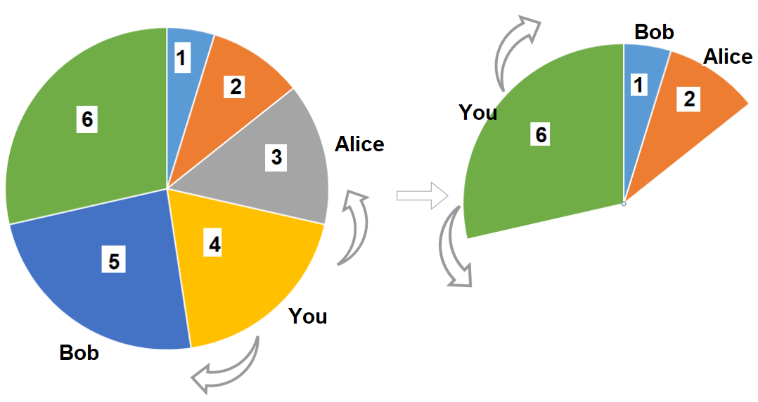
There is a pizza with 3n slices of varying size, you and your friends will take slices of pizza as follows:

* You will pick **any** pizza slice.
* Your friend Alice will pick next slice in anti clockwise direction of your pick.
* Your friend Bob will pick next slice in clockwise direction of your pick.
* Repeat until there are no more slices of pizzas.

Sizes of Pizza slices is represented by circular array slices in clockwise direction.

Return the maximum possible sum of slice sizes which you can have.

**Example 1:**

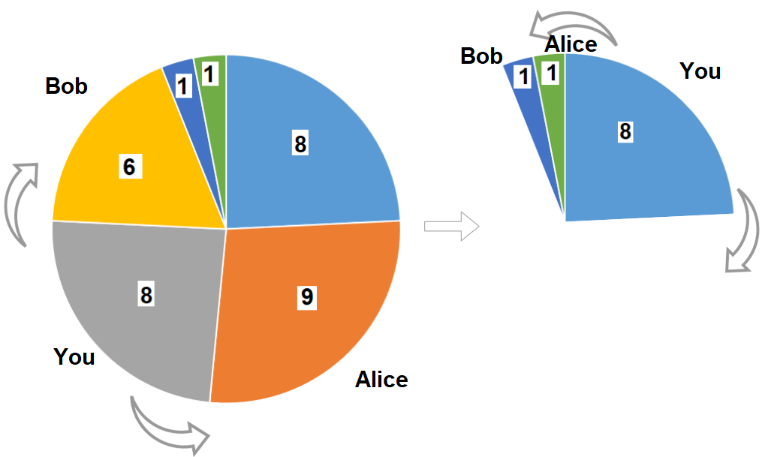


**Input:** slices = [1,2,3,4,5,6]

**Output:** 10

**Explanation:** Pick pizza slice of size 4, Alice and Bob will pick slices with size 3 and 5 respectively. Then Pick slices with size 6, finally Alice and Bob will pick slice of size 2 and 1 respectively. Total = 4 + 6.

**Example 2:**

****

**Input:** slices = [8,9,8,6,1,1]

**Output:** 16

**Output:** Pick pizza slice of size 8 in each turn. If you pick slice with size 9 your partners will pick slices of size 8.

**Example 3:**

**Input:** slices = [4,1,2,5,8,3,1,9,7]

**Output:** 21

**Example 4:**

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

**Output:** 3

**Constraints:**

* 1 <= slices.length <= 500
* slices.length % 3 == 0
* 1 <= slices[i] <= 1000

//解题技巧：环形变线性 掐头去尾

//注意题目每次取都可以取任意一块，即最后所有取的不相邻即可

class Solution {

public:

    int maxSizeSlices(vector<int>& v) {

        int n = v.size() / 3;

        auto l1 = vector<int>(v.begin(), v.end()-1);

        auto l2 = vector<int>(v.begin()+1, v.end());

        return max(linear(l1, n), linear(l2, n));

    }

private:

    int linear(vector<int>& v, int n) {

        vector<vector<int>> dp(v.size()+1, vector<int>(n+1));

        //dp[i][j]到第i个吃了j个的最大值

        int res = 0;

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

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

                if (i == 1) dp[i][j] = v[0];

                else dp[i][j] = max(dp[i-1][j], dp[i-2][j-1] + v[i-1]);

            }

        }

        return dp.back().back();

    }

};

### 1395. Count Number of Teams

Medium

There are n soldiers standing in a line. Each soldier is assigned a **unique** rating value.

You have to form a team of 3 soldiers amongst them under the following rules:

* Choose 3 soldiers with index (i, j, k) with rating (rating[i], rating[j], rating[k]).
* A team is valid if:  (rating[i] < rating[j] < rating[k]) or (rating[i] > rating[j] > rating[k]) where (0 <= i < j < k < n).

Return the number of teams you can form given the conditions. (soldiers can be part of multiple teams).

**Example 1:**

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

**Output:** 3

**Explanation:** We can form three teams given the conditions. (2,3,4), (5,4,1), (5,3,1).

**Example 2:**

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

**Output:** 0

**Explanation:** We can't form any team given the conditions.

**Example 3:**

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

**Output:** 4

**Constraints:**

* n == rating.length
* 1 <= n <= 200
* 1 <= rating[i] <= 10^5

class Solution {

public:

    int numTeams(vector<int>& rating) {

        int res = 0;

        for (auto i = 1; i < rating.size() - 1; ++i) {

            int less[2] = {}, greater[2] = {};

            for (auto j = 0; j < rating.size(); ++j) {

                if (rating[i] < rating[j])

                    ++less[j > i];

                if (rating[i] > rating[j])

                    ++greater[j > i];

            }

            res += less[0] \* greater[1] + greater[0] \* less[1];

        }

        return res;

    }

};

class SolutionOfLeetCode315 {

public:

    using pii = pair<int, int>;

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

        int n = nums.size();

        vector<pii> v(n);

        vector<vector<int>> res(2, vector<int>(n, 0));

        for (int i = 0; i < n; i++)  v[i] = {nums[i], i};

        merge\_sort(v.begin(), v.end(), res, true);

        for (int i = 0; i < n; i++)  v[i] = {nums[i], i};

        merge\_sort(v.begin(), v.end(), res, false);

        return res;

    }

private:

    void merge\_sort(vector<pii>::iterator l, vector<pii>::iterator r,

                    vector<vector<int>> &res, bool IsGreater) {

        if (l+1 < r) {

            auto mid = l + (r-l)/2;

            merge\_sort(l, mid, res, IsGreater);

            merge\_sort(mid, r, res, IsGreater);

            merge(l, r, res, IsGreater);

        }

    }

    void merge(vector<pii>::iterator l, vector<pii>::iterator r,

               vector<vector<int>> &res, bool IsGreater) {

        int cnt = 0;

        auto mid = l + (r-l)/2, p = l, q = mid;

        while (p < mid || q < r) {

            if (p < mid && q < r) {

                if (q->first >= p->first) {

                    if (!IsGreater) res[1][p->second] += r-q;

                    IsGreater ? ++q : ++p;

                }

                else {

                    if (IsGreater) res[0][p->second] += r-q;

                    IsGreater ? ++p : ++q;

                }

            }

            else if (p < mid) p++;

            else q++;

        }

        if (IsGreater) inplace\_merge(l, mid, r, std::greater<>());

        else inplace\_merge(l, mid, r, std::less<>());

    }

};

class Solution {

public:

    int numTeams(vector<int>& rating) {

        //preprocess

        class SolutionOfLeetCode315 A;

        auto temp = A.countSmaller(rating);

        auto &t0  = temp[0], t1 = temp[1];

        int n = rating.size(), res = 0;

        vector<int> sorted\_rating(rating);

        sort(sorted\_rating.begin(), sorted\_rating.end());

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

            int pos = lower\_bound(sorted\_rating.begin(), sorted\_rating.end(), rating[i])

                      - sorted\_rating.begin();

            res += t0[i]\*(n-1-pos-t1[i]) + t1[i]\*(pos-t0[i]);

        }

        return res;

    }

};

### 1397. Find All Good Strings

Hard

Given the strings s1 and s2 of size n, and the string evil. *Return the number of****good****strings*.

A **good** string has size n, it is alphabetically greater than or equal to s1, it is alphabetically smaller than or equal to s2, and it does not contain the string evil as a substring. Since the answer can be a huge number, return this modulo 10^9 + 7.

**Example 1:**

**Input:** n = 2, s1 = "aa", s2 = "da", evil = "b"

**Output:** 51

**Explanation:** There are 25 good strings starting with 'a': "aa","ac","ad",...,"az". Then there are 25 good strings starting with 'c': "ca","cc","cd",...,"cz" and finally there is one good string starting with 'd': "da".

**Example 2:**

**Input:** n = 8, s1 = "leetcode", s2 = "leetgoes", evil = "leet"

**Output:** 0

**Explanation:** All strings greater than or equal to s1 and smaller than or equal to s2 start with the prefix "leet", therefore, there is not any good string.

**Example 3:**

**Input:** n = 2, s1 = "gx", s2 = "gz", evil = "x"

**Output:** 2

**Constraints:**

* s1.length == n
* s2.length == n
* s1 <= s2
* 1 <= n <= 500
* 1 <= evil.length <= 50
* All strings consist of lowercase English letters.

class Solution {

public:

    int dp[501][51][2][2] = {};

    int dfs(const string &s1, const string &s2, const string &evil, int sp, int ep, bool l1, bool l2, const vector<int> &kmp) {

        if (ep == evil.size())

            return 0;

        if (sp == s1.size())

            return 1;

        if (!dp[sp][ep][l1][l2]) {

            for (char ch = (l1 ? s1[sp] : 'a'); ch <= (l2 ? s2[sp] : 'z'); ++ch) {

                auto n\_ep = ep;

                while (n\_ep > 0 && ch != evil[n\_ep])

                    n\_ep = kmp[n\_ep - 1];

                dp[sp][ep][l1][l2] = (dp[sp][ep][l1][l2] +

                    dfs(s1, s2, evil, sp + 1, ch == evil[n\_ep] ? n\_ep + 1 : 0,

                        l1 & (ch == s1[sp]), l2 & (ch == s2[sp]), kmp)) % 1000000007;

            }

        }

        return dp[sp][ep][l1][l2];

    }

    int findGoodStrings(int n, string &s1, string &s2, string &evil) {

        vector<int> kmp(evil.size());

        for (auto j = 0, i = 1; i < kmp.size(); ++i) {

            if (evil[i] == evil[j])

                kmp[i] = ++j;

            else if (j > 0) {

                j = kmp[j - 1];

                --i;

            }

        }

        return dfs(s1, s2, evil, 0, 0, true, true, kmp);

    }

};

### 1402. Reducing Dishes★★

Hard

A chef has collected data on the satisfaction level of his n dishes. Chef can cook any dish in 1 unit of time.

*Like-time coefficient* of a dish is defined as the time taken to cook that dish including previous dishes multiplied by its satisfaction level  i.e.  time[i]\*satisfaction[i]

Return the maximum sum of *Like-time coefficient*that the chef can obtain after dishes preparation.

Dishes can be prepared in **any**order and the chef can discard some dishes to get this maximum value.

**Example 1:**

**Input:** satisfaction = [-1,-8,0,5,-9]

**Output:** 14

**Explanation:** After Removing the second and last dish, the maximum total *Like-time coefficient* will be equal to (-1\*1 + 0\*2 + 5\*3 = 14). Each dish is prepared in one unit of time.

**Example 2:**

**Input:** satisfaction = [4,3,2]

**Output:** 20

**Explanation:** Dishes can be prepared in any order, (2\*1 + 3\*2 + 4\*3 = 20)

**Example 3:**

**Input:** satisfaction = [-1,-4,-5]

**Output:** 0

**Explanation:** People don't like the dishes. No dish is prepared.

* n == satisfaction.length
* 1 <= n <= 500
* -10^3 <= satisfaction[i] <= 10^3

class Solution {

public:

    int maxSatisfaction(vector<int>& A) {

        sort(A.begin(), A.end());

        int res = 0, total = 0, n = A.size();

        for (int i = n - 1; i >= 0 && A[i] > -total; --i) {

            total += A[i];

            res += total;

        }

        return res;

    }

};

### 1405. Longest Happy String

Medium

A string is called *happy* if it does not have any of the strings 'aaa', 'bbb' or 'ccc' as a substring.

Given three integers a, b and c, return **any** string s, which satisfies following conditions:

* s is *happy*and longest possible.
* s contains **at most** a occurrences of the letter 'a', **at most** b occurrences of the letter 'b' and **at most** c occurrences of the letter 'c'.
* s will only contain 'a', 'b' and 'c' letters.

If there is no such string s return the empty string "".

**Example 1:**

**Input:** a = 1, b = 1, c = 7

**Output:** "ccaccbcc"

**Explanation:** "ccbccacc" would also be a correct answer.

**Example 2:**

**Input:** a = 2, b = 2, c = 1

**Output:** "aabbc"

**Example 3:**

**Input:** a = 7, b = 1, c = 0

**Output:** "aabaa"

**Explanation:** It's the only correct answer in this case.

**Constraints:**

* 0 <= a, b, c <= 100
* a + b + c > 0

class Solution {

public:

    string longestDiverseString(int a, int b, int c, char aa = 'a', char bb = 'b', char cc = 'c') {

        if (a < b)

            return longestDiverseString(b, a, c, bb, aa, cc);

        if (b < c)

            return longestDiverseString(a, c, b, aa, cc, bb);

        if (b == 0) return string(min(2, a), aa);

        auto use\_a = min(2, a), use\_b = a - use\_a >= b ? 1 : 0;

        return string(use\_a, aa) +  string(use\_b, bb) +

             longestDiverseString(a-use\_a, b-use\_b, c, aa, bb, cc);

    }

};

### 1406. Stone Game III★

Hard

Alice and Bob continue their games with piles of stones. There are several stones **arranged in a row**, and each stone has an associated value which is an integer given in the array stoneValue.

Alice and Bob take turns, with **Alice** starting first. On each player's turn, that player can take **1, 2 or 3 stones** from the **first** remaining stones in the row.

The score of each player is the sum of values of the stones taken. The score of each player is **0** initially.

The objective of the game is to end with the highest score, and the winner is the player with the highest score and there could be a tie. The game continues until all the stones have been taken.

Assume Alice and Bob **play optimally**.

Return *"Alice"* if Alice will win, *"Bob"* if Bob will win or *"Tie"* if they end the game with the same score.

**Example 1:**

**Input:** values = [1,2,3,7]

**Output:** "Bob"

**Explanation:** Alice will always lose. Her best move will be to take three piles and the score become 6. Now the score of Bob is 7 and Bob wins.

**Example 2:**

**Input:** values = [1,2,3,-9]

**Output:** "Alice"

**Explanation:** Alice must choose all the three piles at the first move to win and leave Bob with negative score.

If Alice chooses one pile her score will be 1 and the next move Bob's score becomes 5. The next move Alice will take the pile with value = -9 and lose.

If Alice chooses two piles her score will be 3 and the next move Bob's score becomes 3. The next move Alice will take the pile with value = -9 and also lose.

Remember that both play optimally so here Alice will choose the scenario that makes her win.

**Example 3:**

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

**Output:** "Tie"

**Explanation:** Alice cannot win this game. She can end the game in a draw if she decided to choose all the first three piles, otherwise she will lose.

**Example 4:**

**Input:** values = [1,2,3,-1,-2,-3,7]

**Output:** "Alice"

**Example 5:**

**Input:** values = [-1,-2,-3]

**Output:** "Tie"

**Constraints:**

* 1 <= values.length <= 50000
* -1000 <= values[i] <= 1000

//dp[i]表示Alice和Bob的差值

class Solution {

public:

    string stoneGameIII(vector<int>& A) {

        int n = A.size();

        vector<int> dp(n+1, -1e9);

        dp[n] = 0;

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

            int S = 0;

            for (int k = 0, j = i; k < 3 && j < n; ++k, ++j) {

                S += A[j];

                dp[i] = max(dp[i], S - dp[j + 1]);

            }

        }

        return dp[0] == 0 ? "Tie" : (dp[0] > 0 ? "Alice" : "Bob");

    }

};

class Solution {

public:

    string stoneGameIII(vector<int>& A) {

        vector<int> dp(4);

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

            dp[i % 4] = -1e9;

            for (int k = 0, take = 0; k < 3 && i + k < A.size(); ++k)

take += A[i + k];

                dp[i % 4] = max(dp[i % 4], take - dp[(i + k + 1) % 4]);

        }

        return dp[0] == 0 ? "Tie" : (dp[0] > 0 ? "Alice" : "Bob");

    }

};

### 1411. Number of Ways to Paint N × 3 Grid

Hard

You have a grid of size n x 3 and you want to paint each cell of the grid with exactly one of the three colours: **Red**, **Yellow** or **Green** while making sure that no two adjacent cells have the same colour (i.e no two cells that share vertical or horizontal sides have the same colour).

You are given n the number of rows of the grid.

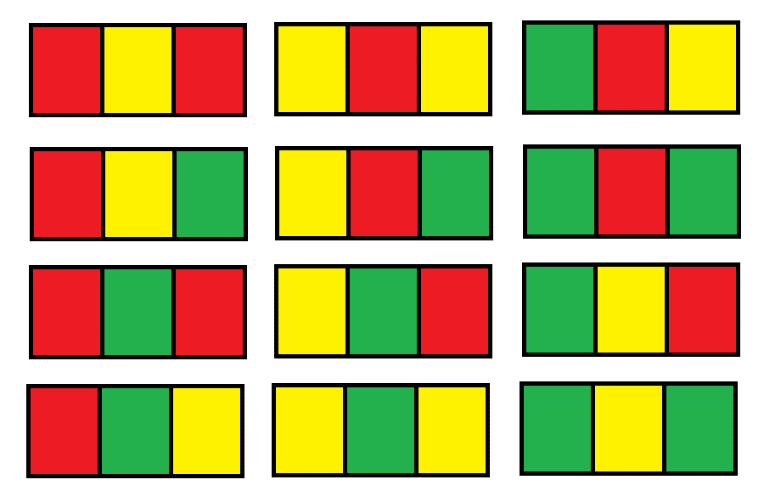
Return *the number of ways* you can paint this grid. As the answer may grow large, the answer **must be** computed modulo 10^9 + 7.

**Example 1:**

**Input:** n = 1

**Output:** 12

**Explanation:** There are 12 possible way to paint the grid as shown:



**Example 2:**

**Input:** n = 2

**Output:** 54

**Example 3:**

**Input:** n = 3

**Output:** 246

**Example 4:**

**Input:** n = 7

**Output:** 106494

**Example 5:**

**Input:** n = 5000

**Output:** 30228214

**Constraints:**

* n == grid.length
* grid[i].length == 3
* 1 <= n <= 5000

class Solution {

public:

    int numOfWays(int n) {

        long a121 = 6, a123 = 6, b121, b123, mod = 1e9 + 7;

        for (int i = 1; i < n; ++i) {

            b121 = a121 \* 3 + a123 \* 2;

            b123 = a121 \* 2 + a123 \* 2;

            a121 = b121 % mod;

            a123 = b123 % mod;

        }

        return (a121 + a123) % mod;

    }

};

### 1414. Find the Minimum Number of Fibonacci Numbers Whose Sum Is K★

Medium

Given the number k, *return the minimum number of Fibonacci numbers whose sum is equal to*k, whether a Fibonacci number could be used multiple times.

The Fibonacci numbers are defined as:

* F1 = 1
* F2 = 1
* Fn = Fn-1 + Fn-2 , for n > 2.

It is guaranteed that for the given constraints we can always find such fibonacci numbers that sum k.

**Example 1:**

**Input:** k = 7

**Output:** 2

**Explanation:** The Fibonacci numbers are: 1, 1, 2, 3, 5, 8, 13, ...

For k = 7 we can use 2 + 5 = 7.

**Example 2:**

**Input:** k = 10

**Output:** 2

**Explanation:** For k = 10 we can use 2 + 8 = 10.

**Example 3:**

**Input:** k = 19

**Output:** 3

**Explanation:** For k = 19 we can use 1 + 5 + 13 = 19.

**Constraints:**

* 1 <= k <= 10^9

class Solution {

public:

    int findMinFibonacciNumbers(int k) {

        if (k < 2) return k;

        int a = 1, b = 1;

        while (b <= k) {

            swap(a, b);

            b += a;

        }

        return 1 + findMinFibonacciNumbers(k - a);

    }

};

### 1415. The k-th Lexicographical String of All Happy Strings of Length n★

Medium

A **happy string** is a string that:

* consists only of letters of the set ['a', 'b', 'c'].
* s[i] != s[i + 1] for all values of i from 1 to s.length - 1 (string is 1-indexed).

For example, strings **"abc", "ac", "b"** and **"abcbabcbcb"** are all happy strings and strings **"aa", "baa"** and **"ababbc"** are not happy strings.

Given two integers n and k, consider a list of all happy strings of length n sorted in lexicographical order.

Return *the kth string* of this list or return an **empty string** if there are less than k happy strings of length n.

**Example 1:**

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

**Output:** "c"

**Explanation:** The list ["a", "b", "c"] contains all happy strings of length 1. The third string is "c".

**Example 2:**

**Input:** n = 1, k = 4

**Output:** ""

**Explanation:** There are only 3 happy strings of length 1.

**Example 3:**

**Input:** n = 3, k = 9

**Output:** "cab"

**Explanation:** There are 12 different happy string of length 3 ["aba", "abc", "aca", "acb", "bab", "bac", "bca", "bcb", "cab", "cac", "cba", "cbc"]. You will find the 9th string = "cab"

**Example 4:**

**Input:** n = 2, k = 7

**Output:** ""

**Example 5:**

**Input:** n = 10, k = 100

**Output:** "abacbabacb"

**Constraints:**

* 1 <= n <= 10
* 1 <= k <= 100

class Solution {

public:

    string getHappyString(int n, int k) {

        auto prem = 1 << (n - 1);

        --k;

        if (k >= 3 \* prem) return "";

        string s = string(1, 'a' + k / prem);

        while (prem > 1) {

            k %= prem;

            prem >>= 1;

            s += k / prem == 0 ? 'a' + (s.back() == 'a')

: 'b' + (s.back() != 'c');

        }

        return s;

    }

};

### 1416. Restore The Array

Hard

A program was supposed to print an array of integers. The program forgot to print whitespaces and the array is printed as a string of digits and all we know is that all integers in the array were in the range [1, k] and there are no leading zeros in the array.

Given the string s and the integer k. There can be multiple ways to restore the array.

Return *the number of possible array* that can be printed as a string s using the mentioned program.

The number of ways could be very large so return it **modulo** 10^9 + 7

**Example 1:**

**Input:** s = "1000", k = 10000

**Output:** 1

**Explanation:** The only possible array is [1000]

**Example 2:**

**Input:** s = "1000", k = 10

**Output:** 0

**Explanation:** There cannot be an array that was printed this way and has all integer >= 1 and <= 10.

**Example 3:**

**Input:** s = "1317", k = 2000

**Output:** 8

**Explanation:** Possible arrays are [1317],[131,7],[13,17],[1,317],[13,1,7],[1,31,7],[1,3,17],[1,3,1,7]

**Example 4:**

**Input:** s = "2020", k = 30

**Output:** 1

**Explanation:** The only possible array is [20,20]. [2020] is invalid because 2020 > 30. [2,020] is ivalid because 020 contains leading zeros.

**Example 5:**

**Input:** s = "1234567890", k = 90

**Output:** 34

**Constraints:**

* 1 <= s.length <= 10^5.
* s consists of only digits and doesn't contain leading zeros.
* 1 <= k <= 10^9.

class Solution {

public:

    int numberOfArrays(string s, int k) {

        vector<int> dp(s.size() + 1);

        dp[s.size()] = 1;

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

            if (s[i] == '0') continue;

            for (long sz = 1, n = 0; i + sz <= s.size(); ++sz) {

                n = n \* 10 + s[i + sz - 1] - '0';

                if (n > k) break;

                dp[i] = (dp[i] + dp[i + sz]) % 1000000007;

            }

        }

        return dp[0];

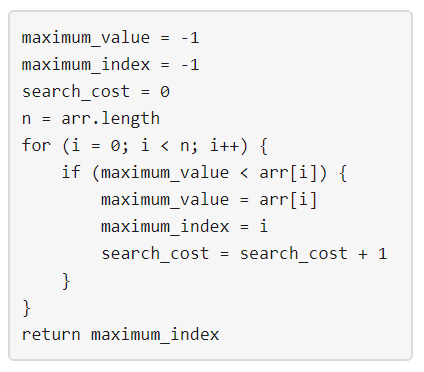
    }

};

### 1420. Build Array Where You Can Find The Maximum Exactly K Comparisons★★

Hard

Given three integers n, m and k. Consider the following algorithm to find the maximum element of an array of positive integers:



You should build the array arr which has the following properties:

* arr has exactly n integers.
* 1 <= arr[i] <= m where (0 <= i < n).
* After applying the mentioned algorithm to arr, the value search\_cost is equal to k.

Return *the number of ways* to build the array arr under the mentioned conditions. As the answer may grow large, the answer **must be** computed modulo 10^9 + 7.

**Example 1:**

**Input:** n = 2, m = 3, k = 1

**Output:** 6

**Explanation:** The possible arrays are [1, 1], [2, 1], [2, 2], [3, 1], [3, 2] [3, 3]

**Example 2:**

**Input:** n = 5, m = 2, k = 3

**Output:** 0

**Explanation:** There are no possible arrays that satisify the mentioned conditions.

**Example 3:**

**Input:** n = 9, m = 1, k = 1

**Output:** 1

**Explanation:** The only possible array is [1, 1, 1, 1, 1, 1, 1, 1, 1]

**Example 4:**

**Input:** n = 50, m = 100, k = 25

**Output:** 34549172

**Explanation:** Don't forget to compute the answer modulo 1000000007

**Example 5:**

**Input:** n = 37, m = 17, k = 7

**Output:** 418930126

**Constraints:**

* 1 <= n <= 50
* 1 <= m <= 100
* 0 <= k <= n

class Solution {

public:

    long long ways[51][101][51];

    const int MOD = 1e9 + 7;

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

        for (int j = 1; j <= m; j++) ways[1][j][1] = 1;

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

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

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

                    long long s = 0;

                    s = (s + b \* ways[a - 1][b][c]) % MOD;

                    for (int x = 1; x < b; x++) {

                        s = (s + ways[a - 1][x][c - 1]) % MOD;

                    }

                    ways[a][b][c] = (ways[a][b][c] + s) % MOD;

                }

            }

        }

        long long ans = 0;

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

            ans = (ans + ways[n][j][k]) % MOD;

        }

        return int(ans);

    }

};

### 1423. Maximum Points You Can Obtain from Cards

Medium

There are several cards **arranged in a row**, and each card has an associated number of points The points are given in the integer array cardPoints.

In one step, you can take one card from the beginning or from the end of the row. You have to take exactly k cards.

Your score is the sum of the points of the cards you have taken.

Given the integer array cardPoints and the integer k, return the *maximum score* you can obtain.

**Example 1:**

**Input:** cardPoints = [1,2,3,4,5,6,1], k = 3

**Output:** 12

**Explanation:** After the first step, your score will always be 1. However, choosing the rightmost card first will maximize your total score. The optimal strategy is to take the three cards on the right, giving a final score of 1 + 6 + 5 = 12.

**Example 2:**

**Input:** cardPoints = [2,2,2], k = 2

**Output:** 4

**Explanation:** Regardless of which two cards you take, your score will always be 4.

**Example 3:**

**Input:** cardPoints = [9,7,7,9,7,7,9], k = 7

**Output:** 55

**Explanation:** You have to take all the cards. Your score is the sum of points of all cards.

**Example 4:**

**Input:** cardPoints = [1,1000,1], k = 1

**Output:** 1

**Explanation:** You cannot take the card in the middle. Your best score is 1.

**Example 5:**

**Input:** cardPoints = [1,79,80,1,1,1,200,1], k = 3

**Output:** 202

**Constraints:**

* 1 <= cardPoints.length <= 10^5
* 1 <= cardPoints[i] <= 10^4
* 1 <= k <= cardPoints.length

class Solution {

public:

    int maxScore(vector<int>& cp, int k) {

        // Time Complexity: O(k)

        // Space Complexity: O(1)

        int n(cp.size()), left(0), right(0);

        for (int i = 0; i < k; ++i) left += cp[i];

        int ans(left);

        for (int i = 0; i < k; ++i) {

            left -= cp[k - i - 1];

            right += cp[n - i - 1];

            ans = max(ans, left + right);

        }

        return ans;

    }

};

### 1425. Constrained Subsequence Sum★★

Hard

Given an integer array nums and an integer k, return the maximum sum of a **non-empty** subsequence of that array such that for every two **consecutive** integers in the subsequence, nums[i] and nums[j], where i < j, the condition j - i <= k is satisfied.

A *subsequence* of an array is obtained by deleting some number of elements (can be zero) from the array, leaving the remaining elements in their original order.

**Example 1:**

**Input:** nums = [10,2,-10,5,20], k = 2

**Output:** 37

**Explanation:** The subsequence is [10, 2, 5, 20].

**Example 2:**

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

**Output:** -1

**Explanation:** The subsequence must be non-empty, so we choose the largest number.

**Example 3:**

**Input:** nums = [10,-2,-10,-5,20], k = 2

**Output:** 23

**Explanation:** The subsequence is [10, -2, -5, 20].

**Constraints:**

* 1 <= k <= nums.length <= 10^5
* -10^4 <= nums[i] <= 10^4

class Solution {

public:

    int constrainedSubsetSum(vector<int>& A, int k) {

        vector<int> dp(A);

        int res = A[0];

        deque<int> deq;

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

            if (!deq.empty()) dp[i] = max(dp[i], dp[deq.front()]+A[i]);

            res = max(res, dp[i]);

            while (!deq.empty() && dp[deq.back()] < dp[i])

                deq.pop\_back();

            deq.push\_back(i);

            if ((i+1) - deq.front() > k) deq.pop\_front();

        }

        return res;

    }

};

class Solution {

public:

    int constrainedSubsetSum(vector<int>& A, int k) {

        deque<int> deq;

        int res = A[0];

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

            A[i] += deq.size() ? deq.front() : 0;

            res = max(res, A[i]);

            while (!deq.empty() && A[i] > deq.back())

                deq.pop\_back();

            if (A[i] > 0) deq.push\_back(A[i]);

            if (i >= k && !deq.empty() && deq.front() == A[i - k])

                deq.pop\_front();

        }

        return res;

    }

};

### 1434. Number of Ways to Wear Different Hats to Each Other★★

Hard

There are n people and 40 types of hats labeled from 1 to 40.

Given a list of list of integers hats, where hats[i] is a list of all hats preferred by the i-th person.

Return the number of ways that the n people wear different hats to each other.

Since the answer may be too large, return it modulo 10^9 + 7.

**Example 1:**

**Input:** hats = [[3,4],[4,5],[5]]

**Output:** 1

**Explanation:** There is only one way to choose hats given the conditions.

First person choose hat 3, Second person choose hat 4 and last one hat 5.

**Example 2:**

**Input:** hats = [[3,5,1],[3,5]]

**Output:** 4

**Explanation:** There are 4 ways to choose hats

(3,5), (5,3), (1,3) and (1,5)

**Example 3:**

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

**Output:** 24

**Explanation:** Each person can choose hats labeled from 1 to 4.

Number of Permutations of (1,2,3,4) = 24.

**Example 4:**

**Input:** hats = [[1,2,3],[2,3,5,6],[1,3,7,9],[1,8,9],[2,5,7]]

**Output:** 111

**Constraints:**

* n == hats.length
* 1 <= n <= 10
* 1 <= hats[i].length <= 40
* 1 <= hats[i][j] <= 40
* hats[i] contains a list of **unique** integers.

class Solution {

public:

    int numberWays(vector<vector<int>>& hats) {

        // Time Complexity: O(2^n \* h \* n)

        // Space Complexity: O(2^n)

        vector<vector<int>> persons(40);

        const int n(hats.size()), mod(1e9 + 7);

        vector<int> masks(1 << n);

        masks[0] = 1;

        for (int i = 0; i < n; ++i)

            for (const int& h: hats[i])

                persons[h - 1].emplace\_back(i);

        for (int i = 0; i < 40; ++i) {

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

                for (const int& p: persons[i]) {

                    if ((j & (1 << p)) == 0) {

                        masks[j | (1 << p)] += masks[j];

                        masks[j | (1 << p)] %= mod;

                    }

                }

            }

        }

        return masks[(1 << n) - 1];

    }

};

### 1438. Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit★★

Medium

Given an array of integers nums and an integer limit, return the size of the longest **non-empty** subarray such that the absolute difference between any two elements of this subarray is less than or equal to limit*.*

**Example 1:**

**Input:** nums = [8,2,4,7], limit = 4

**Output:** 2

**Explanation:** All subarrays are:

[8] with maximum absolute diff |8-8| = 0 <= 4.

[8,2] with maximum absolute diff |8-2| = 6 > 4.

[8,2,4] with maximum absolute diff |8-2| = 6 > 4.

[8,2,4,7] with maximum absolute diff |8-2| = 6 > 4.

[2] with maximum absolute diff |2-2| = 0 <= 4.

[2,4] with maximum absolute diff |2-4| = 2 <= 4.

[2,4,7] with maximum absolute diff |2-7| = 5 > 4.

[4] with maximum absolute diff |4-4| = 0 <= 4.

[4,7] with maximum absolute diff |4-7| = 3 <= 4.

[7] with maximum absolute diff |7-7| = 0 <= 4.

Therefore, the size of the longest subarray is 2.

**Example 2:**

**Input:** nums = [10,1,2,4,7,2], limit = 5

**Output:** 4

**Explanation:** The subarray [2,4,7,2] is the longest since the maximum absolute diff is |2-7| = 5 <= 5.

**Example 3:**

**Input:** nums = [4,2,2,2,4,4,2,2], limit = 0

**Output:** 3

**Constraints:**

* 1 <= nums.length <= 10^5
* 1 <= nums[i] <= 10^9
* 0 <= limit <= 10^9

//nlogn

class Solution {

public:

    int longestSubarray(vector<int>& A, int limit) {

        int i = 0, res = 0;

        multiset<int> m;

        for (int j = 0; j < A.size(); ++j) {

            m.insert(A[j]);

            while (\*m.rbegin() - \*m.begin() > limit)

                m.erase(m.find(A[i++]));

            res = max(res, j-i+1);

        }

        return res;

    }

};

//n

class Solution {

public:

    int longestSubarray(vector<int>& A, int limit) {

        deque<int> maxd, mind;

        int i = 0, res = 0;

        for (int j = 0; j < A.size(); ++j) {

            while (!maxd.empty() && A[j] > maxd.back())maxd.pop\_back();

            while (!mind.empty() && A[j] < mind.back())mind.pop\_back();

            maxd.push\_back(A[j]);

            mind.push\_back(A[j]);

            while (maxd.front() - mind.front() > limit) {

                if (maxd.front() == A[i]) maxd.pop\_front();

                if (mind.front() == A[i]) mind.pop\_front();

                ++i;

            }

            res = max(res, j-i+1);

        }

        return res;

    }

};

### 1439. Find the Kth Smallest Sum of a Matrix With Sorted Rows★

Hard

You are given an m \* n matrix, mat, and an integer k, which has its rows sorted in non-decreasing order.

You are allowed to choose exactly 1 element from each row to form an array. Return the Kth **smallest** array sum among all possible arrays.

**Example 1:**

**Input:** mat = [[1,3,11],[2,4,6]], k = 5

**Output:** 7

**Explanation:** Choosing one element from each row, the first k smallest sum are:

[1,2], [1,4], [3,2], [3,4], [1,6]. Where the 5th sum is 7.

**Example 2:**

**Input:** mat = [[1,3,11],[2,4,6]], k = 9

**Output:** 17

**Example 3:**

**Input:** mat = [[1,10,10],[1,4,5],[2,3,6]], k = 7

**Output:** 9

**Explanation:** Choosing one element from each row, the first k smallest sum are:

[1,1,2], [1,1,3], [1,4,2], [1,4,3], [1,1,6], [1,5,2], [1,5,3]. Where the 7th sum is 9.

**Example 4:**

**Input:** mat = [[1,1,10],[2,2,9]], k = 7

**Output:** 12

**Constraints:**

* m == mat.length
* n == mat.length[i]
* 1 <= m, n <= 40
* 1 <= k <= min(200, n ^ m)
* 1 <= mat[i][j] <= 5000
* mat[i] is a non decreasing array.

class Solution {

public:

    int kthSmallest(vector<vector<int>>& mat, size\_t k) {

        vector<int> row{0};

        for(const auto &t : mat) {

            vector<int> v;

            for(const auto &i : t) {

                for (const auto &pathSum : row) {

                    v.emplace\_back(i + pathSum);

                }

            }

            partial\_sort(v.begin(),v.begin()+min(k,v.size()),v.end());

            v.resize(min(k, v.size()));

            row = move(v);

        }

        return row[k-1];

    }

};

### 1442. Count Triplets That Can Form Two Arrays of Equal XOR★★

Medium

Given an array of integers arr.

We want to select three indices i, j and k where (0 <= i < j <= k < arr.length).

Let's define a and b as follows:

* a = arr[i] ^ arr[i + 1] ^ ... ^ arr[j - 1]
* b = arr[j] ^ arr[j + 1] ^ ... ^ arr[k]

Note that **^** denotes the **bitwise-xor** operation.

Return *the number of triplets* (i, j and k) Where a == b.

**Example 1:**

**Input:** arr = [2,3,1,6,7]

**Output:** 4

**Explanation:** The triplets are (0,1,2), (0,2,2), (2,3,4) and (2,4,4)

**Example 2:**

**Input:** arr = [1,1,1,1,1]

**Output:** 10

**Example 3:**

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

**Output:** 0

**Example 4:**

**Input:** arr = [1,3,5,7,9]

**Output:** 3

**Example 5:**

**Input:** arr = [7,11,12,9,5,2,7,17,22]

**Output:** 8

**Constraints:**

* 1 <= arr.length <= 300
* 1 <= arr[i] <= 10^8

class Solution {

public:

    int countTriplets(vector<int>& A) {

        int n = A.size(), res = 0, prefix = 0;

        unordered\_map<int, int> count = {{0, 1}}, total;

        for (int i = 0; i < n; ++i) {

            prefix ^= A[i];

            res += count[prefix]++ \* i - total[prefix];

            total[prefix] += i + 1;

        }

        return res;

    }

};

### 1444. Number of Ways of Cutting a Pizza★★

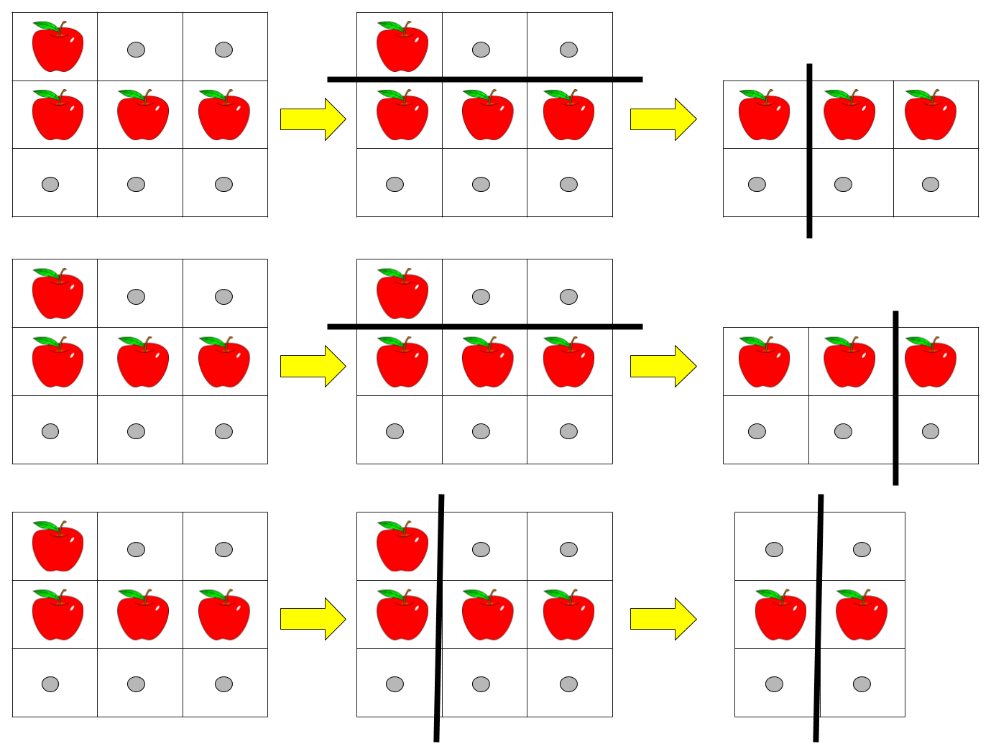
Hard

Given a rectangular pizza represented as a rows x cols matrix containing the following characters: 'A' (an apple) and '.' (empty cell) and given the integer k. You have to cut the pizza into k pieces using k-1 cuts.

For each cut you choose the direction: vertical or horizontal, then you choose a cut position at the cell boundary and cut the pizza into two pieces. If you cut the pizza vertically, give the left part of the pizza to a person. If you cut the pizza horizontally, give the upper part of the pizza to a person. Give the last piece of pizza to the last person.

*Return the number of ways of cutting the pizza such that each piece contains****at least****one apple.*Since the answer can be a huge number, return this modulo 10^9 + 7.

**Example 1:**

****

**Input:** pizza = ["A..","AAA","..."], k = 3

**Output:** 3

**Explanation:** The figure above shows the three ways to cut the pizza. Note that pieces must contain at least one apple.

**Example 2:**

**Input:** pizza = ["A..","AA.","..."], k = 3

**Output:** 1

**Example 3:**

**Input:** pizza = ["A..","A..","..."], k = 1

**Output:** 1

**Constraints:**

* 1 <= rows, cols <= 50
* rows == pizza.length
* cols == pizza[i].length
* 1 <= k <= 10
* pizza consists of characters 'A' and '.' only.

class Solution {

    void add(long &a, long &b) { a = (a + b) % ((int) 1e9+7); }

public:

    int ways(vector<string>& A, int K) {

        int M = A.size(), N = A[0].size();

        vector<vector<int>> cnt(M + 1, vector<int>(N + 1));

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

            int s = 0;

            for (int j = N - 1; j >= 0; --j) {

                s += A[i][j] == 'A';

                cnt[i][j] = cnt[i + 1][j] + s;

            }

        }

        vector<vector<vector<long>>> dp(M + 1, vector<vector<long>>(N + 1, vector<long>(K + 1)));

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

            for (int j = N - 1; j >= 0; --j) {

                dp[i][j][1] = cnt[i][j] > 0;

                for (int k = 2; k <= K; ++k) {

                    for (int t = i + 1; t < M; ++t) {

                        if (cnt[i][j] == cnt[t][j]) continue;

                        if (cnt[t][j] == 0) break;

                        add(dp[i][j][k], dp[t][j][k - 1]);

                    }

                    for (int t = j + 1; t < N; ++t) {

                        if (cnt[i][j] == cnt[i][t]) continue;

                        if (cnt[i][t] == 0) break;

                        add(dp[i][j][k], dp[i][t][k - 1]);

                    }

                }

            }

        }

        return dp[0][0][K];

    }

};

### 1449. Form Largest Integer With Digits That Add up to Target★★

Hard

Given an array of integers cost and an integer target. Return the **maximum** integer you can paint under the following rules:

* The cost of painting a digit (i+1) is given by cost[i] (0 indexed).
* The total cost used must be equal to target.
* Integer does not have digits 0.

Since the answer may be too large, return it as string.

If there is no way to paint any integer given the condition, return "0".

**Example 1:**

**Input:** cost = [4,3,2,5,6,7,2,5,5], target = 9

**Output:** "7772"

**Explanation:**  The cost to paint the digit '7' is 2, and the digit '2' is 3. Then cost("7772") = 2\*3+ 3\*1 = 9. You could also paint "977", but "7772" is the largest number.

**Digit cost**

1 -> 4

2 -> 3

3 -> 2

4 -> 5

5 -> 6

6 -> 7

7 -> 2

8 -> 5

9 -> 5

**Example 2:**

**Input:** cost = [7,6,5,5,5,6,8,7,8], target = 12

**Output:** "85"

**Explanation:** The cost to paint the digit '8' is 7, and the digit '5' is 5. Then cost("85") = 7 + 5 = 12.

**Example 3:**

**Input:** cost = [2,4,6,2,4,6,4,4,4], target = 5

**Output:** "0"

**Explanation:** It's not possible to paint any integer with total cost equal to target.

**Example 4:**

**Input:** cost = [6,10,15,40,40,40,40,40,40], target = 47

**Output:** "32211"

**Constraints:**

* cost.length == 9
* 1 <= cost[i] <= 5000
* 1 <= target <= 5000

class Solution {

public:

    string largestNumber(vector<int>& cost, int target) {

        vector<string> dp(target + 1, "NAN");

        dp[0] = "";

        auto largeThan = [](string& s1, string& s2) {

            if (s1.size() > s2.size()) return true;

            else if (s1.size() < s2.size()) return false;

            else return s1 > s2;

        };

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

            for (int j = cost[i - 1]; j <= target; ++j) {

                if (dp[j-cost[i-1]] != "NAN") {

                    string newVal = to\_string(i) + dp[j-cost[i-1]];

                    if (dp[j] == "NAN" || largeThan(newVal, dp[j])) {

                        dp[j] = newVal;

                    }

                }

            }

        }

        return dp[target] == "NAN" ? "0" : dp[target];

    }

};

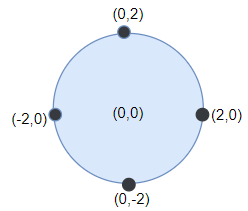
### 1453. Maximum Number of Darts Inside of a Circular Dartboard★★

Hard

You have a very large square wall and a circular dartboard placed on the wall. You have been challenged to throw darts into the board blindfolded. Darts thrown at the wall are represented as an array of points on a 2D plane.

Return the maximum number of points that are within or lie on **any** circular dartboard of radius r.

**Example 1:**

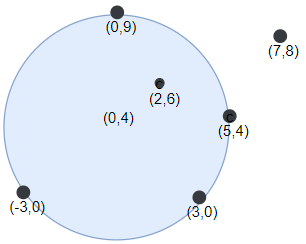


**Input:** points = [[-2,0],[2,0],[0,2],[0,-2]], r = 2

**Output:** 4

**Explanation:** Circle dartboard with center in (0,0) and radius = 2 contain all points.

**Example 2:**

****

**Input:** points = [[-3,0],[3,0],[2,6],[5,4],[0,9],[7,8]], r = 5

**Output:** 5

**Explanation:** Circle dartboard with center in (0,4) and radius = 5 contain all points except the point (7,8).

**Example 3:**

**Input:** points = [[-2,0],[2,0],[0,2],[0,-2]], r = 1

**Output:** 1

**Example 4:**

**Input:** points = [[1,2],[3,5],[1,-1],[2,3],[4,1],[1,3]], r = 2

**Output:** 4

**Constraints:**

* 1 <= points.length <= 100
* points[i].length == 2
* -10^4 <= points[i][0], points[i][1] <= 10^4
* 1 <= r <= 5000

//O（N^3）

class Solution {

public:

const double tol = 1e-6;

vector<Point> pnt;

    struct Point {

        Point() {}

        Point(double a, double b) : x(a), y(b) {}

        double x, y;

    };

    double dis(const Point& a, const Point& b) {

        return sqrt((a.x - b.x)\*(a.x - b.x) + (a.y - b.y)\*(a.y - b.y));

    }

    pair<Point, Point> getCenter(const Point& a, const Point& b, int R){

        Point mid((a.x + b.x) / 2, (a.y + b.y) / 2);

        double theta = atan2(a.y - b.y, b.x - a.x);

        double tmp = dis(a, b) / 2;

        double d = sqrt(R \* R - tmp\*tmp);

        Point t0 = {mid.x - d\*sin(theta), mid.y - d\*cos(theta)};

        Point t1 = {mid.x + d\*sin(theta), mid.y + d\*cos(theta)};

        return {t0, t1};

    }

    int numPoints(vector<vector<int>>& points, int R) {

        int n = points.size(), ans = 1;

        for (const auto &p : points) pnt.emplace\_back(p[0], p[1]);

        for (int i = 0; i < n; i++) {

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

                if (i == j || dis(pnt[i], pnt[j]) - 2\*R > tol)continue;

                int c0 = 0, c1 = 0;

                auto p = getCenter(pnt[i], pnt[j], R);

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

                    if (dis(pnt[k], p.first) - R <= tol) ++c0;

                    if (dis(pnt[k], p.second) - R <= tol) ++c1;

                }

                ans = max({ans, c0, c1});

            }

        }

        return ans;

    }

};

//O(N^2logN)

class Solution {

public:

    using pdb = pair<double, bool>;

    double getDist(const vector<int> &a, const vector<int> &b) {

        return sqrt((a[0]-b[0])\*(a[0]-b[0]) + (a[1]-b[1])\*(a[1]-b[1]));

    }

    int numPoints(vector<vector<int>>& p, int r) {

        int n = p.size();

        vector<vector<double>> dist(n, vector<double>(n));

        for (int i = 0; i < n; ++i) {

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

                dist[i][j] = dist[j][i] = getDist(p[i], p[j]);

            }

        }

        for (int i = 0; i < n; ++i) {

            vector<pdb> angles;

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

                if (j != i && dist[i][j] <= 2 \* r) {

                    double A = atan2(p[j][1]-p[i][1], p[j][0]-p[i][0]);

                    double B = acos(dist[i][j] / (2.0 \* r));

                    double alpha = A-B, beta = A+B;

                    angles.emplace\_back(alpha, false);

                    // "false" means this point is entering

                    angles.emplace\_back(beta, true);

                    // "true" means this point is exiting

                }

            }

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

            int res = 1, cnt = 1;

            for (const auto &[a, b] : angles) {

                res = max(res, b ? --cnt : ++cnt);

            }

        }

        return res;

    }

};

### 1458. Max Dot Product of Two Subsequences

Hard

Given two arrays nums1 and nums2.

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

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

**Example 1:**

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

**Output:** 18

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

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

**Example 2:**

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

**Output:** 21

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

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

**Example 3:**

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

**Output:** -1

**Constraints:**

* 1 <= nums1.length, nums2.length <= 500
* -1000 <= nums1[i], nums2[i] <= 1000

class Solution {

public:

    int maxDotProduct(vector<int>& A, vector<int>& B) {

        int n = A.size(), m = B.size();

        vector<vector<int>> dp(2, vector<int>(m));

        int k = 0;

        for (int i = 0; i < n; ++i) {

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

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

                cur[j] = A[i] \* B[j];

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

                if (i) cur[j] = max(cur[j], pre[j]);

                if (j) cur[j] = max(cur[j], cur[j - 1]);

            }

            k ^= 1;

        }

        return dp[k][m - 1];

    }

};

### 1462. Course Schedule IV

Medium

There are a total of n courses you have to take, labeled from 0 to n-1.

Some courses may have direct prerequisites, for example, to take course 0 you have first to take course 1, which is expressed as a pair: [1,0]

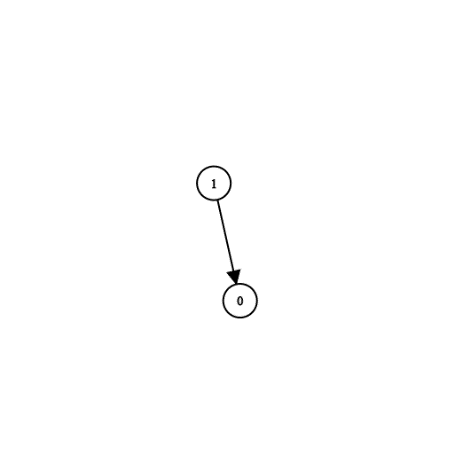
Given the total number of courses n, a list of direct prerequisite **pairs** and a list of queries **pairs**.

You should answer for each queries[i] whether the course queries[i][0] is a prerequisite of the course queries[i][1] or not.

Return *a list of boolean*, the answers to the given queries.

Please note that if course **a** is a prerequisite of course **b** and course **b** is a prerequisite of course **c**, then, course **a** is a prerequisite of course **c**.

**Example 1:**



**Input:** n = 2, prerequisites = [[1,0]], queries = [[0,1],[1,0]]

**Output:** [false,true]

**Explanation:** course 0 is not a prerequisite of course 1 but the opposite is true.

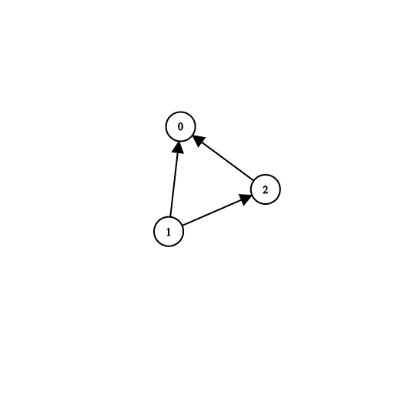
**Example 2:**

**Input:** n = 2, prerequisites = [], queries = [[1,0],[0,1]]

**Output:** [false,false]

**Explanation:** There are no prerequisites and each course is independent.

**Example 3:**



**Input:** n = 3, prerequisites = [[1,2],[1,0],[2,0]], queries = [[1,0],[1,2]]

**Output:** [true,true]

**Example 4:**

**Input:** n = 3, prerequisites = [[1,0],[2,0]], queries = [[0,1],[2,0]]

**Output:** [false,true]

**Example 5:**

**Input:** n = 5, prerequisites = [[0,1],[1,2],[2,3],[3,4]], queries = [[0,4],[4,0],[1,3],[3,0]]

**Output:** [true,false,true,false]

**Constraints:**

* 2 <= n <= 100
* 0 <= prerequisite.length <= (n \* (n - 1) / 2)
* 0 <= prerequisite[i][0], prerequisite[i][1] < n
* prerequisite[i][0] != prerequisite[i][1]
* The prerequisites graph has no cycles.
* The prerequisites graph has no repeated edges.
* 1 <= queries.length <= 10^4
* queries[i][0] != queries[i][1]

class Solution {

public:

    using bits = bitset<100>;

    vector<bool> checkIfPrerequisite(int n, vector<vector<int>>& prerequisites, vector<vector<int>>& queries) {

        vector<vector<int>> g(n);

        for(auto v: prerequisites) {

            g[v[0]].emplace\_back(v[1]);

        }

        vector<bits> cache(n, 0);

        function<bits(int i)> dfs = [&](int i) {

            if (cache[i][i]) return cache[i];

            cache[i][i] = true;

            for (int v: g[i])  cache[i] |= dfs(v);

            return cache[i];

        };

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

        vector<bool> res;

        for(auto &q: queries)  res.emplace\_back(cache[q[0]][q[1]]);

        return res;

    }

};

### 1463. Cherry Pickup II

Hard

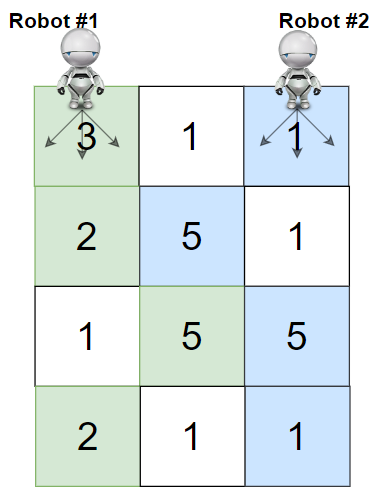
Given a rows x cols matrix grid representing a field of cherries. Each cell in grid represents the number of cherries that you can collect.

You have two robots that can collect cherries for you, Robot #1 is located at the top-left corner (0,0) , and Robot #2 is located at the top-right corner (0, cols-1) of the grid.

Return the maximum number of cherries collection using both robots  by following the rules below:

* From a cell (i,j), robots can move to cell (i+1, j-1) , (i+1, j) or (i+1, j+1).
* When any robot is passing through a cell, It picks it up all cherries, and the cell becomes an empty cell (0).
* When both robots stay on the same cell, only one of them takes the cherries.
* Both robots cannot move outside of the grid at any moment.
* Both robots should reach the bottom row in the grid.

**Example 1:**

****

**Input:** grid = [[3,1,1],[2,5,1],[1,5,5],[2,1,1]]

**Output:** 24

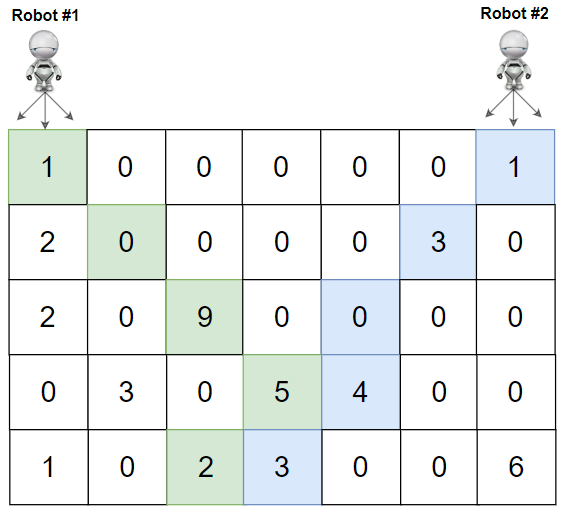
**Explanation:** Path of robot #1 and #2 are described in color green and blue respectively.

Cherries taken by Robot #1, (3 + 2 + 5 + 2) = 12.

Cherries taken by Robot #2, (1 + 5 + 5 + 1) = 12.

Total of cherries: 12 + 12 = 24.

**Example 2:**

****

**Input:** grid = [[1,0,0,0,0,0,1],[2,0,0,0,0,3,0],[2,0,9,0,0,0,0],[0,3,0,5,4,0,0],[1,0,2,3,0,0,6]]

**Output:** 28

**Explanation:** Path of robot #1 and #2 are described in color green and blue respectively.

Cherries taken by Robot #1, (1 + 9 + 5 + 2) = 17.

Cherries taken by Robot #2, (1 + 3 + 4 + 3) = 11.

Total of cherries: 17 + 11 = 28.

**Example 3:**

**Input:** grid = [[1,0,0,3],[0,0,0,3],[0,0,3,3],[9,0,3,3]]

**Output:** 22

**Example 4:**

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

**Output:** 4

**Constraints:**

* rows == grid.length
* cols == grid[i].length
* 2 <= rows, cols <= 70
* 0 <= grid[i][j] <= 100

class Solution {

public:

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

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

        int f[2][n][n];

        int res = 0, p = 0;

        memset(f, -1, sizeof(f));

        f[0][0][n - 1] = v[0][0] + v[0][n-1];

        for (int i = 1; i < m; i++) {

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

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

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

                    int value = (j == k) ? v[i][j] : v[i][j] + v[i][k];

                    for (int col1 = j-1; col1 <= j+1; col1++) {

                        for (int col2 = k-1; col2 <= k+1; col2++) {

                            if (col1>=0 && col1<n && col2>=0

&& col2<n && pre[col1][col2]>=0){

                                cur[j][k] = max(cur[j][k],

pre[col1][col2] + value);

                                if (i == m-1) res=max(res, cur[j][k]);

                            }

                        }

}

                }

            }

            p ^= 1;

        }

        return res;

    }

};

### 1467. Probability of a Two Boxes Having The Same Number of Distinct Balls

Hard

Given 2n balls of k distinct colors. You will be given an integer array balls of size k where balls[i] is the number of balls of color i.

All the balls will be **shuffled uniformly at random**, then we will distribute the first n balls to the first box and the remaining n balls to the other box (Please read the explanation of the second example carefully).

Please note that the two boxes are considered different. For example, if we have two balls of colors a and b, and two boxes [] and (), then the distribution [a] (b) is considered different than the distribution [b] (a) (Please read the explanation of the first example carefully).

We want to *calculate the probability* that the two boxes have the same number of distinct balls.

**Example 1:**

**Input:** balls = [1,1]

**Output:** 1.00000

**Explanation:** Only 2 ways to divide the balls equally:

- A ball of color 1 to box 1 and a ball of color 2 to box 2

- A ball of color 2 to box 1 and a ball of color 1 to box 2

In both ways, the number of distinct colors in each box is equal. The probability is 2/2 = 1

**Example 2:**

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

**Output:** 0.66667

**Explanation:** We have the set of balls [1, 1, 2, 3]

This set of balls will be shuffled randomly and we may have one of the 12 distinct shuffles with equale probability (i.e. 1/12):

[1,1 / 2,3], [1,1 / 3,2], [1,2 / 1,3], [1,2 / 3,1], [1,3 / 1,2], [1,3 / 2,1], [2,1 / 1,3], [2,1 / 3,1], [2,3 / 1,1], [3,1 / 1,2], [3,1 / 2,1], [3,2 / 1,1]

After that we add the first two balls to the first box and the second two balls to the second box.

We can see that 8 of these 12 possible random distributions have the same number of distinct colors of balls in each box.

Probability is 8/12 = 0.66667

**Example 3:**

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

**Output:** 0.60000

**Explanation:** The set of balls is [1, 2, 2, 3, 4, 4]. It is hard to display all the 180 possible random shuffles of this set but it is easy to check that 108 of them will have the same number of distinct colors in each box.

Probability = 108 / 180 = 0.6

**Example 4:**

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

**Output:** 0.30000

**Explanation:** The set of balls is [1, 1, 1, 2, 2, 3]. It is hard to display all the 60 possible random shuffles of this set but it is easy to check that 18 of them will have the same number of distinct colors in each box.

Probability = 18 / 60 = 0.3

**Example 5:**

**Input:** balls = [6,6,6,6,6,6]

**Output:** 0.90327

**Constraints:**

* 1 <= balls.length <= 8
* 1 <= balls[i] <= 6
* sum(balls) is even.
* Answers within 10^-5 of the actual value will be accepted as correct.

class Solution {

public:

    double getProbability(vector<int>& balls) {

        int N = accumulate(balls.begin(), balls.end(), 0), n = N / 2;

        return dfs(0, 0, 0, 0, 0, balls, n);

    }

    double dp[8][50][50][8][8];

    double dfs(int i, int l, int r, int cl, int cr, vector<int> &balls, int n) {

        if (i == balls.size()) {

            return (l == r && cl == cr) ? 1.0 : 0;

        }

        if (dp[i][l][r][cl][cr] != 0) return dp[i][l][r][cl][cr];

        double ret = 0;

        double a = C(2\*n - l - r, balls[i]);

        for (int k = 0; k <= balls[i]; ++k) {

            int new\_l = l+k, new\_r = r+balls[i]-k;

            if (new\_l > n) break;

            if (new\_r > n) continue;

            int new\_cl = cl + (k == 0 ? 0 : 1);

            int new\_cr = cr + (k == balls[i] ? 0 : 1);

            double b = C(n - l, k);

            double c = C(n - r, balls[i]-k);

            ret += dfs(i+1, new\_l, new\_r, new\_cl, new\_cr, balls, n)

/ a \* b \* c;

        }

        return dp[i][l][r][cl][cr] = ret;

    }

    double C(int n, int i) {

        if (i > n) return 0;

        if (i == 0) return 1;

        return C(n-1, i-1) + C(n-1, i);

    }

};

### 1477. Find Two Non-overlapping Sub-arrays Each With Target Sum★★

Medium

Given an array of integers arr and an integer target.

You have to find **two non-overlapping sub-arrays** of arr each with sum equal target. There can be multiple answers so you have to find an answer where the sum of the lengths of the two sub-arrays is **minimum**.

Return *the minimum sum of the lengths* of the two required sub-arrays, or return ***-1*** if you cannot find such two sub-arrays.

**Example 1:**

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

**Output:** 2

**Explanation:** Only two sub-arrays have sum = 3 ([3] and [3]). The sum of their lengths is 2.

**Example 2:**

**Input:** arr = [7,3,4,7], target = 7

**Output:** 2

**Explanation:** Although we have three non-overlapping sub-arrays of sum = 7 ([7], [3,4] and [7]), but we will choose the first and third sub-arrays as the sum of their lengths is 2.

**Example 3:**

**Input:** arr = [4,3,2,6,2,3,4], target = 6

**Output:** -1

**Explanation:** We have only one sub-array of sum = 6.

**Example 4:**

**Input:** arr = [5,5,4,4,5], target = 3

**Output:** -1

**Explanation:** We cannot find a sub-array of sum = 3.

**Example 5:**

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

**Output:** 3

**Explanation:** Note that sub-arrays [1,2] and [2,1] cannot be an answer because they overlap.

**Constraints:**

* 1 <= arr.length <= 10^5
* 1 <= arr[i] <= 1000
* 1 <= target <= 10^8

class Solution {

public:

    unordered\_map<int,int> mp{{0, 0}};

    int minSumOfLengths(vector<int>& arr, int tar) {

        int n = arr.size(), cursum = 0;

        int k = 2;

        vector<vector<int>> dp(n+1, vector<int> (k+1, 1e9));

        for (int i = 0; i < dp.size(); i++) dp[i][0] = 0;

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

            int d = -1;

            cursum += arr[i-1];

            mp[cursum] = i;

            if (mp.count(cursum - tar)) d = mp[cursum-tar];

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

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

                if (d != -1) {

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

                }

            }

        }

        if (dp[n][k] >= 1e9) return -1;

        return dp[n][k];

    }

};

### 1478. Allocate Mailboxes★★

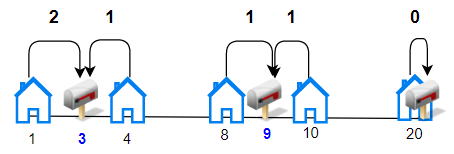
Hard

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

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

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

**Example 1:**



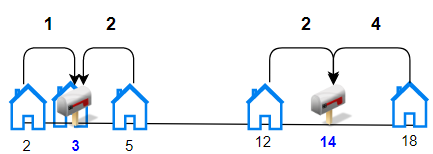
**Input:** houses = [1,4,8,10,20], k = 3

**Output:** 5

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

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

**Example 2:**

****

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

**Output:** 9

**Explanation:** Allocate mailboxes in position 3 and 14.

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

**Example 3:**

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

**Output:** 8

**Example 4:**

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

**Output:** 0

**Constraints:**

* n == houses.length
* 1 <= n <= 100
* 1 <= houses[i] <= 10^4
* 1 <= k <= n
* Array houses contain unique integers.

class Solution {

public:

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

        sort(A.begin(), A.end());

        int n = A.size();

        vector<int> B(n + 1, 0), dp(n, 1e6);

        for (int i = 0; i < n; ++i)

            B[i + 1] = B[i] + A[i];

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

            for (int j = n - 1; j > k - 2; --j) {

                for (int i = k - 2; i < j; ++i) {

                    int m1 =  (i + j + 1) / 2, m2 = (i + j + 2) / 2;

                    int last = (B[j + 1]-B[m2])-(B[m1 + 1]-B[i + 1]);

                    dp[j] = min(dp[j], (i >= 0 ? dp[i] : 0) + last);

                }

            }

        }

        return dp[n - 1];

    }

};

### 1482. Minimum Number of Days to Make m Bouquets

Medium

Given an integer array bloomDay, an integer m and an integer k.

We need to make m bouquets. To make a bouquet, you need to use k **adjacent flowers** from the garden.

The garden consists of n flowers, the ith flower will bloom in the bloomDay[i] and then can be used in **exactly one** bouquet.

Return *the minimum number of days* you need to wait to be able to make m bouquets from the garden. If it is impossible to make m bouquets return **-1**.

**Example 1:**

**Input:** bloomDay = [1,10,3,10,2], m = 3, k = 1

**Output:** 3

**Explanation:** Let's see what happened in the first three days. x means flower bloomed and \_ means flower didn't bloom in the garden.

We need 3 bouquets each should contain 1 flower.

After day 1: [x, \_, \_, \_, \_] // we can only make one bouquet.

After day 2: [x, \_, \_, \_, x] // we can only make two bouquets.

After day 3: [x, \_, x, \_, x] // we can make 3 bouquets. The answer is 3.

**Example 2:**

**Input:** bloomDay = [1,10,3,10,2], m = 3, k = 2

**Output:** -1

**Explanation:** We need 3 bouquets each has 2 flowers, that means we need 6 flowers. We only have 5 flowers so it is impossible to get the needed bouquets and we return -1.

**Example 3:**

**Input:** bloomDay = [7,7,7,7,12,7,7], m = 2, k = 3

**Output:** 12

**Explanation:** We need 2 bouquets each should have 3 flowers.

Here's the garden after the 7 and 12 days:

After day 7: [x, x, x, x, \_, x, x]

We can make one bouquet of the first three flowers that bloomed. We cannot make another bouquet from the last three flowers that bloomed because they are not adjacent.

After day 12: [x, x, x, x, x, x, x]

It is obvious that we can make two bouquets in different ways.

**Example 4:**

**Input:** bloomDay = [1000000000,1000000000], m = 1, k = 1

**Output:** 1000000000

**Explanation:** You need to wait 1000000000 days to have a flower ready for a bouquet.

**Example 5:**

**Input:** bloomDay = [1,10,2,9,3,8,4,7,5,6], m = 4, k = 2

**Output:** 9

**Constraints:**

* bloomDay.length == n
* 1 <= n <= 10^5
* 1 <= bloomDay[i] <= 10^9
* 1 <= m <= 10^6
* 1 <= k <= n

class Solution {

public:

    int minDays(vector<int>& v, int m, int k) {

        int n = v.size();

        if (n < m\*k) return -1;

        int lo = \*min\_element(v.begin(), v.end());

        int hi = \*max\_element(v.begin(), v.end());

        while (lo < hi) {

            int mid = lo + (hi-lo)/2;

            if (check(v, m, k, mid)) hi = mid;

            else lo = mid + 1;

        }

        return hi;

    }

    bool check(vector<int>& v, int m, int k, int day) {

        int sum = 0;

        for (int i : v) {

            if (i <= day) {

                if (++sum == k) {

                    if (--m == 0) return true;

                    sum = 0;

                }

            }

            else sum = 0;

        }

        return false;

    }

};

### 1483. Kth Ancestor of a Tree Node★★

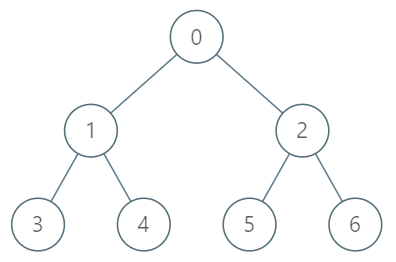
Hard

You are given a tree with n nodes numbered from 0 to n-1 in the form of a parent array where parent[i] is the parent of node i. The root of the tree is node 0.

Implement the function getKthAncestor(int node, int k) to return the k-th ancestor of the given node. If there is no such ancestor, return -1.

The *k-th ancestor* of a tree node is the k-th node in the path from that node to the root.

**Example:**

****

**Input:**

["TreeAncestor","getKthAncestor","getKthAncestor","getKthAncestor"]

[[7,[-1,0,0,1,1,2,2]],[3,1],[5,2],[6,3]]

**Output:**

[null,1,0,-1]

**Explanation:**

TreeAncestor treeAncestor = new TreeAncestor(7, [-1, 0, 0, 1, 1, 2, 2]);

treeAncestor.getKthAncestor(3, 1); // returns 1 which is the parent of 3

treeAncestor.getKthAncestor(5, 2); // returns 0 which is the grandparent of 5

treeAncestor.getKthAncestor(6, 3); // returns -1 because there is no such ancestor

**Constraints:**

* 1 <= k <= n <= 5\*10^4
* parent[0] == -1 indicating that 0 is the root node.
* 0 <= parent[i] < n for all 0 < i < n
* 0 <= node < n
* There will be at most 5\*10^4 queries.

class TreeAncestor {

public:

    vector<vector<int>> v;

    TreeAncestor(int n, vector<int>& parent) {

        vector<vector<int>> par(n, vector<int>(20));

        for (int i = 0; i < n; i++) par[i][0] = parent[i];

        for (int j = 1; j < 20; j++) {

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

                if (par[i][j - 1] == -1) par[i][j] = -1;

                else par[i][j] = par[par[i][j - 1]][j - 1];

            }

        }

        swap(v, par);

    }

    int getKthAncestor(int node, int k) {

        for (int i = 0; i < 20; i++) {

            if ((k >> i) & 1) {

                node = v[node][i];

                if (node == -1) return -1;

            }

        }

        return node;

    }

};

### 1488. Avoid Flood in The City

Medium

Your country has an infinite number of lakes. Initially, all the lakes are empty, but when it rains over the nth lake, the nth lake becomes full of water. If it rains over a lake which is **full of water**, there will be a **flood**. Your goal is to avoid the flood in any lake.

Given an integer array rains where:

* rains[i] > 0 means there will be rains over the rains[i] lake.
* rains[i] == 0 means there are no rains this day and you can choose **one lake** this day and **dry it**.

Return *an array ans* where:

* ans.length == rains.length
* ans[i] == -1 if rains[i] > 0.
* ans[i] is the lake you choose to dry in the ith day if rains[i] == 0.

If there are multiple valid answers return **any** of them. If it is impossible to avoid flood return **an empty array**.

Notice that if you chose to dry a full lake, it becomes empty, but if you chose to dry an empty lake, nothing changes. (see example 4)

**Example 1:**

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

**Output:** [-1,-1,-1,-1]

**Explanation:** After the first day full lakes are [1]

After the second day full lakes are [1,2]

After the third day full lakes are [1,2,3]

After the fourth day full lakes are [1,2,3,4]

There's no day to dry any lake and there is no flood in any lake.

**Example 2:**

**Input:** rains = [1,2,0,0,2,1]

**Output:** [-1,-1,2,1,-1,-1]

**Explanation:** After the first day full lakes are [1]

After the second day full lakes are [1,2]

After the third day, we dry lake 2. Full lakes are [1]

After the fourth day, we dry lake 1. There is no full lakes.

After the fifth day, full lakes are [2].

After the sixth day, full lakes are [1,2].

It is easy that this scenario is flood-free. [-1,-1,1,2,-1,-1] is another acceptable scenario.

**Example 3:**

**Input:** rains = [1,2,0,1,2]

**Output:** []

**Explanation:** After the second day, full lakes are [1,2]. We have to dry one lake in the third day.

After that, it will rain over lakes [1,2]. It's easy to prove that no matter which lake you choose to dry in the 3rd day, the other one will flood.

**Example 4:**

**Input:** rains = [69,0,0,0,69]

**Output:** [-1,69,1,1,-1]

**Explanation:** Any solution on one of the forms [-1,69,x,y,-1], [-1,x,69,y,-1] or [-1,x,y,69,-1] is acceptable where 1 <= x,y <= 10^9

**Example 5:**

**Input:** rains = [10,20,20]

**Output:** []

**Explanation:** It will rain over lake 20 two consecutive days. There is no chance to dry any lake.

**Constraints:**

* 1 <= rains.length <= 10^5

class Solution {

public:

    vector<int> avoidFlood(vector<int>& rains) {

        int n = rains.size();

        vector<int> res(n, -1);

        map<int, int> mp;

        set<int> st;

        for (int i = 0; i < n; ++i) {

            if (rains[i] == 0) st.insert(i);

            else if (mp.count(rains[i])) {

                auto it = st.upper\_bound(mp[rains[i]]);

                if (it == st.end()) return {};

                res[\*it] = rains[i];

                st.erase(it);

                mp[rains[i]] = i;

            }

            else mp[rains[i]] = i;

        }

        if (st.empty()) return res;

        for (auto it = st.rbegin(); it != st.rend(); ++it) {

            res[\*it] = 1;

        }

        return res;

    }

};

### 1489. Find Critical and Pseudo-Critical Edges in Minimum Spanning Tree

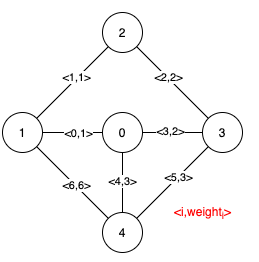
Hard

Given a weighted undirected connected graph with n vertices numbered from 0 to n-1, and an array edges where edges[i] = [fromi, toi, weighti] represents a bidirectional and weighted edge between nodes fromi and toi. A minimum spanning tree (MST) is a subset of the edges of the graph that connects all vertices without cycles and with the minimum possible total edge weight.

Find *all the critical and pseudo-critical edges in the minimum spanning tree (MST) of the given graph*. An MST edge whose deletion from the graph would cause the MST weight to increase is called a *critical edge*. A *pseudo-critical edge*, on the other hand, is that which can appear in some MSTs but not all.

Note that you can return the indices of the edges in any order.

**Example 1:**

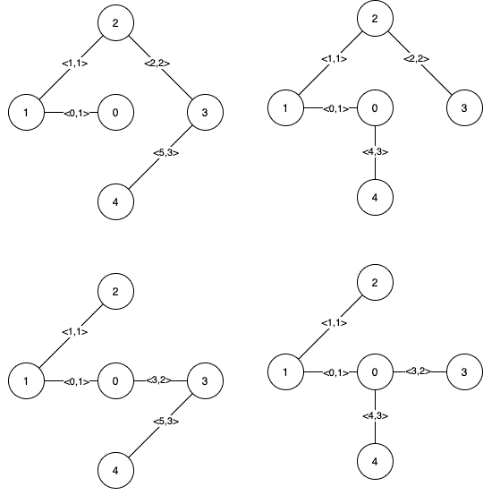


**Input:** n = 5, edges = [[0,1,1],[1,2,1],[2,3,2],[0,3,2],[0,4,3],[3,4,3],[1,4,6]]

**Output:** [[0,1],[2,3,4,5]]

**Explanation:** The figure above describes the graph.

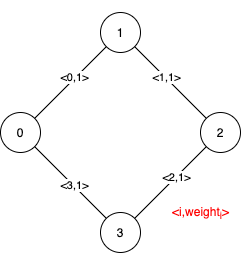
The following figure shows all the possible MSTs:



Notice that the two edges 0 and 1 appear in all MSTs, therefore they are critical edges, so we return them in the first list of the output.

The edges 2, 3, 4, and 5 are only part of some MSTs, therefore they are considered pseudo-critical edges. We add them to the second list of the output.

**Example 2:**



**Input:** n = 4, edges = [[0,1,1],[1,2,1],[2,3,1],[0,3,1]]

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

**Explanation:** We can observe that since all 4 edges have equal weight, choosing any 3 edges from the given 4 will yield an MST. Therefore all 4 edges are pseudo-critical.

**Constraints:**

* 2 <= n <= 100
* 1 <= edges.length <= min(200, n \* (n - 1) / 2)
* edges[i].length == 3
* 0 <= fromi < toi < n
* 1 <= weighti <= 1000
* All pairs (fromi, toi) are distinct.

class Solution {

public:

    vector<vector<int>> findCriticalAndPseudoCriticalEdges(int n, vector<vector<int>>& edges) {

        int m = edges.size();

        for (int i = 0; i < m; ++i) edges[i].push\_back(i);

        auto cmp = [](const vector<int> &lhs, const vector<int> &rhs) {

            return lhs[2] < rhs[2];

        };

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

        int minMST = MST(n, edges, -1, -1);

        vector<vector<int>> res(2);

        for (int i = 0; i < m; ++i) {

            if (minMST < MST(n, edges, i, -1))

                res[0].push\_back(edges[i][3]);

            else if (minMST == MST(n, edges, -1, i))

                res[1].push\_back(edges[i][3]);

        }

        return res;

    }

private:

    int MST(int n, const vector<vector<int>>& edges, int not\_use, int must\_use) {

        vector<int> fa(n, -1);

        int res = 0, cnt = 0;

        if (must\_use != -1) {

            Union(edges[must\_use][0], edges[must\_use][1], fa);

            res += edges[must\_use][2];

            cnt = 1;

        }

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

            if (i == not\_use || i == must\_use) continue;

            const vector<int> &t = edges[i];

            if (!Union(t[0], t[1], fa)) continue;

            res += t[2];

            ++cnt;

        }

        return cnt == n-1 ? res : INT\_MAX;

}

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

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

    }

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

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

        if (fax == fay) return false;

        fa[fax] += fa[fay];

        fa[fay] = fax;

        return true;

    }

};

### 1494. Parallel Courses II★★

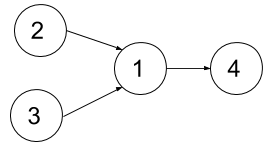
Hard

Given the integer n representing the number of courses at some university labeled from 1 to n, and the array dependencies where dependencies[i] = [xi, yi]  represents a prerequisite relationship, that is, the course xi must be taken before the course yi.  Also, you are given the integer k.

In one semester you can take **at most** k courses as long as you have taken all the prerequisites for the courses you are taking.

*Return the minimum number of semesters to take all courses*. It is guaranteed that you can take all courses in some way.

**Example 1:**

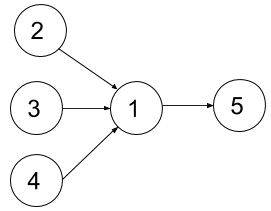
****

**Input:** n = 4, dependencies = [[2,1],[3,1],[1,4]], k = 2

**Output:** 3

**Explanation:** The figure above represents the given graph. In this case we can take courses 2 and 3 in the first semester, then take course 1 in the second semester and finally take course 4 in the third semester.

**Example 2:**

****

**Input:** n = 5, dependencies = [[2,1],[3,1],[4,1],[1,5]], k = 2

**Output:** 4

**Explanation:** The figure above represents the given graph. In this case one optimal way to take all courses is: take courses 2 and 3 in the first semester and take course 4 in the second semester, then take course 1 in the third semester and finally take course 5 in the fourth semester.

**Example 3:**

**Input:** n = 11, dependencies = [], k = 2

**Output:** 6

**Constraints:**

* 1 <= n <= 15
* 1 <= k <= n
* 0 <= dependencies.length <= n \* (n-1) / 2
* dependencies[i].length == 2
* 1 <= xi, yi <= n
* xi != yi
* All prerequisite relationships are distinct, that is, dependencies[i] != dependencies[j].
* The given graph is a directed acyclic graph.

class Solution {

public:

    int minNumberOfSemesters(int n, vector<vector<int>>& dependencies, int k) {

        vector<int> pre(n);

        for(const auto &e : dependencies){

            pre[e[1]-1] |= 1 << (e[0]-1);

        }

        vector<int> dp(1 << n, n);

        dp[0] = 0;

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

            int ex = 0;

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

                if ((i & pre[j]) == pre[j])

                    ex |= 1 << j;

            }

            ex &= ~i;

            for (int s = ex; s; s = (s - 1) & ex) {

                if (\_\_builtin\_popcount(s) <= k) {

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

                }

            }

        }

        return dp.back();

    }

};

### 1498. Number of Subsequences That Satisfy the Given Sum Condition

Medium

Given an array of integers nums and an integer target.

Return the number of **non-empty** subsequences of nums such that the sum of the minimum and maximum element on it is less or equal than target.

Since the answer may be too large, return it modulo 10^9 + 7.

**Example 1:**

**Input:** nums = [3,5,6,7], target = 9

**Output:** 4

**Explanation:** There are 4 subsequences that satisfy the condition.

[3] -> Min value + max value <= target (3 + 3 <= 9)

[3,5] -> (3 + 5 <= 9)

[3,5,6] -> (3 + 6 <= 9)

[3,6] -> (3 + 6 <= 9)

**Example 2:**

**Input:** nums = [3,3,6,8], target = 10

**Output:** 6

**Explanation:** There are 6 subsequences that satisfy the condition. (nums can have repeated numbers).

[3] , [3] , [3,3], [3,6] , [3,6] , [3,3,6]

**Example 3:**

**Input:** nums = [2,3,3,4,6,7], target = 12

**Output:** 61

**Explanation:** There are 63 non-empty subsequences, two of them don't satisfy the condition ([6,7], [7]).

Number of valid subsequences (63 - 2 = 61).

**Example 4:**

**Input:** nums = [5,2,4,1,7,6,8], target = 16

**Output:** 127

**Explanation:** All non-empty subset satisfy the condition (2^7 - 1) = 127

**Constraints:**

* 1 <= nums.length <= 10^5
* 1 <= nums[i] <= 10^6
* 1 <= target <= 10^6

class Solution {

public:

    int numSubseq(vector<int>& A, int target) {

        sort(A.begin(), A.end());

        int res = 0, n = A.size(), l = 0, r = n - 1, mod = 1e9 + 7;

        vector<int> pows(n, 1);

        for (int i = 1 ; i < n ; ++i)

            pows[i] = pows[i - 1] \* 2 % mod;

        while (l <= r) {

            if (A[l] + A[r] > target) r--;

            else {

                res = (res + pows[r - l++]) % mod;

            }

        }

        return res;

    }

};

### 1499. Max Value of Equation★★

Hard

Given an array points containing the coordinates of points on a 2D plane, sorted by the x-values, where points[i] = [xi, yi] such that xi < xj for all 1 <= i < j <= points.length. You are also given an integer k.

Find the *maximum value of the equation*yi + yj + |xi - xj| where |xi - xj| <= k and 1 <= i < j <= points.length. It is guaranteed that there exists at least one pair of points that satisfy the constraint |xi - xj| <= k.

**Example 1:**

**Input:** points = [[1,3],[2,0],[5,10],[6,-10]], k = 1

**Output:** 4

**Explanation:** The first two points satisfy the condition |xi - xj| <= 1 and if we calculate the equation we get 3 + 0 + |1 - 2| = 4. Third and fourth points also satisfy the condition and give a value of 10 + -10 + |5 - 6| = 1.

No other pairs satisfy the condition, so we return the max of 4 and 1.

**Example 2:**

**Input:** points = [[0,0],[3,0],[9,2]], k = 3

**Output:** 3

**Explanation:** Only the first two points have an absolute difference of 3 or less in the x-values, and give the value of 0 + 0 + |0 - 3| = 3.

**Constraints:**

* 2 <= points.length <= 10^5
* points[i].length == 2
* -10^8 <= points[i][0], points[i][1] <= 10^8
* 0 <= k <= 2 \* 10^8
* points[i][0] < points[j][0] for all 1 <= i < j <= points.length
* xi form a strictly increasing sequence.

class Solution {

public:

    using PII = pair<int, int>;

    int findMaxValueOfEquation(vector<vector<int>>& points, int k) {

        deque<PII> dq;

        int res = -2e9;

        for(const auto &pnt : points) {

            while (!dq.empty() && pnt[0] - dq.front().first > k)

dq.pop\_front();

            if (!dq.empty())

                res = max(res, dq.front().second + pnt[0] + pnt[1]);

            while (!dq.empty() && dq.back().second <= pnt[1] - pnt[0])

                dq.pop\_back();

            dq.emplace\_back(pnt[0], pnt[1] - pnt[0]);

        }

        return res;

    }

};

### 1504. Count Submatrices With All Ones★★

Medium

Given a rows \* columns matrix mat of ones and zeros, return how many **submatrices** have all ones.

**Example 1:**

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

  [1,1,0],

  [1,1,0]]

**Output:** 13

**Explanation:**

There are **6** rectangles of side 1x1.

There are **2** rectangles of side 1x2.

There are **3** rectangles of side 2x1.

There is **1** rectangle of side 2x2.

There is **1** rectangle of side 3x1.

Total number of rectangles = 6 + 2 + 3 + 1 + 1 = **13.**

**Example 2:**

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

  [0,1,1,1],

  [1,1,1,0]]

**Output:** 24

**Explanation:**

There are **8** rectangles of side 1x1.

There are **5** rectangles of side 1x2.

There are **2** rectangles of side 1x3.

There are **4** rectangles of side 2x1.

There are **2** rectangles of side 2x2.

There are **2** rectangles of side 3x1.

There is **1** rectangle of side 3x2.

Total number of rectangles = 8 + 5 + 2 + 4 + 2 + 2 + 1 = 24**.**

**Example 3:**

**Input:** mat = [[1,1,1,1,1,1]]

**Output:** 21

**Example 4:**

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

**Output:** 5

**Constraints:**

* 1 <= rows <= 150
* 1 <= columns <= 150
* 0 <= mat[i][j] <= 1

//O(n^3)

class Solution {

public:

    int numSubmat(vector<vector<int>>& mat) {

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

        int res = 0;

        vector<int> h(m, 0);

        for (int i = 0; i < n; ++i) {

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

                h[j] = mat[i][j] ? (h[j]+1) : 0;

            }

            res += solve(h);

        }

        return res;

    }

private:

   int solve(const vector<int> &h) {

       int res = 0;

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

           int t = h[i];

           for (int j = i; j >= 0; --j) {

               res += (t = min(h[j], t));

           }

       }

       return res;

   }

    int solve\_0(const vector<int> &h) {

       vector<int> sum(h.size(), 0);

       stack<int> stk;

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

           int t = h[i];

           for (int j = i; j >= 0; --j) {

               if (h[j] >= t) sum[i] += t;

               else {

                   sum[i] += sum[j];

                   break;

               }

           }

       }

       return accumulate(sum.begin(), sum.end(), 0);

   }

};

//O(n^2)

class Solution {

public:

    int numSubmat(vector<vector<int>>& mat) {

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

        int res = 0;

        vector<int> h(m, 0);

        for (int i = 0; i < n; ++i) {

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

                h[j] = mat[i][j] ? (h[j]+1) : 0;

            }

            res += solve(h);

        }

        return res;

    }

private:

   int solve(const vector<int> &h) {

       vector<int> sum(h.size(), 0);

       stack<int> stk;

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

           while (!stk.empty() && h[stk.top()] >= h[i]) stk.pop();

           if (stk.empty()) sum[i] = h[i] \* (i + 1);

           else {

               sum[i] = sum[stk.top()] + h[i] \* (i - stk.top());

           }

           stk.push(i);

       }

       return accumulate(sum.begin(), sum.end(), 0);

   }

};

### 1505. Minimum Possible Integer After at Most K Adjacent Swaps On Digits

Hard

Given a string num representing **the digits** of a very large integer and an integer k.

You are allowed to swap any two adjacent digits of the integer **at most** k times.

Return *the minimum integer* you can obtain also as a string.

**Example 1:**

https://assets.leetcode.com/uploads/2020/06/17/q4_1.jpg

**Input:** num = "4321", k = 4

**Output:** "1342"

**Explanation:** The steps to obtain the minimum integer from 4321 with 4 adjacent swaps are shown.

**Example 2:**

**Input:** num = "100", k = 1

**Output:** "010"

**Explanation:** It's ok for the output to have leading zeros, but the input is guaranteed not to have any leading zeros.

**Example 3:**

**Input:** num = "36789", k = 1000

**Output:** "36789"

**Explanation:** We can keep the number without any swaps.

**Example 4:**

**Input:** num = "22", k = 22

**Output:** "22"

**Example 5:**

**Input:** num = "9438957234785635408", k = 23

**Output:** "0345989723478563548"

**Constraints:**

* 1 <= num.length <= 30000
* num contains **digits** only and doesn't have **leading zeros**.
* 1 <= k <= 10^9

class Solution {

    vector<pair<int, int>> resort;

    priority\_queue<int, vector<int>, greater<int>> nums[10];

    int used[30001], n;

    int getSum(int index)  {

        int sum = 0;

        while (index > 0) {

            sum += used[index];

            index -= index & (-index);

        }

        return sum;

    }

    void updateBIT(int index, int val) {

        while (index <= n) {

         used[index] += val;

         index += index & (-index);

        }

    }

public:

    string minInteger(string num, int k) {

        memset(used, 0, sizeof(used));

        int ctr = 0;

        n = num.size();

        for (int i = 0; i < n; i++)

            nums[num[i] - '0'].push(i + 1);

        string res;

        while (ctr < n && k > 0) {

            for (int i = 0; i <= 9; i++) {

                if (!nums[i].empty()) {

                    int cur = nums[i].top();

                    int holes = getSum(cur - 1);

                    int act = cur - holes;

                    if (act - 1 <= k) {

                        res += ('0' + i);

                        k -= (act - 1);

                        updateBIT(cur, 1);

                        nums[i].pop();

                        break;

                    }

                }

            }

            ctr++;

        }

        for(int i = 0; i <= 9; i++) {

            while (!nums[i].empty()) {

                resort.emplace\_back(nums[i].top(), i);

                nums[i].pop();

            }

        }

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

        for (auto &p : resort) {

            res += ('0' + p.second);

        }

        return res;

    }

};

### 1514. Path with Maximum Probability

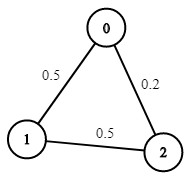
Medium

You are given an undirected weighted graph of n nodes (0-indexed), represented by an edge list where edges[i] = [a, b] is an undirected edge connecting the nodes a and b with a probability of success of traversing that edge succProb[i].

Given two nodes start and end, find the path with the maximum probability of success to go from start to end and return its success probability.

If there is no path from start to end, **return 0**. Your answer will be accepted if it differs from the correct answer by at most **1e-5**.

**Example 1:**

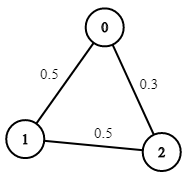
****

**Input:** n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.2], start = 0, end = 2

**Output:** 0.25000

**Explanation:** There are two paths from start to end, one having a probability of success = 0.2 and the other has 0.5 \* 0.5 = 0.25.

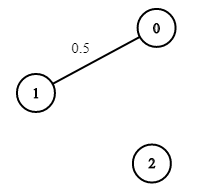
**Example 2:**

****

**Input:** n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.3], start = 0, end = 2

**Output:** 0.30000

**Example 3:**

****

**Input:** n = 3, edges = [[0,1]], succProb = [0.5], start = 0, end = 2

**Output:** 0.00000

**Explanation:** There is no path between 0 and 2.

**Constraints:**

* 2 <= n <= 10^4
* 0 <= start, end < n
* start != end
* 0 <= a, b < n
* a != b
* 0 <= succProb.length == edges.length <= 2\*10^4
* 0 <= succProb[i] <= 1
* There is at most one edge between every two nodes.

class Solution {

public:

    using pdi = pair<double, int>;

    double maxProbability(int n, vector<vector<int>>& edges, vector<double>& pro, int start, int end) {

        vector<vector<pdi>> g(n);

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

            g[edges[i][0]].emplace\_back(pro[i], edges[i][1]);

            g[edges[i][1]].emplace\_back(pro[i], edges[i][0]);

        }

        vector<bool> seen(n, false);

        priority\_queue<pdi> pq;

        pq.emplace(1.0, start);

        vector<double> mx(n, 0.0);

        mx[start] = 1.0;

        while (!pq.empty()) {

            auto [proba, node] = pq.top();

            pq.pop();

            if (!seen[node]) {

                seen[node] = true;

                for (const auto &[i, j] : g[node]) {

                    if (mx[j] < i\*proba) {

                        mx[j] = i\*proba;

                        pq.emplace(mx[j],j);

                    }

                }

            }

        }

        return mx[end];

    }

};

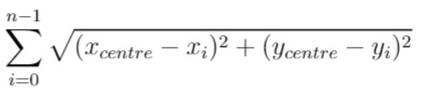
### 1515. Best Position for a Service Centre

Hard

A delivery company wants to build a new service centre in a new city. The company knows the positions of all the customers in this city on a 2D-Map and wants to build the new centre in a position such that **the sum of the euclidean distances to all customers is minimum**.

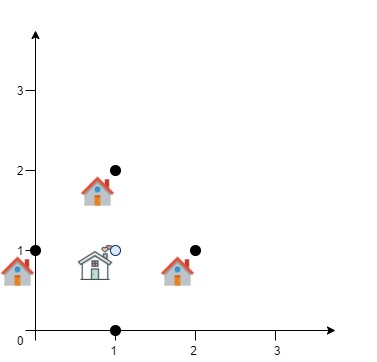
Given an array positions where positions[i] = [xi, yi] is the position of the ith customer on the map, return *the minimum sum of the euclidean distances* to all customers.

In other words, you need to choose the position of the service centre [xcentre, ycentre] such that the following formula is minimized:



Answers within 10^-5 of the actual value will be accepted.

**Example 1:**

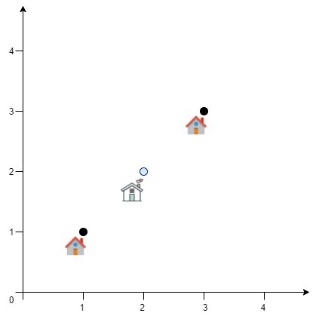


**Input:** positions = [[0,1],[1,0],[1,2],[2,1]]

**Output:** 4.00000

**Explanation:** As shown, you can see that choosing [xcentre, ycentre] = [1, 1] will make the distance to each customer = 1, the sum of all distances is 4 which is the minimum possible we can achieve.

**Example 2:**



**Input:** positions = [[1,1],[3,3]]

**Output:** 2.82843

**Explanation:** The minimum possible sum of distances = sqrt(2) + sqrt(2) = 2.82843

**Example 3:**

**Input:** positions = [[1,1]]

**Output:** 0.00000

**Example 4:**

**Input:** positions = [[1,1],[0,0],[2,0]]

**Output:** 2.73205

**Explanation:** At the first glance, you may think that locating the centre at [1, 0] will achieve the minimum sum, but locating it at [1, 0] will make the sum of distances = 3.

Try to locate the centre at [1.0, 0.5773502711] you will see that the sum of distances is 2.73205.

Be careful with the precision!

**Example 5:**

**Input:** positions = [[0,1],[3,2],[4,5],[7,6],[8,9],[11,1],[2,12]]

**Output:** 32.94036

**Explanation:** You can use [4.3460852395, 4.9813795505] as the position of the centre.

**Constraints:**

* 1 <= positions.length <= 50
* positions[i].length == 2
* 0 <= positions[i][0], positions[i][1] <= 100

class Solution {

public:

    double getMinDistSum(vector<vector<int>>& pos) {

        double left = 100, bottom = 100, right = 0, top = 0;

        for (auto &p : pos) {

            left = min(left, (double)p[0]);

            bottom = min(bottom, (double)p[1]);

            right = max(right, (double)p[0]);

            top = max(top, (double)p[1]);

        }

        double res = DBL\_MAX, res\_x = 0, res\_y = 0;

        for (double delta = 10; delta >= 0.00001; delta /= 10) {

            for (double x = left; x <= right; x += delta)

                for (double y = bottom; y <= top; y += delta) {

                    double d = 0;

                    for (auto &p : pos)

                        d += sqrt((p[0] - x) \* (p[0] - x)

+ (p[1] - y) \* (p[1] - y));

                    if (res > d) {

                        res = d;

                        res\_x = x;

                        res\_y = y;

                    }

                }

            left = res\_x - delta;

            bottom = res\_y - delta;

            right = res\_x + delta \* 2;

            top = res\_y + delta \* 2;

        }

        return res == DBL\_MAX ? 0 : res;

    }

};

### 1520. Maximum Number of Non-Overlapping Substrings

Hard

Given a string s of lowercase letters, you need to find the maximum number of **non-empty** substrings of s that meet the following conditions:

1. The substrings do not overlap, that is for any two substrings s[i..j] and s[k..l], either j < k or i > l is true.
2. A substring that contains a certain character c must also contain all occurrences of c.

Find *the maximum number of substrings that meet the above conditions*. If there are multiple solutions with the same number of substrings, *return the one with minimum total length.*It can be shown that there exists a unique solution of minimum total length.

Notice that you can return the substrings in **any** order.

**Example 1:**

**Input:** s = "adefaddaccc"

**Output:** ["e","f","ccc"]

**Explanation:** The following are all the possible substrings that meet the conditions:

[

  "adefaddaccc"

  "adefadda",

  "ef",

  "e",

"f",

  "ccc",

]

If we choose the first string, we cannot choose anything else and we'd get only 1. If we choose "adefadda", we are left with "ccc" which is the only one that doesn't overlap, thus obtaining 2 substrings. Notice also, that it's not optimal to choose "ef" since it can be split into two. Therefore, the optimal way is to choose ["e","f","ccc"] which gives us 3 substrings. No other solution of the same number of substrings exist.

**Example 2:**

**Input:** s = "abbaccd"

**Output:** ["d","bb","cc"]

**Explanation:** Notice that while the set of substrings ["d","abba","cc"] also has length 3, it's considered incorrect since it has larger total length.

**Constraints:**

* 1 <= s.length <= 10^5
* s contains only lowercase English letters.

class Solution {

public:

    vector<string> maxNumOfSubstrings(string s) {

        vector<int> start(26, INT\_MAX), end(26, INT\_MIN), skip(26, 0);

        vector<string> result;

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

            start[s[i] - 'a'] = min(start[s[i] - 'a'], i);

end[s[i] - 'a'] = i;

}

        for (int i = 0; i < 26; ++i)

            for (int j = start[i]; j <= end[i]; ++j)

                if (start[s[j] - 'a'] < start[i]) skip[i] = 1;

                else end[i] = max(end[i], end[s[j] - 'a']);

        for (int i = s.size() - 1, cut = INT\_MAX; i >= 0; --i)

            if (i == start[s[i] - 'a'] && end[s[i] - 'a'] < cut

&& !skip[s[i] - 'a'])

                result.push\_back(s.substr((cut = i), end[s[i] - 'a']

- i + 1));

        return result;

    }

};

class Solution {

public:

    int checkSubstr(string &s, int i, vector<int> &l, vector<int> &r) {

        int right = r[s[i] - 'a'];

        for (auto j = i; j <= right; ++j) {

            if (l[s[j] - 'a'] < i)

                return -1;

            right = max(right, r[s[j] - 'a']);

        }

        return right;

    }

    vector<string> maxNumOfSubstrings(string s) {

        vector<int> l(26, INT\_MAX), r(26, INT\_MIN);

        vector<string> res;

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

            l[s[i] - 'a'] = min(l[s[i] - 'a'], i);

            r[s[i] - 'a'] = i;

        }

        int right = -1;

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

            if (i == l[s[i] - 'a']) {

                int new\_right = checkSubstr(s, i, l, r);

                if (new\_right != -1) {

                    if (i > right)

                        res.push\_back("");

                    right = new\_right;

                    res.back() = s.substr(i, right - i + 1);

                }

            }

        }

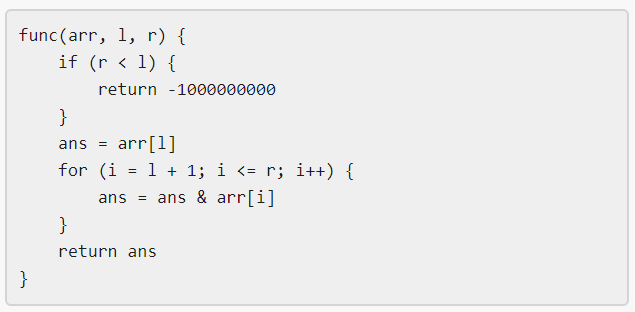
        return res;

    }

};

### 1521. Find a Value of a Mysterious Function Closest to Target

Hard



Winston was given the above mysterious function func. He has an integer array arr and an integer target and he wants to find the values l and r that make the value |func(arr, l, r) - target| minimum possible.

Return *the minimum possible value* of |func(arr, l, r) - target|.

Notice that func should be called with the values l and r where 0 <= l, r < arr.length.

**Example 1:**

**Input:** arr = [9,12,3,7,15], target = 5

**Output:** 2

**Explanation:** Calling func with all the pairs of [l,r] = [[0,0],[1,1],[2,2],[3,3],[4,4],[0,1],[1,2],[2,3],[3,4],[0,2],[1,3],[2,4],[0,3],[1,4],[0,4]], Winston got the following results [9,12,3,7,15,8,0,3,7,0,0,3,0,0,0]. The value closest to 5 is 7 and 3, thus the minimum difference is 2.

**Example 2:**

**Input:** arr = [1000000,1000000,1000000], target = 1

**Output:** 999999

**Explanation:** Winston called the func with all possible values of [l,r] and he always got 1000000, thus the min difference is 999999.

**Example 3:**

**Input:** arr = [1,2,4,8,16], target = 0

**Output:** 0

**Constraints:**

* 1 <= arr.length <= 10^5
* 1 <= arr[i] <= 10^6
* 0 <= target <= 10^7

class Solution {

public:

    int closestToTarget(vector<int>& arr, int target) {

        int len = (int)arr.size();

        vector<vector<int> > tab(30);

        for(int i = 0; i < len; i++)

            for(int j = 0; j < 30; j++) if((arr[i] >> j) & 1)

                tab[j].push\_back(i);

        vector<int> dp(len);

        int ans = abs(arr[0] - target);

        for(int i = len - 1; i >= 0; i--)

            //In round i, dp[i], dp[i+1]...dp[N-1] is our new sequence of arr[i], arr[i]&arr[i+1],...,arr[i]&...&arr[N-1]

        {

            for(int j = 0; j < 30; j++) if(!((arr[i] >> j) & 1))

                while(!tab[j].empty() && tab[j].back() > i)

                //Find elements whose j'th bit need to be set to zero

                {

                    dp[tab[j].back()] -= 1 << j;

                    tab[j].pop\_back();

 //After set this element's j'th bit to zero, we need not consider this bit again

                }

            dp[i] = arr[i];

            //Binary search

            int l = i, r = len;

            while(l < r)

            {

                int m = l + (r - l) / 2;

                if(dp[m] > target) l = m + 1;

                else r = m;

            }

            if(l == len) ans = min(ans, abs(target - dp[l - 1]));

            else if(l == i) ans = min(ans, abs(target - dp[l]));

            else ans = min(ans, min(abs(target - dp[l]), abs(target - dp[l - 1])));

        }

        return ans;

    }

};

class Solution {

public:

    int closestToTarget(vector<int>& arr, int target) {

        int len = (int)arr.size();

        vector<vector<int> > tab(30);

        for(int i = 0; i < len; i++)

            for(int j = 0; j < 30; j++) if((arr[i] >> j) & 1)

                tab[j].push\_back(i);

        vector<int> dp(len + 1); dp[len] = arr[0];

        //dummy value to avoid "index out of range"

        int ans = abs(arr[0] - target);

        int l = len - 1;

        for(int i = len - 1; i >= 0; i--)

        {

            for(int j = 0; j < 30; j++) if(!((arr[i] >> j) & 1))

                while(!tab[j].empty() && tab[j].back() > i)

                {

                    dp[tab[j].back()] -= 1 << j;

                    tab[j].pop\_back();

                }

            dp[i] = arr[i];

            while(l > i && dp[l] < target) l--;

            //After this line, we have dp[l] >= target, dp[l + 1] < target,

            //or l == i and all elements in this new sequence is smaller than target

            ans = min(ans, min(abs(target - dp[l]), abs(target - dp[l + 1])));

        }

        return ans;

    }

};

### 1526. Minimum Number of Increments on Subarrays to Form a Target Array

Hard

Given an array of positive integers target and an array initial of same size with all zeros.

Return the minimum number of operations to form a target array from initial if you are allowed to do the following operation:

* Choose **any** subarray from initial and increment each value by one.

The answer is guaranteed to fit within the range of a 32-bit signed integer.

**Example 1:**

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

**Output:** 3

**Explanation:** We need at least 3 operations to form the target array from the initial array.

[0,0,0,0,0] increment 1 from index 0 to 4 (inclusive).

[1,1,1,1,1] increment 1 from index 1 to 3 (inclusive).

[1,2,2,2,1] increment 1 at index 2.

[1,2,3,2,1] target array is formed.

**Example 2:**

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

**Output:** 4

**Explanation:** (initial)[0,0,0,0] -> [1,1,1,1] -> [1,1,1,2] -> [2,1,1,2] -> [3,1,1,2] (target).

**Example 3:**

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

**Output:** 7

**Explanation:** (initial)[0,0,0,0,0] -> [1,1,1,1,1] -> [2,1,1,1,1] -> [3,1,1,1,1]

-> [3,1,2,2,2] -> [3,1,3,3,2] -> [3,1,4,4,2] -> [3,1,5,4,2] (target).

**Example 4:**

**Input:** target = [1,1,1,1]

**Output:** 1

**Constraints:**

* 1 <= target.length <= 10^5
* 1 <= target[i] <= 10^5

class Solution {

public:

    int minNumberOperations(vector<int>& v) {

        int res  = v[0];

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

            if (v[i] > v[i-1]) res += v[i] - v[i-1];

        }

        return res;

    }

};

### 1531. String Compression II★★

Hard

[Run-length encoding](http://en.wikipedia.org/wiki/Run-length_encoding) is a string compression method that works by replacing consecutive identical characters (repeated 2 or more times) with the concatenation of the character and the number marking the count of the characters (length of the run). For example, to compress the string "aabccc" we replace "aa" by "a2" and replace "ccc" by "c3". Thus the compressed string becomes "a2bc3".

Notice that in this problem, we are not adding '1' after single characters.

Given a string s and an integer k. You need to delete **at most** k characters from s such that the run-length encoded version of s has minimum length.

Find the minimum length of the run-length encoded version of s after deleting at most k characters.

**Example 1:**

**Input:** s = "aaabcccd", k = 2

**Output:** 4

**Explanation:** Compressing s without deleting anything will give us "a3bc3d" of length 6. Deleting any of the characters 'a' or 'c' would at most decrease the length of the compressed string to 5, for instance delete 2 'a' then we will have s = "abcccd" which compressed is abc3d. Therefore, the optimal way is to delete 'b' and 'd', then the compressed version of s will be "a3c3" of length 4.

**Example 2:**

**Input:** s = "aabbaa", k = 2

**Output:** 2

**Explanation:** If we delete both 'b' characters, the resulting compressed string would be "a4" of length 2.

**Example 3:**

**Input:** s = "aaaaaaaaaaa", k = 0

**Output:** 3

**Explanation:** Since k is zero, we cannot delete anything. The compressed string is "a11" of length 3.

**Constraints:**

* 1 <= s.length <= 100
* 0 <= k <= s.length
* s contains only lowercase English letters.

class Solution {

public:

    // dp[left][k] means the minimal coding size for substring

    // s[left:] and removing at most k chars

    int getLengthOfOptimalCompression(string s, int k) {

        memset(dp, -1, sizeof(dp));

        return dfs(s, 0, k);

    }

private:

    int dp[101][101];

    int dfs(string &s, int left, int K) {

        if (s.size() - left <= K) return 0;

        if (dp[left][K] >= 0) return dp[left][K];

        int k = K, cnt = 1;

        int res = k ? dfs(s, left + 1, k - 1) : 10000;

        for (int i = left + 1; i <= s.size(); ++i) {

            res = min(res, dfs(s, i, k) + 1 + xs(cnt));

            if (i == s.size()) break;

            if (s[i] == s[left]) ++cnt;

            else if (--k < 0) break;

        }

        return dp[left][K] = res;

    }

    int xs(int x) { return x == 1 ? 0 : x < 10 ? 1 : x < 100 ? 2 : 3; }

};

class Solution {

public:

    int xs(int x) { return x == 1 ? 0 : x < 10 ? 1 : x < 100 ? 2 : 3; }

    const int INF = 0x3f3f3f3f;

    int getLengthOfOptimalCompression(string s, int k) {

        int n = s.size();

        vector<vector<int>> dp(n+1,vector<int>(k+2, INF));

        dp[0][0] = 0;

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

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

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

                int cnt = 0, del = j;

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

                    s[m-1] == s[i-1] ? ++cnt : ++del;

                    if (del > k) break;

                    dp[m][del] = min(dp[m][del],

xs(cnt) + 1 + dp[i-1][j]);

                }

            }

        }

        return dp[n][k];

    }

};

### 1537. Get the Maximum Score

Hard

You are given two **sorted** arrays of distinct integers nums1 and nums2.

A **validpath** is defined as follows:

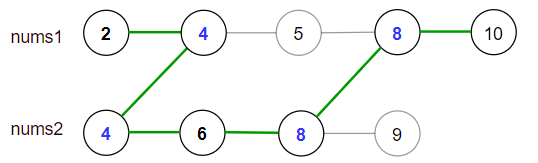
* Choose array nums1 or nums2 to traverse (from index-0).
* Traverse the current array from left to right.
* If you are reading any value that is present in nums1 and nums2 you are allowed to change your path to the other array. (Only one repeated value is considered in the valid path).

*Score* is defined as the sum of uniques values in a valid path.

Return the maximum *score* you can obtain of all possible **valid paths**.

Since the answer may be too large, return it modulo 10^9 + 7.

**Example 1:**

****

**Input:** nums1 = [2,4,5,8,10], nums2 = [4,6,8,9]

**Output:** 30

**Explanation:** Valid paths:

[2,4,5,8,10], [2,4,5,8,9], [2,4,6,8,9], [2,4,6,8,10], (starting from nums1)

[4,6,8,9], [4,5,8,10], [4,5,8,9], [4,6,8,10] (starting from nums2)

The maximum is obtained with the path in green **[2,4,6,8,10]**.

**Example 2:**

**Input:** nums1 = [1,3,5,7,9], nums2 = [3,5,100]

**Output:** 109

**Explanation:** Maximum sum is obtained with the path **[1,3,5,100]**.

**Example 3:**

**Input:** nums1 = [1,2,3,4,5], nums2 = [6,7,8,9,10]

**Output:** 40

**Explanation:** There are no common elements between nums1 and nums2.

Maximum sum is obtained with the path [6,7,8,9,10].

**Example 4:**

**Input:** nums1 = [1,4,5,8,9,11,19], nums2 = [2,3,4,11,12]

**Output:** 61

**Constraints:**

* 1 <= nums1.length <= 10^5
* 1 <= nums2.length <= 10^5
* 1 <= nums1[i], nums2[i] <= 10^7
* nums1 and nums2 are strictly increasing.

class Solution {

public:

    int maxSum(vector<int>& A, vector<int>& B) {

        int i = 0, j = 0, n = A.size(), m = B.size();

        long a = 0, b = 0, mod = 1e9 + 7;

        while (i < n || j < m) {

            if (i < n && (j == m || A[i] < B[j])) {

                a += A[i++];

            } else if (j < m && (i == n || A[i] > B[j])) {

                b += B[j++];

            } else {

                a = b = max(a, b) + A[i];

                i++, j++;

            }

        }

        return max(a, b) % mod;

    }

};

### 1541. Minimum Insertions to Balance a Parentheses String

Medium

Given a parentheses string s containing only the characters '(' and ')'. A parentheses string is **balanced** if:

* Any left parenthesis '(' must have a corresponding two consecutive right parenthesis '))'.
* Left parenthesis '(' must go before the corresponding two consecutive right parenthesis '))'.

In other words, we treat '(' as openning parenthesis and '))' as closing parenthesis.

For example, "())", "())(())))" and "(())())))" are balanced, ")()", "()))" and "(()))" are not balanced.

You can insert the characters '(' and ')' at any position of the string to balance it if needed.

Return *the minimum number of insertions* needed to make s balanced.

**Example 1:**

**Input:** s = "(()))"

**Output:** 1

**Explanation:** The second '(' has two matching '))', but the first '(' has only ')' matching. We need to to add one more ')' at the end of the string to be "(())))" which is balanced.

**Example 2:**

**Input:** s = "())"

**Output:** 0

**Explanation:** The string is already balanced.

**Example 3:**

**Input:** s = "))())("

**Output:** 3

**Explanation:** Add '(' to match the first '))', Add '))' to match the last '('.

**Example 4:**

**Input:** s = "(((((("

**Output:** 12

**Explanation:** Add 12 ')' to balance the string.

**Example 5:**

**Input:** s = ")))))))"

**Output:** 5

**Explanation:** Add 4 '(' at the beginning of the string and one ')' at the end. The string becomes "(((())))))))".

**Constraints:**

* 1 <= s.length <= 10^5
* s consists of '(' and ')' only.

class Solution {

public:

    int minInsertions(string s) {

        int res = 0, right = 0;

        for (char &c: s) {

            if (c == '(') {

                if (right % 2 > 0) {

                    right--;

                    res++;

                }

                right += 2;

            } else {

                right--;

                if (right < 0) {

                    right += 2;

                    res++;

                }

            }

        }

        return right + res;

    }

};

### 1542. Find Longest Awesome Substring★★

Hard

Given a string s. An *awesome* substring is a non-empty substring of s such that we can make any number of swaps in order to make it palindrome.

Return the length of the maximum length **awesome substring** of s.

**Example 1:**

**Input:** s = "3242415"

**Output:** 5

**Explanation:** "24241" is the longest awesome substring, we can form the palindrome "24142" with some swaps.

**Example 2:**

**Input:** s = "12345678"

**Output:** 1

**Example 3:**

**Input:** s = "213123"

**Output:** 6

**Explanation:** "213123" is the longest awesome substring, we can form the palindrome "231132" with some swaps.

**Example 4:**

**Input:** s = "00"

**Output:** 2

**Constraints:**

* 1 <= s.length <= 10^5
* s consists only of digits.

class Solution {

public:

    int longestAwesome(string s) {

        vector<int> dp(1024, s.size());

        int res = 0, mask = 0;

        dp[0] = -1;

        for (auto i = 0; i < s.size(); ++i) {

            mask ^= 1 << (s[i] - '0');

            res = max(res, i - dp[mask]);

            for (auto j = 0; j <= 9; ++j)

                res = max(res, i - dp[mask ^ (1 << j)]);

            dp[mask] = min(dp[mask], i);

        }

        return res;

    }

};

### 1546. Maximum Number of NonOverlapping Subarrays With Sum Equals Target★★

Medium

Given an array nums and an integer target.

Return the maximum number of **non-empty** **non-overlapping** subarrays such that the sum of values in each subarray is equal to target.

**Example 1:**

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

**Output:** 2

**Explanation:** There are 2 non-overlapping subarrays [**1,1**,1,**1,1**] with sum equals to target(2).

**Example 2:**

**Input:** nums = [-1,3,5,1,4,2,-9], target = 6

**Output:** 2

**Explanation:** There are 3 subarrays with sum equal to 6.

([5,1], [4,2], [3,5,1,4,2,-9]) but only the first 2 are non-overlapping.

**Example 3:**

**Input:** nums = [-2,6,6,3,5,4,1,2,8], target = 10

**Output:** 3

**Example 4:**

**Input:** nums = [0,0,0], target = 0

**Output:** 3

**Constraints:**

* 1 <= nums.length <= 10^5
* -10^4 <= nums[i] <= 10^4
* 0 <= target <= 10^6

class Solution {

public:

    unordered\_map<int, int> mp{{0, 0}};

    int maxNonOverlapping(vector<int>& nums, int target) {

        int n = nums.size(), sum = 0;

        vector<int> dp(n+1, 0);

        for(int i = 1; i <= n; i++) {

            sum += nums[i-1];

            auto it = mp.find(sum - target);

            if (it == mp.end()) dp[i] = dp[i-1];

            else {

                dp[i] = max(dp[i-1], dp[it->second] + 1);

            }

            mp[sum] = i;

        }

        return dp.back();

    }

};

class Solution {

public:

    int maxNonOverlapping(vector<int>& nums, int target) {

        int n = nums.size(), res = 0;

        int pre = -1, sum = 0;

        unordered\_map<int, int> mp{{0, -1}};

        for (int i = 0; i < n; ++i) {

            sum += nums[i];

            if (mp.count(sum - target)) {

                int p = mp[sum - target];

                if (p >= pre) {

                    ++res;

                    pre = i;

                }

            }

            mp[sum] = i;

        }

        return res;

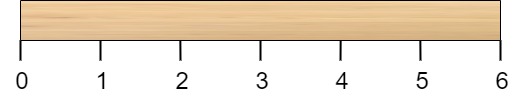
    }

};

### 1547. Minimum Cost to Cut a Stick

Hard

Given a wooden stick of length n units. The stick is labelled from 0 to n. For example, a stick of length **6** is labelled as follows:



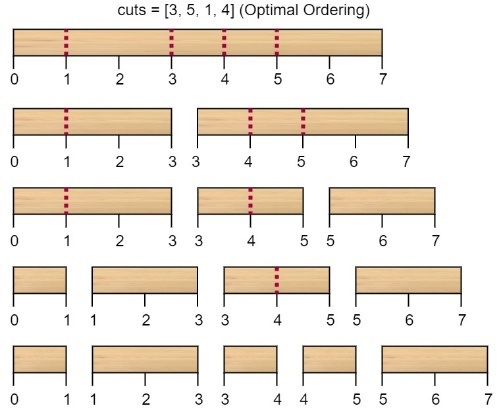
Given an integer array cuts where cuts[i] denotes a position you should perform a cut at.

You should perform the cuts in order, you can change the order of the cuts as you wish.

The cost of one cut is the length of the stick to be cut, the total cost is the sum of costs of all cuts. When you cut a stick, it will be split into two smaller sticks (i.e. the sum of their lengths is the length of the stick before the cut). Please refer to the first example for a better explanation.

Return *the minimum total cost* of the cuts.

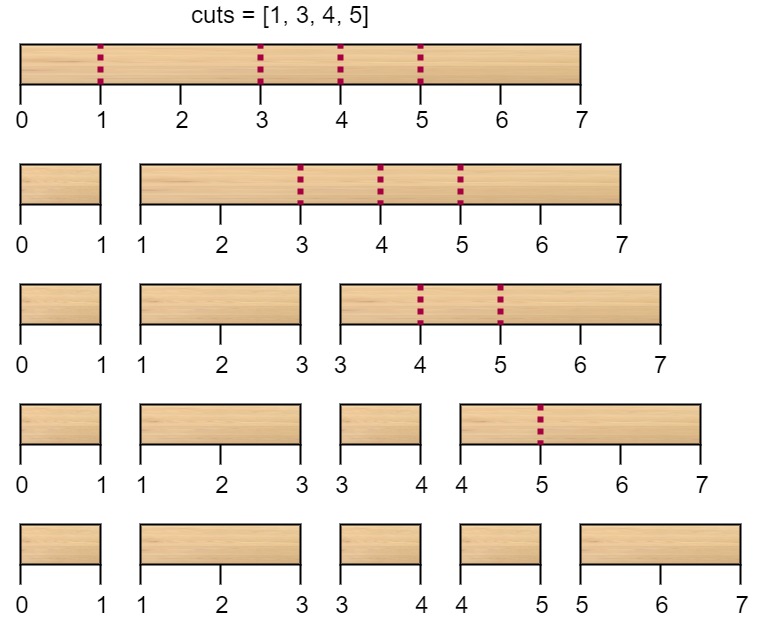
**Example 1:**



**Input:** n = 7, cuts = [1,3,4,5]

**Output:** 16

**Explanation:** Using cuts order = [1, 3, 4, 5] as in the input leads to the following scenario:



The first cut is done to a rod of length 7 so the cost is 7. The second cut is done to a rod of length 6 (i.e. the second part of the first cut), the third is done to a rod of length 4 and the last cut is to a rod of length 3. The total cost is 7 + 6 + 4 + 3 = 20.

Rearranging the cuts to be [3, 5, 1, 4] for example will lead to a scenario with total cost = 16 (as shown in the example photo 7 + 4 + 3 + 2 = 16).

**Example 2:**

**Input:** n = 9, cuts = [5,6,1,4,2]

**Output:** 22

**Explanation:** If you try the given cuts ordering the cost will be 25.

There are much ordering with total cost <= 25, for example, the order [4, 6, 5, 2, 1] has total cost = 22 which is the minimum possible.

**Constraints:**

* 2 <= n <= 10^6
* 1 <= cuts.length <= min(n - 1, 100)
* 1 <= cuts[i] <= n - 1
* All the integers in cuts array are **distinct**.

class Solution {

public:

    int minCost(int n, vector<int>& A) {

        A.push\_back(0);  A.push\_back(n);

        sort(A.begin(), A.end());

        int k = A.size();

        vector<vector<int>> dp(k, vector<int>(k, 0));

        for (int d = 2; d < k; ++d) {

            for (int i = 0; i < k - d; ++i) {

                dp[i][i + d] = 1e9;

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

                    dp[i][i + d] = min(dp[i][i + d],

dp[i][m] + dp[m][i + d] + A[i + d] - A[i]);

                }

            }

        }

        return dp[0][k - 1];

    }

};

### 1553. Minimum Number of Days to Eat N Oranges★

Hard

There are n oranges in the kitchen and you decided to eat some of these oranges every day as follows:

* Eat one orange.
* If the number of remaining oranges (n) is divisible by 2 then you can eat  n/2 oranges.
* If the number of remaining oranges (n) is divisible by 3 then you can eat  2\*(n/3) oranges.

You can only choose one of the actions per day.

Return the minimum number of days to eat n oranges.

**Example 1:**

**Input:** n = 10

**Output:** 4

**Explanation:** You have 10 oranges.

Day 1: Eat 1 orange, 10 - 1 = 9.

Day 2: Eat 6 oranges, 9 - 2\*(9/3) = 9 - 6 = 3. (Since 9 is divisible by 3)

Day 3: Eat 2 oranges, 3 - 2\*(3/3) = 3 - 2 = 1.

Day 4: Eat the last orange 1 - 1 = 0.

You need at least 4 days to eat the 10 oranges.

**Example 2:**

**Input:** n = 6

**Output:** 3

**Explanation:** You have 6 oranges.

Day 1: Eat 3 oranges, 6 - 6/2 = 6 - 3 = 3. (Since 6 is divisible by 2).

Day 2: Eat 2 oranges, 3 - 2\*(3/3) = 3 - 2 = 1. (Since 3 is divisible by 3)

Day 3: Eat the last orange 1 - 1 = 0.

You need at least 3 days to eat the 6 oranges.

**Example 3:**

**Input:** n = 1

**Output:** 1

**Example 4:**

**Input:** n = 56

**Output:** 6

**Constraints:**

* 1 <= n <= 2\*10^9

class Solution {

public:

    unordered\_map<int, int> dp;

    int minDays(int n) {

        if (n <= 1)  return n;

        if (!dp.count(n))

            dp[n] = 1 + min(n % 2 + minDays(n / 2),

n % 3 + minDays(n / 3));

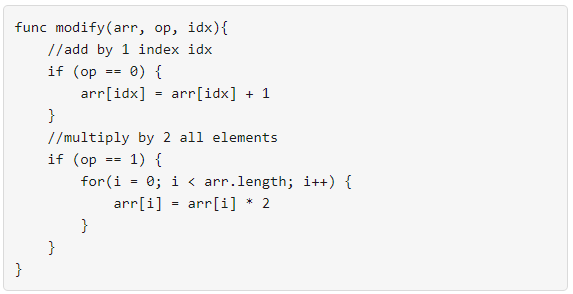
        return dp[n];

    }

};

**1558. Minimum Numbers of Function Calls to Make Target Array**★

Medium



Your task is to form an integer array nums from an initial array of zeros arr that is the same size as nums.

Return the minimum number of function calls to make nums from arr.

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

**Example 1:**

**Input:** nums = [1,5]

**Output:** 5

**Explanation:** Increment by 1 (second element): [0, 0] to get [0, 1] (1 operation).

Double all the elements: [0, 1] -> [0, 2] -> [0, 4] (2 operations).

Increment by 1 (both elements) [0, 4] -> [1, 4] -> **[1, 5]** (2 operations).

Total of operations: 1 + 2 + 2 = 5.

**Example 2:**

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

**Output:** 3

**Explanation:** Increment by 1 (both elements) [0, 0] -> [0, 1] -> [1, 1] (2 operations).

Double all the elements: [1, 1] -> **[2, 2]** (1 operation).

Total of operations: 2 + 1 = 3.

**Example 3:**

**Input:** nums = [4,2,5]

**Output:** 6

**Explanation:** (initial)[0,0,0] -> [1,0,0] -> [1,0,1] -> [2,0,2] -> [2,1,2] -> [4,2,4] -> **[4,2,5]**(nums).

**Example 4:**

**Input:** nums = [3,2,2,4]

**Output:** 7

**Example 5:**

**Input:** nums = [2,4,8,16]

**Output:** 8

**Constraints:**

* 1 <= nums.length <= 10^5
* 0 <= nums[i] <= 10^9

class Solution {

public:

    int minOperations(vector<int>& nums) {

        int n = nums.size(), res = 0, B = 0;

        for (auto i : nums) {

            int a = 0, b = 0;

            while (i) {

                if (i % 2 == 0) i /= 2, ++b;

                else --i, ++a;

            }

            res += a;

            B = max(B, b);

        }

        return res + B;

    }

};

### 1559. Detect Cycles in 2D Grid

Hard

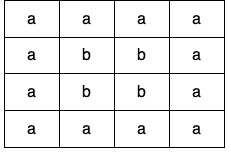
Given a 2D array of characters grid of size m x n, you need to find if there exists any cycle consisting of the **same value** in grid.

A cycle is a path of **length 4 or more** in the grid that starts and ends at the same cell. From a given cell, you can move to one of the cells adjacent to it - in one of the four directions (up, down, left, or right), if it has the **same value** of the current cell.

Also, you cannot move to the cell that you visited in your last move. For example, the cycle (1, 1) -> (1, 2) -> (1, 1) is invalid because from (1, 2) we visited (1, 1) which was the last visited cell.

Return true if any cycle of the same value exists in grid, otherwise, return false.

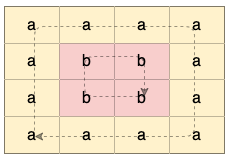
**Example 1:**

****

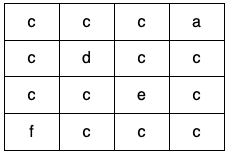
**Input:** grid = [["a","a","a","a"],["a","b","b","a"],["a","b","b","a"],["a","a","a","a"]]

**Output:** true

**Explanation:** There are two valid cycles shown in different colors in the image below:



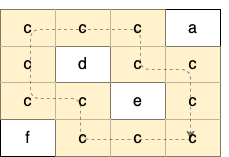
**Example 2:**

****

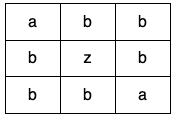
**Input:** grid = [["c","c","c","a"],["c","d","c","c"],["c","c","e","c"],["f","c","c","c"]]

**Output:** true

**Explanation:** There is only one valid cycle highlighted in the image below:



**Example 3:**

****

**Input:** grid = [["a","b","b"],["b","z","b"],["b","b","a"]]

**Output:** false

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m <= 500
* 1 <= n <= 500
* grid consists only of lowercase English letters.

class Solution {

public:

    vector<vector<int>> v;

    int n, m;

    bool containsCycle(vector<vector<char>>& grid) {

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

        v.resize(n, vector<int> (m, -1));

        for (int i = 0; i < n; ++i) {

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

                if (v[i][j] != -1) continue;

                if (dfs(i, j, i\*m + n, grid)) return true;

            }

        }

        return  false;

    }

    const vector<int> dx{-1,1,0,0};

    const vector<int> dy{0,0,-1,1};

    bool dfs(int i, int j, int cnt, vector<vector<char>>& grid) {

        v[i][j] = ++cnt;

        for (int k = 0; k < 4; ++k) {

            int x = i + dx[k], y = j + dy[k];

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

            if (grid[x][y] != grid[i][j]) continue;

            if (v[x][y] != -1) {

                if (cnt - v[x][y] >= 3) return true;

            }

            else {

                if (dfs(x, y, cnt, grid)) return true;

            }

        }

        return false;

    }

};

### 1562. Find Latest Group of Size M★★

Medium

Given an array arr that represents a permutation of numbers from 1 to n. You have a binary string of size n that initially has all its bits set to zero.

At each step i (assuming both the binary string and arr are 1-indexed) from 1 to n, the bit at position arr[i] is set to 1. You are given an integer m and you need to find the latest step at which there exists a group of ones of length m. A group of ones is a contiguous substring of 1s such that it cannot be extended in either direction.

Return *the latest step at which there exists a group of ones of length****exactly*** m. *If no such group exists, return* -1.

**Example 1:**

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

**Output:** 4

**Explanation:**

Step 1: "00100", groups: ["1"]

Step 2: "00101", groups: ["1", "1"]

Step 3: "10101", groups: ["1", "1", "1"]

Step 4: "11101", groups: ["111", "1"]

Step 5: "11111", groups: ["11111"]

The latest step at which there exists a group of size 1 is step 4.

**Example 2:**

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

**Output:** -1

**Explanation:**

Step 1: "00100", groups: ["1"]

Step 2: "10100", groups: ["1", "1"]

Step 3: "10101", groups: ["1", "1", "1"]

Step 4: "10111", groups: ["1", "111"]

Step 5: "11111", groups: ["11111"]

No group of size 2 exists during any step.

**Example 3:**

**Input:** arr = [1], m = 1

**Output:** 1

**Example 4:**

**Input:** arr = [2,1], m = 2

**Output:** 2

**Constraints:**

* n == arr.length
* 1 <= n <= 10^5
* 1 <= arr[i] <= n
* All integers in arr are **distinct**.
* 1 <= m <= arr.length

class Solution {

public:

    int findLatestStep(vector<int>& A, int m) {

        int res = -1, n = A.size();

        if (n == m) return n;

        vector<int> length(n + 2);

        for (int i = 0; i < n; ++i) {

            int a = A[i], left = length[a - 1], right = length[a + 1];

            length[a - left] = length[a + right] = left + right + 1;

            if (left == m || right == m)

                res = i;

        }

        return res;

    }

};

### 1563. Stone Game V

Hard

There are several stones **arranged in a row**, and each stone has an associated value which is an integer given in the array stoneValue.

In each round of the game, Alice divides the row into **two non-empty rows** (i.e. left row and right row), then Bob calculates the value of each row which is the sum of the values of all the stones in this row. Bob throws away the row which has the maximum value, and Alice's score increases by the value of the remaining row. If the value of the two rows are equal, Bob lets Alice decide which row will be thrown away. The next round starts with the remaining row.

The game ends when there is only **one stone remaining**. Alice's is initially **zero**.

Return *the maximum score that Alice can obtain*.

**Example 1:**

**Input:** stoneValue = [6,2,3,4,5,5]

**Output:** 18

**Explanation:** In the first round, Alice divides the row to [6,2,3], [4,5,5]. The left row has the value 11 and the right row has value 14. Bob throws away the right row and Alice's score is now 11.

In the second round Alice divides the row to [6], [2,3]. This time Bob throws away the left row and Alice's score becomes 16 (11 + 5).

The last round Alice has only one choice to divide the row which is [2], [3]. Bob throws away the right row and Alice's score is now 18 (16 + 2). The game ends because only one stone is remaining in the row.

**Example 2:**

**Input:** stoneValue = [7,7,7,7,7,7,7]

**Output:** 28

**Example 3:**

**Input:** stoneValue = [4]

**Output:** 0

**Constraints:**

* 1 <= stoneValue.length <= 500
* 1 <= stoneValue[i] <= 10^6

class Solution {

public:

    int dp[505][505] = {0};

    vector<int> pre{0};

    int stoneGameV(vector<int>& v) {

        int n = v.size(), res = 0;

        for (auto i : v) pre.push\_back(pre.back() + i);

        return solve(1, v.size(), v);

    }

    int solve(int i, int j, vector<int>& v) {

        if (i > j) return 0;

        else if (i == j) return 0;

        else if (dp[i][j] != 0) return dp[i][j];

        int ret = 0;

        for (int k = i; k < j; ++k) {

            int a = pre[k] - pre[i-1];

            int b = pre[j] - pre[k];

            if (a <= b) ret = max(ret, solve(i, k, v) + a);

            if (a >= b) ret = max(ret, solve(k+1, j, v) + b);

        }

        return dp[i][j] = ret;

    }

};

//n\*n\*logn

class Solution {

    public:

    int getsum(vector<int>& sum, int l, int r) {

        if (l > r) return 0;

        if (l == 0) return sum[r];

        return sum[r] - sum[l - 1];

    }

    int stoneGameV(vector<int>& stoneValue) {

        int n = stoneValue.size();

        vector<int> sum(n, 0);

        for (int i = 0; i < n; ++i) {

            if (i > 0) sum[i] += sum[i - 1];

            sum[i] += stoneValue[i];

        }

        vector<vector<int>> f=vector<vector<int>>(n,vector<int>(n,0));

        vector<vector<int>> lf=vector<vector<int>>(n,vector<int>(n,0));

        vector<vector<int>> rf=vector<vector<int>>(n,vector<int>(n,0));

        for (int i = 0; i < n; ++i) {

            lf[i][i] = rf[i][i] = stoneValue[i];

        }

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

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

            int segsum = getsum(sum, i, j);

            int l = i - 1, r = j;

            while (l < r - 1) {

                int mid = (l + r) / 2;

                int left = getsum(sum, i, mid);

                if (left \* 2 <= segsum) l = mid;

                else r = mid;

                if (l >= i) f[i][j] = max(f[i][j], lf[i][l]);

                int rst = l;

                if (getsum(sum, i, l) \* 2 < segsum) rst += 2;

                else rst += 1;

                if (rst <= j)

                    f[i][j] = max(f[i][j], rf[rst][j]);

                lf[i][j] = max(lf[i][max(i, j - 1)], f[i][j] + segsum);

                rf[i][j] = max(rf[max(0, i + 1)][j], f[i][j] + segsum);

            }

        }

        return f[0][n - 1];

    }

};

### 1568. Minimum Number of Days to Disconnect Island

Hard

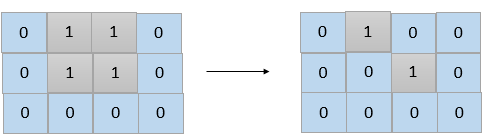
Given a 2D grid consisting of 1s (land) and 0s (water).  An *island* is a maximal 4-directionally (horizontal or vertical) connected group of 1s.

The grid is said to be **connected** if we have **exactly one island**, otherwise is said **disconnected**.

In one day, we are allowed to change **any**single land cell (1) into a water cell (0).

Return *the minimum number of days* to disconnect the grid.

**Example 1:**

****

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

**Output:** 2

**Explanation:** We need at least 2 days to get a disconnected grid.

Change land grid[1][1] and grid[0][2] to water and get 2 disconnected island.

**Example 2:**

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

**Output:** 2

**Explanation:** Grid of full water is also disconnected ([[1,1]] -> [[0,0]]), 0 islands.

**Example 3:**

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

**Output:** 0

**Example 4:**

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

  [1,1,1,1,1],

  [1,1,0,1,1],

  [1,1,0,1,1]]

**Output:** 1

**Example 5:**

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

  [1,1,1,1,1],

  [1,1,0,1,1],

  [1,1,1,1,1]]

**Output:** 2

**Constraints:**

* 1 <= grid.length, grid[i].length <= 30
* grid[i][j] is 0 or 1.

class Solution {

public:

    vector<int> dx = {-1,1,0,0}, dy = {0,0,-1,1};

    int n, m, p = -1;

    vector<pair<int, int>> v;

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

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

        int t = 0, cnt, cnt0;

        for (int i = 0; i < n; ++i) {

            for (int j = 0; j < m; ++j) if (grid[i][j] == 1) {

                ++t;

                cnt = dfs(i, j, grid, -1);

            }

        }

        if (t != 1) return 0;

        if (cnt == 1) return 1;

        for (int i = 0; i < n; ++i) {

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

                grid[i][j] = 0;

                int pp = p == -1?1:-1;

                if (i == v[0].first && j == v[0].second)

                    cnt0 = dfs(v[1].first, v[1].second, grid, pp);

                else cnt0 = dfs(v[0].first, v[0].second, grid, pp);

                if (cnt0 != cnt-1) return 1;

                grid[i][j] = p = pp;

            }

        }

        return 2;

    }

    int dfs(int i, int j, vector<vector<int>>& grid, int p) {

        grid[i][j] = p;

        if (v.size() < 2) v.emplace\_back(i, j);

        int res = 1;

        for (int k = 0; k < 4; ++k) {

            int x = i + dx[k], y = j + dy[k];

            if (x < 0 || y < 0 || x >= n || y >= m || grid[x][y] != (p==1?-1:1)) continue;

            res += dfs(x, y, grid, p);

        }

        return res;

    }

};

### 1569. Number of Ways to Reorder Array to Get Same BST★★

Hard

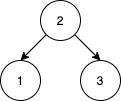
Given an array nums that represents a permutation of integers from 1 to n. We are going to construct a binary search tree (BST) by inserting the elements of nums in order into an initially empty BST. Find the number of different ways to reorder nums so that the constructed BST is identical to that formed from the original array nums.

For example, given nums = [2,1,3], we will have 2 as the root, 1 as a left child, and 3 as a right child. The array [2,3,1] also yields the same BST but [3,2,1] yields a different BST.

Return *the number of ways to reorder* nums *such that the BST formed is identical to the original BST formed from* nums.

Since the answer may be very large, **return it modulo**10^9 + 7.

**Example 1:**

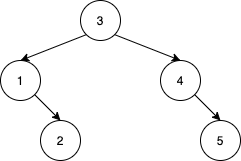


**Input:** nums = [2,1,3]

**Output:** 1

**Explanation:** We can reorder nums to be [2,3,1] which will yield the same BST. There are no other ways to reorder nums which will yield the same BST.

**Example 2:**

****

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

**Output:** 5

**Explanation:** The following 5 arrays will yield the same BST:

[3,1,2,4,5]

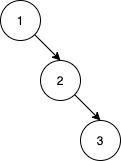
[3,1,4,2,5]

[3,1,4,5,2]

[3,4,1,2,5]

[3,4,1,5,2]

**Example 3:**

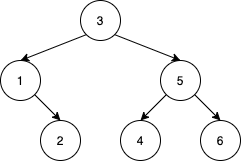
****

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

**Output:** 0

**Explanation:** There are no other orderings of nums that will yield the same BST.

**Example 4:**

****

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

**Output:** 19

**Example 5:**

**Input:** nums = [9,4,2,1,3,6,5,7,8,14,11,10,12,13,16,15,17,18]

**Output:** 216212978

**Explanation:** The number of ways to reorder nums to get the same BST is 3216212999. Taking this number modulo 10^9 + 7 gives 216212978.

**Constraints:**

* 1 <= nums.length <= 1000
* 1 <= nums[i] <= nums.length
* All integers in nums are **distinct**.

class Solution {

public:

    using ll = long long;

    const ll mod = 1e9 + 7;

    ll dp[1002][1002] = {0};

    int helper(int i, int j) {

        if (i == 0 && j == 0) return 0;

        if (i == 0 || j == 0) return 1;

        if (dp[i][j]) return dp[i][j];

        return dp[i][j] = (helper(i-1, j) + helper(i, j-1)) % mod;

    }

    int numOfWays(vector<int>& nums) {

        return static\_cast<int> (dfs(nums) - 1);

    }

    ll dfs(vector<int>& nums) {

        int i = 0, j = nums.size()-1;

        if (i >= j) return 1;

        vector<int> v[2];

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

            int t = nums[k] < nums[i] ? 1 : 0;

            v[t].push\_back(nums[k]);

        }

        ll a = dfs(v[0]), b = dfs(v[1]);

        return (a\*b % mod) \* helper(v[0].size(), v[1].size()) % mod;

    }

};

### 1574. Shortest Subarray to be Removed to Make Array Sorted★★

Medium

Given an integer array arr, remove a subarray (can be empty) from arr such that the remaining elements in arr are **non-decreasing**.

A subarray is a contiguous subsequence of the array.

Return *the length of the shortest subarray to remove*.

**Example 1:**

**Input:** arr = [1,2,3,10,4,2,3,5]

**Output:** 3

**Explanation:** The shortest subarray we can remove is [10,4,2] of length 3. The remaining elements after that will be [1,2,3,3,5] which are sorted.

Another correct solution is to remove the subarray [3,10,4].

**Example 2:**

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

**Output:** 4

**Explanation:** Since the array is strictly decreasing, we can only keep a single element. Therefore we need to remove a subarray of length 4, either [5,4,3,2] or [4,3,2,1].

**Example 3:**

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

**Output:** 0

**Explanation:** The array is already non-decreasing. We do not need to remove any elements.

**Example 4:**

**Input:** arr = [1]

**Output:** 0

**Constraints:**

* 1 <= arr.length <= 10^5
* 0 <= arr[i] <= 10^9

class Solution {

public:

    int findLengthOfShortestSubarray(vector<int>& n) {

        int sz = n.size(), r = sz - 1;

        for (; r > 0 && n[r - 1] <= n[r]; --r) ;

        auto res = r;

        for (int l = 0; l < r && (l == 0 || n[l - 1] <= n[l]); ++l) {

            while (r < sz && n[r] < n[l])

                ++r;

            res = min(res, r - l - 1);

        }

        return res;

    }

};

### 1575. Count All Possible Routes

Hard

You are given an array of **distinct** positive integers locations where locations[i] represents the position of city i. You are also given integers start, finish and fuel representing the starting city, ending city, and the initial amount of fuel you have, respectively.

At each step, if you are at city i, you can pick any city j such that j != i and 0 <= j < locations.length and move to city j. Moving from city i to city j reduces the amount of fuel you have by |locations[i] - locations[j]|. Please notice that |x| denotes the absolute value of x.

Notice that fuel **cannot** become negative at any point in time, and that you are **allowed** to visit any city more than once (including start and finish).

Return *the count of all possible routes from*start *to* finish.

Since the answer may be too large, return it modulo 10^9 + 7.

**Example 1:**

**Input:** locations = [2,3,6,8,4], start = 1, finish = 3, fuel = 5

**Output:** 4

**Explanation:** The following are all possible routes, each uses 5 units of fuel:

1 -> 3

1 -> 2 -> 3

1 -> 4 -> 3

1 -> 4 -> 2 -> 3

**Example 2:**

**Input:** locations = [4,3,1], start = 1, finish = 0, fuel = 6

**Output:** 5

**Explanation:** The following are all possible routes:

1 -> 0, used fuel = 1

1 -> 2 -> 0, used fuel = 5

1 -> 2 -> 1 -> 0, used fuel = 5

1 -> 0 -> 1 -> 0, used fuel = 3

1 -> 0 -> 1 -> 0 -> 1 -> 0, used fuel = 5

**Example 3:**

**Input:** locations = [5,2,1], start = 0, finish = 2, fuel = 3

**Output:** 0

**Explanation:** It's impossible to get from 0 to 2 using only 3 units of fuel since the shortest route needs 4 units of fuel.

**Example 4:**

**Input:** locations = [2,1,5], start = 0, finish = 0, fuel = 3

**Output:** 2

**Explanation:** There are two possible routes, 0 and 0 -> 1 -> 0.

**Example 5:**

**Input:** locations = [1,2,3], start = 0, finish = 2, fuel = 40

**Output:** 615088286

**Explanation:** The total number of possible routes is 2615088300. Taking this number modulo 10^9 + 7 gives us 615088286.

**Constraints:**

* 2 <= locations.length <= 100
* 1 <= locations[i] <= 10^9
* All integers in locations are **distinct**.
* 0 <= start, finish < locations.length
* 1 <= fuel <= 200

class Solution {

    const int N=1000000007;

public:

    int countRoutes(vector<int>& locations, int start, int finish, int fuel) {

        start=locations[start];

        finish=locations[finish];

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

        start=find(locations.begin(),locations.end(),start)

-locations.begin();

        finish=find(locations.begin(),locations.end(),finish)

-locations.begin();

        int size=locations.size();

        int j,f,d;

        vector<vector<int>> Ldp(size,vector<int>(fuel+1,0));

        vector<vector<int>> Rdp(size,vector<int>(fuel+1,0));

        for(f=1;f<=fuel;f++){

            Ldp[size-1][f]=0;

            Rdp[0][f]=0;

            for(j=0;j<size-1;j++){

                d=locations[j+1]-locations[j];

                if(f>d)

                    Ldp[j][f]=(Rdp[j+1][f-d]+2\*Ldp[j+1][f-d]%N)%N;

                else

                    Ldp[j][f]=1\*(f==d)\*(start==(j+1));

            }

            for(j=1;j<size;j++){

                d=locations[j]-locations[j-1];

                if(f>d)

                    Rdp[j][f]=(Ldp[j-1][f-d]+2\*Rdp[j-1][f-d]%N)%N;

                else

                    Rdp[j][f]=1\*(f==d)\*(start==(j-1));

            }

        }

        int ans=1\*(start==finish);

        for(j=1;j<=fuel;j++)

ans=((ans+Ldp[finish][j])%N+Rdp[finish][j])%N;

        return ans;

    }

};

### 1579. Remove Max Number of Edges to Keep Graph Fully Traversable★★

Hard

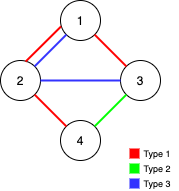
Alice and Bob have an undirected graph of n nodes and 3 types of edges:

* Type 1: Can be traversed by Alice only.
* Type 2: Can be traversed by Bob only.
* Type 3: Can by traversed by both Alice and Bob.

Given an array edges where edges[i] = [typei, ui, vi] represents a bidirectional edge of type typei between nodes ui and vi, find the maximum number of edges you can remove so that after removing the edges, the graph can still be fully traversed by both Alice and Bob. The graph is fully traversed by Alice and Bob if starting from any node, they can reach all other nodes.

Return *the maximum number of edges you can remove, or return* -1 *if it's impossible for the graph to be fully traversed by Alice and Bob.*

**Example 1:**

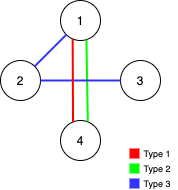
****

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

**Output:** 2

**Explanation:** If we remove the 2 edges [1,1,2] and [1,1,3]. The graph will still be fully traversable by Alice and Bob. Removing any additional edge will not make it so. So the maximum number of edges we can remove is 2.

**Example 2:**

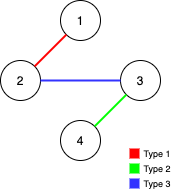
****

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

**Output:** 0

**Explanation:** Notice that removing any edge will not make the graph fully traversable by Alice and Bob.

**Example 3:**

****

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

**Output:** -1

**Explanation:** In the current graph, Alice cannot reach node 4 from the other nodes. Likewise, Bob cannot reach 1. Therefore it's impossible to make the graph fully traversable.

**Constraints:**

* 1 <= n <= 10^5
* 1 <= edges.length <= min(10^5, 3 \* n \* (n-1) / 2)
* edges[i].length == 3
* 1 <= edges[i][0] <= 3
* 1 <= edges[i][1] < edges[i][2] <= n
* All tuples (typei, ui, vi) are distinct.

class Solution {

public:

    int find(vector<int> &ds, int i) {

        return ds[i] < 0 ? i : ds[i] = find(ds, ds[i]);

    }

    int maxNumEdgesToRemove(int n, vector<vector<int>>& edges) {

        vector<int> ds\_both(n + 1, -1);

        int used = 0;

        for (int type = 3; type > 0; --type) {

            auto ds\_one = ds\_both;

            auto &ds = type == 3 ? ds\_both : ds\_one;

            for (auto &e : edges) {

                if (e[0] == type) {

                    int i = find(ds, e[1]), j = find(ds, e[2]);

                    if (i != j) {

                        ++used;

                        ds[i] += ds[j];

                        ds[j] = i;

                    }

                }

            }

            if (type != 3 && ds[find(ds, 1)] != -n)

                return -1;

        }

        return edges.size() - used;

    }

};

### 1589. Maximum Sum Obtained of Any Permutation★★

Medium

We have an array of integers, nums, and an array of requests where requests[i] = [starti, endi]. The ith request asks for the sum of nums[starti] + nums[starti + 1] + ... + nums[endi - 1] + nums[endi]. Both starti and endi are *0-indexed*.

Return *the maximum total sum of all requests****among all permutations****of* nums.

Since the answer may be too large, return it **modulo** 109 + 7.

**Example 1:**

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

**Output:** 19

**Explanation:** One permutation of nums is [2,1,3,4,5] with the following result:

requests[0] -> nums[1] + nums[2] + nums[3] = 1 + 3 + 4 = 8

requests[1] -> nums[0] + nums[1] = 2 + 1 = 3

Total sum: 8 + 3 = 11.

A permutation with a higher total sum is [3,5,4,2,1] with the following result:

requests[0] -> nums[1] + nums[2] + nums[3] = 5 + 4 + 2 = 11

requests[1] -> nums[0] + nums[1] = 3 + 5 = 8

Total sum: 11 + 8 = 19, which is the best that you can do.

**Example 2:**

**Input:** nums = [1,2,3,4,5,6], requests = [[0,1]]

**Output:** 11

**Explanation:** A permutation with the max total sum is [6,5,4,3,2,1] with request sums [11].

**Example 3:**

**Input:** nums = [1,2,3,4,5,10], requests = [[0,2],[1,3],[1,1]]

**Output:** 47

**Explanation:** A permutation with the max total sum is [4,10,5,3,2,1] with request sums [19,18,10].

**Constraints:**

* n == nums.length
* 1 <= n <= 105
* 0 <= nums[i] <= 105
* 1 <= requests.length <= 105
* requests[i].length == 2
* 0 <= starti <= endi < n

class Solution {

public:

    int maxSumRangeQuery(vector<int>& A, vector<vector<int>>& req) {

        int res = 0, mod = 1e9 + 7, n = A.size();

        vector<int> count(A.size());

        for (auto &r: req) {

            count[r[0]] += 1;

            if (r[1] + 1 < n)

                count[r[1] + 1] -= 1;

        }

        for (int i = 1; i < n; ++i)

            count[i] += count[i - 1];

        sort(begin(count), end(count));

        sort(begin(A), end(A));

        for (int i = 0; i < n; ++i)

            res = (res + A[i] \* count[i]) % mod;

        return res;

    }

};

### 1591. Strange Printer II★★

Hard

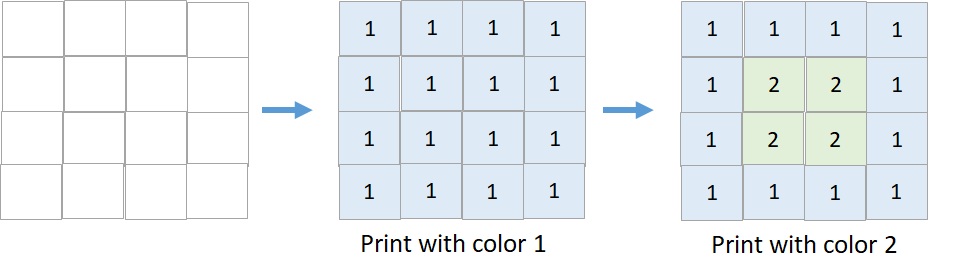
There is a strange printer with the following two special requirements:

* On each turn, the printer will print a solid rectangular pattern of a single color on the grid. This will cover up the existing colors in the rectangle.
* Once the printer has used a color for the above operation, **the same color cannot be used again**.

You are given a m x n matrix targetGrid, where targetGrid[row][col] is the color in the position (row, col) of the grid.

Return true*if it is possible to print the matrix*targetGrid*, otherwise, return*false.

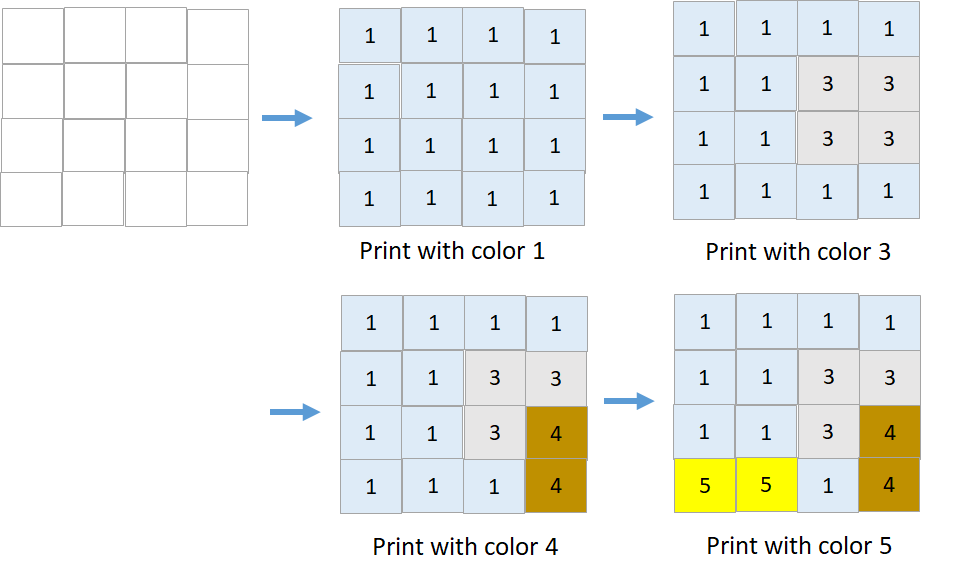
**Example 1:**



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

**Output:** true

**Example 2:**



**Input:** targetGrid = [[1,1,1,1],[1,1,3,3],[1,1,3,4],[5,5,1,4]]

**Output:** true

**Example 3:**

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

**Output:** false

**Explanation:** It is impossible to form targetGrid because it is not allowed to print the same color in different turns.

**Example 4:**

**Input:** targetGrid = [[1,1,1],[3,1,3]]

**Output:** false

**Constraints:**

* m == targetGrid.length
* n == targetGrid[i].length
* 1 <= m, n <= 60
* 1 <= targetGrid[row][col] <= 60

class Solution {

private:

    vector<int> vis; // visitation state for each color

    int n, m;

    bool dfs(vector<vector<int>>& targetGrid, int color) {

        if(vis[color] == -1) return true;

        if(vis[color] == 1) return false;

        // find bounds

        int l = INT\_MAX, r = INT\_MIN, u = INT\_MAX, d = INT\_MIN;

        for(int i = 0; i < n; i ++)

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

                if(targetGrid[i][j] == color) l = min(l, j), r = max(r, j), u = min(u, i), d = max(d, i);

        if(l == INT\_MAX) return true; // none of this color

        vis[color] = 1;

        // now, start exploring inside my maximal rectangle

        for(int i = u; i <= d; i ++)

            for(int j = l; j <= r; j ++)

                if(targetGrid[i][j] != color && !dfs(targetGrid, targetGrid[i][j])) return false;

        vis[color] = -1;

        return true;

    }

public:

    bool isPrintable(vector<vector<int>>& targetGrid) {

        n = targetGrid.size();

        m = targetGrid[0].size();

        vis = vector<int> (61, 0); // 0: didnt open, 1: open, -1: closed

        for(int color = 1; color <= 60; color ++) if(!dfs(targetGrid, color)) return false;

        return true;

    }

};

### 1595. Minimum Cost to Connect Two Groups of Points

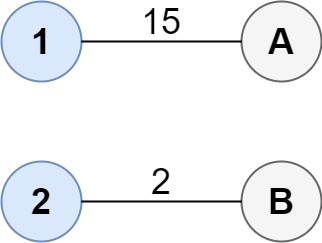
Hard

You are given two groups of points where the first group has size1 points, the second group has size2 points, and size1 >= size2.

The cost of the connection between any two points are given in an size1 x size2 matrix where cost[i][j] is the cost of connecting point i of the first group and point j of the second group. The groups are connected if **each point in both groups is connected to one or more points in the opposite group**. In other words, each point in the first group must be connected to at least one point in the second group, and each point in the second group must be connected to at least one point in the first group.

Return *the minimum cost it takes to connect the two groups*.

**Example 1:**



**Input:** cost = [[15, 96], [36, 2]]

**Output:** 17

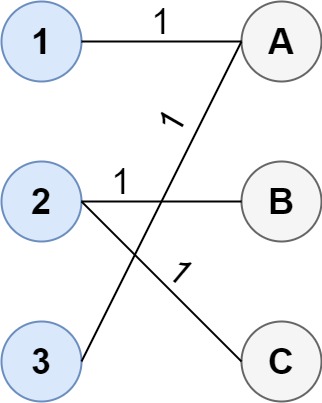
**Explanation**: The optimal way of connecting the groups is:

1--A

2--B

This results in a total cost of 17.

**Example 2:**



**Input:** cost = [[1, 3, 5], [4, 1, 1], [1, 5, 3]]

**Output:** 4

**Explanation**: The optimal way of connecting the groups is:

1--A

2--B

2--C

3--A

This results in a total cost of 4.

Note that there are multiple points connected to point 2 in the first group and point A in the second group. This does not matter as there is no limit to the number of points that can be connected. We only care about the minimum total cost.

**Example 3:**

**Input:** cost = [[2, 5, 1], [3, 4, 7], [8, 1, 2], [6, 2, 4], [3, 8, 8]]

**Output:** 10

**Constraints:**

* size1 == cost.length
* size2 == cost[i].length
* 1 <= size1, size2 <= 12
* size1 >= size2
* 0 <= cost[i][j] <= 100

class Solution {

public:

    int connectTwoGroups(vector<vector<int>>& cost) {

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

        vector<vector<int>> dp(2, vector<int>(1 << m, 1e9));

        dp[0][0] = 0;

        for(int i = 0; i < n; ++i) {

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

            for (int mask = 0; mask < 1<<m; ++mask) {

                cur[mask] = 1e9;

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

                    cur[mask] = min({cur[mask],

cost[i][j] + pre[mask & ~(1 << j)],

cost[i][j] + cur[mask & ~(1 << j)]});

             }

            }

            k ^= 1;

        }

        return dp[k].back();

    }

};

### 1601. Maximum Number of Achievable Transfer Requests

Hard

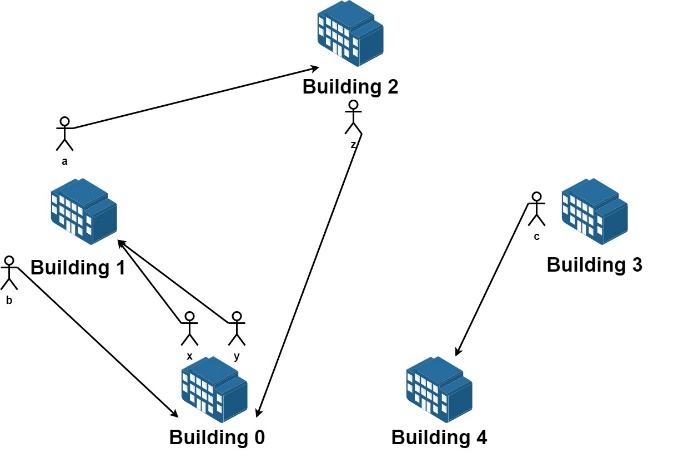
We have n buildings numbered from 0 to n - 1. Each building has a number of employees. It's transfer season, and some employees want to change the building they reside in.

You are given an array requests where requests[i] = [fromi, toi] represents an employee's request to transfer from building fromi to building toi.

**All buildings are full**, so a list of requests is achievable only if for each building, the **net change in employee transfers is zero**. This means the number of employees **leaving** is **equal** to the number of employees **moving in**. For example if n = 3 and two employees are leaving building 0, one is leaving building 1, and one is leaving building 2, there should be two employees moving to building 0, one employee moving to building 1, and one employee moving to building 2.

Return *the maximum number of achievable requests*.

**Example 1:**



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

**Output:** 5

**Explantion:** Let's see the requests:

From building 0 we have employees x and y and both want to move to building 1.

From building 1 we have employees a and b and they want to move to buildings 2 and 0 respectively.

From building 2 we have employee z and they want to move to building 0.

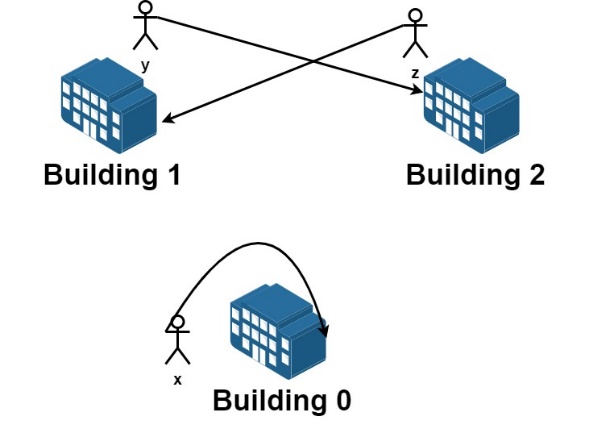
From building 3 we have employee c and they want to move to building 4.

From building 4 we don't have any requests.

We can achieve the requests of users x and b by swapping their places.

We can achieve the requests of users y, a and z by swapping the places in the 3 buildings.

**Example 2:**



**Input:** n = 3, requests = [[0,0],[1,2],[2,1]]

**Output:** 3

**Explantion:** Let's see the requests:

From building 0 we have employee x and they want to stay in the same building 0.

From building 1 we have employee y and they want to move to building 2.

From building 2 we have employee z and they want to move to building 1.

We can achieve all the requests.

**Example 3:**

**Input:** n = 4, requests = [[0,3],[3,1],[1,2],[2,0]]

**Output:** 4

**Constraints:**

* 1 <= n <= 20
* 1 <= requests.length <= 16
* requests[i].length == 2
* 0 <= fromi, toi < n

class Solution {

public:

    int maximumRequests(int N, vector<vector<int>>& a) {

        int n = a.size();

        vector<int> ind(N, 0);

        int res = 0;

        for (int mask = 0; mask < 1<<n; ++mask) {

            fill(ind.begin(), ind.end(), 0);

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

                if (mask>>j & 1) {

                    ++ind[a[j][1]];

                    --ind[a[j][0]];

                }

            }

            bool ok = true;

            for (int x : ind) {

                if (x > 0) {

                    ok = false;

                    break;

                }

            }

            if (ok) {

                res = max(res, \_\_builtin\_popcount(mask));

            }

        }

        return res;

    }

};

### 1610. Maximum Number of Visible Points

Hard

You are given an array points, an integer angle, and your location, where location = [posx, posy] and points[i] = [xi, yi] both denote **integral coordinates** on the X-Y plane.

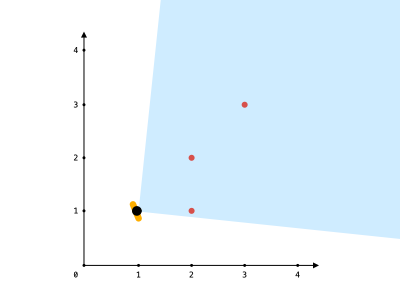
Initially, you are facing directly east from your position. You **cannot move** from your position, but you can **rotate**. In other words, posx and posy cannot be changed. Your field of view in **degrees** is represented by angle, determining how wide you can see from any given view direction. Let d be the amount in degrees that you rotate counterclockwise. Then, your field of view is the **inclusive** range of angles [d - angle/2, d + angle/2].

You can **see** some set of points if, for each point, the **angle** formed by the point, your position, and the immediate east direction from your position is **in your field of view**.

There can be multiple points at one coordinate. There may be points at your location, and you can always see these points regardless of your rotation. Points do not obstruct your vision to other points.

Return *the maximum number of points you can see*.

**Example 1:**



**Input:** points = [[2,1],[2,2],[3,3]], angle = 90, location = [1,1]

**Output:** 3

**Explanation:** The shaded region represents your field of view. All points can be made visible in your field of view, including [3,3] even though [2,2] is in front and in the same line of sight.

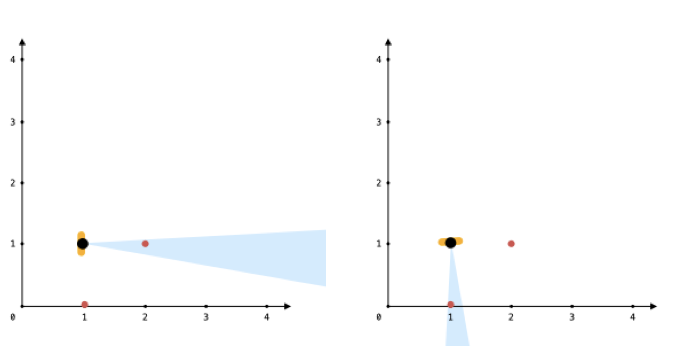
**Example 2:**

**Input:** points = [[2,1],[2,2],[3,4],[1,1]], angle = 90, location = [1,1]

**Output:** 4

**Explanation:** All points can be made visible in your field of view, including the one at your location.

**Example 3:**



**Input:** points = [[1,0],[2,1]], angle = 13, location = [1,1]

**Output:** 1

**Explanation:** You can only see one of the two points, as shown above.

**Constraints:**

* 1 <= points.length <= 105
* points[i].length == 2
* location.length == 2
* 0 <= angle < 360
* 0 <= posx, posy, xi, yi <= 109

class Solution {

public:

    int visiblePoints(vector<vector<int>>& points, int angle, vector<int>& loc) {

        vector<double> angs;

        int max\_points = 0, overlap = 0;

        for (auto &p : points) {

            if (p[1] == loc[1] && p[0] == loc[0])

                ++overlap;

            else

               angs.push\_back(atan2(p[1]-loc[1],p[0]-loc[0])\*180/M\_PI);

        }

        sort(begin(angs), end(angs));

        int sz = angs.size();

        for (int i = 0, j = 0; i < sz \* 2; ++i) {

            while (360 \* (i >= sz) + angs[i % sz]

- 360 \* (j >= sz) - angs[j % sz] > angle)

                ++j;

            max\_points = max(max\_points, i - j + 1);

        }

        return max\_points + overlap;

    }

};

### 1617. Count Subtrees With Max Distance Between Cities

Hard

There are n cities numbered from 1 to n. You are given an array edges of size n-1, where edges[i] = [ui, vi] represents a bidirectional edge between cities uiand vi. There exists a unique path between each pair of cities. In other words, the cities form a **tree**.

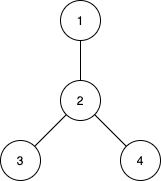
A **subtree** is a subset of cities where every city is reachable from every other city in the subset, where the path between each pair passes through only the cities from the subset. Two subtrees are different if there is a city in one subtree that is not present in the other.

For each d from 1 to n-1, find the number of subtrees in which the **maximum distance** between any two cities in the subtree is equal to d.

Return *an array of size* n-1 *where the*dth*element****(1-indexed)****is the number of subtrees in which the****maximum distance****between any two cities is equal to*d.

**Notice** that the **distance** between the two cities is the number of edges in the path between them.

**Example 1:**

****

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

**Output:** [3,4,0]

**Explanation:**

The subtrees with subsets {1,2}, {2,3} and {2,4} have a max distance of 1.

The subtrees with subsets {1,2,3}, {1,2,4}, {2,3,4} and {1,2,3,4} have a max distance of 2.

No subtree has two nodes where the max distance between them is 3.

**Example 2:**

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

**Output:** [1]

**Example 3:**

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

**Output:** [2,1]

**Constraints:**

* 2 <= n <= 15
* edges.length == n-1
* edges[i].length == 2
* 1 <= ui, vi <= n
* All pairs (ui, vi) are distinct.

class Solution {

public:

    vector<int> countSubgraphsForEachDiameter(int n, vector<vector<int>>& edges) {

        vector<vector<int>> dis(n, vector<int> (n, 1e9));

        vector<int> neig(n, 0);

        for (auto &v : edges) {

            dis[v[0]-1][v[1]-1] = dis[v[1]-1][v[0]-1] = 1;

            neig[v[0]-1] |= 1 << (v[1]-1);

            neig[v[1]-1] |= 1 << (v[0]-1);

        }

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

        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]);

                }

            }

        }

        vector<bool> vis(1<<n, false);

        vector<int> res(n-1, 0);

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

            if (\_\_builtin\_popcount(k) == 1) vis[k] = true;

            if (!vis[k]) continue;

            int t = 0;

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

                if (k & (1 << i)) {

                    t |= neig[i];

                }

            }

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

                if ((t & (1 << i)) && (k & (1 << i)) == 0) {

                    vis[k | (1 << i)] = true;

                }

            }

            int d = 0;

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

                if ((k & (1 << i))) {

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

                        if ((k & (1 << j))) {

                            d = max(d, dis[i][j]);

                        }

                    }

                }

            }

            if (d == 0) continue;

            ++res[d-1];

        }

        return res;

    }

};

class Solution {

public:

    vector<int> countSubgraphsForEachDiameter(int n, vector<vector<int>>& edges) {

        int INF = n; // Since cities form a tree so maximum distance between 2 cities always < n

        vector<vector<int>> dist(n, vector<int>(n, INF));

        for (auto& e : edges)

            dist[e[0]-1][e[1]-1] = dist[e[1]-1][e[0]-1] = 1;

        for (int k = 0; k < n; k++)

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

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

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

        vector<int> ans(n - 1, 0);

        for (int state = 0; state < (1 << n); state++) {

            int d = maxDistance(state, dist, n);

            if (d > 0) ans[d - 1] += 1;

        }

        return ans;

    }

    int maxDistance(int state, vector<vector<int>>& dist, int n) {

        int cntEdge = 0, cntCity = 0, maxDist = 0;

        for (int i = 0; i < n; i++) {

            if (((state >> i) & 1) == 0) continue;

// Skip if city `i` not in our subset

            cntCity += 1;

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

                if (((state >> j) & 1) == 0) continue;

// Skip if city `j` not in our subset

                cntEdge += dist[i][j] == 1;

                maxDist = max(maxDist, dist[i][j]);

            }

        }

        if (cntEdge != cntCity - 1) return 0;

// Subset form an invalid subtree!

        return maxDist;

    }

};

class Solution {

public:

    vector<int> countSubgraphsForEachDiameter(int n, vector<vector<int>>& edges) {

        vector<vector<int>> graph(n);

        for (auto& e : edges) {

            graph[e[0]-1].push\_back(e[1]-1);

            graph[e[1]-1].push\_back(e[0]-1);

        }

        vector<int> ans(n - 1, 0);

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

            int d = maxDistance(state, graph, n);

            if (d > 0) ans[d - 1] += 1;

        }

        return ans;

    }

    int maxDistance(int state, vector<vector<int>>& graph, int n) {

        int anyNode = 0, cntCity = 0;

        for (int i = 0; i < n; i++) if ((state >> i) & 1) {

            anyNode = i;

            cntCity += 1;

        }

        auto [farthestNode,\_,visitedSize]=bfs(anyNode,state,graph,n);

        if (visitedSize < cntCity) return 0;

// Can't visit all nodes of the tree -> Invalid tree

        auto [\_ig1, dist, \_ig2] = bfs(farthestNode, state, graph, n);

        return dist;

    }

    tuple<int, int, int> bfs(int src, int state, vector<vector<int>>& graph, int n) {

        unordered\_set<int> visited{src};

        queue<pair<int,int>> q;

        q.push({src, 0});

        int farthestNode = -1, farthestDist = 0;

        while(!q.empty()){

            auto [u, d] = q.front(); q.pop();

            farthestNode = u, farthestDist = d;

            for (int v : graph[u]){

                if (visited.find(v) == visited.end()&&(state >> v)& 1){

                    q.push({v, d + 1});

                    visited.insert(v);

                }

            }

        }

        return make\_tuple(farthestNode, farthestDist, visited.size());

    }

};

### 1627. Graph Connectivity With Threshold

Hard

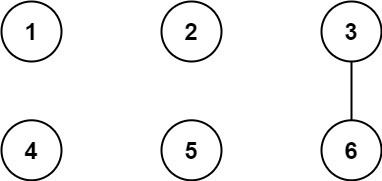
We have n cities labeled from 1 to n. Two different cities with labels x and y are directly connected by a bidirectional road if and only if x and y share a common divisor **strictly greater** than some threshold. More formally, cities with labels xand y have a road between them if there exists an integer z such that all of the following are true:

* x % z == 0,
* y % z == 0, and
* z > threshold.

Given the two integers, n and threshold, and an array of queries, you must determine for each queries[i] = [ai, bi] if cities ai and bi are connected (i.e. there is some path between them).

Return *an array*answer*, where*answer.length == queries.length*and*answer[i]*is*true*if for the*ith*query, there is a path between*ai*and*bi*, or*answer[i]*is*false*if there is no path.*

**Example 1:**



**Input:** n = 6, threshold = 2, queries = [[1,4],[2,5],[3,6]]

**Output:** [false,false,true]

**Explanation:** The divisors for each number:

1: 1

2: 1, 2

3: 1, 3

4: 1, 2, 4

5: 1, 5

6: 1, 2, 3, 6

Using the underlined divisors above the threshold, only cities 3 and 6 share a common divisor, so they are the

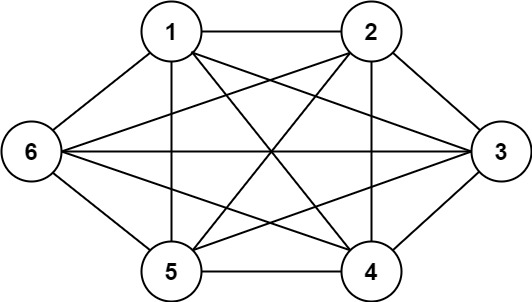
only ones directly connected. The result of each query:

[1,4] 1 is not connected to 4

[2,5] 2 is not connected to 5

[3,6] 3 is connected to 6 through path 3--6

**Example 2:**



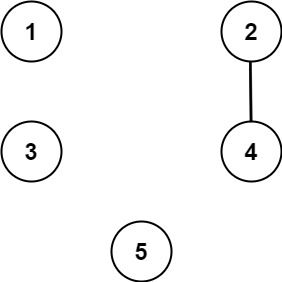
**Input:** n = 6, threshold = 0, queries = [[4,5],[3,4],[3,2],[2,6],[1,3]]

**Output:** [true,true,true,true,true]

**Explanation:** The divisors for each number are the same as the previous example. However, since the threshold is 0,

all divisors can be used. Since all numbers share 1 as a divisor, all cities are connected.

**Example 3:**



**Input:** n = 5, threshold = 1, queries = [[4,5],[4,5],[3,2],[2,3],[3,4]]

**Output:** [false,false,false,false,false]

**Explanation:** Only cities 2 and 4 share a common divisor 2 which is strictly greater than the threshold 1, so they are the only ones directly connected.

Please notice that there can be multiple queries for the same pair of nodes [x, y], and that the query [x, y] is equivalent to the query [y, x].

**Constraints:**

* 2 <= n <= 104
* 0 <= threshold <= n
* 1 <= queries.length <= 105
* queries[i].length == 2
* 1 <= ai, bi <= cities
* ai != bi

class Solution {

public:

    vector<bool> areConnected(int n, int threshold, vector<vector<int>>& queries) {

        vector<int> fa(n+1, -1);

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

            for (int j=2;j\*i<=n;++j) {

                Union(i, i\*j, fa);

            }

        }

        vector<bool> res;

        for (auto &q : queries) {

            int xx=find(q[0], fa);

            int yy=find(q[1], fa);

            res.push\_back(xx == yy);

        }

        return 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 xx=find(x, fa);

        int yy=find(y, fa);

        if (xx==yy)return;

        fa[yy] += fa[xx];

        fa[xx] = yy;

    }

};

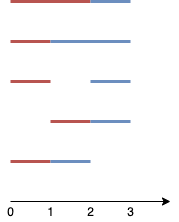
### 1621. Number of Sets of K Non-Overlapping Line Segments

Medium

Given n points on a 1-D plane, where the ith point (from 0 to n-1) is at x = i, find the number of ways we can draw **exactly** k **non-overlapping** line segments such that each segment covers two or more points. The endpoints of each segment must have **integral coordinates**. The k line segments **do not** have to cover all npoints, and they are **allowed** to share endpoints.

Return *the number of ways we can draw*k*non-overlapping line segments.* Since this number can be huge, return it **modulo** 109 + 7.

**Example 1:**



**Input:** n = 4, k = 2

**Output:** 5

**Explanation:**

The two line segments are shown in red and blue.

The image above shows the 5 different ways {(0,2),(2,3)}, {(0,1),(1,3)}, {(0,1),(2,3)}, {(1,2),(2,3)}, {(0,1),(1,2)}.

**Example 2:**

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

**Output:** 3

**Explanation:** The 3 ways are {(0,1)}, {(0,2)}, {(1,2)}.

**Example 3:**

**Input:** n = 30, k = 7

**Output:** 796297179

**Explanation:** The total number of possible ways to draw 7 line segments is 3796297200. Taking this number modulo 109 + 7 gives us 796297179.

**Example 4:**

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

**Output:** 7

**Example 5:**

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

**Output:** 1

**Constraints:**

* 2 <= n <= 1000
* 1 <= k <= n-1

class Solution {

public:

    constexpr static int mod = 1e9 + 7, N = 2050;

    int C[N][N];

    void Init () {

        for (int i = 1; i < N; i++) {

            C[i][0] = C[i][i] = 1;

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

                C[i][j] = (C[i-1][j-1]+C[i-1][j]) % mod;

        }

    }

    int numberOfSets(int n, int k) {

        Init();

        return C[n + k - 1][2 \* k];

    }

};

class Solution {

public:

    constexpr static int mod=1e9+7, MX=2e3+10;

    int mult(int a,int b){return (1ll\*a\*b)%mod;}

    int fact[MX],inv[MX],invfact[MX];

    void init\_INV(){

        fact[0] = invfact[0] = fact[1] = invfact[1] = inv[1] = 1;

        for (int i = 2; i < MX; i++) {

            fact[i] = mult(fact[i - 1], i);

            inv[i] = mult(inv[mod % i], mod - mod / i);

            invfact[i] = mult(invfact[i - 1], inv[i]);

        }

    }

    int ncr(int n,int r){

        if(r > n) return 0;

        return (1LL \* fact[n] \* invfact[n-r] % mod)

\* 1LL \*invfact[r] % mod;

    }

    int numberOfSets(int n, int k) {

        init\_INV();

        return ncr(n + k - 1, 2 \* k);

    }

};

class Solution {

public:

    int numberOfSets(int n, int K) {

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

        int kMod = 1e9 + 7;

        dp[1][1] = 1;

        for (int i = 2; i < n; i++)

            dp[i][1] = (i + 1) \* i / 2;

        vector<vector<int>> sum(n, vector<int>(K + 1));

        for (int i = 2; i < n; i++) {

            for (int k = 2; k <= K; k++) {

                if (k <= i)

                    sum[i][k] = (sum[i - 1][k]+dp[i - 1][k - 1])%kMod;

                dp[i][k] = (sum[i][k] + dp[i - 1][k]) % kMod;

            }

        }

        return dp[n - 1][K];

    }

};

### 1622. Fancy Sequence

Hard

8114Add to ListShare

Write an API that generates fancy sequences using the append, addAll, and multAll operations.

Implement the Fancy class:

* Fancy() Initializes the object with an empty sequence.
* void append(val) Appends an integer val to the end of the sequence.
* void addAll(inc) Increments all existing values in the sequence by an integer inc.
* void multAll(m) Multiplies all existing values in the sequence by an integer m.
* int getIndex(idx) Gets the current value at index idx (0-indexed) of the sequence **modulo** 109 + 7. If the index is greater or equal than the length of the sequence, return -1.

**Example 1:**

**Input**

["Fancy", "append", "addAll", "append", "multAll", "getIndex", "addAll", "append", "multAll", "getIndex", "getIndex", "getIndex"]

[[], [2], [3], [7], [2], [0], [3], [10], [2], [0], [1], [2]]

**Output**

[null, null, null, null, null, 10, null, null, null, 26, 34, 20]

**Explanation**

Fancy fancy = new Fancy();

fancy.append(2); // fancy sequence: [2]

fancy.addAll(3); // fancy sequence: [2+3] -> [5]

fancy.append(7); // fancy sequence: [5, 7]

fancy.multAll(2); // fancy sequence: [5\*2, 7\*2] -> [10, 14]

fancy.getIndex(0); // return 10

fancy.addAll(3); // fancy sequence: [10+3, 14+3] -> [13, 17]

fancy.append(10); // fancy sequence: [13, 17, 10]

fancy.multAll(2); // fancy sequence: [13\*2, 17\*2, 10\*2] -> [26, 34, 20]

fancy.getIndex(0); // return 26

fancy.getIndex(1); // return 34

fancy.getIndex(2); // return 20

**Constraints:**

* 1 <= val, inc, m <= 100
* 0 <= idx <= 105
* At most 105 calls total will be made to append, addAll, multAll, and getIndex.

class Fancy {

public:

    using ll = long long;

    const ll mod = 1e9 + 7;

    Fancy() {}

    ll inv(ll a){

        ll n = mod-2, res = 1;

        while(n) {

            if (n&1) {

                res = (res\*a)%mod;

            }

            n >>= 1;

            a = (a\*a) % mod;

        }

        return res;

    }

    void append(int val) {

        ll p = (mod + val - add) % mod;

        v.push\_back((inv(mul)\*p) % mod);

    }

    void addAll(int inc) {

        add = (add + inc) % mod;

    }

    void multAll(int m) {

        add = (add \* m) % mod;

        mul = (mul \* m) % mod;

    }

    int getIndex(int idx) {

        if (idx >= v.size()) return -1;

        else return (v[idx]\*mul + add) % mod;

    }

    vector<ll> v;

    ll add = 0, mul = 1;

};

### 1626. Best Team With No Conflicts

Medium

You are the manager of a basketball team. For the upcoming tournament, you want to choose the team with the highest overall score. The score of the team is the **sum**of scores of all the players in the team.

However, the basketball team is not allowed to have **conflicts**. A **conflict** exists if a younger player has a **strictly higher** score than an older player. A conflict does **not**occur between players of the same age.

Given two lists, scores and ages, where each scores[i] and ages[i] represents the score and age of the ith player, respectively, return *the highest overall score of all possible basketball teams*.

**Example 1:**

**Input:** scores = [1,3,5,10,15], ages = [1,2,3,4,5]

**Output:** 34

**Explanation:** You can choose all the players.

**Example 2:**

**Input:** scores = [4,5,6,5], ages = [2,1,2,1]

**Output:** 16

**Explanation:** It is best to choose the last 3 players. Notice that you are allowed to choose multiple people of the same age.

**Example 3:**

**Input:** scores = [1,2,3,5], ages = [8,9,10,1]

**Output:** 6

**Explanation:** It is best to choose the first 3 players.

**Constraints:**

* 1 <= scores.length, ages.length <= 1000
* scores.length == ages.length
* 1 <= scores[i] <= 106
* 1 <= ages[i] <= 1000

class Solution {

public:

    int bestTeamScore(vector<int>& scores, vector<int>& ages) {

        vector<pair<int, int>> players;

        int n = scores.size();

        for (int i=0; i<n; i++) {

            players.emplace\_back(ages[i], scores[i]);

        }

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

        int ans = 0;

        vector<int> dp(n);

        for (int i = 0; i < n; i++) {

            int score = players[i].second;

            dp[i] = score;

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

                if (players[j].second >= players[i].second) {

    // age of j is certainly >= i, so only important part to check

  //  before we add i and j in the same team is the score.

                    dp[i] = max(dp[i], dp[j] + score);

                }

            }

            ans = max(ans, dp[i]);

        }

        return ans;

    }

};

### 1631. Path With Minimum Effort

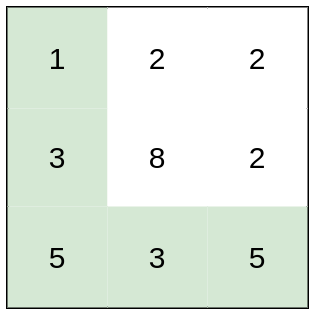
Medium

You are a hiker preparing for an upcoming hike. You are given heights, a 2D array of size rows x columns, where heights[row][col] represents the height of cell (row, col). You are situated in the top-left cell, (0, 0), and you hope to travel to the bottom-right cell, (rows-1, columns-1) (i.e., **0-indexed**). You can move **up**, **down**, **left**, or **right**, and you wish to find a route that requires the minimum **effort**.

A route's **effort** is the **maximum absolute difference**in heights between two consecutive cells of the route.

Return *the minimum****effort****required to travel from the top-left cell to the bottom-right cell.*

**Example 1:**



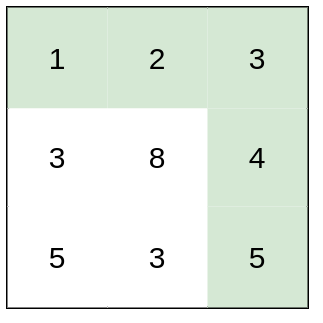
**Input:** heights = [[1,2,2],[3,8,2],[5,3,5]]

**Output:** 2

**Explanation:** The route of [1,3,5,3,5] has a maximum absolute difference of 2 in consecutive cells.

This is better than the route of [1,2,2,2,5], where the maximum absolute difference is 3.

**Example 2:**

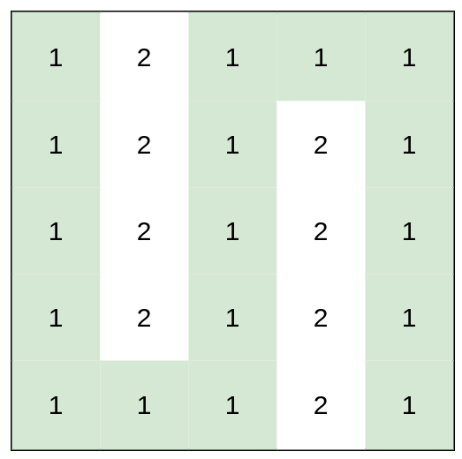


**Input:** heights = [[1,2,3],[3,8,4],[5,3,5]]

**Output:** 1

**Explanation:** The route of [1,2,3,4,5] has a maximum absolute difference of 1 in consecutive cells, which is better than route [1,3,5,3,5].

**Example 3:**



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

**Output:** 0

**Explanation:** This route does not require any effort.

**Constraints:**

* rows == heights.length
* columns == heights[i].length
* 1 <= rows, columns <= 100
* 1 <= heights[i][j] <= 106

class Solution {

public:

    int n, m;

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

    vector<vector<bool>> v;

    bool dfs(int i,int j, int mid, vector<vector<int>> &h) {

        if (i == n-1 && j == m-1) return true;

        v[i][j] = true;

        bool ok = false;

        for (int k = 0; k < 4; ++k) {

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

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

            if (v[x][y] || abs(h[i][j]-h[x][y]) > mid) continue;

            if (dfs(x, y, mid, h)) {

                ok = true;

                break;

            }

        }

        return ok;

    }

    int minimumEffortPath(vector<vector<int>>& h) {

        int low = 0, high = INT\_MAX;

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

        v.resize(n, vector<bool> (m));

        while(low <= high) {

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

            for (auto &t : v) fill(t.begin(), t.end(), 0);

            if (dfs(0, 0, mid, h))

                high = mid-1;

            else low = mid+1;

        }

        return low;

    }

};

class Solution {

public:

    int minimumEffortPath(vector<vector<int>>& h) {

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

        vector<vector<int>> dis(n, vector<int>(m, 1000000));

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

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

return p1[0] >= p2[0];

};

        priority\_queue<array<int, 3>, vector<array<int, 3>>, decltype(cmp)> pq(cmp);

        dis[0][0] = 0;

        pq.push({0, 0, 0});

        while (!pq.empty()) {

           auto [e, i, j] = pq.top();

           pq.pop();

           if (i == n-1 && j == m-1) break;

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

               int x = i + dir[d], y = j + dir[d + 1];

               if (x >= 0 && y >= 0 && x < n && y < m) {

                   int d = max(dis[i][j], abs(h[i][j] - h[x][y]));

                   if (d < dis[x][y]) {

                       dis[x][y] = d;

                       pq.push({d, x, y});

                   }

               }

            }

        }

        return dis[n-1][m-1];

    }

};

### 1632. Rank Transform of a Matrix

Hard

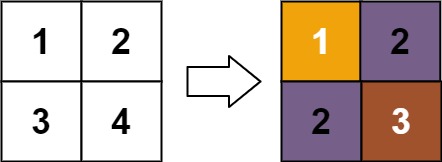
Given an m x n matrix, return *a new matrix*answer*where*answer[row][col]*is the****rank****of*matrix[row][col].

The **rank** is an **integer** that represents how large an element is compared to other elements. It is calculated using the following rules:

* If an element is the smallest element in its row and column, then its **rank** is 1.
* If two elements p and q are in the **same row or column**, then:
  + If p < q then rank(p) < rank(q)
  + If p == q then rank(p) == rank(q)
  + If p > q then rank(p) > rank(q)
* The **rank** should be as **small** as possible.

It is guaranteed that answer is unique under the given rules.

**Example 1:**



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

**Output:** [[1,2],[2,3]]

**Explanation:**

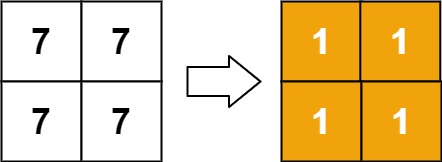
The rank of matrix[0][0] is 1 because it is the smallest integer in its row and column.

The rank of matrix[0][1] is 2 because matrix[0][1] > matrix[0][0] and matrix[0][0] is rank 1.

The rank of matrix[1][0] is 2 because matrix[1][0] > matrix[0][0] and matrix[0][0] is rank 1.

The rank of matrix[1][1] is 3 because matrix[1][1] > matrix[0][1], matrix[1][1] > matrix[1][0], and both matrix[0][1] and matrix[1][0] are rank 2.

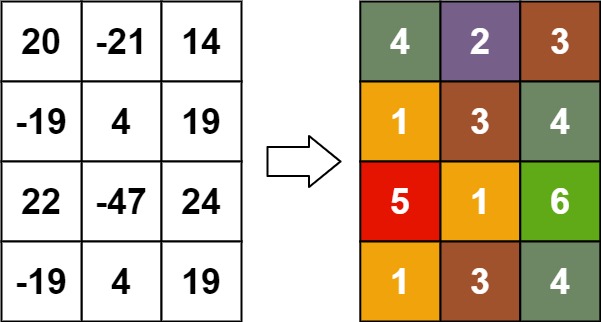
**Example 2:**



**Input:** matrix = [[7,7],[7,7]]

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

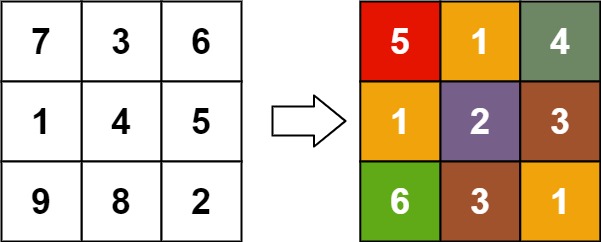
**Example 3:**



**Input:** matrix = [[20,-21,14],[-19,4,19],[22,-47,24],[-19,4,19]]

**Output:** [[4,2,3],[1,3,4],[5,1,6],[1,3,4]]

**Example 4:**



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

**Output:** [[5,1,4],[1,2,3],[6,3,1]]

**Constraints:**

* m == matrix.length
* n == matrix[i].length
* 1 <= m, n <= 500
* -109 <= matrix[row][col] <= 109

class Solution {

public:

    int n, m;

    vector<int> rows, cols;

    int find(vector<int> &ds, int i) {

        return ds[i] < 0 ? i : ds[i] = find(ds, ds[i]);

    }

   void process(vector<vector<int>>&matrix,vector<pair<int,int>>&crds){

        vector<int> ds(n+m, -1);

        for (auto [i, j] : crds) {

            auto pi = find(ds, i), pj = find(ds, n + j);

            if (pi != pj) {

                ds[pi] = min({ds[pi], ds[pj],

-max(rows[i], cols[j]) - 1});

                ds[pj] = pi;

            }

        }

        for (auto [i, j] : crds)

            matrix[i][j] = rows[i] = cols[j] = -ds[find(ds, i)];

    }

    vector<vector<int>>matrixRankTransform(vector<vector<int>>&matrix){

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

        rows.resize(n); cols.resize(m);

        vector<array<int, 3>> v;

        for (auto i = 0; i < n; ++i)

            for (auto j = 0; j < m; ++j)

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

        sort(begin(v), end(v), [](const auto &a, const auto &b){return a[0] < b[0]; });

        vector<pair<int, int>> crds;

        int last\_val = INT\_MAX;

        for (auto &[val, i, j] : v) {

            if (val != last\_val) {

                process(matrix, crds);

                last\_val = val;

                crds.clear();

            }

            crds.emplace\_back(i, j);

        }

        process(matrix, crds);

        return matrix;

    }

};

### 毕业旅行问题

小明目前在做一份毕业旅行的规划。打算从北京出发，分别去若干个城市，然后再回到北京，每个城市之间均乘坐高铁，且每个城市只去一次。由于经费有限，希望能够通过合理的路线安排尽可能的省一些路上的花销。给定一组城市和每对城市之间的火车票的价钱，找到每个城市只访问一次并返回起点的最小车费花销。

##### 输入描述:

城市个数n（1<n≤20，包括北京）  
  
城市间的车票价钱 n行n列的矩阵 m[n][n]

##### 输出描述:

最小车费花销 s

##### 输入例子1:

4

0 2 6 5

2 0 4 4

6 4 0 2

5 4 2 0

##### 输出例子1:

13

##### 例子说明1:

共 4 个城市，城市 1 和城市 1 的车费为0，城市 1 和城市 2 之间的车费为 2，城市 1 和城市 3 之间的车费为 6，城市 1 和城市 4 之间的车费为 5，依次类推。假设任意两个城市之间均有单程票可购买，且票价在1000元以内，无需考虑极端情况。

#include <bits/stdc++.h>

using namespace std;

int main() {

    int n;

    cin >> n;

    vector<vector<int>> nums(n,vector<int>(n, 0));

    for(int i = 0; i < n; i++) {

        for(int j = 0; j < n; j++)

            cin >> nums[i][j];

    }

    int maxstate = 1 << n;

//dp[i][j] 标识从0开始，经过 i 所表示的状态城市，到达j需要的最小花销

//因为是环 所以从哪开始都一样！

    vector<vector<int>> dp(maxstate, vector<int>(n, INT\_MAX));

    dp[1][0] = 0;

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

        for (int j = 0; j < n; j++) {

            if (dp[i][j] != INT\_MAX) {

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

                    if ((i & (1<<k)) == 0) {

                        int t = i | (1<<k);

                        dp[t][k] = min(dp[t][k], dp[i][j]+nums[j][k]);

                    }

                }

            }

        }

    }

    int res = INT\_MAX;

    for(int i = 1; i < n; i++) {

        if(dp[maxstate-1][i] != INT\_MAX) {

            res = min(res, dp[maxstate-1][i] + nums[i][0]);

        }

    }

    cout << res << endl;

    return 0;

}

### [1655. Distribute Repeating Integers](https://leetcode-cn.com/problems/distribute-repeating-integers/)

You are given an array of n integers, nums, where there are at most 50 unique values in the array. You are also given an array of m customer order quantities, quantity, where quantity[i] is the amount of integers the ith customer ordered. Determine if it is possible to distribute nums such that:

* The ith customer gets **exactly** quantity[i] integers,
* The integers the ith customer gets are **all equal**, and
* Every customer is satisfied.

Return true if it is possible to distribute nums according to the above conditions.

**Example 1:**

**Input:** nums = [1,2,3,4], quantity = [2]

**Output:** false

**Explanation:** The 0th customer cannot be given two different integers.

**Example 2:**

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

**Output:** true

**Explanation:** The 0th customer is given [3,3]. The integers [1,2] are not used.

**Example 3:**

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

**Output:** true

**Explanation:** The 0th customer is given [1,1], and the 1st customer is given [2,2].

**Example 4:**

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

**Output:** false

**Explanation:** Although the 0th customer could be given [1,1], the 1st customer cannot be satisfied.

**Example 5:**

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

**Output:** true

**Explanation:** The 0th customer is given [1,1], and the 1st customer is given [1,1,1].

**Constraints:**

* n == nums.length
* 1 <= n <= 105
* 1 <= nums[i] <= 1000
* m == quantity.length
* 1 <= m <= 10
* 1 <= quantity[i] <= 105
* There are at most 50 unique values in nums.

class Solution {

public:

    bool canDistribute(vector<int>& nums, vector<int>& quantity) {

        unordered\_map<int, int> mp;

        for (int x: nums) ++mp[x];

        vector<int> cnt;

        for (auto& i: mp) {

            cnt.emplace\_back(i.second);

        }

        int n = cnt.size(), m = quantity.size();

        vector<int> sum(1 << m, 0);

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

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

                if (i & (1 << j)) {

                    int left = i - (1 << j);

                    sum[i] = sum[left] + quantity[j];

                    break;

                }

            }

        }

        vector<vector<bool>> dp(2, vector<bool>(1 << m, false));

        dp[0][0] = true;

        int k = 0;

        for (int i = 0; i < n; i++) {

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

            for (int j = 0; j < (1 << m); j++) {

                cur[j] = j == 0 ? true : false;

                if (i > 0 && pre[j]) cur[j] = true;

                else for (int s = j; s != 0; s = ((s - 1) & j)) {

                    int prev = j - s;

                    // cnt[0..i-1] 能否满足子集 prev

                    bool last = (i == 0) ? (prev == 0) : pre[prev];

                    // cnt[i] 能否满足子集 s

                    bool need = sum[s] <= cnt[i];

                    if (last && need) {

                        cur[j] = true;

                        break;

                    }

                }

            }

            k ^= 1;

        }

        return dp[k].back();

    }

};

### [1665. Minimum Initial Energy to Finish Tasks](https://leetcode-cn.com/problems/minimum-initial-energy-to-finish-tasks/)★★

You are given an array tasks where tasks[i] = [actuali, minimumi]:

* actuali is the actual amount of energy you **spend to finish** the ith task.
* minimumi is the minimum amount of energy you **require to begin** the ith task.

For example, if the task is [10, 12] and your current energy is 11, you cannot start this task. However, if your current energy is 13, you can complete this task, and your energy will be 3 after finishing it.

You can finish the tasks in **any order** you like.

Return the ***minimum*** initial amount of energy you will need to finish all the tasks.

**Example 1:**

**Input:** tasks = [[1,2],[2,4],[4,8]]

**Output:** 8

**Explanation:**

Starting with 8 energy, we finish the tasks in the following order:

- 3rd task. Now energy = 8 - 4 = 4.

- 2nd task. Now energy = 4 - 2 = 2.

- 1st task. Now energy = 2 - 1 = 1.

Notice that even though we have leftover energy, starting with 7 energy does not work because we cannot do the 3rd task.

**Example 2:**

**Input:** tasks = [[1,3],[2,4],[10,11],[10,12],[8,9]]

**Output:** 32

**Explanation:**

Starting with 32 energy, we finish the tasks in the following order:

- 1st task. Now energy = 32 - 1 = 31.

- 2nd task. Now energy = 31 - 2 = 29.

- 3rd task. Now energy = 29 - 10 = 19.

- 4th task. Now energy = 19 - 10 = 9.

- 5th task. Now energy = 9 - 8 = 1.

**Example 3:**

**Input:** tasks = [[1,7],[2,8],[3,9],[4,10],[5,11],[6,12]]

**Output:** 27

**Explanation:**

Starting with 27 energy, we finish the tasks in the following order:

- 5th task. Now energy = 27 - 5 = 22.

- 2nd task. Now energy = 22 - 2 = 20.

- 3rd task. Now energy = 20 - 3 = 17.

- 1st task. Now energy = 17 - 1 = 16.

- 4th task. Now energy = 16 - 4 = 12.

- 6th task. Now energy = 12 - 6 = 6.

**Constraints:**

* 1 <= tasks.length <= 105
* 1 <= actual​i <= minimumi <= 104

class Solution {

public:

    int minimumEffort(vector<vector<int>>& tasks) {

        auto cmp = [](const vector<int> &lhs, const vector<int> &rhs) {

            return lhs[1] - lhs[0] < rhs[1] - rhs[0];

        };

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

        int res = 0;

        for (auto &v : tasks) res = max(res+v[0], v[1]);

        return res;

    }

};

### [5559. Minimum Number of Removals to Make Mountain Array](https://leetcode-cn.com/problems/minimum-number-of-removals-to-make-mountain-array/)

You may recall that an array arr is a **mountain array** if and only if:

* arr.length >= 3
* There exists some index i (**0-indexed**) with 0 < i < arr.length - 1 such that:
  + arr[0] < arr[1] < ... < arr[i - 1] < arr[i]
  + arr[i] > arr[i + 1] > ... > arr[arr.length - 1]

Given an integer array nums​​​, return the ***minimum*** number of elements to remove to make nums​​​ a ***mountain array***.

**Example 1:**

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

**Output:** 0

**Explanation:** The array itself is a mountain array so we do not need to remove any elements.

**Example 2:**

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

**Output:** 3

**Example 3:**

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

**Output:** 4

**Example 4:**

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

**Output:** 1

**Constraints:**

* 3 <= nums.length <= 1000
* 1 <= nums[i] <= 109
* It is guaranteed that you can make a mountain array out of nums.

class Solution {

public:

    int minimumMountainRemovals(vector<int>& nums) {

        int n = nums.size(), res = 1;

        vector<int> L(n), v;

        for (int i = 0; i < n; ++i) {

            auto it = lower\_bound(v.begin(), v.end(), nums[i]);

            L[i] = it - v.begin() + 1;

            if (it == v.end()) v.push\_back(nums[i]);

            else \*it = nums[i];

        }

        v.clear();

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

            auto it = lower\_bound(v.begin(), v.end(), nums[i]);

            if (L[i] > 1 && it-v.begin() > 0)

                res = max(res, (int)(it-v.begin()) + L[i]);

            if (it == v.end()) v.push\_back(nums[i]);

            else \*it = nums[i];

        }

        return n - res;

    }

};

### [5614. Find the Most Competitive Subsequence](https://leetcode-cn.com/problems/find-the-most-competitive-subsequence/)★

Given an integer array nums and a positive integer k, return the most***competitive*** subsequence of nums of size k.

An array's subsequence is a resulting sequence obtained by erasing some (possibly zero) elements from the array.

We define that a subsequence a is more **competitive** than a subsequence b (of the same length) if in the first position where a and b differ, subsequence a has a number **less** than the corresponding number in b. For example, [1,3,4] is more competitive than [1,3,5] because the first position they differ is at the final number, and 4 is less than 5.

**Example 1:**

**Input:** nums = [3,5,2,6], k = 2

**Output:** [2,6]

**Explanation:** Among the set of every possible subsequence: {[3,5], [3,2], [3,6], [5,2], [5,6], [2,6]}, [2,6] is the most competitive.

**Example 2:**

**Input:** nums = [2,4,3,3,5,4,9,6], k = 4

**Output:** [2,3,3,4]

**Constraints:**

* 1 <= nums.length <= 105
* 0 <= nums[i] <= 109
* 1 <= k <= nums.length

class Solution {

public:

    vector<int> mostCompetitive(vector<int>& v, int k) {

        int n = v.size();

        vector<int> stk;

        for (int i = 0; i < n; ++i) {

            while (!stk.empty()&&v[i]<stk.back()&&n-i+stk.size()>=k+1){

                stk.pop\_back();

            }

            stk.push\_back(v[i]);

        }

        stk.resize(k);

        return stk;

    }

};

### [5615. Minimum Moves to Make Array Complementary](https://leetcode-cn.com/problems/minimum-moves-to-make-array-complementary/)★★

You are given an integer array nums of **even** length n and an integer limit. In one move, you can replace any integer from nums with another integer between 1 and limit, inclusive.

The array nums is **complementary** if for all indices i (**0-indexed**), nums[i] + nums[n - 1 - i] equals the same number. For example, the array [1,2,3,4] is complementary because for all indices i, nums[i] + nums[n - 1 - i] = 5.

Return the ***minimum*** number of moves required to make nums ***complementary***.

**Example 1:**

**Input:** nums = [1,2,4,3], limit = 4

**Output:** 1

**Explanation:** In 1 move, you can change nums to [1,2,2,3] (underlined elements are changed).

nums[0] + nums[3] = 1 + 3 = 4.

nums[1] + nums[2] = 2 + 2 = 4.

nums[2] + nums[1] = 2 + 2 = 4.

nums[3] + nums[0] = 3 + 1 = 4.

Therefore, nums[i] + nums[n-1-i] = 4 for every i, so nums is complementary.

**Example 2:**

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

**Output:** 2

**Explanation:** In 2 moves, you can change nums to [2,2,2,2]. You cannot change any number to 3 since 3 > limit.

**Example 3:**

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

**Output:** 0

**Explanation:** nums is already complementary.

**Constraints:**

* n == nums.length
* 2 <= n <= 105
* 1 <= nums[i] <= limit <= 105
* n is even.

class Solution {

public:

    int minMoves(vector<int>& nums, int limit) {

        int n = nums.size(), N = limit\*2 + 2;

        vector<int> d(N), cnt(N);

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

            ++cnt[nums[i] + nums[n - i - 1]];

            ++d[1 + min(nums[i], nums[n - i - 1])];

            --d[limit + max(nums[i], nums[n - i - 1]) + 1];

        }

        int res = n \* 2, Sum = 0;

        for (int K = 2; K < N; ++K) {

            Sum += d[K];

            res = min(res, n - Sum - cnt[K]);

        }

        return res;

    }

};