Top Google Questions – Part 3

- 609. Find Duplicate File in System
- 616. Add Bold Tag in String
- 658. Find K Closest Elements
- 669. Trim a Binary Search Tree
- 684. Redundant Connection
- 679. 24 Game
- 693. Binary Number with Alternating Bits
- 694. Number of Distinct Islands
- 700. Search in a Binary Search Tree
- 702. Search in a Sorted Array of Unknown Size
- 705. Design HashSet
- 706. Design HashMap
- 711. Number of Distinct Islands II
- 717. 1-bit and 2-bit Characters
- 720. Longest Word in Dictionary
- 721. Accounts Merge
- 733. Flood Fill
- 734. Sentence Similarity
- 737. Sentence Similarity II
- 748. Shortest Completing Word
- 758. Bold Words in String
- 773. Sliding Puzzle
- 787. Cheapest Flights Within K Stops
- 819. Most Common Word
- 844. Backspace String Compare
- 868. Binary Gap
- 917. Reverse Only Letters
- 973. K Closest Points to Origin
- 986. Interval List Intersections

- 988. Smallest String Starting From Leaf
- 997. Find the Town Judge
- 1029. Two City Scheduling
- 1057. Campus Bikes
- 1065. Index Pairs of a String
- 1066. Campus Bikes II
- 1143. Longest Common Subsequence
- 1170. Compare Strings by Frequency of the Smallest Character
- 1277. Count Square Submatrices with All Ones
- 1287. Element Appearing More Than 25\x In Sorted Array
- 1314. Matrix Block Sum
- 1331. Rank Transform of an Array
- 1422. Maximum Score After Splitting a String
- 1424. Diagonal Traverse II
- 1438. Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit
- 1441. Build an Array With Stack Operations
- 1447. Simplified Fractions
- 1452. People Whose List of Favorite Companies Is Not a Subset of Another List
- 1461. Check If a String Contains All Binary Codes of Size K
- 1463. Cherry Pickup II
- 1471. The k Strongest Values in an Array
- Check If a String Is a Valid Sequence from Root to Leaves Path in a Binary Tree
- 1477. Find Two Non-overlapping Sub-arrays Each With Target Sum
- 1480. Running Sum of 1d Array
- 1482. Minimum Number of Days to Make m Bouquets
- 1483. Kth Ancestor of a Tree Node
- 1488. Avoid Flood in The City

609. Find Duplicate File in System

Description

Given a list of directory info including directory path, and all the files with contents in this directory, you need to find out all the groups of duplicate files in the file system in terms of their paths.

A group of duplicate files consists of at least two files that have exactly the same content.

A single directory info string in the input list has the following format:

"root/d1/d2/.../dm f1.txt(f1_content) f2.txt(f2_content) ... fn.txt(fn_content)"

It means there are n files (f1.txt, f2.txt ... fn.txt with content f1_content, f2_content ... fn_content, respectively) in directory root/d1/d2/.../dm. Note that n >= 1 and m >= 0. If m = 0, it means the directory is just the root directory.

The output is a list of group of duplicate file paths. For each group, it contains all the file paths of the files that have the same content. A file path is a string that has the following format:

"directory_path/file_name.txt"

Example 1:

Input:
["root/a 1.txt(abcd) 2.txt(efgh)", "root/c 3.txt(abcd)", "root/c/d 4.txt(efgh)", "root 4.txt(efgh)"]
Output:
[["root/a/2.txt","root/c/d/4.txt","root/4.txt"],["root/a/1.txt","root/c/3.txt"]]

Note:

No order is required for the final output.
You may assume the directory name, file name and file content only has letters and digits, and the length of file content is in the range of [1,50].
The number of files given is in the range of [1,20000].
You may assume no files or directories share the same name in the same directory.
You may assume each given directory info represents a unique directory. Directory path and file info are separated by a single blank space.

Follow-up beyond contest:
Imagine you are given a real file system, how will you search files? DFS or BFS?
If the file content is very large (GB level), how will you modify your solution?
If you can only read the file by 1kb each time, how will you modify your solution?
What is the time complexity of your modified solution? What is the most time-consuming part and memory consuming part of it? How to optimize?
How to make sure the duplicated files you find are not false positive?

Solution

05/20/2020:

```cpp
class Solution {
public:
  vector<vector<string>> findDuplicate(vector<string>& paths) {
    unordered_map<string, vector<string>> files;
    for (auto& p : paths) {
      istringstream iss(p);
      string base, file_content;
      iss >> base;
      while (iss >> file_content) {
        int i = 0, n = file_content.size();
        for (; file_content[i] != '(' && i < n; ++i);
        string filename = file_content.substr(0, i);
        string content = file_content.substr(i, n - i);
        files[content].push_back(base + "/" + filename);
      }
    }
    vector<vector<string>> ret;
    for (auto& f : files)
      if (f.second.size() > 1)
        ret.push_back(f.second);
    return ret;
  }
};
```

616. Add Bold Tag in String

Description

Given a string s and a list of strings dict, you need to add a closed pair of bold tag and to wrap the substrings in s that exist in dict. If two such substrings overlap, you need to wrap them together by only one pair of closed bold tag. Also, if two substrings wrapped by bold tags are consecutive, you need to combine them.

Example 1:

Input:
s = "abcxyz123"
dict = ["abc","123"]
Output:
"abcxyz123"

Example 2:

Input:

s = "aaabbcc"
dict = ["aaa","aab","bc"]
Output:
"aaabbcc"

Constraints:

The given dict won't contain duplicates, and its length won't exceed 100.
All the strings in input have length in range [1, 1000].
Note: This question is the same as 758: https://leetcode.com/problems/bold-words-in-string/

Solution

06/09/2020:

```
class Solution {
public:
  string addBoldTag(string s, vector<string>& dict) {
    unordered_set<string> dicts(dict.begin(), dict.end());
    vector<int> lengths;
    for (auto& d : dict) lengths.push_back(d.size());
    sort(lengths.rbegin(), lengths.rend());
    int n = s.size();
    vector<bool> isBold(n, false);
    for (int i = 0; i < n; ++i) {
      bool wrap = false;
      int wrapLength = 0;
      for (auto& len : lengths) {
        if (i + len <= n && dicts.count(s.substr(i, len)) > 0) {
          wrap = true;
          wrapLength = len;
          break;
        }
      }
      if (wrap) {
        for (int j = i; j < i + wrapLength; ++j)
          isBold[j] = true;
      }
    }
    string ret;
    int last = 0;
    for (int i = 0; i < n;) {
      if (isBold[i]) {
        while (i + 1 < n && isBold[i + 1]) ++i;
        ret += "<b>";
        ret += s.substr(last, i - last + 1);
```

```
            ret += "</b>";
            last = ++i;
          } else {
            ret += s[i];
            last = ++i;
          }
        }
        return ret;
    }
};
```

658. Find K Closest Elements

Description

Given a sorted array, two integers k and x, find the k closest elements to x in the array. The result should also be sorted in ascending order. If there is a tie, the smaller elements are always preferred.

Example 1:

Input: [1,2,3,4,5], k=4, x=3
Output: [1,2,3,4]
Example 2:

Input: [1,2,3,4,5], k=4, x=-1
Output: [1,2,3,4]
Note:

The value k is positive and will always be smaller than the length of the sorted array.
Length of the given array is positive and will not exceed 104
Absolute value of elements in the array and x will not exceed 104

Solution

04/28/2020:

```
class Solution {
public:
    vector<int> findClosestElements(vector<int>& arr, int k, int x) {
        if (arr.empty()) return {};
        int n = arr.size();
        int lo = 0, hi = n - 1;
        int l, r;
```

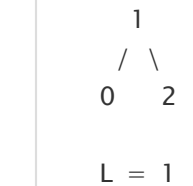
```
    if (x < arr.front()) {
      l = 0;
      r = 0;
    } else if (x > arr.back()) {
      l = n - 1;
      r = n - 1;
    } else {
      while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (arr[mid] == x) {
          l = mid;
          r = mid;
          break;
        } else if (arr[mid] > x) {
          hi = mid - 1;
          r = hi;
        } else {
          lo = mid + 1;
          l = lo;
        }
      }
      if (abs(arr[l] - x) < abs(arr[r] - x)) {
        r = l;
      } else {
        l = r;
      }
    }
    deque<int> q;
    if (k-- > 0) {
      q.push_front(arr[l]);
      l--;
      r++;
    }
    while (k > 0) {
      if (l < 0) {
        q.push_back(arr[r++]);
      } else if (r > n - 1) {
        q.push_front(arr[l--]);
      } else if (abs(arr[l] - x) <= abs(arr[r] - x)) {
        q.push_front(arr[l--]);
      } else if (abs(arr[l] - x) > abs(arr[r] - x)) {
        q.push_back(arr[r++]);
      }
      --k;
    }
    return vector<int>(q.begin(), q.end());
  }
};
```

669. Trim a Binary Search Tree

Description

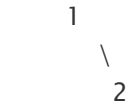
Given a binary search tree and the lowest and highest boundaries as L and R, trim the tree so that all its elements lies in [L, R] (R >= L). You might need to change the root of the tree, so the result should return the new root of the trimmed binary search tree.

Example 1:
Input:

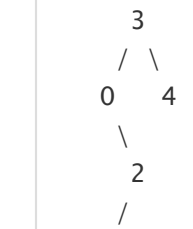


```
  L = 1
  R = 2
```

Output:

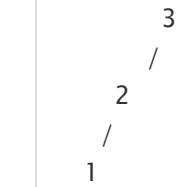


Example 2:
Input:



```
  L = 1
  R = 3
```

Output:



Solution

05/10/2020:

```
/**
```

```
 * 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:
  TreeNode* trimBST(TreeNode* root, int L, int R) {
    if (root == nullptr) return root;
    if (root->val < L) return trimBST(root->right, L, R);
    if (root->val > R) return trimBST(root->left, L, R);
    root->left = trimBST(root->left, L, R);
    root->right = trimBST(root->right, L, R);
    return root;
  }
};
```

684. Redundant Connection

Description

In this problem, a tree is an undirected graph that is connected and has no cycles.

The given input is a graph that started as a tree with N nodes (with distinct values 1, 2, ..., N), with one additional edge added. The added edge has two different vertices chosen from 1 to N, and was not an edge that already existed.

The resulting graph is given as a 2D-array of edges. Each element of edges is a pair [u, v] with u < v, that represents an undirected edge connecting nodes u and v.

Return an edge that can be removed so that the resulting graph is a tree of N nodes. If there are multiple answers, return the answer that occurs last in the given 2D-array. The answer edge [u, v] should be in the same format, with u < v.

Example 1:
Input: [[1,2], [1,3], [2,3]]
Output: [2,3]
Explanation: The given undirected graph will be like this:

```
  1
 / \
2 - 3
```

Example 2:
Input: [[1,2], [2,3], [3,4], [1,4], [1,5]]
Output: [1,4]
Explanation: The given undirected graph will be like this:
5 - 1 - 2
    | |
    4 - 3
Note:
The size of the input 2D-array will be between 3 and 1000.
Every integer represented in the 2D-array will be between 1 and N, where N is the size of the input array.

Solution

06/10/2020:

```
class UnionFind {
private:
  vector<int> id;
  vector<int> sz;

public:
  UnionFind(int n) {
    id.resize(n);
    iota(id.begin(), id.end(), 0);
    sz.resize(n, 1);
  }

  int find(int x) {
    if (x == id[x]) return x;
    return id[x] = find(id[x]);
  }

  bool connected(int x, int y) {
    return find(x) == find(y);
  }

  bool merge(int x, int y) {
    int i = find(x), j = find(y);
    if (i == j) return false;
    if (sz[i] > sz[j]) {
      sz[i] += sz[j];
      id[j] = i;
    } else {
      sz[j] += sz[i];
      id[i] = j;
    }
    return true;
```

```
  }
};

class Solution {
public:
  vector<int> findRedundantConnection(vector<vector<int>>& edges) {
    int n = edges.size();
    UnionFind uf(n + 1);
    vector<int> ret;
    for (auto& e : edges)
      if (uf.connected(e[0], e[1]))
        ret = e;
      else
        uf.merge(e[0], e[1]);
    return ret;
  }
};
```

679. 24 Game

Description

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

Example 1:
Input: [4, 1, 8, 7]
Output: True
Explanation: (8-4) * (7-1) = 24
Example 2:
Input: [1, 2, 1, 2]
Output: False
Note:
The division operator / represents real division, not integer division. For example, 4 / (1 - 2/3) = 12.
Every operation done is between two numbers. In particular, we cannot use - as a unary operator. For example, with [1, 1, 1, 1] as input, the expression -1 - 1 - 1 - 1 is not allowed.
You cannot concatenate numbers together. For example, if the input is [1, 2, 1, 2], we cannot write this as 12 + 12.

Solution

06/10/2020:

```cpp
class Solution {
public:
  bool judgePoint24(vector<int>& nums) {
    unordered_set<string> seen;
    string operators = "+-*/";
    sort(nums.begin(), nums.end());
    do {
      int a = nums[0], b = nums[1], c = nums[2], d = nums[3];
      string key = to_string(a) + "," + to_string(b) + "," + to_string(c) + ","
+  to_string(d);
      if (seen.count(key) > 0) continue;
      seen.insert(key);
      for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
          for (int k = 0; k < 4; ++k) {
            vector<double> results;
            results.push_back(op(op(op(a, b, operators[i]), c, operators[j]), d,
operators[k]));
            results.push_back(op(op(a, b, operators[i]), op(c, d, operators[k]),
operators[j]));
            results.push_back(op(op(a, op(b, c, operators[j]), operators[i]), d,
operators[k]));
            results.push_back(op(a, op(op(b, c, operators[j]), d, operators[k]),
operators[i]));
            results.push_back(op(a, op(b, op(c, d, operators[k]),
operators[j]),operators[i]));
            for (auto& r : results)
              if (fabs(r - 24) < 1e-6)
                return true;
          }
        }
      }
    } while (next_permutation(nums.begin(), nums.end()));
    return false;
  }

  double op(double a, double b, char op) {
    switch(op) {
      case '+': return a + b;
      case '-': return a - b;
      case '*': return a * b;
      case '/': return b != 0 ? a / b : -1e6;
    }
    return 0;
  }
};
```

```cpp
class Solution {
public:
  bool judgePoint24(vector<int>& nums) {
    bool res = false;
    double eps = 0.001;
    vector<double> arr(nums.begin(), nums.end());
    dfs(arr, eps, res);
    return res;
  }

  void dfs(vector<double>& nums, double eps, bool& res) {
    if(res) return;
    if(nums.size()==1){
      if(abs(nums[0]-24)<eps) res=true;
      return;
    }
    for(int i=0;i<nums.size();i++){
      for(int j=0;j<i;j++){
        double p=nums[i], q=nums[j];
        vector<double>t{p+q, p-q, q-p, p*q};
        if (p > eps) t.push_back(q / p);
        if (q > eps) t.push_back(p / q);
        nums.erase(nums.begin() + i);
        nums.erase(nums.begin() + j);
        for (double d : t) {
          nums.push_back(d);
          dfs(nums, eps, res);
          nums.pop_back();
        }
        nums.insert(nums.begin() + j, q);
        nums.insert(nums.begin() + i, p);
      }
    }
  }
};
```

693. Binary Number with Alternating Bits

Description

Given a positive integer, check whether it has alternating bits: namely, if two adjacent bits will always have different values.

Example 1:

```
Input: 5
Output: True
Explanation:
The binary representation of 5 is: 101
Example 2:

Input: 7
Output: False
Explanation:
The binary representation of 7 is: 111.
Example 3:

Input: 11
Output: False
Explanation:
The binary representation of 11 is: 1011.
Example 4:

Input: 10
Output: True
Explanation:
The binary representation of 10 is: 1010.
```

Solution

05/10/2020:

```cpp
class Solution {
public:
  bool hasAlternatingBits(int n) {
    int cnt = n & 1 ? 1 : 0;
    while (n != 0) {
      n >>= 1;
      if (n & 1) {
        ++cnt;
      } else {
        --cnt;
      }
      if (cnt < 0 || cnt > 1) return false;
    }
    return true;
  }
};
```

694. Number of Distinct Islands

Description

Given a non-empty 2D array grid of 0's and 1's, an island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

Count the number of distinct islands. An island is considered to be the same as another if and only if one island can be translated (and not rotated or reflected) to equal the other.

Example 1:

```
11000
11000
00011
00011
```

Given the above grid map, return 1.

Example 2:

```
11011
10000
00001
11011
```

Given the above grid map, return 3.

Notice that:

```
11
1
```

and

```
 1
11
```

are considered different island shapes, because we do not consider reflection / rotation.

Note: The length of each dimension in the given grid does not exceed 50.

Solution

05/08/2020 Discussion:

1. Use a UnionFind set to find all the islands.
2. Eliminate the islands that are the "same" by hashing: translate all the islands to the top-left corner (subtract offset from the id for each cell). Then hash the sorted sequence of the cell. Note: the offset is obtained from the minimum of the row index mini and column index minj: offset = mini * n + minj.

```
class UnionFind {
private:
  vector<int> id;
```

```
  vector<int> sz;

public:
  UnionFind(int n) {
    id.resize(n);
    iota(id.begin(), id.end(), 0);
    sz.resize(n, 1);
  }

  int find(int x) {
    while (x != id[x]) {
      id[x] = id[id[x]];
      x = id[x];
    }
    return x;
  }

  bool merge(int x, int y) {
    int i = find(x), j = find(y);
    if (i == j) return false;
    if (sz[i] > sz[j]) {
      sz[i] += sz[j];
      id[j] = i;
    } else {
      sz[j] += sz[i];
      id[i] = j;
    }
    return true;
  }
};

class Solution {
public:
  int numDistinctIslands(vector<vector<int>>& grid) {
    if (grid.empty() || grid[0].empty()) return 0;
    int m = grid.size(), n = grid[0].size();

    UnionFind uf(m * n);
    int dir[4][2] = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };
    for (int i = 0; i < m; ++i) {
      for (int j = 0; j < n; ++j) {
        if (grid[i][j] == 0) continue;
        int island = i * n + j;
        for (int d = 0; d < 4; ++d) {
          int ni = i + dir[d][0], nj = j + dir[d][1], neighbor = ni * n + nj;
          if (ni >= 0 && ni < m && nj >= 0 && nj < n && grid[ni][nj] == 1) {
            bool merged = uf.merge(island, neighbor);
          }
        }
```

```
      }
    }

    unordered_map<int, vector<int>> components;
    for (int i = 0; i < m; ++i) {
      for (int j = 0; j < n; ++j) {
        if (grid[i][j] == 0) continue;
        int island = i * n + j;
        components[uf.find(island)].push_back(island);
      }
    }

    unordered_set<int> islands;
    for (auto& c : components) {
      sort(c.second.begin(), c.second.end());
      int mini = c.second.front() / n, minj = n;
      for (auto& i : c.second) minj = min(minj, i % n);
      int offset = mini * n + minj;
      long long hash = 1;
      const int MOD = 1e9 + 7;
      for (auto& i : c.second) hash = (hash * 31 + i - offset) % MOD;
      islands.insert(hash);
    }
    return islands.size();
  }
};
```

```
class UnionFind {
private:
  vector<int> id;
  vector<int> sz;

public:
  UnionFind(int n) {
    id.resize(n);
    iota(id.begin(), id.end(), 0);
    sz.resize(n, 1);
  }

  int find(int x) {
    if (x == id[x]) return x;
    return id[x] = find(id[x]);
  }

  bool merge(int x, int y) {
    int i = find(x), j = find(y);
    if (i == j) return false;
    if (sz[i] > sz[j]) {
```

```
      sz[i] += sz[j];
      id[j] = i;
    } else {
      sz[j] += sz[i];
      id[i] = j;
    }
    return true;
  }
};

class Solution {
private:
  int m, n;
public:
  int numDistinctIslands(vector<vector<int>>& grid) {
    if (grid.empty() || grid[0].empty()) return 0;
    m = grid.size(), n = grid[0].size();
    UnionFind uf(m * n);
    int dir[4][2] = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };
    for (int i = 0; i < m; ++i) {
      for (int j = 0; j < n; ++j) {
        if (grid[i][j] == 0) continue;
        int island = i * n + j;
        for (int d = 0; d < 4; ++d) {
          int ni = i + dir[d][0], nj = j + dir[d][1], neighbor = ni * n + nj;
          if (ni >= 0 && ni < m && nj >= 0 && nj < n && grid[ni][nj] == 1) {
            uf.merge(island, neighbor);
          }
        }
      }
    }

    unordered_map<int, vector<pair<int, int>>> components;
    for (int i = 0; i < m; ++i) {
      for (int j = 0; j < n; ++j) {
        if (grid[i][j] == 0) continue;
        components[uf.find(i * n + j)].emplace_back(i, j);
      }
    }

    unordered_set<int> seen;
    int cnt = 0;
    for (auto& c : components) {
      vector<int> hashes = hash(c.second);
      bool exists = false;
      for (auto& h : hashes) {
        if (seen.count(h)) {
          exists = true;
          break;
```

```
        }
      }
      for (auto& h : hashes) seen.insert(h);
      if (!exists) ++cnt;
    }
    return cnt;
  }

  vector<int> hash(vector<pair<int, int>> nums) {
    const int MOD = 1e9 + 7;
    int ops[4][2] = { {1, 1}, {-1, 1}, {-1, -1}, {1, -1} };
    int N = 1;
    vector<long long> hashes(N, 1);
    for (int d = 0; d < N; ++d) {
      int offset_x = INT_MAX, offset_y = INT_MAX;
      vector<pair<int, int>> t_nums(nums);
      for (auto& p : t_nums) {
        int& x = p.first;
        int& y = p.second;
        if (d >= 4) swap(x, y);
        x *= ops[d % 4][0];
        y *= ops[d % 4][1];
        offset_x = min(offset_x, x);
        offset_y = min(offset_y, y);
      }
      sort(t_nums.begin(), t_nums.end());
      for (auto& p : t_nums) {
        int& x = p.first;
        int& y = p.second;
        hashes[d] = (hashes[d] * 31 + (x - offset_x) * n + (y - offset_y)) %
MOD;

      }
    }
    return vector<int>(hashes.begin(), hashes.end());
  }
};
```

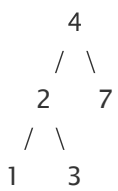
700. Search in a Binary Search Tree

Description

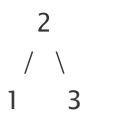
Given the root node of a binary search tree (BST) and a value. You need to find the node in the BST that the node's value equals the given value. Return the subtree rooted with that node. If such node doesn't exist, you should return NULL.

For example,

Given the tree:



And the value to search: 2
You should return this subtree:



In the example above, if we want to search the value 5, since there is no node with value 5, we should return NULL.

Note that an empty tree is represented by NULL, therefore you would see the expected output (serialized tree format) as [], not null.

Solution

06/15/2020:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
  TreeNode* searchBST(TreeNode* root, int val) {
    if (!root) return nullptr;
    if (root->val == val) return root;
    if (root->val > val) return searchBST(root->left, val);
    if (root->val < val) return searchBST(root->right, val);
    return nullptr;
  }
};
```

702. Search in a Sorted Array of Unknown Size

Description

Given an integer array sorted in ascending order, write a function to search target in nums. If target exists, then return its index, otherwise return -1. However, the array size is unknown to you. You may only access the array using an ArrayReader interface, where ArrayReader.get(k) returns the element of the array at index k (0-indexed).

You may assume all integers in the array are less than 10000, and if you access the array out of bounds, ArrayReader.get will return 2147483647.

Example 1:

Input: array = [-1,0,3,5,9,12], target = 9
Output: 4
Explanation: 9 exists in nums and its index is 4
Example 2:

Input: array = [-1,0,3,5,9,12], target = 2
Output: -1
Explanation: 2 does not exist in nums so return -1

Note:

You may assume that all elements in the array are unique.
The value of each element in the array will be in the range [-9999, 9999].

Solution

04/26/2020:

```
/**
 * // This is the ArrayReader's API interface.
 * // You should not implement it, or speculate about its implementation
 * class ArrayReader {
 *   public:
 *     int get(int index);
 * };
 */

class Solution {
public:
```

```
  int search(const ArrayReader& reader, int target) {
    int lo = 0, hi = INT_MAX;
    while (lo <= hi) {
      int mid = lo + (hi - lo) / 2;
      int val = reader.get(mid);
      if (val == target) {
        return mid;
      } else if (val == INT_MAX || val > target) {
        hi = mid - 1;
      } else {
        lo = mid + 1;
      }
    }
    return -1;
  }
};
```

705. Design HashSet

Description

Design a HashSet without using any built-in hash table libraries.

To be specific, your design should include these functions:

add(value): Insert a value into the HashSet.
contains(value) : Return whether the value exists in the HashSet or not.
remove(value): Remove a value in the HashSet. If the value does not exist in the HashSet, do nothing.

Example:

```
MyHashSet hashSet = new MyHashSet();
hashSet.add(1);
hashSet.add(2);
hashSet.contains(1);    // returns true
hashSet.contains(3);    // returns false (not found)
hashSet.add(2);
hashSet.contains(2);    // returns true
hashSet.remove(2);
hashSet.contains(2);    // returns false (already removed)
```

Note:

All values will be in the range of [0, 1000000].
The number of operations will be in the range of [1, 10000].

> Please do not use the built-in HashSet library.

Solution

05/10/2020:

```
struct Node {
  int key;
  Node* next;
  Node (int k) : key(k), next(nullptr) {}
};

class MyHashSet {
private:
  vector<Node*> hashset;
  int N;

public:
  /** Initialize your data structure here. */
  MyHashSet() {
    N = 100003;
    hashset.resize(N, nullptr);
  }

  void add(int key) {
    Node* cur = hashset[key % N];
    if (cur == nullptr) {
      hashset[key % N] = new Node(key);
    } else {
      while (cur->key != key && cur->next != nullptr) {
        cur = cur->next;
      }
      if (cur->key != key) {
        cur->next = new Node(key);
      }
    }
  }

  void remove(int key) {
    Node* cur = hashset[key % N];
    if (cur == nullptr) return;
    if (cur->key == key) {
      hashset[key % N] = cur->next;
    } else {
      while (cur->next != nullptr && cur->next->key != key) {
        cur = cur->next;
      }
      if (cur->next != nullptr && cur->next->key == key) {
```

```
        cur->next = cur->next->next;
      }
    }
  }

  /** Returns true if this set contains the specified element */
  bool contains(int key) {
    Node* cur = hashset[key % N];
    if (cur == nullptr) return false;
    if (cur->key == key) {
      return true;
    } else {
      while (cur->key != key && cur->next != nullptr) {
        cur = cur->next;
      }
      return cur->key == key;
    }
  }
};

/**
 * Your MyHashSet object will be instantiated and called as such:
 * MyHashSet* obj = new MyHashSet();
 * obj->add(key);
 * obj->remove(key);
 * bool param_3 = obj->contains(key);
 */
```

706. Design HashMap

Description

Design a HashMap without using any built-in hash table libraries.

To be specific, your design should include these functions:

put(key, value) : Insert a (key, value) pair into the HashMap. If the value already exists in the HashMap, update the value.
get(key): Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key.
remove(key) : Remove the mapping for the value key if this map contains the mapping for the key.

Example:

```
MyHashMap hashMap = new MyHashMap();
```

```
hashMap.put(1, 1);
hashMap.put(2, 2);
hashMap.get(1);            // returns 1
hashMap.get(3);            // returns -1 (not found)
hashMap.put(2, 1);         // update the existing value
hashMap.get(2);            // returns 1
hashMap.remove(2);         // remove the mapping for 2
hashMap.get(2);            // returns -1 (not found)
```

Note:

All keys and values will be in the range of [0, 1000000].
The number of operations will be in the range of [1, 10000].
Please do not use the built-in HashMap library.

Solution

05/10/2020:

```
struct Node {
  int key, val;
  Node* next;
  Node (int k, int v) : key(k), val(v), next(nullptr) {}
};

class MyHashMap {
private:
  vector<Node*> hashmap;
  int N, n;

public:
  /** Initialize your data structure here. */
  MyHashMap() {
    N = 100003;
    n = 0;
    hashmap.resize(N, nullptr);
  }

  /** value will always be non-negative. */
  void put(int key, int value) {
    Node* cur = hashmap[key % N];
    if (cur == nullptr) {
      hashmap[key % N] = new Node(key, value);
    } else {
      while (cur->key != key && cur->next != nullptr) {
        cur = cur->next;
      }
      if (cur->key == key) {
```

```
        cur->val = value;
      } else if (cur->next == nullptr) {
        cur->next = new Node(key, value);
      }
    }
  }

  /** Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key */
  int get(int key) {
    Node* cur = hashmap[key % N];
    if (cur == nullptr) {
      return -1;
    } else if (cur->key == key) {
      return cur->val;
    } else {
      while (cur->key != key && cur->next != nullptr) {
        cur = cur->next;
      }
      return cur->key == key ? cur->val : -1;
    }
  }

  /** Removes the mapping of the specified value key if this map contains a mapping for the key */
  void remove(int key) {
    Node* cur = hashmap[key % N];
    if (cur == nullptr) return;
    if (cur->key == key) {
      hashmap[key % N] = cur->next;
    } else {
      while (cur->next != nullptr && cur->next->key != key) {
        cur = cur->next;
      }
      if (cur->next != nullptr && cur->next->key == key) {
        cur->next = cur->next->next;
      }
    }
  }
};

/**
 * Your MyHashMap object will be instantiated and called as such:
 * MyHashMap* obj = new MyHashMap();
 * obj->put(key,value);
 * int param_2 = obj->get(key);
 * obj->remove(key);
 */
```

711. Number of Distinct Islands II

Description

Given a non-empty 2D array grid of 0's and 1's, an island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

Count the number of distinct islands. An island is considered to be the same as another if they have the same shape, or have the same shape after rotation (90, 180, or 270 degrees only) or reflection (left/right direction or up/down direction).

Example 1:

```
11000
10000
00001
00011
```

Given the above grid map, return 1.

Notice that:

```
11
1
```

and

```
 1
11
```

are considered same island shapes. Because if we make a 180 degrees clockwise rotation on the first island, then two islands will have the same shapes.

Example 2:

```
11100
10001
01001
01110
```

Given the above grid map, return 2.

Here are the two distinct islands:

```
111
1
```

and

```
1
1
```

Notice that:

```
111
1
```

and

```
1
111
```
are considered same island shapes. Because if we flip the first array in the up/down direction, then they have the same shapes.

Note: The length of each dimension in the given grid does not exceed 50.

Solution

05/08/2020 Discussion:

1. Use a UnionFind set to find all the islands.
2. Eliminate the islands that are the "same" by hashing: translate all the transformed islands to the top-left corner (subtract offset_x from the x and substract offset_y from y for each cell). Then hash the sorted sequence of the cell. Note: the offset_x and offset_y are obtained from the minimum of the row index and column index.

```
class UnionFind {
private:
  vector<int> id;
  vector<int> sz;

public:
  UnionFind(int n) {
    id.resize(n);
    iota(id.begin(), id.end(), 0);
    sz.resize(n, 1);
  }

  int find(int x) {
    if (x == id[x]) return x;
    return id[x] = find(id[x]);
  }

  bool merge(int x, int y) {
    int i = find(x), j = find(y);
    if (i == j) return false;
    if (sz[i] > sz[j]) {
      sz[i] += sz[j];
      id[j] = i;
    } else {
      sz[j] += sz[i];
      id[i] = j;
    }
    return true;
  }
};

class Solution {
```

```
private:
  int m, n;
public:
  int numDistinctIslands2(vector<vector<int>>& grid) {
    if (grid.empty() || grid[0].empty()) return 0;
    m = grid.size(), n = grid[0].size();
    UnionFind uf(m * n);
    int dir[4][2] = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };
    for (int i = 0; i < m; ++i) {
      for (int j = 0; j < n; ++j) {
        if (grid[i][j] == 0) continue;
        int island = i * n + j;
        for (int d = 0; d < 4; ++d) {
          int ni = i + dir[d][0], nj = j + dir[d][1], neighbor = ni * n + nj;
          if (ni >= 0 && ni < m && nj >= 0 && nj < n && grid[ni][nj] == 1) {
            uf.merge(island, neighbor);
          }
        }
      }
    }

    unordered_map<int, vector<pair<int, int>>> components;
    for (int i = 0; i < m; ++i) {
      for (int j = 0; j < n; ++j) {
        if (grid[i][j] == 0) continue;
        components[uf.find(i * n + j)].emplace_back(i, j);
      }
    }

    unordered_set<int> seen;
    int cnt = 0;
    for (auto& c : components) {
      vector<int> hashes = hash(c.second);
      bool exists = false;
      for (auto& h : hashes) {
        if (seen.count(h)) {
          exists = true;
          break;
        }
      }
      for (auto& h : hashes) seen.insert(h);
      if (!exists) ++cnt;
    }
    return cnt;
  }

  vector<int> hash(vector<pair<int, int>> nums) {
    const int MOD = 1e9 + 7;
    int ops[4][2] = { {1, 1}, {-1, 1}, {-1, -1}, {1, -1} };
```

```
    vector<long long> hashes(8, 1);
    for (int d = 0; d < 8; ++d) {
      int offset_x = INT_MAX, offset_y = INT_MAX;
      vector<pair<int, int>> t_nums(nums);
      for (auto& p : t_nums) {
        int& x = p.first;
        int& y = p.second;
        if (d >= 4) swap(x, y);
        x *= ops[d % 4][0];
        y *= ops[d % 4][1];
        offset_x = min(offset_x, x);
        offset_y = min(offset_y, y);
      }
      sort(t_nums.begin(), t_nums.end());
      for (auto& p : t_nums) {
        int& x = p.first;
        int& y = p.second;
        hashes[d] = (hashes[d] * 31 + (x - offset_x) * n + (y - offset_y)) %
MOD;
      }
    }
    return vector<int>(hashes.begin(), hashes.end());
  }
};
```

717. 1-bit and 2-bit Characters

Description

We have two special characters. The first character can be represented by one bit 0. The second character can be represented by two bits (10 or 11).

Now given a string represented by several bits. Return whether the last character must be a one-bit character or not. The given string will always end with a zero.

Example 1:
Input:
bits = [1, 0, 0]
Output: True
Explanation:
The only way to decode it is two-bit character and one-bit character. So the last character is one-bit character.
Example 2:
Input:
bits = [1, 1, 1, 0]

Output: False
Explanation:
The only way to decode it is two-bit character and two-bit character. So the last character is NOT one-bit character.
Note:

1 <= len(bits) <= 1000.
bits[i] is always 0 or 1.

Solution

06/09/2020:

```
class Solution {
public:
  bool isOneBitCharacter(vector<int>& bits) {
    int i = 0, n = bits.size();
    for (; i < n - 1; ++i) {
      if (bits[i] == 1) {
        ++i;
      }
    }
    return i == n - 1;
  }
};
```

720. Longest Word in Dictionary

Description

Given a list of strings words representing an English Dictionary, find the longest word in words that can be built one character at a time by other words in words. If there is more than one possible answer, return the longest word with the smallest lexicographical order.

If there is no answer, return the empty string.
Example 1:
Input:
words = ["w","wo","wor","worl", "world"]
Output: "world"
Explanation:
The word "world" can be built one character at a time by "w", "wo", "wor", and "worl".
Example 2:
Input:

words = ["a", "banana", "app", "appl", "ap", "apply", "apple"]
Output: "apple"
Explanation:
Both "apply" and "apple" can be built from other words in the dictionary.
However, "apple" is lexicographically smaller than "apply".
Note:

All the strings in the input will only contain lowercase letters.
The length of words will be in the range [1, 1000].
The length of words[i] will be in the range [1, 30].

Solution

05/18/2020:

```
class Solution {
public:
  string longestWord(vector<string>& words) {
    unordered_set<string> s(words.begin(), words.end());
    sort(words.begin(), words.end(), [](const string& s1, const string& s2) {
      if (s1.size() == s2.size()) {
        return s1 < s2;
      }
      return s1.size() > s2.size();
    });
    for (auto& w : words) {
      bool ok = true;
      for (int i = 1; i < (int)w.size(); ++i) {
        if (s.count(w.substr(0, i)) == 0) {
          ok = false;
          break;
        }
      }
      if (ok) return w;
    }
    return "";
  }
};
```

721. Accounts Merge

Description

Given a list accounts, each element accounts[i] is a list of strings, where the first element accounts[i][0] is a name, and the rest of the elements are emails representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some email that is common to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in sorted order. The accounts themselves can be returned in any order.

Example 1:
Input:
accounts = [["John", "johnsmith@mail.com", "john00@mail.com"], ["John", "johnnybravo@mail.com"], ["John", "johnsmith@mail.com", "john_newyork@mail.com"], ["Mary", "mary@mail.com"]]
Output: [["John", 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com'], ["John", "johnnybravo@mail.com"], ["Mary", "mary@mail.com"]]
Explanation:
The first and third John's are the same person as they have the common email "johnsmith@mail.com".
The second John and Mary are different people as none of their email addresses are used by other accounts.
We could return these lists in any order, for example the answer [['Mary', 'mary@mail.com'], ['John', 'johnnybravo@mail.com'], ['John', 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com']] would still be accepted.
Note:

The length of accounts will be in the range [1, 1000].
The length of accounts[i] will be in the range [1, 10].
The length of accounts[i][j] will be in the range [1, 30].

Solution

06/10/2020:

Define UnionFind:

```
class UnionFind {
private:
  vector<int> id;
  vector<int> sz;
```

```
public:
  UnionFind(int n) {
    id.resize(n);
    iota(id.begin(), id.end(), 0);
    sz.resize(n, 1);
  }

  int find(int x) {
    if (x == id[x]) return x;
    return id[x] = find(id[x]);
  }

  bool connected(int x, int y) {
    return find(x) == find(y);
  }

  bool merge(int x, int y) {
    int i = find(x), j = find(y);
    if (i == j) return false;
    if (sz[i] > sz[j]) {
      id[j] = i;
      sz[i] += sz[j];
    } else {
      id[i] = j;
      sz[j] += sz[i];
    }
    return true;
  }
};
```

```
class Solution {
public:
  vector<vector<string>> accountsMerge(vector<vector<string>>& accounts) {
    int id = 0;
    UnionFind uf(10000);
    unordered_map<string, string> email_user;
    unordered_map<string, int> email_id;
    unordered_map<int, string> id_email;
    for (auto& emails : accounts) {
      for (int i = 1; i < (int)emails.size(); ++i) {
        email_user[emails[i]] = emails[0];
        if (email_id.count(emails[i]) == 0) {
          email_id[emails[i]] = id++;
          id_email[id - 1] = emails[i];
        }
        if (i > 1) uf.merge(email_id[emails[i - 1]], email_id[emails[i]]);
      }
    }
```

```
    unordered_map<int, vector<int>> connected_components;
    for (int i = 0; i < id; ++i) connected_components[uf.find(i)].push_back(i);

    vector<vector<string>> ret;
    for (auto& c : connected_components) {
      vector<string> account;
      account.push_back(email_user[id_email[c.first]]);
      for (auto& i : c.second) account.push_back(id_email[i]);
      sort(account.begin() + 1, account.end());
      ret.push_back(account);
    }
    return ret;
  }
};
```

```
class Solution {
public:
  vector<vector<string>> accountsMerge(vector<vector<string>>& accounts) {
    int n = accounts.size();
    UnionFind uf(n);
    for (int i = 0; i < n; ++i) {
      unordered_set<string> accounts_set(accounts[i].begin() + 1,
accounts[i].end());
      for (int j = i + 1; j < n; ++j)
        if (accounts[i][0] == accounts[j][0])
          for (int k = 1; !uf.connected(i, j) && k < (int)accounts[j].size();
++k)
            if (accounts_set.count(accounts[j][k]) > 0)
              uf.merge(i, j);
    }


    unordered_map<int, vector<int>> connected_components;
    for (int i = 0; i < n; ++i)
      if (connected_components.count(uf.find(i)) == 0)
        connected_components[uf.find(i)] = {i};
      else
        connected_components[uf.find(i)].push_back(i);

    vector<vector<string>> ret;
    for (auto& c : connected_components) {
      unordered_set<string> acct;
      for (auto i : c.second)
        for (int j = 1; j < (int)accounts[i].size(); ++j)
          acct.insert(accounts[i][j]);
      vector<string> sorted_acct(acct.begin(), acct.end());
      sort(sorted_acct.rbegin(), sorted_acct.rend());
      sorted_acct.push_back(accounts[c.first][0]);
```

```
        reverse(sorted_acct.begin(), sorted_acct.end());
        ret.push_back(sorted_acct);
      }
      return ret;
    }
};
```

733. Flood Fill

Description

An image is represented by a 2-D array of integers, each integer representing the pixel value of the image (from 0 to 65535).

Given a coordinate (sr, sc) representing the starting pixel (row and column) of the flood **fill**, and a pixel value newColor, "flood **fill**" the image.

To perform a "flood **fill**", consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color as the starting pixel), and so on. Replace the color of **all** of the aforementioned pixels with the newColor.

At the end, return the modified image.

Example 1:
Input:
image = [[1,1,1],[1,1,0],[1,0,1]]
sr = 1, sc = 1, newColor = 2
Output: [[2,2,2],[2,2,0],[2,0,1]]
Explanation:
From the center of the image (with position (sr, sc) = (1, 1)), all pixels connected
by a path of the same color as the starting pixel are colored with the new color.
Note the bottom corner **is** not colored 2, because **it is** not 4-directionally connected
to the starting pixel.
Note:

The length of image and image[0] will be in the range [1, 50].
The given starting pixel will satisfy 0 <= sr < image.length and 0 <= sc < image[0].length.
The value of each color in image[i][j] and newColor will be an integer in [0, 65535].

Solution

05/11/2020:

```
class Solution {
public:
  vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int
newColor) {
    if (image.empty() || image[0].empty()) return image;
    if (image[sr][sc] == newColor) return image;
    int m = image.size(), n = image[0].size();
    int oldColor = image[sr][sc];
    image[sr][sc] = newColor;
    queue<pair<int, int>> q;
    q.emplace(sr, sc);
    int dir[4][2] = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };
    while (!q.empty()) {
      int sz = q.size();
      for (int s = 0; s < sz; ++s) {
        pair<int, int> cur = q.front(); q.pop();
        int i = cur.first, j = cur.second;
        for (int d = 0; d < 4; ++d) {
          int ni = i + dir[d][0], nj = j + dir[d][1];
          if (ni >= 0 && ni < m && nj >= 0 && nj < n && image[ni][nj] ==
oldColor) {
            image[ni][nj] = newColor;
            q.emplace(ni, nj);
          }
        }
      }
    }
    return image;
  }
};
```

734. Sentence Similarity

Description

Given two sentences words1, words2 (each represented as an array of strings), and a list of similar word pairs pairs, determine if two sentences are similar.

For example, "great acting skills" and "fine drama talent" are similar, if the similar word pairs are pairs = [["great", "fine"], ["acting","drama"], ["skills","talent"]].

Note that the similarity relation is not transitive. For example, if "great" and "fine" are similar, and "fine" and "good" are similar, "great" and "good" are not necessarily similar.

However, similarity is symmetric. For example, "great" and "fine" being similar is the same as "fine" and "great" being similar.

Also, a word is always similar with itself. For example, the sentences words1 = ["great"], words2 = ["great"], pairs = [] are similar, even though there are no specified similar word pairs.

Finally, sentences can only be similar if they have the same number of words. So a sentence like words1 = ["great"] can never be similar to words2 = ["doubleplus","good"].

Note:

The length of words1 and words2 will not exceed 1000.
The length of pairs will not exceed 2000.
The length of each pairs[i] will be 2.
The length of each words[i] and pairs[i][j] will be in the range [1, 20].

Solution

06/10/2020:

```
class Solution {
public:
  bool areSentencesSimilar(vector<string>& words1, vector<string>& words2,
vector<vector<string>>& pairs) {
    if (words1.size() != words2.size()) return false;
    bool isSimilar = true;
    unordered_map<string, unordered_set<string>> similarWords;
    for (auto& p : pairs) {
      similarWords[p[0]].insert(p[1]);
      similarWords[p[1]].insert(p[0]);
    }
    for (int i = 0; i < words1.size(); ++i)
      if (words1[i] != words2[i])
        if (similarWords[words1[i]].count(words2[i]) == 0 &&
similarWords[words2[i]].count(words1[i]) == 0)
          return false;
    return true;
  }
};
```

05/20/2020:

```
class Solution {
public:
  bool areSentencesSimilar(vector<string>& words1, vector<string>& words2,
vector<vector<string>>& pairs) {
    if (words1.empty() && words2.empty()) return true;
    if (words1.size() != words2.size()) return false;
    unordered_map<string, unordered_set<string>> similarWords;
    for (auto& p : pairs) {
      if (p.empty()) continue;
      similarWords[p[0]].insert(p[1]);
      similarWords[p[1]].insert(p[0]);
    }
    int n = words1.size();
    for (int i = 0; i < n; ++i) {
      bool isSimilar = false;
      if (words1[i] == words2[i]) {
        isSimilar = true;
      } else {
        if (similarWords[words1[i]].count(words2[i]) > 0)
          isSimilar = true;
      }
      if (isSimilar == false) return false;
    }
    return true;
  }
};
```

737. Sentence Similarity II

Description

Given two sentences words1, words2 (each represented as an array of strings), and a list of similar word pairs pairs, determine if two sentences are similar.

For example, words1 = ["great", "acting", "skills"] and words2 = ["fine", "drama", "talent"] are similar, if the similar word pairs are pairs = [["great", "good"], ["fine", "good"], ["acting","drama"], ["skills","talent"]].

Note that the similarity relation is transitive. For example, if "great" and "good" are similar, and "fine" and "good" are similar, then "great" and "fine" are similar.

Similarity is also symmetric. For example, "great" and "fine" being similar is the same as "fine" and "great" being similar.

Also, a word is always similar with itself. For example, the sentences words1 = ["great"], words2 = ["great"], pairs = [] are similar, even though there are no specified similar word pairs.

Finally, sentences can only be similar if they have the same number of words. So a sentence like words1 = ["great"] can never be similar to words2 = ["doubleplus","good"].

Note:

The length of words1 and words2 will not exceed 1000.
The length of pairs will not exceed 2000.
The length of each pairs[i] will be 2.
The length of each words[i] and pairs[i][j] will be in the range [1, 20].

Solution

06/10/2020:

```
class UnionFind {
private:
  vector<int> id;
  vector<int> sz;

public:
  UnionFind(int n) {
    id.resize(n);
    iota(id.begin(), id.end(), 0);
    sz.resize(n, 1);
  }

  int find(int x) {
    if (x == id[x]) return x;
    return id[x] = find(id[x]);
  }

  bool connected(int x, int y) {
    return find(x) == find(y);
  }

  bool merge(int x, int y) {
    int i = find(x), j = find(y);
    if (i == j) return false;
    if (sz[i] > sz[j]) {
      sz[i] += sz[j];
      id[j] = i;
    } else {
      sz[j] += sz[i];
```

```
      id[i] = j;
    }
    return true;
  }
};

class Solution {
public:
  bool areSentencesSimilarTwo(vector<string>& words1, vector<string>& words2,
vector<vector<string>>& pairs) {
    if (words1.size() != words2.size()) return false;
    UnionFind uf(6000);
    unordered_map<string, int> word_id;
    int id = 0;
    for (auto& w : words1) assign_id(w, id, word_id);
    for (auto& w : words2) assign_id(w, id, word_id);
    for (auto& p : pairs) {
      assign_id(p[0], id, word_id);
      assign_id(p[1], id, word_id);
      uf.merge(word_id[p[0]], word_id[p[1]]);
    }
    for (int i = 0; i < (int)words1.size(); ++i)
      if (words1[i] != words2[i] && !uf.connected(word_id[words1[i]],
word_id[words2[i]]))
        return false;
    return true;
  }

  void assign_id(const string& word, int& id, unordered_map<string, int>&
word_id) {
    if (word_id.count(word) == 0) word_id[word] = id++;
  }
};
```

748. Shortest Completing Word

Description

Find the minimum length word from a given dictionary words, which has all the letters from the string licensePlate. Such a word is said to complete the given string licensePlate

Here, for letters we ignore case. For example, "P" on the licensePlate still matches "p" on the word.

It is guaranteed an answer exists. If there are multiple answers, return the one that occurs first in the array.

The license plate might have the same letter occurring multiple times. For example, given a licensePlate of "PP", the word "pair" does not complete the licensePlate, but the word "supper" does.

Example 1:
Input: licensePlate = "1s3 PSt", words = ["step", "steps", "stripe", "stepple"]
Output: "steps"
Explanation: The smallest length word that contains the letters "S", "P", "S", and "T".
Note that the answer is not "step", because the letter "s" must occur in the word twice.
Also note that we ignored case for the purposes of comparing whether a letter exists in the word.
Example 2:
Input: licensePlate = "1s3 456", words = ["looks", "pest", "stew", "show"]
Output: "pest"
Explanation: There are 3 smallest length words that contains the letters "s".
We return the one that occurred first.
Note:
licensePlate will be a string with length in range [1, 7].
licensePlate will contain digits, spaces, or letters (uppercase or lowercase).
words will have a length in the range [10, 1000].
Every words[i] will consist of lowercase letters, and have length in range [1, 15].

Solution

05/17/2020:

```
class Solution {
public:
  string shortestCompletingWord(string licensePlate, vector<string>& words) {
    vector<int> trimmedLicensePlate(26, 0);
    for (auto& l : licensePlate) {
      if (isalpha(l)) {
        ++trimmedLicensePlate[tolower(l) - 'a'];
      }
    }
    vector<pair<string, int>> validWords;
    for (int k = 0; k < (int)words.size(); ++k) {
      string w = words[k];
      vector<int> cnt(26, 0);
      for (auto& c : w) {
        ++cnt[c - 'a'];
      }
```

```
      bool isComplete = true;
      for (int i = 0; i < 26; ++i) {
        if (cnt[i] < trimmedLicensePlate[i]) {
          isComplete = false;
          break;
        }
      }
      if (isComplete) validWords.emplace_back(w, k);
    }
    sort(validWords.begin(), validWords.end(), [](pair<string, int>& p1,
pair<string, int>& p2) {
      if (p1.first.size() == p2.first.size()) {
        return p1.second < p2.second;
      }
      return p1.first.size() < p2.first.size();
    });
    return validWords.front().first;
  }
};
```

758. Bold Words in String

Description

Given a set of keywords words and a string S, make all appearances of all keywords in S bold. Any letters between and tags become bold.

The returned string should use the least number of tags possible, and of course the tags should form a valid combination.

For example, given that words = ["ab", "bc"] and S = "aabcd", we should return "aabcd". Note that returning "aabcd" would use more tags, so it is incorrect.

Constraints:

words has length in range [0, 50].
words[i] has length in range [1, 10].
S has length in range [0, 500].
All characters in words[i] and S are lowercase letters.
Note: This question is the same as 616: https://leetcode.com/problems/add-bold-tag-in-string/

Solution

06/09/2020:

```cpp
class Solution {
public:
  string boldWords(vector<string>& dict, string s) {
    unordered_set<string> dicts(dict.begin(), dict.end());
    vector<int> lengths;
    for (auto& d : dict) lengths.push_back(d.size());
    sort(lengths.rbegin(), lengths.rend());
    int n = s.size();
    vector<bool> isBold(n, false);
    for (int i = 0; i < n; ++i) {
      bool wrap = false;
      int wrapLength = 0;
      for (auto& len : lengths) {
        if (i + len <= n && dicts.count(s.substr(i, len)) > 0) {
          wrap = true;
          wrapLength = len;
          break;
        }
      }
      if (wrap) {
        for (int j = i; j < i + wrapLength; ++j)
          isBold[j] = true;
      }
    }
    string ret;
    int last = 0;
    for (int i = 0; i < n;) {
      if (isBold[i]) {
        while (i + 1 < n && isBold[i + 1]) ++i;
        ret += "<b>";
        ret += s.substr(last, i - last + 1);
        ret += "</b>";
        last = ++i;
      } else {
        ret += s[i];
        last = ++i;
      }
    }
    return ret;
  }
};
```

773. Sliding Puzzle

Description

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

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

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

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

Examples:

Input: board = [[1,2,3],[4,0,5]]
Output: 1
Explanation: Swap the 0 and the 5 in one move.
Input: board = [[1,2,3],[5,4,0]]
Output: -1
Explanation: No number of moves will make the board solved.
Input: board = [[4,1,2],[5,0,3]]
Output: 5
Explanation: 5 is the smallest number of moves that solves the board.
An example path:
After move 0: [[4,1,2],[5,0,3]]
After move 1: [[4,1,2],[0,5,3]]
After move 2: [[0,1,2],[4,5,3]]
After move 3: [[1,0,2],[4,5,3]]
After move 4: [[1,2,0],[4,5,3]]
After move 5: [[1,2,3],[4,5,0]]
Input: board = [[3,2,4],[1,5,0]]
Output: 14
Note:

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

Solution

05/08/2020 Discussion:

This is the C++ version solution for one of the assignments of the course Algorithms (Part 1):

```
class Board {
public:
  vector<vector<int>> board;
  pair<int, int> zeroTile;
  int rows, cols, manhattanDist, moves, priority;

  Board(const vector<vector<int>>& board) {
    this->board = board;
    rows = board.size();
    cols  = board[0].size();
    zeroTile = findZeroTile();
    manhattanDist = manhattan();
    moves = 0;
  }

  bool operator<(const Board& that) const { return this->priority <
that.priority; }

  bool operator>(const Board& that) const { return this->priority >
that.priority; }

  bool operator!=(const Board& that) const {
    for (int i = 0; i < rows; ++i)
      for (int j = 0; j < cols; ++j)
        if (that.board[i][j] != board[i][j])
          return true;
    return false;
  }

  pair<int, int> findZeroTile() {
    for (int i = 0; i < rows; ++i)
      for (int j = 0; j < cols; ++j)
        if (board[i][j] == 0)
          return {i, j};
    return {-1, -1};
  }

  int manhattan() {
    int d = 0;
    for (int i = 0; i < rows; ++i)
      for (int j = 0; j < cols; ++j)
        if (board[i][j] != 0)
          d += abs((board[i][j] - 1) / cols - i) + abs((board[i][j] - 1) % cols
- j);
    return d;
  }

  void updateMoves(int moves) {
```

```
    this->moves = moves;
    priority = manhattanDist + moves;
  }

  vector<Board> neighbor() {
    int dir[4][2] = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };
    int i = zeroTile.first, j = zeroTile.second;
    vector<Board> neighborBoards;
    for (int d = 0; d < 4; ++d) {
      int ni = dir[d][0] + i, nj = dir[d][1] + j;
      vector<vector<int>> newBoard(board);
      if (ni >= 0 && ni < rows && nj >= 0 && nj < cols) {
        swap(newBoard[i][j], newBoard[ni][nj]);
        neighborBoards.push_back(Board(newBoard));
        neighborBoards.back().updateMoves(moves + 1);
      }
    }
    return neighborBoards;
  }

  Board twin () {
    for (int i = 0; i < rows; ++i) {
      for (int j = 0; j < cols - 1; ++j) {
        if (board[i][j] != 0 && board[i][j + 1] != 0) {
          vector<vector<int>> twinBoard(board);
          swap(twinBoard[i][j], twinBoard[i][j + 1]);
          return Board(twinBoard);
        }
      }
    }
    return Board(board);
  }
};

class Solution {
public:
  int slidingPuzzle(vector<vector<int>>& board) {
    priority_queue<pair<Board, Board>, vector<pair<Board, Board>>,
greater<pair<Board, Board>>> pq, pqTwin;
    Board initialBoard(board), twinBoard = initialBoard.twin();
    pq.emplace(initialBoard, initialBoard);
    pqTwin.emplace(twinBoard, twinBoard);
    while (!pq.empty() && pq.top().first.manhattanDist > 0 && !pqTwin.empty() &&
pqTwin.top().first.manhattanDist > 0) {
      pair<Board, Board> cur = pq.top(); pq.pop();
      for(auto& p : cur.first.neighbor())
        if (p != cur.second) pq.emplace(p, cur.first);
      cur = pqTwin.top(); pqTwin.pop();
      for(auto& p : cur.first.neighbor())
```

```
            if (p != cur.second) pqTwin.emplace(p, cur.first);
        }
        return pqTwin.top().first.manhattanDist == 0 ? -1 : pq.top().first.moves;
    }
};
```

787. Cheapest Flights Within K Stops

Description

There are n cities connected by m flights. Each flight starts from city u and arrives at v with a price w.

Now given all the cities and flights, together with starting city src and the destination dst, your task is to find the cheapest price from src to dst with up to k stops. If there is no such route, output -1.

Example 1:
Input:
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
src = 0, dst = 2, k = 1
Output: 200
Explanation:
The graph looks like this:

The cheapest price from city 0 to city 2 with at most 1 stop costs 200, as marked red in the picture.

Example 2:
Input:
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
src = 0, dst = 2, k = 0
Output: 500
Explanation:
The graph looks like this:

The cheapest price from city 0 to city 2 with at most 0 stop costs 500, as marked blue in the picture.

Constraints:

The number of nodes n will be in range [1, 100], with nodes labeled from 0 to n - 1.
The size of flights will be in range [0, n * (n - 1) / 2].

The format of each flight will be (src, dst, price).
The price of each flight will be in the range [1, 10000].
k is in the range of [0, n - 1].
There will not be any duplicated flights or self cycles.

Solution

06/14/2020:

```
typedef tuple<int, int, int> ti;
class Solution {
public:
  int findCheapestPrice(int n, vector<vector<int>>& flights, int src, int dst,
int K) {
    vector<vector<pair<int, int>>> adj(n);
    for(const auto& f : flights) adj[f[0]].emplace_back(f[1], f[2]);
    priority_queue<ti, vector<ti>, greater<ti>> pq;
    pq.emplace(0, src, K + 1);
    while(!pq.empty()) {
      auto [cost, u, stops] = pq.top(); pq.pop();
      if(u == dst) return cost;
      if(!stops) continue;
      for(auto& to : adj[u]) {
        auto [v, w] = to;
        pq.emplace(cost + w, v, stops - 1);
      }
    }
    return -1;
  }
};
```

819. Most Common Word

Description

Given a paragraph and a list of banned words, return the most frequent word that is not in the list of banned words. It is guaranteed there is at least one word that isn't banned, and that the answer is unique.

Words in the list of banned words are given in lowercase, and free of punctuation. Words in the paragraph are not case sensitive. The answer is in lowercase.

Example:

Input:
paragraph = "Bob hit a ball, the hit BALL flew far after it was hit."
banned = ["hit"]
Output: "ball"
Explanation:
"hit" occurs 3 times, but it is a banned word.
"ball" occurs twice (and no other word does), so it is the most frequent non-banned word in the paragraph.
Note that words in the paragraph are not case sensitive,
that punctuation is ignored (even if adjacent to words, such as "ball,"),
and that "hit" isn't the answer even though it occurs more because it is banned.

Note:

1 <= paragraph.length <= 1000.
0 <= banned.length <= 100.
1 <= banned[i].length <= 10.
The answer is unique, and written in lowercase (even if its occurrences in paragraph may have uppercase symbols, and even if it is a proper noun.)
paragraph only consists of letters, spaces, or the punctuation symbols !?',;.
There are no hyphens or hyphenated words.
Words only consist of letters, never apostrophes or other punctuation symbols.

Solution

05/10/2020:

```
class Solution {
public:
  string mostCommonWord(string paragraph, vector<string>& banned) {
    for (auto& c : paragraph) c = isalpha(c) ? tolower(c) : ' ';
    unordered_set<string> b(banned.begin(), banned.end());
    unordered_map<string, int> mp;
    string ret, s;
    int freq = 0;
    istringstream iss(paragraph);
    while (iss >> s) {
      if (b.count(s) == 0 && ++mp[s] > freq) {
        freq = mp[s];
        ret = s;
      }
    }
    return ret;
  }
};
```

844. Backspace String Compare

Description

Given two strings S and T, return if they are equal when both are typed into empty text editors. # means a backspace character.

Example 1:

Input: S = "ab#c", T = "ad#c"
Output: true
Explanation: Both S and T become "ac".
Example 2:

Input: S = "ab##", T = "c#d#"
Output: true
Explanation: Both S and T become "".
Example 3:

Input: S = "a##c", T = "#a#c"
Output: true
Explanation: Both S and T become "c".
Example 4:

Input: S = "a#c", T = "b"
Output: false
Explanation: S becomes "c" while T becomes "b".
Note:

1 <= S.length <= 200
1 <= T.length <= 200
S and T only contain lowercase letters and '#' characters.
Follow up:

Can you solve it in O(N) time and O(1) space?

Solution

02/05/2020:

```
class Solution {
public:
  bool backspaceCompare(string S, string T) {
    string s, t;
    for (auto& c : S) {
      if (c == '#') {
        if (!s.empty()) {
```

```
            s.pop_back();
          }
        } else {
          s.push_back(c);
        }
      }
      for (auto& c : T) {
        if (c == '#') {
          if (!t.empty()) {
            t.pop_back();
          }
        } else {
          t.push_back(c);
        }
      }
      return s == t;
    }
};
```

868. Binary Gap

Description

Given a positive integer N, find and return the longest distance between two consecutive 1's in the binary representation of N.

If there aren't two consecutive 1's, return 0.

Example 1:

Input: 22
Output: 2
Explanation:
22 in binary is 0b10110.
In the binary representation of 22, there are three ones, and two consecutive pairs of 1's.
The first consecutive pair of 1's have distance 2.
The second consecutive pair of 1's have distance 1.
The answer is the largest of these two distances, which is 2.
Example 2:

Input: 5
Output: 2
Explanation:

5 in binary is 0b101.
Example 3:

Input: 6
Output: 1
Explanation:
6 in binary is 0b110.
Example 4:

Input: 8
Output: 0
Explanation:
8 in binary is 0b1000.
There aren't any consecutive pairs of 1's in the binary representation of 8, so we return 0.

Note:

1 <= N <= 10^9

Solution

05/10/2020:

```
class Solution {
public:
  int binaryGap(int N) {
    if (N == 0) return 0;
    int ret = INT_MIN;
    while ((N & 1) == 0) {
      N >>= 1;
    }
    int cnt = 0;
    while (N != 0) {
      N >>= 1;
      if ((N & 1) == 1) {
        ret = max(ret, cnt);
        cnt = 0;
      } else {
        ++cnt;
      }
    }
    return ret == INT_MIN ? 0 : ret + 1;
  }
};
```

917. Reverse Only Letters

Description

Given a string S, return the "reversed" string where all characters that are not a letter stay in the same place, and all letters reverse their positions.

Example 1:

Input: "ab-cd"
Output: "dc-ba"
Example 2:

Input: "a-bC-dEf-ghIj"
Output: "j-Ih-gfE-dCba"
Example 3:

Input: "Test1ng-Leet=code-Q!"
Output: "Qedo1ct-eeLg=ntse-T!"

Note:

S.length <= 100
33 <= S[i].ASCIIcode <= 122
S doesn't contain \ or "

Solution

05/11/2020:

```
class Solution {
public:
  string reverseOnlyLetters(string S) {
    int n = S.size(), l = 0, r = n - 1;
    while (l < r) {
      if (isalpha(S[l]) && isalpha(S[r])) {
        swap(S[l++], S[r--]);
      } else if (!isalpha(S[l])) {
        ++l;
      } else { // if (!isalpha(S[r]))
        --r;
      }
    }
    return S;
```

```
  }
};
```

```
class Solution {
public:
  string reverseOnlyLetters(string S) {
    int n = S.size(), l = 0, r = n - 1;
    string ret(S);
    while (l < n && r >= 0) {
      for (; l < n && !isalpha(ret[l]); ++l);
      for (; r >= 0 && !isalpha(S[r]); --r);
      if (l < n && r >= 0) ret[l++] = S[r--];
    }
    return ret;
  }
};
```

973. K Closest Points to Origin

Description

We have a list of points on the plane. Find the K closest points to the origin (0, 0).

(Here, the distance between two points on a plane is the Euclidean distance.)

You may return the answer in any order. The answer is guaranteed to be unique (except for the order that it is in.)

Example 1:

Input: points = [[1,3],[-2,2]], K = 1
Output: [[-2,2]]
Explanation:
The distance between (1, 3) and the origin is sqrt(10).
The distance between (-2, 2) and the origin is sqrt(8).
Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.
We only want the closest K = 1 points from the origin, so the answer is just[[-2,2]].
Example 2:

Input: points = [[3,3],[5,-1],[-2,4]], K = 2
Output: [[3,3],[-2,4]]

(The answer [[-2,4],[3,3]] would also be accepted.)

Note:

```
1 <= K <= points.length <= 10000
-10000 < points[i][0] < 10000
-10000 < points[i][1] < 10000
```

Solution

05/28/2020:

```
class Solution {
public:
  vector<vector<int>> kClosest(vector<vector<int>>& points, int K) {
    priority_queue<pair<double, vector<int>>, vector<pair<double, vector<int>>>,
greater<pair<double, vector<int>>>> q;
    for (auto& p : points) q.push({hypot(p[0], p[1]), p});
    vector<vector<int>> ret;
    while (K-- > 0) {
      pair<double, vector<int>> cur = q.top(); q.pop();
      ret.push_back(cur.second);
    }
    return ret;
  }
};
```

986. Interval List Intersections

Description

Given two lists of closed intervals, each list of intervals is pairwise disjoint and in sorted order.

Return the intersection of these two interval lists.

(Formally, a closed interval [a, b] (with a <= b) denotes the set of real numbers x with a <= x <= b. The intersection of two closed intervals is a set of real numbers that is either empty, or can be represented as a closed interval. For example, the intersection of [1, 3] and [2, 4] is [2, 3].)

Example 1:

Input: A = [[0,2],[5,10],[13,23],[24,25]], B = [[1,5],[8,12],[15,24],[25,26]]
Output: [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]
Reminder: The inputs and the desired output are lists of Interval objects, and not arrays or lists.

Note:

0 <= A.length < 1000
0 <= B.length < 1000
0 <= A[i].start, A[i].end, B[i].start, B[i].end < 10^9
NOTE: input types have been changed on April 15, 2019. Please reset to default code definition to get new method signature.

Solution

05/23/2020:

```
class Solution {
public:
  vector<vector<int>> intervalIntersection(vector<vector<int>>& A,
vector<vector<int>>& B) {
    auto ia = A.begin();
    auto ib = B.begin();
    vector<vector<int>> ret;
    for (; ia != A.end() && ib != B.end();) {
      if ((*ia)[1] <= (*ib)[1]) {
        if ((*ib)[0] <= (*ia)[1]) {
          ret.push_back({max((*ia)[0], (*ib)[0]), (*ia)[1]});
        }
        ++ia;
      } else {
        if ((*ib)[1] >= (*ia)[0]) {
          ret.push_back({max((*ia)[0], (*ib)[0]), (*ib)[1]});
        }
        ++ib;
      }
    }
    return ret;
  }
};
```

988. Smallest String Starting From Leaf

Description

Given the root of a binary tree, each node has a value from 0 to 25 representing the letters 'a' to 'z': a value of 0 represents 'a', a value of 1 represents 'b', and so on.

Find the lexicographically smallest string that starts at a leaf of this tree and ends at the root.

(As a reminder, any shorter prefix of a string is lexicographically smaller: for example, "ab" is lexicographically smaller than "aba". A leaf of a node is a node that has no children.)

Example 1:

Input: [0,1,2,3,4,3,4]
Output: "dba"
Example 2:

Input: [25,1,3,1,3,0,2]
Output: "adz"
Example 3:

Input: [2,2,1,null,1,0,null,0]
Output: "abc"

Note:

The number of nodes in the given tree will be between 1 and 8500.
Each node in the tree will have a value between 0 and 25.

Solution

06/15/2020:

```
/**
```

```
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
  string smallestFromLeaf(TreeNode* root) {
    if (!root) return "";
    string ret(8500, 'z'), s;
    backtrack(s, ret, root);
    return ret;
  }

  void backtrack(string& s, string& ret, TreeNode* root) {
    if (!root) return;
    s.push_back(root->val + 'a');
    if (!root->left && !root->right) ret = min(string(s.rbegin(), s.rend()),
ret);
    backtrack(s, ret, root->left);
    backtrack(s, ret, root->right);
    s.pop_back();
  }
};
```

997. Find the Town Judge

Description

In a town, there are N people labelled from 1 to N. There is a rumor that one of these people is secretly the town judge.

If the town judge exists, then:

The town judge trusts nobody.
Everybody (except for the town judge) trusts the town judge.
There is exactly one person that satisfies properties 1 and 2.
You are given trust, an array of pairs trust[i] = [a, b] representing that the person labelled a trusts the person labelled b.

If the town judge exists and can be identified, return the label of the town judge. Otherwise, return -1.

Example 1:

Input: N = 2, trust = [[1,2]]
Output: 2
Example 2:

Input: N = 3, trust = [[1,3],[2,3]]
Output: 3
Example 3:

Input: N = 3, trust = [[1,3],[2,3],[3,1]]
Output: -1
Example 4:

Input: N = 3, trust = [[1,2],[2,3]]
Output: -1
Example 5:

Input: N = 4, trust = [[1,3],[1,4],[2,3],[2,4],[4,3]]
Output: 3

Note:

1 <= N <= 1000
trust.length <= 10000
trust[i] are all different
trust[i][0] != trust[i][1]
1 <= trust[i][0], trust[i][1] <= N

Solution

05/10/2020:

```
class Solution {
public:
  int findJudge(int N, vector<vector<int>>& trust) {
    if (N == 1 && trust.empty()) return 1;
    if (trust.empty() || trust[0].empty()) return -1;
    unordered_map<int, vector<int>> r;
    unordered_set<int> truster;
    for (auto& t : trust) {
      r[t[1]].push_back(t[0]);
```

```
      truster.insert(t[0]);
    }
    int judge = -1;
    int cnt = 0;
    for (auto& s : r) {
      if ((int)s.second.size() == N - 1 && truster.count(s.first) == 0) {
        ++cnt;
        judge = s.first;
      }
    }
    return cnt == 1 ? judge : -1;
  }
};
```

1029. Two City Scheduling

Description

There are 2N people a company is planning to interview. The cost of flying the i-th person to city A is costs[i][0], and the cost of flying the i-th person to city B is costs[i][1].

Return the minimum cost to fly every person to a city such that exactly N people arrive in each city.

Example 1:

Input: [[10,20],[30,200],[400,50],[30,20]]
Output: 110
Explanation:
The first person goes to city A for a cost of 10.
The second person goes to city A for a cost of 30.
The third person goes to city B for a cost of 50.
The fourth person goes to city B for a cost of 20.

The total minimum cost is 10 + 30 + 50 + 20 = 110 to have half the people interviewing in each city.

Note:

1 <= costs.length <= 100
It is guaranteed that costs.length is even.
1 <= costs[i][0], costs[i][1] <= 1000

Solution

06/03/2020:

Backtracking (TLE):

```
class Solution {
public:
  int ret, numOfCityA;
  int twoCitySchedCost(vector<vector<int>>& costs) {
    int n = costs.size(), totalCost = 0;
    ret = INT_MAX, numOfCityA = 0;
    backtrack(0, n, numOfCityA, totalCost, costs);
    return ret;
  }

  void backtrack(int k, int n, int numOfCityA, int totalCost,
vector<vector<int>>& costs) {
    if (k == n) {
      if (numOfCityA == n / 2) ret = min(ret, totalCost);
      return;
    }
    if (numOfCityA > n / 2) return;
    backtrack(k + 1, n, numOfCityA + 1, totalCost + costs[k][0], costs);
    backtrack(k + 1, n, numOfCityA, totalCost + costs[k][1], costs);
  }
};
```

Dynamic Programming:

```
class Solution {
public:
  int twoCitySchedCost(vector<vector<int>>& costs) {
    int n = costs.size() / 2;
    vector<vector<int>> dp(2 * n + 1, vector<int>(n + 1, INT_MAX));
    dp[0][0] = 0;
    // dp[i][j]: the total cost among costs[0..i - 1]: pick j to city A.
    // dp[i][j] = min(dp[i - 1][j - 1] + costs[i - 1][0], dp[i - 1][j] + costs[i
- 1][1])
    for (int i = 1; i <= 2 * n; ++i) {
      for (int j = 0; j <= i && j <= n; ++j) {
        if (i - j > n) continue;
        if (dp[i - 1][j] != INT_MAX) {
          dp[i][j] = min(dp[i][j], dp[i - 1][j] + costs[i - 1][1]);
        }
        if (j > 0 && dp[i - 1][j - 1] != INT_MAX) {
          dp[i][j] = min(dp[i][j], dp[i - 1][j - 1] + costs[i - 1][0]);
```

```
        }
      }
    }
    return dp[2 * n][n];
  }
};
```

Greedy:

```
class Solution {
public:
  int twoCitySchedCost(vector<vector<int>>& costs) {
    sort(costs.begin(), costs.end(), [](vector<int>& nums1, vector<int>& nums2)
{
      return nums1[0] - nums1[1] < nums2[0] - nums2[1];
    });
    int n = costs.size(), ret = 0;
    for (int i = 0; i < n; ++i) ret += i < n / 2 ? costs[i][0] : costs[i][1];
    return ret;
  }
};
```

1057. Campus Bikes

Description

On a campus represented as a 2D grid, there are N workers and M bikes, with N <= M. Each worker and bike is a 2D coordinate on this grid.

Our goal is to assign a bike to each worker. Among the available bikes and workers, we choose the (worker, bike) pair with the shortest Manhattan distance between each other, and assign the bike to that worker. (If there are multiple (worker, bike) pairs with the same shortest Manhattan distance, we choose the pair with the smallest worker index; if there are multiple ways to do that, we choose the pair with the smallest bike index). We repeat this process until there are no available workers.

The Manhattan distance between two points p1 and p2 is Manhattan(p1, p2) = |p1.x - p2.x| + |p1.y - p2.y|.

Return a vector ans of length N, where ans[i] is the index (0-indexed) of the bike that the i-th worker is assigned to.

Example 1:

Input: workers = [[0,0],[2,1]], bikes = [[1,2],[3,3]]
Output: [1,0]
Explanation:
Worker 1 grabs Bike 0 as they are closest (without ties), and Worker 0 is assigned Bike 1. So the output is [1, 0].
Example 2:

Input: workers = [[0,0],[1,1],[2,0]], bikes = [[1,0],[2,2],[2,1]]
Output: [0,2,1]
Explanation:
Worker 0 grabs Bike 0 at first. Worker 1 and Worker 2 share the same distance to Bike 2, thus Worker 1 is assigned to Bike 2, and Worker 2 will take Bike 1. So the output is [0,2,1].

Note:

0 <= workers[i][j], bikes[i][j] < 1000
All worker and bike locations are distinct.
1 <= workers.length <= bikes.length <= 1000

Solution

06/11/2020: Using bucket-sort:

```
class Solution {
public:
  vector<int> assignBikes(vector<vector<int>>& workers, vector<vector<int>>&
bikes) {
    vector<vector<pair<int, int>>> bucket(2001);
    int m = workers.size(), n = bikes.size();
    for (int i = 0; i < m; ++i)
      for (int j = 0; j < n; ++j)
        bucket[abs(workers[i][0] - bikes[j][0]) + abs(workers[i][1] - bikes[j]
[1])].emplace_back(i, j);
    vector<bool> chosen_workers(m, false), chosen_bikes(n, false);
    vector<int> ret(m, -1);
    for (int i = 0; i < 2001; ++i) {
      for (int j = 0; j < (int)bucket[i].size(); ++j) {
        int w = bucket[i][j].first, b = bucket[i][j].second;
        if (!chosen_workers[w] && !chosen_bikes[b]) {
          ret[w] = b;
```

```
            chosen_workers[w] = chosen_bikes[b] = true;
          }
        }
      }
      return ret;
    }
};
```

Using sort:

```
class Solution {
public:
  vector<int> assignBikes(vector<vector<int>>& workers, vector<vector<int>>&
bikes) {
    int m = workers.size(), n = bikes.size();
    vector<pair<int, pair<int, int>>> dist;
    for (int i = 0; i < m; ++i)
      for (int j = 0; j < n; ++j)
        dist.push_back({abs(workers[i][0] - bikes[j][0]) + abs(workers[i][1] -
bikes[j][1]), {i, j}});
    sort(dist.begin(), dist.end());
    vector<bool> chosen_workers(m, false), chosen_bikes(n, false);
    vector<int> ret(m, -1);
    for (auto& d : dist) {
      int i = d.second.first, j = d.second.second;
      if (!chosen_workers[i] && !chosen_bikes[j]) {
        ret[i] = j;
        chosen_workers[i] = chosen_bikes[j] = true;
      }
    }
    return ret;
  }
};
```

Using prioirty_queue (much slower than sort):

```
class Solution {
public:
  vector<int> assignBikes(vector<vector<int>>& workers, vector<vector<int>>&
bikes) {
    int m = workers.size(), n = bikes.size();
    priority_queue<pair<int, pair<int, int>>, vector<pair<int, pair<int, int>>>,
greater<pair<int, pair<int, int>>>> pq;
    for (int i = 0; i < m; ++i)
      for (int j = 0; j < n; ++j)
        pq.push({abs(workers[i][0] - bikes[j][0]) + abs(workers[i][1] - bikes[j]
[1]), {i, j}});
```

```
    vector<bool> chosen_workers(m, false), chosen_bikes(n, false);
    vector<int> ret(m, -1);
    while (!pq.empty()) {
      pair<int, pair<int, int>> p = pq.top(); pq.pop();
      int i = p.second.first, j = p.second.second;
      if (!chosen_workers[i] && !chosen_bikes[j]) {
        ret[i] = j;
        chosen_workers[i] = chosen_bikes[j] = true;
      }
    }
    return ret;
  }
};
```

1065. Index Pairs of a String

Description

Given a text string and words (a list of strings), return all index pairs [i, j] so that the substring text[i]...text[j] is in the list of words.

Example 1:

Input: text = "thestoryofleetcodeandme", words = ["story","fleet","leetcode"]
Output: [[3,7],[9,13],[10,17]]
Example 2:

Input: text = "ababa", words = ["aba","ab"]
Output: [[0,1],[0,2],[2,3],[2,4]]
Explanation:
Notice that matches can overlap, see "aba" is found in [0,2] and [2,4].

Note:

All strings contains only lowercase English letters.
It's guaranteed that all strings in words are different.
1 <= text.length <= 100
1 <= words.length <= 20
1 <= words[i].length <= 50
Return the pairs [i,j] in sorted order (i.e. sort them by their first coordinate in case of ties sort them by their second coordinate).

Solution

05/10/2020:

```
class Solution {
public:
  vector<vector<int>> indexPairs(string text, vector<string>& words) {
    vector<vector<int>> ret;
    int n = text.size();
    for (auto& w : words) {
      int m = w.size();
      if (m > n) continue;
      for (int i = 0; i < n - m + 1; ++i) {
        if (text.substr(i, m) == w) {
          ret.push_back({i, i + m - 1});
        }
      }
    }
    sort(ret.begin(), ret.end());
    return ret;
  }
};
```

1066. Campus Bikes II

Description

On a campus represented as a 2D grid, there are N workers and M bikes, with N <= M. Each worker and bike is a 2D coordinate on this grid.

We assign one unique bike to each worker so that the sum of the Manhattan distances between each worker and their assigned bike is minimized.

The Manhattan distance between two points p1 and p2 is Manhattan(p1, p2) = |p1.x - p2.x| + |p1.y - p2.y|.

Return the minimum possible sum of Manhattan distances between each worker and their assigned bike.

Example 1:

Input: workers = [[0,0],[2,1]], bikes = [[1,2],[3,3]]

Output: 6
Explanation:
We assign bike 0 to worker 0, bike 1 to worker 1. The Manhattan distance of both assignments is 3, so the output is 6.
Example 2:

Input: workers = [[0,0],[1,1],[2,0]], bikes = [[1,0],[2,2],[2,1]]
Output: 4
Explanation:
We first assign bike 0 to worker 0, then assign bike 1 to worker 1 or worker 2, bike 2 to worker 2 or worker 1. Both assignments lead to sum of the Manhattan distances as 4.

Note:

0 <= workers[i][0], workers[i][1], bikes[i][0], bikes[i][1] < 1000
All worker and bike locations are distinct.
1 <= workers.length <= bikes.length <= 10

Solution

06/11/2020:

Using Dynamic Programming:

```
class Solution {
public:
  int assignBikes(vector<vector<int>>& workers, vector<vector<int>>& bikes) {
    function dist = [](vector<int>& worker, vector<int>& bike) {
      return abs(worker[0] - bike[0]) + abs(worker[1] - bike[1]);
    };
    int m = workers.size(), n = bikes.size();
    vector<vector<int>> dp(m + 1, vector<int>(1 << n, INT_MAX - 4000));
    dp[0][0] = 0;
    int minDist = INT_MAX;
    for (int i = 1; i <= m; ++i) {
      for (int s = 1; s < (1 << n); ++s) {
        for (int j = 0; j < n; ++j) {
          if ((s & (1 << j)) != 0) {
            int prev = s ^ (1 << j);
            dp[i][s] = min(dp[i][s], dp[i - 1][prev] + dist(workers[i - 1],
bikes[j]));
            minDist = i == m ? min(minDist, dp[i][s]) : minDist;
          }
        }
```

```
        }
      }
      return minDist;
    }
};
```

Using backtracking (TLE):

```
class Solution {
public:
  int minDist = INT_MAX;
  vector<bool> chosen_bikes;
  int assignBikes(vector<vector<int>>& workers, vector<vector<int>>& bikes) {
    int m = workers.size(), n = bikes.size(), dist = 0;
    chosen_bikes.assign(1000, false);
    backtrack(0, dist, workers, bikes);
    return minDist;
  }
  void backtrack(int i, int dist, vector<vector<int>>& workers,
vector<vector<int>>& bikes) {
    if (i == workers.size()) {
      minDist = min(dist, minDist);
      return;
    }
    if (dist > minDist) return;  // pruning
    for (int j = 0; j < (int)bikes.size(); ++j) {
      if (chosen_bikes[j] == false) {
        chosen_bikes[j] = true;
        backtrack(i + 1, dist + abs(workers[i][0] - bikes[j][0]) +
abs(workers[i][1] - bikes[j][1]), workers, bikes);
        chosen_bikes[j] = false;
      }
    }
  }
};
```

1143. Longest Common Subsequence

Description

Given two strings text1 and text2, return the length of their longest common subsequence.

A subsequence of a string is a new string generated from the original string with some characters(can be none) deleted without changing the relative order of the remaining characters. (eg, "ace" is a subsequence of "abcde" while "aec" is not). A common subsequence of two strings is a subsequence that is common to both strings.

If there is no common subsequence, return 0.

Example 1:

Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.
Example 2:

Input: text1 = "abc", text2 = "abc"
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.
Example 3:

Input: text1 = "abc", text2 = "def"
Output: 0
Explanation: There is no such common subsequence, so the result is 0.

Constraints:

1 <= text1.length <= 1000
1 <= text2.length <= 1000
The input strings consist of lowercase English characters only.

Solution

Discussion:

```cpp
class Solution {
public:
  int longestCommonSubsequence(string text1, string text2) {
    int m = text1.size(), n = text2.size();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 0; i < m; ++i)
      for (int j = 0; j < n; ++j)
        dp[i + 1][j + 1] = text1[i] == text2[j] ? dp[i][j] + 1 : max(dp[i][j + 1], dp[i + 1][j]);
    return dp.back().back();
  }
};
```

```cpp
class Solution {
public:
  int longestCommonSubsequence(string text1, string text2) {
    int m = text1.size(), n = text2.size();
    vector<int> dp(n + 1, 0);
    for (int i = 0; i < m; ++i) {
      vector<int> tmp(dp);
      for (int j = 0; j < n; ++j) {
        dp[j + 1] = text1[i] == text2[j] ? tmp[j] + 1 : max(dp[j + 1], dp[j]);
      }
    }
    return dp.back();
  }
};
```

1170. Compare Strings by Frequency of the Smallest Character

Description

Let's define a function f(s) over a non-empty string s, which calculates the frequency of the smallest character in s. For example, if s = "dcce" then f(s) = 2 because the smallest character is "c" and its frequency is 2.

Now, given string arrays queries and words, return an integer array answer, where each answer[i] is the number of words such that f(queries[i]) < f(W), where W is a word in words.

Example 1:

Input: queries = ["cbd"], words = ["zaaaz"]
Output: [1]
Explanation: On the first query we have f("cbd") = 1, f("zaaaz") = 3 so f("cbd") < f("zaaaz").
Example 2:

Input: queries = ["bbb","cc"], words = ["a","aa","aaa","aaaa"]
Output: [1,2]
Explanation: On the first query only f("bbb") < f("aaaa"). On the second query both f("aaa") and f("aaaa") are both > f("cc").

Constraints:

1 <= queries.length <= 2000
1 <= words.length <= 2000
1 <= queries[i].length, words[i].length <= 10
queries[i][j], words[i][j] are English lowercase letters.

Solution

02/05/2020:

```
class Solution {
public:
  vector<int> numSmallerByFrequency(vector<string>& queries, vector<string>&
words) {
    int n = queries.size(), N = 11;
    vector<int> ret(n, 0), cnt(N, 0);
    for (auto& w : words) ++cnt[f(w) - 1];
    for (int i = N - 2; i > -1; --i) cnt[i] += cnt[i + 1];
    for (int i = 0; i < n; ++i) ret[i] = cnt[f(queries[i])];
    return ret;
  }

  int f(string& s) {
    char ch = *min_element(s.begin(), s.end());
    return count(s.begin(), s.end(), ch);
  }
};
```

1277. Count Square Submatrices with All Ones

Description

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

Example 1:

Input: matrix =
[
  [0,1,1,1],
  [1,1,1,1],
  [0,1,1,1]
]
Output: 15
Explanation:
There are 10 squares of side 1.
There are 4 squares of side 2.
There is 1 square of side 3.
Total number of squares = 10 + 4 + 1 = 15.
Example 2:

Input: matrix =
[
  [1,0,1],
  [1,1,0],
  [1,1,0]
]
Output: 7
Explanation:
There are 6 squares of side 1.
There is 1 square of side 2.
Total number of squares = 6 + 1 = 7.

Constraints:

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

Solution

01/14/2020 (Dynamic Programming):

```
class Solution {
public:
```

```
  int countSquares(vector<vector<int>>& matrix) {
    int m = matrix.size(), n = matrix[0].size(), ret = 0;;
    vector<vector<int>> dp(m, vector<int>(n, 0));
    for (int i = 0; i < m; ++i) {
      for (int j = 0; j < n; ++j) {
        if (i == 0 || j == 0)
          dp[i][j] = matrix[i][j];
        else
          dp[i][j] = matrix[i][j] == 0 ? 0 : min(min(dp[i - 1][j - 1], dp[i - 1]
[j]), dp[i][j - 1]) + 1;
        ret += dp[i][j];
      }
    }
    return ret;
  }
};
```

```
class Solution {
public:
  int countSquares(vector<vector<int>>& matrix) {
    if (matrix.empty() || matrix[0].empty()) return 0;
    int m = matrix.size(), n = matrix[0].size();
    int ret = 0;
    for (int i = 0; i < m; ++i) {
      for (int j = 0; j < n; ++j) {
        if (i >= 1 && j >= 1 && matrix[i][j] == 1)
          matrix[i][j] = min(min(matrix[i - 1][j], matrix[i][j - 1]), matrix[i -
1][j - 1]) + 1;
        ret += matrix[i][j];
      }
    }
    return ret;
  }
};
```

1287. Element Appearing More Than 25% In Sorted Array

Description

Given an integer array sorted in non-decreasing order, there is exactly one integer in the array that occurs more than 25% of the time.

Return that integer.

Example 1:

Input: arr = [1,2,2,6,6,6,6,7,10]
Output: 6

Constraints:

1 <= arr.length <= 10^4
0 <= arr[i] <= 10^5

Solution

05/10/2020:

```
class Solution {
public:
  int findSpecialInteger(vector<int>& arr) {
    int n = arr.size();
    int m = (n + 1) / 4;
    for (int i = 0; i < n - m; ++i) {
      if (arr[i] == arr[i + m]) {
        return arr[i];
      }
    }
    return -1;
  }
};
```

1314. Matrix Block Sum

Description

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
```

Solution

01/14/2020 (Definition):

```
class Solution {
public:
  vector<vector<int>> matrixBlockSum(vector<vector<int>>& mat, int K) {
    int m = mat.size(), n = mat[0].size();
    vector<vector<int>> ret(m, vector<int>(n, 0));
    for (int i = 0; i < m; ++i) {
      for (int j = 0; j < n; ++j) {
        for (int k = -K; k <= K; ++k) {
          for (int l = -K; l <= K; ++l) {
            if (i + k >= 0 && i + k < m && j + l >= 0 && j + l < n) {
              ret[i][j] += mat[i + k][j + l];
            }
          }
        }
      }
    }
    return ret;
  }
};
```

01/14/2020 (Dynamic Programming):

```
class Solution {
public:
  vector<vector<int>> matrixBlockSum(vector<vector<int>>& mat, int K) {
    int m = mat.size(), n = mat[0].size();
    vector<vector<int>> dp(m, vector<int>(n, 0));
    vector<vector<int>> ret(m, vector<int>(n, 0));
    dp[0][0] = mat[0][0];
    for (int i = 1; i < m; ++i) dp[i][0] += dp[i - 1][0] + mat[i][0];
    for (int j = 1; j < n; ++j) dp[0][j] += dp[0][j - 1] + mat[0][j];
    for (int i = 1; i < m; ++i)
```

```
      for (int j = 1; j < n; ++j)
        dp[i][j] = mat[i][j] + dp[i][j - 1] + dp[i - 1][j] - dp[i - 1][j - 1];
    for (int i = 0; i < m; ++i) {
      for (int j = 0; j < n; ++j) {
        int r1 = max(i - K, 0);
        int c1 = max(j - K, 0);
        int r2 = min(i + K, m - 1);
        int c2 = min(j + K, n - 1);
        ret[i][j] = dp[r2][c2];
        if (r1 > 0) ret[i][j] -= dp[r1 - 1][c2];
        if (c1 > 0) ret[i][j] -= dp[r2][c1 - 1];
        if (r1 > 0 && c1 > 0) ret[i][j] += dp[r1 - 1][c1 - 1];
      }
    }
    return ret;
  }
};
```

1331. Rank Transform of an Array

Description

Given an array of integers arr, replace each element with its rank.

The rank represents how large the element is. The rank has the following rules:

Rank is an integer starting from 1.
The larger the element, the larger the rank. If two elements are equal, their rank must be the same.
Rank should be as small as possible.

Example 1:

Input: arr = [40,10,20,30]
Output: [4,1,2,3]
Explanation: 40 is the largest element. 10 is the smallest. 20 is the second smallest. 30 is the third smallest.
Example 2:

Input: arr = [100,100,100]
Output: [1,1,1]
Explanation: Same elements share the same rank.
Example 3:

Input: arr = [37,12,28,9,100,56,80,5,12]

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

Constraints:

0 <= arr.length <= 105
-109 <= arr[i] <= 109

Solution

05/10/2020:

```
class Solution {
public:
  vector<int> arrayRankTransform(vector<int>& arr) {
    if (arr.empty()) return {};
    set<int> keys(arr.begin(), arr.end());
    unordered_map<int, int> mp;
    int i = 1;
    for (auto& k : keys) {
      mp[k] = i++;
    }
    vector<int> ret;
    for (auto& a : arr) {
      ret.push_back(mp[a]);
    }
    return ret;
  }
};
```

1422. Maximum Score After Splitting a String

Description

Given a string s of zeros and ones, return the maximum score after splitting the string into two non-empty substrings (i.e. left substring and right substring).

The score after splitting a string is the number of zeros in the left substring plus the number of ones in the right substring.

Example 1:

Input: s = "011101"

Output: 5
Explanation:
All possible ways of splitting s into two non-empty substrings are:
left = "0" and right = "11101", score = 1 + 4 = 5
left = "01" and right = "1101", score = 1 + 3 = 4
left = "011" and right = "101", score = 1 + 2 = 3
left = "0111" and right = "01", score = 1 + 1 = 2
left = "01110" and right = "1", score = 2 + 1 = 3

Example 2:

Input: s = "00111"
Output: 5
Explanation: When left = "00" and right = "111", we get the maximum score = 2 + 3 = 5

Example 3:

Input: s = "1111"
Output: 3

Constraints:

2 <= s.length <= 500
The string s consists of characters '0' and '1' only.

Solution

04/25/2020:

```
class Solution {
public:
  int maxScore(string s) {
    int zeros = 0, ones = 0;
    for (auto& c : s) {
      if (c == '0') {
        ++zeros;
      } else {
        ++ones;
      }
    }
    int n = s.size();
    int cur_zeros = 0;
    int ret = 0;
    for (int i = 0; i < n - 1; ++i) {
      if (s[i] == '0') {
        ++cur_zeros;
      }
      ret = max(ret, cur_zeros + (n - (i + 1) - (zeros - cur_zeros)));
```

```
    }
    return ret;
  }
};
```

1424. Diagonal Traverse II

Description

Given a list of lists of integers, nums, return all elements of nums in diagonal order as shown in the below images.

Example 1:

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

Example 2:

Input: nums = [[1,2,3,4,5],[6,7],[8],[9,10,11],[12,13,14,15,16]]
Output: [1,6,2,8,7,3,9,4,12,10,5,13,11,14,15,16]

Example 3:

Input: nums = [[1,2,3],[4],[5,6,7],[8],[9,10,11]]
Output: [1,4,2,5,3,8,6,9,7,10,11]

Example 4:

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

Constraints:

1 <= nums.length <= 10^5
1 <= nums[i].length <= 10^5
1 <= nums[i][j] <= 10^9
There at most 10^5 elements in nums.

Solution

04/25/2020:

```cpp
class Solution {
public:
  vector<int> findDiagonalOrder(vector<vector<int>>& nums) {
    int m = nums.size();
    if (m <= 0) return {};
    map<int, vector<int>> mp;
    for (int i = 0; i < m; ++i) {
      int n = nums[i].size();
      for (int j = 0; j < n; ++j) {
        mp[i + j].push_back(nums[i][j]);
      }
    }
    vector<int> ret;
    for (auto& m : mp) {
      ret.insert(ret.end(), m.second.rbegin(), m.second.rend());
    }
    return ret;
  }
};
```

1438. Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit

Description

Given an array of integers nums and an integer limit, return the size of the longest continuous subarray such that the absolute difference between any two elements is less than or equal to limit.

In case there is no subarray satisfying the given condition return 0.

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

Solution

05/03/2020:

```
class Solution {
public:
  int longestSubarray(vector<int>& nums, int limit, int start = 0, int stop =
INT_MAX) {
    if (stop == INT_MAX) stop = nums.size() - 1;
    if (start > stop) return 0;
    auto min_it = min_element(nums.begin() + start, nums.begin() + stop + 1);
    auto max_it = max_element(nums.begin() + start, nums.begin() + stop + 1);
    if (*max_it - *min_it <= limit) return stop - start + 1;
    auto lo = min_it, hi = min_it;
    for (; lo - nums.begin() > start && *(lo - 1) - *min_it <= limit; --lo);
    for (; hi - nums.begin() < stop && *(hi + 1) - *min_it <= limit; ++hi);
    int left = int(min_it - nums.begin() - 1) - start  + 1 >= hi - lo + 1 ?
longestSubarray(nums, limit, start, int(min_it - nums.begin() - 1)) : 0;
    int right = stop - int(min_it - nums.begin() - 1) + 1 >= hi - lo + 1 ?
longestSubarray(nums, limit, int(min_it - nums.begin() + 1), stop) : 0;
    return max(int(hi - lo + 1), max(left, right));
  }
};
```

1441. Build an Array With Stack Operations

Description

Given an array target and an integer n. In each iteration, you will read a number from list = {1,2,3..., n}.

Build the target array using the following operations:

Push: Read a new element from the beginning list, and push it in the array.
Pop: delete the last element of the array.
If the target array is already built, stop reading more elements.
You are guaranteed that the target array is strictly increasing, only containing numbers between 1 to n inclusive.

Return the operations to build the target array.

You are guaranteed that the answer is unique.

Example 1:

Input: target = [1,3], n = 3
Output: ["Push","Push","Pop","Push"]
Explanation:
Read number 1 and automatically push in the array -> [1]
Read number 2 and automatically push in the array then Pop it -> [1]
Read number 3 and automatically push in the array -> [1,3]
Example 2:

Input: target = [1,2,3], n = 3
Output: ["Push","Push","Push"]
Example 3:

Input: target = [1,2], n = 4
Output: ["Push","Push"]
Explanation: You only need to read the first 2 numbers and stop.
Example 4:

Input: target = [2,3,4], n = 4
Output: ["Push","Pop","Push","Push","Push"]

Constraints:

1 <= target.length <= 100
1 <= target[i] <= 100

1 <= n <= 100
target is strictly increasing.

Solution

05/10/2020:

```
class Solution {
public:
  vector<string> buildArray(vector<int>& target, int n) {
    int m = *max_element(target.begin(), target.end());
    unordered_set<int> s(target.begin(), target.end());
    vector<string> ret;
    for (int i = 1; i <= m; ++i) {
      ret.push_back("Push");
      if (s.count(i) == 0) {
        ret.push_back("Pop");
      }
    }
    return ret;
  }
};
```

1447. Simplified Fractions

Description

Given an integer n, return a list of all simplified fractions between 0 and 1 (exclusive) such that the denominator is less-than-or-equal-to n. The fractions can be in any order.

Example 1:

Input: n = 2
Output: ["1/2"]
Explanation: "1/2" is the only unique fraction with a denominator less-than-or-equal-to 2.
Example 2:

Input: n = 3
Output: ["1/2","1/3","2/3"]
Example 3:

Input: n = 4
Output: ["1/2","1/3","1/4","2/3","3/4"]
Explanation: "2/4" is not a simplified fraction because it can be simplified to "1/2".
Example 4:

Input: n = 1
Output: []

Constraints:

1 <= n <= 100

Solution

05/16/2020:

```
class Solution {
public:
  vector<string> simplifiedFractions(int n) {
    unordered_set<string> ret;
    for (int i = 1; i < n; ++i) {
      for (int j = i + 1; j <= n; ++j) {
        int gcd = __gcd(i, j);
        int numerator = i / gcd;
        int denominator = j / gcd;
        ret.insert(to_string(numerator) + "/" + to_string(denominator));
      }
    }
    return vector<string>(ret.begin(), ret.end());
  }
};
```

1452. People Whose List of Favorite Companies Is Not a Subset of Another List

Description

Given the array favoriteCompanies where favoriteCompanies[i] is the list of favorites companies for the ith person (indexed from 0).

Return the indices of people whose list of favorite companies is not a subset of any other list of favorites companies. You must return the indices in increasing order.

Example 1:

Input: favoriteCompanies = [["leetcode","google","facebook"],["google","microsoft"],["google","facebook"],["google"],["amazon"]]
Output: [0,1,4]
Explanation:
Person with index=2 has favoriteCompanies[2]=["google","facebook"] which is a subset of favoriteCompanies[0]=["leetcode","google","facebook"] corresponding to the person with index 0.
Person with index=3 has favoriteCompanies[3]=["google"] which is a subset of favoriteCompanies[0]=["leetcode","google","facebook"] and favoriteCompanies[1]=["google","microsoft"].
Other lists of favorite companies are not a subset of another list, therefore, the answer is [0,1,4].

Example 2:

Input: favoriteCompanies = [["leetcode","google","facebook"],["leetcode","amazon"],["facebook","google"]]
Output: [0,1]
Explanation: In this case favoriteCompanies[2]=["facebook","google"] is a subset of favoriteCompanies[0]=["leetcode","google","facebook"], therefore, the answer is [0,1].

Example 3:

Input: favoriteCompanies = [["leetcode"],["google"],["facebook"],["amazon"]]
Output: [0,1,2,3]

Constraints:

1 <= favoriteCompanies.length <= 100
1 <= favoriteCompanies[i].length <= 500
1 <= favoriteCompanies[i][j].length <= 20
All strings in favoriteCompanies[i] are distinct.
All lists of favorite companies are distinct, that is, If we sort alphabetically each list then favoriteCompanies[i] != favoriteCompanies[j].
All strings consist of lowercase English letters only.

Solution

05/16/2020:

```cpp
class Solution {
public:
  vector<int> peopleIndexes(vector<vector<string>>& favoriteCcompanies) {
    vector<pair<unordered_set<string>, int>> companies;
    int n = favoriteCompanies.size();
    for (int i = 0; i < n; ++i) {
      companies.emplace_back(unordered_set<string>(favoriteCompanies[i].begin(),
favoriteCompanies[i].end()), i);
    }
    sort(companies.begin(), companies.end(), [](pair<unordered_set<string>,
int>& p1, pair<unordered_set<string>, int>& p2) {
      return p1.first.size() > p2.first.size();
    });
    vector<int> ret;
    ret.push_back(companies[0].second);
    for (int i = 1; i < n; ++i) {
      int cnt = 0;
      for (int j = 0; j < i; ++j) {
        bool isSubset = true;
        for (auto& s : companies[i].first) {
          if (companies[j].first.count(s) == 0) {
            isSubset = false;
            break;
          }
        }
        if (!isSubset) {
          ++cnt;
        }
      }
      if (cnt == i) ret.push_back(companies[i].second);
    }
    sort(ret.begin(), ret.end());
    return ret;
  }
};
```

1461. Check If a String Contains All Binary Codes of Size K

Description

Given a binary string s and an integer k.

Return True if all binary codes of length k is a substring of s. Otherwise, return False.

Example 1:

Input: s = "00110110", k = 2
Output: true
Explanation: The binary codes of length 2 are "00", "01", "10" and "11". They can be all found as substrings at indicies 0, 1, 3 and 2 respectively.
Example 2:

Input: s = "00110", k = 2
Output: true
Example 3:

Input: s = "0110", k = 1
Output: true
Explanation: The binary codes of length 1 are "0" and "1", it is clear that both exist as a substring.
Example 4:

Input: s = "0110", k = 2
Output: false
Explanation: The binary code "00" is of length 2 and doesn't exist in the array.
Example 5:

Input: s = "0000000001011100", k = 4
Output: false

Constraints:

1 <= s.length <= 5 * 10^5
s consists of 0's and 1's only.
1 <= k <= 20

Solution

05/30/2020:

```
class Solution {
public:
  bool hasAllCodes(string s, int k) {
    bool ret = true;
    int n = 1 << k;
    int sz = s.size() - k;
    unordered_set<string> required;
    for (int i = 0; i <= sz; ++i) {
      string sub = s.substr(i, k);
```

```
            required.insert(sub);
        }
        return required.size() == n;
    }
};
```

1463. Cherry Pickup II

Description

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

Solution

06/07/2020:

```
int dp[71][71][71];

class Solution {
public:
  int cherryPickup(vector<vector<int>>& grid) {
    fill_n(dp[0][0], 71 * 71 * 71, -1);
    int m = grid.size(), n = grid[0].size();
    dp[0][0][n - 1] = grid[0][0] + grid[0][n - 1];
    for (int i = 1; i < m; ++i) {
      for (int j = 0; j < n; ++j) {
        for (int k = 0; k < n; ++k) {
          int cur = -1;
          for (int u = -1; u <= 1; ++u) {
            int nj = j + u;
            for (int v = -1; v <= 1; ++v) {
              int nk = k + v;
              if (nj >= 0 && nj < n && nk >= 0 && nk < n && dp[i - 1][nj][nk] !=
-1)
                cur = max(cur, dp[i - 1][nj][nk] + grid[i][j] + grid[i][k]);
            }
          }
          if (j == k) cur -= grid[i][j];
```

```
          dp[i][j][k] = max(dp[i][j][k], cur);
        }
      }
    }
    int ret = -1;
    for (int j = 0; j < n; ++j) ret = max(ret, *max_element(dp[m - 1][j], dp[m -
1][j] + n));
    return ret;
  }
};
```

1471. The k Strongest Values in an Array

Description

Given an array of integers arr and an integer k.

A value arr[i] is said to be stronger than a value arr[j] if |arr[i] - m| > |arr[j] - m| where m is the median of the array.
If |arr[i] - m| == |arr[j] - m|, then arr[i] is said to be stronger than arr[j] if arr[i] > arr[j].

Return a list of the strongest k values in the array. return the answer in any arbitrary order.

Median is the middle value in an ordered integer list. More formally, if the length of the list is n, the median is the element in position ((n - 1) / 2) in the sorted list (0-indexed).

For arr = [6, -3, 7, 2, 11], n = 5 and the median is obtained by sorting the array arr = [-3, 2, 6, 7, 11] and the median is arr[m] where m = ((5 - 1) / 2) = 2. The median is 6.
For arr = [-7, 22, 17, 3], n = 4 and the median is obtained by sorting the array arr = [-7, 3, 17, 22] and the median is arr[m] where m = ((4 - 1) / 2) = 1. The median is 3.

Example 1:

Input: arr = [1,2,3,4,5], k = 2
Output: [5,1]
Explanation: Median is 3, the elements of the array sorted by the strongest are [5,1,4,2,3]. The strongest 2 elements are [5, 1]. [1, 5] is also accepted answer.
Please note that although |5 - 3| == |1 - 3| but 5 is stronger than 1 because 5 > 1.

Example 2:

Input: arr = [1,1,3,5,5], k = 2
Output: [5,5]
Explanation: Median is 3, the elements of the array sorted by the strongest are [5,5,1,1,3]. The strongest 2 elements are [5, 5].
Example 3:

Input: arr = [6,7,11,7,6,8], k = 5
Output: [11,8,6,6,7]
Explanation: Median is 7, the elements of the array sorted by the strongest are [11,8,6,6,7,7].
Any permutation of [11,8,6,6,7] is accepted.
Example 4:

Input: arr = [6,-3,7,2,11], k = 3
Output: [-3,11,2]
Example 5:

Input: arr = [-7,22,17,3], k = 2
Output: [22,17]

Constraints:

1 <= arr.length <= 10^5
-10^5 <= arr[i] <= 10^5
1 <= k <= arr.length

Solution

06/06/2020:

```cpp
int m;
class Solution {
public:
  vector<int> getStrongest(vector<int>& arr, int k) {
    int n = arr.size();
    sort(arr.begin(), arr.end());
    m = arr[(n - 1) / 2];
    sort(arr.begin(), arr.end(), [](int a, int b) {
      if (abs(a - m) == abs(b - m)) return a > b;
      return abs(a - m) > abs(b - m);
    });
    vector<int> ret;
    for (int i = 0; i < k; ++i)
      ret.push_back(arr[i]);
    return ret;
```

```
    }
};
```

Check If a String Is a Valid Sequence from Root to Leaves Path in a Binary Tree

Description

Given a binary tree where each path going from the root to any leaf form a valid sequence, check if a given string is a valid sequence in such binary tree.

We get the given string from the concatenation of an array of integers arr and the concatenation of all values of the nodes along a path results in a sequence in the given binary tree.

Example 1:

Input: root = [0,1,0,0,1,0,null,null,1,0,0], arr = [0,1,0,1]
Output: true
Explanation:
The path 0 -> 1 -> 0 -> 1 is a valid sequence (green color in the figure).
Other valid sequences are:
0 -> 1 -> 1 -> 0
0 -> 0 -> 0
Example 2:

Input: root = [0,1,0,0,1,0,null,null,1,0,0], arr = [0,0,1]
Output: false
Explanation: The path 0 -> 0 -> 1 does not exist, therefore it is not even a sequence.
Example 3:

Input: root = [0,1,0,0,1,0,null,null,1,0,0], arr = [0,1,1]
Output: false
Explanation: The path 0 -> 1 -> 1 is a sequence, but it is not a valid sequence.

```
Constraints:

1 <= arr.length <= 5000
0 <= arr[i] <= 9
Each node's value is between [0 - 9].
```

Solution

Discussion

- Using unordered_map is straightforward: convert numbers visited to a string and store seen strings in an unordered_map. Then just check whether the target string generated by the arr is in the unordered_map or not.
- Time complexity: O(n)
- Space complexity: O(n^2)

```
class Solution {
private:
  unordered_set<string> s;
  string target;
public:
  bool isValidSequence(TreeNode* root, vector<int>& arr) {
    for (auto& n : arr) target += to_string(n);  dfs(root);
    return s.count(target);
  }

  void dfs(TreeNode* root, string str="") {
    if (root == nullptr) return;
    if (root->left == nullptr && root->right == nullptr) {
      str += to_string(root->val);
      if (!str.empty()) s.insert(str);
      return;
    }
    dfs(root->left, str + to_string(root->val));
    dfs(root->right, str + to_string(root->val));
  }
};
```

- Another way is more straightforward, we check the elements in the arr while traversing the nodes in the tree:
- Time complexity: O(n)
- Space complexity: O(n)

```cpp
class Solution {
public:
  bool isValidSequence(TreeNode* root, vector<int>& arr, int i = 0) {
    if (root == nullptr) return false;
    if (i >= (int)arr.size()) return false;
    if (root->left == nullptr && root->right == nullptr && i == (int)arr.size() - 1) return arr.back() == root->val;
    return root->val == arr[i] && (isValidSequence(root->left, arr, i + 1) || isValidSequence(root->right, arr, i + 1));
  }
};
```

1477. Find Two Non-overlapping Sub-arrays Each With Target Sum

Description

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

Solution

06/13/2020:

```
class Solution {
public:
  int minSumOfLengths(vector<int>& arr, int target) {
    queue<pair<int, int>> q;
    int ret = INT_MAX, preMin = INT_MAX;
    int lo = 0, hi = 0, cur = 0;
    while (hi < arr.size()) {
      cur += arr[hi];
      ++hi;
      while (cur > target && lo < hi) {
        cur -= arr[lo];
        ++lo;
      }
      while (!q.empty() && q.front().second <= lo) {
        preMin = min(preMin, q.front().second - q.front().first);
        q.pop();
      }
      if (cur == target) {
        if (preMin != INT_MAX) {
          ret = min(ret, preMin + hi - lo);
        }
        q.emplace(lo, hi);
      }
```

```
    }
    return ret == INT_MAX ? -1 : ret;
  }
};
```

```
void solve(vector<int>& v, int t, vector<int>& ret) {
  int n = v.size();
  ret.resize(n + 1);
  fill(ret.begin(), ret.end(), 1e9);
  int cur = 0;
  int lhs = 0;
  for (int i = 1; i <= n; ++i) {
    cur += v[i - 1];
    while (cur > t) cur -= v[lhs++];
    if (cur == t) ret[i] = i - lhs;
  }
}

class Solution {
public:
  int minSumOfLengths(vector<int>& arr, int target) {
    vector<int> lhs, rhs;
    solve(arr, target, lhs);
    reverse(arr.begin(), arr.end());
    solve(arr, target, rhs);
    for (int i = 1; i < (int)lhs.size(); ++i) lhs[i] = min(lhs[i], lhs[i - 1]);
    for (int i = 1; i < (int)rhs.size(); ++i) rhs[i] = min(rhs[i], rhs[i - 1]);
    int ret = 1e9;
    int n = arr.size();
    for (int i = 1; i < n; ++i) ret = min(ret, lhs[i] + rhs[n - i]);
    if (ret >= 1e9) ret = -1;
    return ret;
  }
};
```

1480. Running Sum of 1d Array

Description

Given an array nums. We define a running sum of an array as runningSum[i] = sum(nums[0]…nums[i]).

Return the running sum of nums.

Example 1:

Input: nums = [1,2,3,4]
Output: [1,3,6,10]
Explanation: Running sum is obtained as follows: [1, 1+2, 1+2+3, 1+2+3+4].
Example 2:

Input: nums = [1,1,1,1,1]
Output: [1,2,3,4,5]
Explanation: Running sum is obtained as follows: [1, 1+1, 1+1+1, 1+1+1+1, 1+1+1+1+1].
Example 3:

Input: nums = [3,1,2,10,1]
Output: [3,4,6,16,17]

Constraints:

1 <= nums.length <= 1000
-10^6 <= nums[i] <= 10^6

Solution

06/13/2020:

```
class Solution {
public:
  vector<int> runningSum(vector<int>& nums) {
    partial_sum(nums.begin(), nums.end(), nums.begin());
    return nums;
  }
};
```

1482. Minimum Number of Days to Make m Bouquets

Description

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

Solution

06/13/2020:

```
class Solution {
public:
  int minDays(vector<int>& bloomDay, int m, int k) {
    if (bloomDay.size() < (long long)m * k) return -1;
    int lo = 0, hi = 1e9;
    while (lo < hi) {
      int mid = lo + (hi - lo) / 2;
      int amount = 0, candidate = 0;
      for (auto& b : bloomDay) {
        if (b <= mid)
          ++candidate;
        else
          candidate = 0;
        if (candidate == k) {
          candidate = 0;
          ++amount;
        }
      }
      if (amount >= m)
        hi = mid;
      else
        lo = mid + 1;
    }
    return lo;
  }
};
```

1483. Kth Ancestor of a Tree Node

Description

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.

Solution

06/13/2020:

```
const int MAXD = 16;
int dp[50005][MAXD];

class TreeAncestor {
public:
  TreeAncestor(int n, vector<int>& parent) {
    for (int i = 0; i < n; ++i) dp[i][0] = parent[i];
```

```
    for (int d = 1; d < MAXD; ++d) {
      for (int i = 0; i < n; ++i) {
        if (dp[i][d - 1] == -1)
          dp[i][d] = -1;
        else
          dp[i][d] = dp[dp[i][d - 1]][d - 1];
      }
    }
  }

  int getKthAncestor(int node, int k) {
    for (int d = MAXD - 1; d >= 0; --d) {
      if ((1 << d) <= k) {
        k -= 1 << d;
        node = dp[node][d];
        if (node == -1) return node;
      }
    }
    return node;
  }
};

/**
 * Your TreeAncestor object will be instantiated and called as such:
 * TreeAncestor* obj = new TreeAncestor(n, parent);
 * int param_1 = obj->getKthAncestor(node,k);
 */
```

1488. Avoid Flood in The City

Description

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
0 <= rains[i] <= 10^9

Solution

06/20/2020:

```
class Solution {
public:
  vector<int> avoidFlood(vector<int>& rains) {
    int n = rains.size();
    unordered_map<int, int> bad;
    set<int> canfix;
    vector<int> ret(n, -1);
    for (int i = 0; i < n; ++i) {
      if (rains[i] == 0) {
        canfix.insert(i);
        ret[i] = 1;
      } else {
        if (bad.count(rains[i]) == 0) {
          bad[rains[i]] = i;
        } else {
          int must = bad[rains[i]];
          auto it = canfix.upper_bound(must);
          if (it == canfix.end()) return {};
          ret[*it] = rains[i];
          canfix.erase(it);
          bad[rains[i]] = i;
        }
      }
    }
    return ret;
  }
};
```