# 146. LRU Cache

Design a data structure that follows the constraints of a [**Least Recently Used (LRU) cache**](https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU).

Implement the LRUCache class:

* LRUCache(int capacity) Initialize the LRU cache with **positive** size capacity.
* int get(int key) Return the value of the key if the key exists, otherwise return -1.
* void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, **evict** the least recently used key.

The functions get and put must each run in O(1) average time complexity.

**Example 1:**

**Input**

["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]

[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]

**Output**

[null, null, null, 1, null, -1, null, -1, 3, 4]

**Explanation**

LRUCache lRUCache = new LRUCache(2);

lRUCache.put(1, 1); // cache is {1=1}

lRUCache.put(2, 2); // cache is {1=1, 2=2}

lRUCache.get(1); // return 1

lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}

lRUCache.get(2); // returns -1 (not found)

lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}

lRUCache.get(1); // return -1 (not found)

lRUCache.get(3); // return 3

lRUCache.get(4); // return 4

**Constraints:**

* 1 <= capacity <= 3000
* 0 <= key <= 104
* 0 <= value <= 105
* At most 2 \* 105 calls will be made to get and put

# Answer:

public class LRUCache {

public int Maximum {get; set;}

public Dictionary<int,int> Cache {get; set;}

public List<int> LastUsed {get; set;}

public LRUCache(int capacity) {

Maximum=capacity;

Cache= new Dictionary<int,int>();

LastUsed= new List<int>();

}

public int Get(int key) {

if(!Cache.ContainsKey(key))

return -1;

LastUsed.Remove(key);

LastUsed.Insert(0,key);

return Cache[key];

}

public void Put(int key, int value) {

if(!Cache.ContainsKey(key) && Cache.Count != Maximum){

Cache.Add(key,value);

LastUsed.Insert(0,key);

return;

}

else if(!Cache.ContainsKey(key) && Cache.Count == Maximum){

Cache.Remove(LastUsed.Last());

LastUsed.Remove(LastUsed.Last());

Cache.Add(key,value);

LastUsed.Insert(0,key);

return;

}

else{

Cache[key]=value;

LastUsed.Remove(key);

LastUsed.Insert(0,key);

}

}

}

/\*\*

\* Your LRUCache object will be instantiated and called as such:

\* LRUCache obj = new LRUCache(capacity);

\* int param\_1 = obj.Get(key);

\* obj.Put(key,value);

\*/

# 79. Word Search

Given a 2D board and a word, find if the word exists in the grid. The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

## **Details**

#### **Example**

Given **board** =

[

['A','B','C','E'],

['S','F','C','S'],

['A','D','E','E']

]

**word** = "ABCCED", -> returns true,  
**word** = "SEE", -> returns true,  
**word** = "ABCB", -> returns false.

# Answer:

#### **Program**

namespace WordSearchProblem

{

class WordSearch

{

**static** **void** Main(**string**[] args)

{

**char**[,] arr =

{

{'A','B','C','E'},

{'S','F','C','S'},

{'A','D','E','E'}

};

**string** word = "ABCCED";

**bool** IsExists = Exist(arr, word);

}

/// <summary>

/// Mwthod to find in the given word exists in the given board.

/// </summary>

/// <param name="board">2D board</param>

/// <param name="word">word to find in board</param>

/// <returns></returns>

**static** **public** **bool** **Exist**(**char**[,] board, **string** word)

{

**int** rowCount = board.GetLength(0);

**int** colCount = board.GetLength(1);

**bool**[,] memo = **new** **bool**[rowCount, colCount];

// Begin from the 0th index of the "word"

**for** (**int** i = 0; i < rowCount; i++)

{

**for** (**int** j = 0; j < colCount; j++)

{

**if** (Exist(board, word, memo, 0, i, j, rowCount, colCount))

**return** **true**;

}

}

**return** **false**;

}

/// <summary>

/// Overloaded method to be called recursively.

/// This is based on back tracking dynamic programming concept.

/// </summary>

/// <param name="board">given 2D to find the word into.</param>

/// <param name="word">given word to find in the board</param>

/// <param name="memo">to memorize the indexes in board

/// that have been covered in order to find the word</param>

/// <param name="findIndex">next index of word to find in board</param>

/// <param name="i">row index</param>

/// <param name="j">column index</param>

/// <param name="rowCount">row count in board</param>

/// <param name="colCount">column count in board</param>

/// <returns>true if word's current index found in board at (i,j) position</returns>

**static** **public** **bool** **Exist**(**char**[,] board, **string** word, **bool**[,] memo, **int** findIndex, **int** i, **int** j, **int** rowCount, **int** colCount)

{

// all character till end found in board.

**if** (findIndex == word.Length)

{

**return** **true**;

}

**else**

{

**if** (i < rowCount && j < colCount && i >= 0 && j >= 0)

{

// memo[i,j] is false means that index is not covered yet.

**if** (memo[i, j] == **false** && board[i, j] == word[findIndex])

{

// mark it as covered

memo[i, j] = **true**;

// find next character at same row next column

**if** (Exist(board, word, memo, findIndex + 1, i, j + 1, rowCount, colCount))

{

**return** **true**;

}// find next character at next row same column

**else** **if** (Exist(board, word, memo, findIndex + 1, i + 1, j, rowCount, colCount))

{

**return** **true**;

}// find next character at same row previous column

**else** **if** (Exist(board, word, memo, findIndex + 1, i, j - 1, rowCount, colCount))

{

**return** **true**;

}// find next character at previous row same column

**else** **if** (Exist(board, word, memo, findIndex + 1, i - 1, j, rowCount, colCount))

{

**return** **true**;

}

**else**

{

// mark it as uncovered.

memo[i, j] = **false**;

}

}

**else**

{

**return** **false**;

}

}

}

**return** **false**;

}

}

}

# 200. Number of Islands

Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

**Example 1:**

**Input:** grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

["0","0","0","0","0"]

]

**Output:** 1

**Example 2:**

**Input:** grid = [

["1","1","0","0","0"],

["1","1","0","0","0"],

["0","0","1","0","0"],

["0","0","0","1","1"]

]

**Output:** 3

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 300
* grid[i][j] is '0' or '1'.

# Answer:

public int NumIslands(char[][] grid) {

var count = 0;

for(var i = 0; i < grid.Length; i++) {

for(var j = 0 ; j < grid[i].Length; j++) {

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

count++;

grid[i][j] = '0';

var queue = new Queue<int[]>();

queue.Enqueue(new int[]{ i, j });

while(queue.Count > 0) {

var loop = queue.Count;

for(var k = 0; k < loop; k++) {

var p = queue.Dequeue();

var x = p[0];

var y = p[1];

if (y + 1 < grid[0].Length && grid[x][y+1] == '1') {

grid[x][y+1] = '0';

queue.Enqueue(new int[]{ x, y+1 });

}

if (x + 1 < grid.Length && grid[x+1][y] == '1') {

grid[x+1][y] = '0';

queue.Enqueue(new int[]{ x+1, y });

}

if (y - 1 >= 0 && grid[x][y-1] == '1') {

grid[x][y-1] = '0';

queue.Enqueue(new int[]{ x, y-1 });

}

if (x - 1 >= 0 && grid[x-1][y] == '1') {

grid[x-1][y] = '0';

queue.Enqueue(new int[]{ x-1, y });

}

}

}}}}

return count;

}

# 588. [Design In-Memory File System](https://leetcode.com/problems/design-in-memory-file-system)

Design a data structure that simulates an in-memory file system.

Implement the FileSystem class:

* FileSystem() Initializes the object of the system.
* List<String> ls(String path)
  + If path is a file path, returns a list that only contains this file's name.
  + If path is a directory path, returns the list of file and directory names **in this directory**.

The answer should in **lexicographic order**.

* void mkdir(String path) Makes a new directory according to the given path. The given directory path does not exist. If the middle directories in the path do not exist, you should create them as well.
* void addContentToFile(String filePath, String content)
  + If filePath does not exist, creates that file containing given content.
  + If filePath already exists, appends the given content to original content.
* String readContentFromFile(String filePath) Returns the content in the file at filePath.

**Example 1:**

Table

Description automatically generated

**Input**

["FileSystem", "ls", "mkdir", "addContentToFile", "ls", "readContentFromFile"]

[[], ["/"], ["/a/b/c"], ["/a/b/c/d", "hello"], ["/"], ["/a/b/c/d"]]

**Output**

[null, [], null, null, ["a"], "hello"]

**Explanation**

FileSystem fileSystem = new FileSystem();

fileSystem.ls("/"); // return []

fileSystem.mkdir("/a/b/c");

fileSystem.addContentToFile("/a/b/c/d", "hello");

fileSystem.ls("/"); // return ["a"]

fileSystem.readContentFromFile("/a/b/c/d"); // return "hello"

**Constraints:**

* 1 <= path.length, filePath.length <= 100
* path and filePath are absolute paths which begin with '/' and do not end with '/' except that the path is just "/".
* You can assume that all directory names and file names only contain lowercase letters, and the same names will not exist in the same directory.
* You can assume that all operations will be passed valid parameters, and users will not attempt to retrieve file content or list a directory or file that does not exist.
* 1 <= content.length <= 50
* At most 300 calls will be made to ls, mkdir, addContentToFile, and readContentFromFile.

# Answer:

public class File {

public File(string name) { Name = name; }

public bool IsFile { get; set; }

public string Name { get; set; }

public string Content { get; set; }

public Dictionary<string, File> Files { get; set; } = new Dictionary<string, File>();

}

public class FileSystem {

File root = new File("");

public void Mkdir(string path) => addPath(path);

public void AddContentToFile(string filePath, string content) => addPath(filePath, content);

public string ReadContentFromFile(string filePath) => getPath(filePath).Content;

public IList<string> Ls(string path) {

var target = getPath(path);

if (target.IsFile) return new List<string>() { target.Name };

return target.Files.Keys.OrderBy(k => k).ToList();

}

private File addPath(string path, string content = null) {

var segments = path.Split('/', StringSplitOptions.RemoveEmptyEntries);

var curr = root;

foreach (var segment in segments) {

if (!curr.Files.ContainsKey(segment))

curr.Files.Add(segment, new File(segment));

curr = curr.Files[segment];

}

if (content == null) return curr;

curr.Content += content;

curr.IsFile = true;

return curr;

}

private File getPath(string path) {

var segments = path.Split('/', StringSplitOptions.RemoveEmptyEntries);

var curr = root;

foreach (var segment in segments)

curr = curr.Files[segment];

return curr;

}

}

# 227. Basic Calculator II

Given a string s which represents an expression, *evaluate this expression and return its value*.

The integer division should truncate toward zero.

You may assume that the given expression is always valid. All intermediate results will be in the range of [-231, 231 - 1].

**Note:** You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().

**Example 1:**

**Input:** s = "3+2\*2"

**Output:** 7

**Example 2:**

**Input:** s = " 3/2 "

**Output:** 1

**Example 3:**

**Input:** s = " 3+5 / 2 "

**Output:** 5

**Constraints:**

* 1 <= s.length <= 3 \* 105
* s consists of integers and operators ('+', '-', '\*', '/') separated by some number of spaces.
* s represents **a valid expression**.
* All the integers in the expression are non-negative integers in the range [0, 231 - 1].
* The answer is **guaranteed** to fit in a **32-bit integer**.

# Answer:

public static int Calculate(string s)

{

if (s == null || s.Length == 0)

{

return 0;

}

// trim the string from both end.

s = s.Trim();

// stack to store the operands

Stack<int> operands = new Stack<int>();

// current number

int num = 0;

// last operator encountered

char operation = '+';

// loop throught the given expression

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

{

// if it is just space; move to next index value

if (s[i] == ' ')

{

continue;

}

// if it is digit, append it in the number

if (s[i] >= '0' && s[i] <= '9')

{

num = num \* 10 + (s[i] - '0');

}

// if it is not digit or it is last character in the string

if (!(s[i] >= '0' && s[i] <= '9') || i == s.Length - 1)

{

// calculate based on last operator

switch (operation)

{

case '\*':

// Pop the last value in the stack and multiply it with val

// and Push the product in the stack

operands.Push(operands.Pop() \* num);

break;

case '/':

// Pop the last value in the stack and divide it with val

// and Push the result in the stack

operands.Push(operands.Pop() / num);

break;

case '-':

// make the value -ve and push it in the stack

operands.Push(num \* (-1));

break;

default:

// push the value into the stack

operands.Push(num);

break;

}

// set num to save the next number

num = 0;

}

// if it is a operand, update the operation

if ((s[i] == '+') || (s[i] == '-') || (s[i] == '\*') || (s[i] == '/'))

{

operation = s[i];

}

}

int sum = 0;

// read all the values from the stack and sum them

while (operands.Count > 0)

{

sum += operands.Pop();

}

// return sum

return sum;

}

# 380. Insert Delete GetRandom O(1)

Implement the RandomizedSet class:

* RandomizedSet() Initializes the RandomizedSet object.
* bool insert(int val) Inserts an item val into the set if not present. Returns true if the item was not present, false otherwise.
* bool remove(int val) Removes an item val from the set if present. Returns true if the item was present, false otherwise.
* int getRandom() Returns a random element from the current set of elements (it's guaranteed that at least one element exists when this method is called). Each element must have the **same probability** of being returned.

You must implement the functions of the class such that each function works in **average** O(1) time complexity.

**Example 1:**

**Input**

["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert", "getRandom"]

[[], [1], [2], [2], [], [1], [2], []]

**Output**

[null, true, false, true, 2, true, false, 2]

**Explanation**

RandomizedSet randomizedSet = new RandomizedSet();

randomizedSet.insert(1); // Inserts 1 to the set. Returns true as 1 was inserted successfully.

randomizedSet.remove(2); // Returns false as 2 does not exist in the set.

randomizedSet.insert(2); // Inserts 2 to the set, returns true. Set now contains [1,2].

randomizedSet.getRandom(); // getRandom() should return either 1 or 2 randomly.

randomizedSet.remove(1); // Removes 1 from the set, returns true. Set now contains [2].

randomizedSet.insert(2); // 2 was already in the set, so return false.

randomizedSet.getRandom(); // Since 2 is the only number in the set, getRandom() will always return 2.

**Constraints:**

* -231 <= val <= 231 - 1
* At most 2 \* 105 calls will be made to insert, remove, and getRandom.
* There will be **at least one** element in the data structure when getRandom is called.

# Answer:

private Dictionary<int,int> map ;

private List<int> nums;

private Random rand;

public RandomizedSet() {

map= new Dictionary<int,int>();

nums = new List<int>();

rand = new Random();

}

public bool Insert(int val) {

if(map.ContainsKey(val))

{ return false;

}

nums.Add(val);

map.Add(val, nums.Count-1);

return true;

}

public bool Remove(int val) {

if(!map.ContainsKey(val))

{ return false;

}

int i = map[val];

nums[i]= nums[nums.Count-1];

map[nums[i]]= i;

nums.RemoveAt(nums.Count-1);

map.Remove(val);

return true;

}

public int GetRandom() {

return nums[rand.Next(0 , nums.Count)] ;

}

# 253. Meeting Rooms II

Given an array of meeting time intervals intervals where intervals[i] = [starti, endi], return *the minimum number of conference rooms required*.

**Example 1:**

**Input:** intervals = [[0,30],[5,10],[15,20]]

**Output:** 2

**Example 2:**

**Input:** intervals = [[7,10],[2,4]]

**Output:** 1

**Constraints:**

* 1 <= intervals.length <= 104
* 0 <= starti < endi <= 106

# Answer:

public int MinMeetingRooms(int[][] intervals) {

int result=0;

if(intervals == null || intervals.Length == 0){

return result;

}

var starts = intervals.Select(i => i[0]).OrderBy(i => i).ToArray();

var ends= intervals.Select(i => i[1]).OrderBy(i => i).ToArray();

for (int i = 0, j = 0; i < starts.Length; i++)

if (starts[i] < ends[j])

result++;

else

j++;

return result;

}

# 273. Integer to English Words

Convert a non-negative integer num to its English words representation.

**Example 1:**

**Input:** num = 123

**Output:** "One Hundred Twenty Three"

**Example 2:**

**Input:** num = 12345

**Output:** "Twelve Thousand Three Hundred Forty Five"

**Example 3:**

**Input:** num = 1234567

**Output:** "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"

**Constraints:**

* 0 <= num <= 231 - 1

# Answer:

public string NumberToWords(int num)

{

if (num == 0)

return "Zero";

if (num < 0)

return "Negative " + NumberToWords(Math.Abs(num));

string words = "";

if ((num / 1000000000) > 0)

{

words += NumberToWords(num / 1000000000) + " Billion ";

num %= 1000000000;

}

if ((num / 1000000) > 0)

{

words += NumberToWords(num / 1000000) + " Million ";

num %= 1000000;

}

if ((num / 1000) > 0)

{

words += NumberToWords(num / 1000) + " Thousand ";

num %= 1000;

}

if ((num / 100) > 0)

{

words += NumberToWords(num / 100) + " Hundred ";

num %= 100;

}

if (num > 0)

{

if (words != "")

words += "";

var unitsMap = new[] { "Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen" };

var tensMap = new[] { "Zero", "Ten", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety" };

if (num < 20)

words += unitsMap[num];

else

{

words += tensMap[num / 10];

if ((num % 10) > 0)

words +=" "+ unitsMap[num % 10];

}

}

return words.Trim();

}

# 394. Decode String

Given an encoded string, return its decoded string.

The encoding rule is: k[encoded\_string], where the encoded\_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; there are no extra white spaces, square brackets are well-formed, etc. Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there will not be input like 3a or 2[4].

The test cases are generated so that the length of the output will never exceed 105.

**Example 1:**

**Input:** s = "3[a]2[bc]"

**Output:** "aaabcbc"

**Example 2:**

**Input:** s = "3[a2[c]]"

**Output:** "accaccacc"

**Example 3:**

**Input:** s = "2[abc]3[cd]ef"

**Output:** "abcabccdcdcdef"

**Constraints:**

* 1 <= s.length <= 30
* s consists of lowercase English letters, digits, and square brackets '[]'.
* s is guaranteed to be **a valid** input.
* All the integers in s are in the range [1, 300].

# Answer:

class Solution {

public String decodeString(String s) {

Stack<char> stack = new Stack<char>();

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

if (s[i] == ']') {

List<char> decodedString = new List<char>();

// get the encoded string

while (stack.Peek() != '[') {

decodedString.Add(stack.Pop());

}

// pop from the stack

stack.pop();

int base = 1;

int k = 0;

// get the number k

while (!stack.isEmpty() && char.isDigit(stack.Peek())) {

k = k + (stack.Pop() - '0') \* base;

base \*= 10;

}

// decode k[decodedString], by pushing decodedString k times into stack

while (k != 0) {

for (int j = decodedString.size() - 1; j >= 0; j--) {

stack.push(decodedString.get(j));

}

k--;

}

}

// push the current character to stack

else {

stack.Push(s[i]);

}

}

// get the result from stack

char[] result = new char[stack.Count];

for (int i = result.length - 1; i >= 0; i--) {

result[i] = stack.Pop();

}

return new String(result);

}

}

# 23. Merge k Sorted Lists

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

*Merge all the linked-lists into one sorted linked-list and return it.*

**Example 1:**

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

**Output:** [1,1,2,3,4,4,5,6]

**Explanation:** The linked-lists are:

[

1->4->5,

1->3->4,

2->6

]

merging them into one sorted list:

1->1->2->3->4->4->5->6

**Example 2:**

**Input:** lists = []

**Output:** []

**Example 3:**

**Input:** lists = [[]]

**Output:** []

**Constraints:**

* k == lists.length
* 0 <= k <= 104
* 0 <= lists[i].length <= 500
* -104 <= lists[i][j] <= 104
* lists[i] is sorted in **ascending order**.
* The sum of lists[i].length will not exceed 104.

# Answer:

public ListNode MergeKLists(ListNode[] lists) {

var toMerge = lists?.ToList();

if(lists == null || toMerge.Count == 0){

return null;

}

var mergedLists= new List<ListNode>();

while(toMerge.Count > 1)

{

mergedLists= new List<ListNode>();

for(int i=0; i<toMerge.Count; i+=2){

if(i+1 < toMerge.Count){

var merged = Merge2Lists(toMerge[i], toMerge[i+1]);

mergedLists.Add(merged);

}

else{

mergedLists.Add(toMerge[i]);

}

}

toMerge = mergedLists;

}

return toMerge[0];

}

public ListNode Merge2Lists(ListNode l1, ListNode l2){

if(l1 == null && l2 == null){

return null;

}

if(l1== null){

return l2;

}

if(l2 == null){

return l1;

}

var dummyHead= new ListNode();

var result = dummyHead;

while(l1!=null && l2 != null){

if(l1.val < l2.val){

result.next = l1;

l1 = l1.next;

}

else{

result.next = l2;

l2= l2.next;

}

result= result.next;

}

result.next = l1 ?? l2;

return dummyHead.next;

}

# 17. Letter Combinations of a Phone Number

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**.

A mapping of digit to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

A picture containing electronics, calculator, hand, orange

Description automatically generated

**Example 1:**

**Input:** digits = "23"

**Output:** ["ad","ae","af","bd","be","bf","cd","ce","cf"]

**Example 2:**

**Input:** digits = ""

**Output:** []

**Example 3:**

**Input:** digits = "2"

**Output:** ["a","b","c"]

**Constraints:**

* 0 <= digits.length <= 4
* digits[i] is a digit in the range ['2', '9'].

# Answer:

public IList<string> LetterCombinations(string digits) {

var result = new List<string>();

if(string.IsNullOrEmpty(digits)){

return result;

}

Dictionary<char,char[]> lettersMap=

new Dictionary<char,char[]>();

lettersMap.Add('1', null);

lettersMap.Add('2', new [] {'a','b','c'});

lettersMap.Add('3', new [] {'d','e','f'});

lettersMap.Add('4', new [] {'g','h','i'});

lettersMap.Add('5', new [] {'j','k','l'});

lettersMap.Add('6', new [] {'m','n','o'});

lettersMap.Add('7', new [] {'p','q','r','s'});

lettersMap.Add('8', new [] {'t','u','v'});

lettersMap.Add('9', new [] {'w','x','y','z'});

lettersMap.Add('0', null);

result.Add("");

foreach(var d in digits){

var next = new List<string>();

var letterList = lettersMap.Where(x=>x.Key == d).FirstOrDefault().Value;

foreach(var letter in letterList){

foreach(string s in result){

next.Add(s + letter);

}

}

result= next;

}

return result;

}

# 49. Group Anagrams

Given an array of strings strs, group **the anagrams** together. You can return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Example 1:**

**Input:** strs = ["eat","tea","tan","ate","nat","bat"]

**Output:** [["bat"],["nat","tan"],["ate","eat","tea"]]

**Example 2:**

**Input:** strs = [""]

**Output:** [[""]]

**Example 3:**

**Input:** strs = ["a"]

**Output:** [["a"]]

**Constraints:**

* 1 <= strs.length <= 104
* 0 <= strs[i].length <= 100
* strs[i] consists of lowercase English letters.

# Answer:

public IList<IList<string>> GroupAnagrams(string[] strs) {

if (strs == null)

{

return null;

}

int[] prime = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103};

Dictionary<long, List<string>> res = new Dictionary<long, List<string>>();

foreach(String x in strs)

{

long val = 1;

for(int i= 0; i< x.Length; i++)

{

val \*= prime[x[i]-'a'];

}

List<string> currentValue = null;

if (res.ContainsKey(val))

{

var oldlist = res[val];

oldlist.Add(x);

res[val]=oldlist;

}

else

{

var newlist = new List<string>();

newlist.Add(x);

res[val] = newlist;

}

}

IList<IList<string>> result = new List<IList<string>>();

foreach(var pair in res)

{

IList<string> values = pair.Value;

result.Add(values);

}

return result;

}

# 127. Word Ladder

A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

Every adjacent pair of words differs by a single letter.

Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.

sk == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return *the****number of words****in the****shortest transformation sequence****from* beginWord *to* endWord*, or*0*if no such sequence exists.*

**Example 1:**

**Input:** beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

**Output:** 5

**Explanation:** One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.

**Example 2:**

**Input:** beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]

**Output:** 0

**Explanation:** The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

**Constraints:**

1 <= beginWord.length <= 10

endWord.length == beginWord.length

1 <= wordList.length <= 5000

wordList[i].length == beginWord.length

beginWord, endWord, and wordList[i] consist of lowercase English letters.

beginWord != endWord

All the words in wordList are **unique**.

# Answer:

public int LadderLength(string beginWord, string endWord, IList<string> wordList) {

var count = 1;

var queue = new Queue<string>();

//populate a dictionary of our complete word set that we can reference later

var dictionary = new HashSet<string>(wordList);

queue.Enqueue(beginWord);

while (queue.Count != 0) {

//tracking the level count is important so that we know

//when to increment our total count

var countInLevel = queue.Count;

while (countInLevel > 0) {

var currentWord = queue.Dequeue();

if (currentWord == endWord) return count;

for (var i = 0; i < currentWord.Length; i++) {

var temp = currentWord.ToCharArray();

for (var j = 0; j < 26; j++) {

//adding an int to a char here like this increments the ASCII code

//so 'a' + 1, cast to a char, becomes 'b'

//this allows us to modify each char in the word

//and check for any matches in our dictionary

temp[i] = (char)('a' + j);

//if we find a match in our dictionary, queue that word

//so we can check it during the next iteration of our outer while loop

var ts = new string(temp);

if (dictionary.Contains(ts)) {

queue.Enqueue(ts);

//removing from the dictionary ensures we don't have duplicates

//in any level, and that we check each entry only once

dictionary.Remove(ts);

}

}

}

countInLevel--;

}

count++;

}

return 0;

}

# 41. First Missing Positive

Given an unsorted integer array nums, return the smallest missing positive integer.

You must implement an algorithm that runs in O(n) time and uses constant extra space.

**Example 1:**

**Input:** nums = [1,2,0]

**Output:** 3

**Example 2:**

**Input:** nums = [3,4,-1,1]

**Output:** 2

**Example 3:**

**Input:** nums = [7,8,9,11,12]

**Output:** 1

**Constraints:**

* 1 <= nums.length <= 5 \* 105
* -231 <= nums[i] <= 231 - 1

# Answer:

public class Solution {

public int FirstMissingPositive(int[] nums)

{

HashSet<int> hs = new HashSet<int>();

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

{

if (hs.Contains(nums[i]))

continue;

hs.Add(nums[i]);

}

int minVal = 1;

for (minVal = 1; minVal <= nums.Length; minVal++)

{

if (!hs.Contains(minVal))

return minVal;

}

return minVal;

}

}

# 54. Spiral Matrix

Given an m x n matrix, return *all elements of the* matrix *in spiral order*.

**Example 1:**

A picture containing text, crossword puzzle

Description automatically generated

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

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

**Example 2:**

Shape

Description automatically generated

**Input:** matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]

**Output:** [1,2,3,4,8,12,11,10,9,5,6,7]

**Constraints:**

* m == matrix.length
* n == matrix[i].length
* 1 <= m, n <= 10
* -100 <= matrix[i][j] <= 100

# Answer:

public IList<int> SpiralOrder(int[][] matrix) {

int top=0;

if(matrix== null || matrix.Length==0|| matrix[0].Length==0)

return new List<int>();

int bottom= matrix.Length-1;

int left=0;

int right=matrix[0].Length-1;

IList<int> res= new List<int>();

while(true){

for(int i=left; i<=right;i++){

res.Add(matrix[top][i]);

}

top++;

if(top>bottom) break;

for(int i=top; i<=bottom;i++){

res.Add(matrix[i][right]);

}

right--;

if(left> right) break;

for(int i=right; i>=left;i--){

res.Add(matrix[bottom][i]);

}

bottom--;

if(top>bottom) break;

for(int i=bottom; i>=top;i--){

res.Add(matrix[i][left]);

}

left++;

if(left> right) break;

}

return res;

}

# 362. Design Hit Counter

Design a hit counter which counts the number of hits received in the past 5 minutes (i.e., the past 300 seconds).

Your system should accept a timestamp parameter (**in seconds** granularity), and you may assume that calls are being made to the system in chronological order (i.e., timestamp is monotonically increasing). Several hits may arrive roughly at the same time.

Implement the HitCounter class:

* HitCounter() Initializes the object of the hit counter system.
* void hit(int timestamp) Records a hit that happened at timestamp (**in seconds**). Several hits may happen at the same timestamp.
* int getHits(int timestamp) Returns the number of hits in the past 5 minutes from timestamp (i.e., the past 300 seconds).

**Example 1:**

Input

["HitCounter", "hit", "hit", "hit", "getHits", "hit", "getHits", "getHits"]

[[], [1], [2], [3], [4], [300], [300], [301]]

Output

[null, null, null, null, 3, null, 4, 3]

Explanation

HitCounter hitCounter = new HitCounter();

hitCounter.hit(1); // hit at timestamp 1.

hitCounter.hit(2); // hit at timestamp 2.

hitCounter.hit(3); // hit at timestamp 3.

hitCounter.getHits(4); // get hits at timestamp 4, return 3.

hitCounter.hit(300); // hit at timestamp 300.

hitCounter.getHits(300); // get hits at timestamp 300, return 4.

hitCounter.getHits(301); // get hits at timestamp 301, return 3.

**Constraints:**

* 1 <= timestamp <= 2 \* 109
* All the calls are being made to the system in chronological order (i.e., timestamp is monotonically increasing).
* At most 300 calls will be made to hit and getHits.

**Follow up:** What if the number of hits per second could be huge? Does your design scale?

# Answer:

private Queue<int> queue;

/\*\* Initialize your data structure here. \*/

public HitCounter() {

queue = new Queue<int>();

}

/\*\* Record a hit.

@param timestamp - The current timestamp (in seconds granularity). \*/

public void Hit(int timestamp) {

queue.Enqueue(timestamp);

}

/\*\* Return the number of hits in the past 5 minutes.

@param timestamp - The current timestamp (in seconds granularity). \*/

public int GetHits(int timestamp) {

while(queue.Count > 0 && timestamp - queue.Peek() >= 300)

queue.Dequeue();

return queue.Count;

}

# 224. Basic Calculator

Given a string s representing a valid expression, implement a basic calculator to evaluate it, and return *the result of the evaluation*.

**Note:** You are **not** allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().

**Example 1:**

**Input:** s = "1 + 1"

**Output:** 2

**Example 2:**

**Input:** s = " 2-1 + 2 "

**Output:** 3

**Example 3:**

**Input:** s = "(1+(4+5+2)-3)+(6+8)"

**Output:** 23

**Constraints:**

* 1 <= s.length <= 3 \* 105
* s consists of digits, '+', '-', '(', ')', and ' '.
* s represents a valid expression.
* '+' is **not** used as a unary operation (i.e., "+1" and "+(2 + 3)" is invalid).
* '-' could be used as a unary operation (i.e., "-1" and "-(2 + 3)" is valid).
* There will be no two consecutive operators in the input.
* Every number and running calculation will fit in a signed 32-bit integer.

# Answer:

public int Calculate(string s)

{

int result = 0;

bool minusLastSeen = false;

bool evenNumberOfMinusesOnStack = true;

int currentNum = 0;

Stack<bool> signStack = new Stack<bool>();

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

{

char c = s[i];

if (c == ' ')

continue;

if (char.IsDigit(c))

{

currentNum \*= 10;

currentNum += (currentNum < 0 ? -1 : 1) \* (c - '0');

if (minusLastSeen)

{

currentNum \*= -1;

minusLastSeen = false;

}

}

else

{

result += (evenNumberOfMinusesOnStack ? 1 : -1) \* currentNum;

currentNum = 0;

if (c == '-')

{

minusLastSeen = true;

}

else

{

if (c == '(')

{

if (minusLastSeen)

{

evenNumberOfMinusesOnStack = !evenNumberOfMinusesOnStack;

signStack.Push(false);

}

else

{

signStack.Push(true);

}

}

else if (c == ')')

{

if (!signStack.Pop())

{

evenNumberOfMinusesOnStack = !evenNumberOfMinusesOnStack;

}

}

minusLastSeen = false;

}

}

}

result += currentNum;

return result;

}

# 33. Search in Rotated Sorted Array

There is an integer array nums sorted in ascending order (with **distinct** values).

Prior to being passed to your function, nums is **possibly rotated** at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (**0-indexed**). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums **after** the possible rotation and an integer target, return *the index of*target*if it is in*nums*, or*-1*if it is not in*nums.

You must write an algorithm with O(log n) runtime complexity.

**Example 1:**

**Input:** nums = [4,5,6,7,0,1,2], target = 0

**Output:** 4

**Example 2:**

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

**Output:** -1

**Example 3:**

**Input:** nums = [1], target = 0

**Output:** -1

**Constraints:**

* 1 <= nums.length <= 5000
* -104 <= nums[i] <= 104
* All values of nums are **unique**.
* nums is an ascending array that is possibly rotated.
* -104 <= target <= 104

# Answer:

public int Search(int[] nums, int target) {

int L=0;

int R=nums.Length-1;

if(nums== null || nums.Length == 0){

return 0;

}

while(L<=R)

{

//3

if(nums[L]== target){

return L;

}

L++;

}

return -1;

}

# 443. String Compression

Given an array of characters chars, compress it using the following algorithm:

Begin with an empty string s. For each group of **consecutive repeating characters** in chars:

* If the group's length is 1, append the character to s.
* Otherwise, append the character followed by the group's length.

The compressed string s **should not be returned separately**, but instead, be stored **in the input character array chars**. Note that group lengths that are 10 or longer will be split into multiple characters in chars.

After you are done **modifying the input array,** return *the new length of the array*.

You must write an algorithm that uses only constant extra space.

**Example 1:**

**Input:** chars = ["a","a","b","b","c","c","c"]

**Output:** Return 6, and the first 6 characters of the input array should be: ["a","2","b","2","c","3"]

**Explanation:** The groups are "aa", "bb", and "ccc". This compresses to "a2b2c3".

**Example 2:**

**Input:** chars = ["a"]

**Output:** Return 1, and the first character of the input array should be: ["a"]

**Explanation:** The only group is "a", which remains uncompressed since it's a single character.

**Example 3:**

**Input:** chars = ["a","b","b","b","b","b","b","b","b","b","b","b","b"]

**Output:** Return 4, and the first 4 characters of the input array should be: ["a","b","1","2"].

**Explanation:** The groups are "a" and "bbbbbbbbbbbb". This compresses to "ab12".

**Constraints:**

* 1 <= chars.length <= 2000
* chars[i] is a lowercase English letter, uppercase English letter, digit, or symbol.

# Answer:

public int Compress(char[] chars) {

int i = 0;

int ptr = 0;

while (i < chars.Length)

{

int count = 1;

char ch = chars[i];

i++;

while (i < chars.Length && chars[i] == ch)

{

count++;

i++;

}

chars[ptr++] = ch;

if (count > 1)

{

foreach (var item in count.ToString().ToCharArray())

{

chars[ptr++] = item;

}

}

}

return ptr;

}

# 297. Serialize and Deserialize Binary Tree

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

**Clarification:** The input/output format is the same as [how LeetCode serializes a binary tree](https://leetcode.com/faq/#binary-tree). You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

**Example 1:**

Diagram

Description automatically generated

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

**Output:** [1,2,3,null,null,4,5]

**Example 2:**

**Input:** root = []

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* -1000 <= Node.val <= 1000

# Answer:

public string serialize(TreeNode root) {

if (root == null) return "";

var result = new List<string>();

var queue = new Queue<TreeNode>();

queue.Enqueue(root);

while (queue.Any()) {

var size = queue.Count;

for (int s = 0; s < size; s++) {

var cur = queue.Dequeue();

if (cur == null) {

result.Add("n");

continue;

}

result.Add($"{cur.val}");

queue.Enqueue(cur.left);

queue.Enqueue(cur.right);

}

}

return string.Join(",", result);

}

// Decodes your encoded data to tree.

public TreeNode deserialize(string data) {

if (data == "") return null;

var allNodes = data.Split(',');

var root = new TreeNode(int.Parse(allNodes.First()));

var i = 1;

var queue = new Queue<TreeNode>();

queue.Enqueue(root);

while (queue.Any()) {

var size = queue.Count;

for (int s = 0; s < queue.Count; s++) {

var cur = queue.Dequeue();

var leftValue = allNodes[i];

i++;

if (leftValue != "n") {

}

var rightValue = allNodes[i];

i++;

if (rightValue != "n") {

cur.right = new TreeNode(int.Parse(rightValue));

queue.Enqueue(cur.right);

}

}

}

return root;

}

# 692. Top K Frequent Words

Given an array of strings words and an integer k, return *the*k*most frequent strings*.

Return the answer **sorted** by **the frequency** from highest to lowest. Sort the words with the same frequency by their **lexicographical order**.

**Example 1:**

**Input:** words = ["i","love","leetcode","i","love","coding"], k = 2

**Output:** ["i","love"]

**Explanation:** "i" and "love" are the two most frequent words.

Note that "i" comes before "love" due to a lower alphabetical order.

**Example 2:**

**Input:** words = ["the","day","is","sunny","the","the","the","sunny","is","is"], k = 4

**Output:** ["the","is","sunny","day"]

**Explanation:** "the", "is", "sunny" and "day" are the four most frequent words, with the number of occurrence being 4, 3, 2 and 1 respectively.

**Constraints:**

* 1 <= words.length <= 500
* 1 <= words[i] <= 10
* words[i] consists of lowercase English letters.
* k is in the range [1, The number of **unique** words[i]]

**Follow-up:** Could you solve it in O(n log(k)) time and O(n) extra space?

# Answer:

public IList<string> TopKFrequent(string[] words, int k) {

// Input validation

if(words == null)

return null;

IList<string> result = new List<string>();

if(words.Length == 0 || k == 0)

{

return result;

}

IDictionary<string, int> dict = new Dictionary<string, int>();

foreach(string str in words)

{

if(dict.TryGetValue(str, out int val))

{

dict[str]++;

}

else

{

dict.Add(str, 1);

}

}

result = dict.OrderByDescending(r => r.Value).ThenBy(v => v.Key).Select(s => s.Key).Take(k).ToList();

return result;

}

# 10. Regular Expression Matching

Given an input string s and a pattern p, implement regular expression matching with support for '.' and '\*' where:

'.' Matches any single character.​​​​

'\*' Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial).

**Example 1:**

**Input:** s = "aa", p = "a"

**Output:** false

**Explanation:** "a" does not match the entire string "aa".

**Example 2:**

**Input:** s = "aa", p = "a\*"

**Output:** true

**Explanation:** '\*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

**Example 3:**

**Input:** s = "ab", p = ".\*"

**Output:** true

**Explanation:** ".\*" means "zero or more (\*) of any character (.)".

**Constraints:**

* 1 <= s.length <= 20
* 1 <= p.length <= 30
* s contains only lowercase English letters.
* p contains only lowercase English letters, '.', and '\*'.
* It is guaranteed for each appearance of the character '\*', there will be a previous valid character to match.

# Answer:

public bool IsMatch(String text, String pattern)

{

bool[,] dp = new bool[text.Length + 1, pattern.Length + 1];

dp[text.Length, pattern.Length] = true;

for (int i = text.Length; i >= 0; i--)

{

for (int j = pattern.Length - 1; j >= 0; j--)

{

Boolean first\_match = (i < text.Length &&

(pattern[j] == text[i] ||

pattern[j] == '.'));

if (j + 1 < pattern.Length && pattern[j + 1] == '\*')

{

dp[i, j] = dp[i, j + 2] || first\_match && dp[i + 1, j];

}

else

{

dp[i, j] = first\_match && dp[i + 1, j + 1];

}

}

}

return dp[0, 0];

}

# 312. Burst Balloons

You are given n balloons, indexed from 0 to n - 1. Each balloon is painted with a number on it represented by an array nums. You are asked to burst all the balloons.

If you burst the ith balloon, you will get nums[i - 1] \* nums[i] \* nums[i + 1] coins. If i - 1 or i + 1 goes out of bounds of the array, then treat it as if there is a balloon with a 1 painted on it.

Return *the maximum coins you can collect by bursting the balloons wisely*.

**Example 1:**

**Input:** nums = [3,1,5,8]

**Output:** 167

**Explanation:**

nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []

coins = 3\*1\*5 + 3\*5\*8 + 1\*3\*8 + 1\*8\*1 = 167

**Example 2:**

**Input:** nums = [1,5]

**Output:** 10

**Constraints:**

* n == nums.length
* 1 <= n <= 300
* 0 <= nums[i] <= 100

# Answer:

public int MaxCoins(int[] nums) {

int n=nums.Length;

var dp=new int[n][];

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

{

dp[i]=new int[n];

}

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

{

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

{

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

{

int lv=i==0?1:nums[i-1];

int rv=j==nums.Length-1?1:nums[j+1];

int cur=nums[k]\*lv\*rv;

int rest=(k==0?0:dp[i][k-1])+(k==n-1?0:dp[k+1][j]);

dp[i][j]=Math.Max(cur+rest,dp[i][j]);

}

}

}

return dp[0][n-1];

}

# 460. LFU Cache

Design and implement a data structure for a [Least Frequently Used (LFU)](https://en.wikipedia.org/wiki/Least_frequently_used) cache.

Implement the LFUCache class:

* LFUCache(int capacity) Initializes the object with the capacity of the data structure.
* int get(int key) Gets the value of the key if the key exists in the cache. Otherwise, returns -1.
* void put(int key, int value) Update the value of the key if present, or inserts the key if not already present. When the cache reaches its capacity, it should invalidate and remove the **least frequently used** key before inserting a new item. For this problem, when there is a **tie** (i.e., two or more keys with the same frequency), the **least recently used** key would be invalidated.

To determine the least frequently used key, a **use counter** is maintained for each key in the cache. The key with the smallest **use counter** is the least frequently used key.

When a key is first inserted into the cache, its **use counter** is set to 1 (due to the put operation). The **use counter** for a key in the cache is incremented either a get or put operation is called on it.

The functions get and put must each run in O(1) average time complexity.

**Example 1:**

**Input**

["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get", "get", "get"]

[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]

**Output**

[null, null, null, 1, null, -1, 3, null, -1, 3, 4]

**Explanation**

// cnt(x) = the use counter for key x

// cache=[] will show the last used order for tiebreakers (leftmost element is most recent)

LFUCache lfu = new LFUCache(2);

lfu.put(1, 1); // cache=[1,\_], cnt(1)=1

lfu.put(2, 2); // cache=[2,1], cnt(2)=1, cnt(1)=1

lfu.get(1); // return 1

// cache=[1,2], cnt(2)=1, cnt(1)=2

lfu.put(3, 3); // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2.

  // cache=[3,1], cnt(3)=1, cnt(1)=2

lfu.get(2); // return -1 (not found)

lfu.get(3); // return 3

// cache=[3,1], cnt(3)=2, cnt(1)=2

lfu.put(4, 4); // Both 1 and 3 have the same cnt, but 1 is LRU, invalidate 1.

// cache=[4,3], cnt(4)=1, cnt(3)=2

lfu.get(1); // return -1 (not found)

lfu.get(3); // return 3

// cache=[3,4], cnt(4)=1, cnt(3)=3

lfu.get(4); // return 4

// cache=[4,3], cnt(4)=2, cnt(3)=3

**Constraints:**

* 0 <= capacity <= 104
* 0 <= key <= 105
* 0 <= value <= 109
* At most 2 \* 105 calls will be made to get and put.

# Answer:

Dictionary<int,int> cachemap = new Dictionary<int,int>();

Dictionary<int,int> frequency = new Dictionary<int,int>();

Dictionary<int,LinkedList<int>> freqmap = new Dictionary<int,LinkedList<int>>();

int capacity;

public LFUCache(int capacity) {

this.capacity = capacity;

}

public int Get(int key) {

if (cachemap.ContainsKey(key))

{

updatefreq(key);

return cachemap[key];

}

else

return -1;

}

public void updatefreq(int key)

{

int f = frequency[key];

freqmap[f].Remove(key);

f = ++frequency[key];

if (!freqmap.ContainsKey(f))

freqmap.Add(f, new LinkedList<int>());

freqmap[f].AddLast(key);

}

public void Put(int key, int value) {

if (capacity == 0)

return;

if (cachemap.ContainsKey(key))

{

updatefreq(key);

cachemap[key] = value;

return;

}

if (cachemap.Count >= capacity)

{

//find the item in the freqmap;

int item = 0;

while(freqmap.FirstOrDefault().Value.Count==0)

freqmap.Remove(freqmap.FirstOrDefault().Key);

item = freqmap.FirstOrDefault().Value.First();

int f = freqmap.FirstOrDefault().Key;

freqmap[f].Remove(item);

cachemap.Remove(item);

frequency.Remove(item);

}

cachemap.Add(key, value);

frequency.Add(key, 1);

if (!freqmap.ContainsKey(1))

{

freqmap.Add(1, new LinkedList<int>());

}

freqmap[1].AddLast(key);

}

# 210. Course Schedule II

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you **must** take course bi first if you want to take course ai.

* For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return *the ordering of courses you should take to finish all courses*. If there are many valid answers, return **any** of them. If it is impossible to finish all courses, return **an empty array**.

**Example 1:**

**Input:** numCourses = 2, prerequisites = [[1,0]]

**Output:** [0,1]

**Explanation:** There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1].

**Example 2:**

**Input:** numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]

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

**Explanation:** There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0.

So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3].

**Example 3:**

**Input:** numCourses = 1, prerequisites = []

**Output:** [0]

**Constraints:**

* 1 <= numCourses <= 2000
* 0 <= prerequisites.length <= numCourses \* (numCourses - 1)
* prerequisites[i].length == 2
* 0 <= ai, bi < numCourses
* ai != bi
* All the pairs [ai, bi] are **distinct**.

# Answer:

public int[] FindOrder(int numCourses, int[][] prerequisites) {

List<int>[] g = new List<int>[numCourses];

int[] indegree = new int[numCourses];

Queue<int> q = new Queue<int>();

Stack<int> s = new Stack<int>();

int[] res = new int[numCourses];

foreach (var item in prerequisites)

{

if (g[item[0]] == null)

g[item[0]] = new List<int>();

g[item[0]].Add(item[1]);

indegree[item[1]] += 1;

}

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

if (indegree[i] == 0)

q.Enqueue(i);

while (q.Count > 0)

{

int cur = q.Dequeue();

if (g[cur] != null)

foreach (var item in g[cur])

if (--indegree[item] == 0)

q.Enqueue(item);

s.Push(cur);

}

if (s.Count != numCourses)

return new int[] { };

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

res[i] = s.Pop();

return res;

}

# 317. Shortest Distance from All Buildings

You are given an m x n grid grid of values 0, 1, or 2, where:

* each 0 marks **an empty land** that you can pass by freely,
* each 1 marks **a building** that you cannot pass through, and
* each 2 marks **an obstacle** that you cannot pass through.

You want to build a house on an empty land that reaches all buildings in the **shortest total travel** distance. You can only move up, down, left, and right.

Return the ***shortest travel distance*** for such a house. If it is not possible to build such a house according to the above rules, return -1.

The **total travel distance** is the sum of the distances between the houses of the friends and the meeting point.

The distance is calculated using [Manhattan Distance](http://en.wikipedia.org/wiki/Taxicab_geometry), where distance(p1, p2) = |p2.x - p1.x| + |p2.y - p1.y|.

**Example 1:**

Calendar

Description automatically generated

**Input:** grid = [[1,0,2,0,1],[0,0,0,0,0],[0,0,1,0,0]]

**Output:** 7

**Explanation:** Given three buildings at (0,0), (0,4), (2,2), and an obstacle at (0,2).

The point (1,2) is an ideal empty land to build a house, as the total travel distance of 3+3+1=7 is minimal.

So return 7.

**Example 2:**

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

**Output:** 1

**Example 3:**

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

**Output:** -1

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 50
* grid[i][j] is either 0, 1, or 2.
* There will be **at least one** building in the grid.

# Answer:

public int ShortestDistance(int[][] grid) {

if(grid == null || grid[0].Length == 0)

return -1;

int m = grid.Length, n = grid[0].Length;

int[,] reach = new int[m,n]; // number of building can be reached from [i,j]

int[,] distance = new int[m,n]; // total distance from [i,j] to all buildings

int building = 0; // number of buildings

Queue<(int,int)> queue = new Queue<(int,int)>();

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

{

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

{

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

{

building++;

BFS(grid, i, j, reach, distance);

}

}

}

int minLen = Int32.MaxValue;

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

{

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

{

if(reach[i,j] == building)

{

minLen = Math.Min(minLen, distance[i,j]);

}

}

}

return minLen == Int32.MaxValue? -1 : minLen;

}

private void BFS(int[][] grid, int row, int col, int[,] reach, int[,] distance)

{

int m = grid.Length, n = grid[0].Length;

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

bool[,] visited = new bool[m,n];

Queue<(int,int)> queue = new Queue<(int,int)>();

queue.Enqueue((row,col));

visited[row,col] = true;

int curDistance = 0;

while(queue.Any())

{

curDistance++;

int size = queue.Count;

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

{

var curr = queue.Dequeue();

for(int j = 0; j < 4; j++)

{

int newRow = curr.Item1 + dir[j, 0];

int newCol = curr.Item2 + dir[j, 1];

if(newRow >= 0 && newRow < m && newCol >= 0 && newCol < n && !visited[newRow,newCol] && grid[newRow][newCol] == 0)

{

reach[newRow,newCol]++;

distance[newRow,newCol] += curDistance;

queue.Enqueue((newRow,newCol));

visited[newRow,newCol] = true;

}

}

}

}

}

# 173. Binary Search Tree Iterator

Implement the BSTIterator class that represents an iterator over the [**in-order traversal**](https://en.wikipedia.org/wiki/Tree_traversal#In-order_(LNR)) of a binary search tree (BST):

* BSTIterator(TreeNode root) Initializes an object of the BSTIterator class. The root of the BST is given as part of the constructor. The pointer should be initialized to a non-existent number smaller than any element in the BST.
* boolean hasNext() Returns true if there exists a number in the traversal to the right of the pointer, otherwise returns false.
* int next() Moves the pointer to the right, then returns the number at the pointer.

Notice that by initializing the pointer to a non-existent smallest number, the first call to next() will return the smallest element in the BST.

You may assume that next() calls will always be valid. That is, there will be at least a next number in the in-order traversal when next() is called.

**Example 1:**

Shape

Description automatically generated

**Input**

["BSTIterator", "next", "next", "hasNext", "next", "hasNext", "next", "hasNext", "next", "hasNext"]

[[[7, 3, 15, null, null, 9, 20]], [], [], [], [], [], [], [], [], []]

**Output**

[null, 3, 7, true, 9, true, 15, true, 20, false]

**Explanation**

BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]);

bSTIterator.next(); // return 3

bSTIterator.next(); // return 7

bSTIterator.hasNext(); // return True

bSTIterator.next(); // return 9

bSTIterator.hasNext(); // return True

bSTIterator.next(); // return 15

bSTIterator.hasNext(); // return True

bSTIterator.next(); // return 20

bSTIterator.hasNext(); // return False

**Constraints:**

* The number of nodes in the tree is in the range [1, 105].
* 0 <= Node.val <= 106
* At most 105 calls will be made to hasNext, and next.

**Follow up:**

* Could you implement next() and hasNext() to run in average O(1) time and use O(h) memory, where h is the height of the tree?

# Answer:

public class BSTIterator {

Stack<TreeNode> stack = new Stack<TreeNode>();

public BSTIterator(TreeNode root) {

while (root != null) {

stack.Push(root);

root = root.left;

}

}

/\*\* @return the next smallest number \*/

public int Next() {

var next = stack.Pop();

if (next.right != null) {

var cur = next.right;

while (cur != null) {

stack.Push(cur);

cur = cur.left;

}

}

return next.val;

}

/\*\* @return whether we have a next smallest number \*/

public bool HasNext() {

return stack.Any();

}

}

# 1405. Longest Happy String

A string s is called **happy** if it satisfies the following conditions:

* s only contains the letters 'a', 'b', and 'c'.
* s does not contain any of "aaa", "bbb", or "ccc" as a substring.
* s contains **at most** a occurrences of the letter 'a'.
* s contains **at most** b occurrences of the letter 'b'.
* s contains **at most** c occurrences of the letter 'c'.

Given three integers a, b, and c, return *the****longest possible happy****string*. If there are multiple longest happy strings, return *any of them*. If there is no such string, return *the empty string*"".

A **substring** is a contiguous sequence of characters within a string.

**Example 1:**

**Input:** a = 1, b = 1, c = 7

**Output:** "ccaccbcc"

**Explanation:** "ccbccacc" would also be a correct answer.

**Example 2:**

**Input:** a = 7, b = 1, c = 0

**Output:** "aabaa"

**Explanation:** It is the only correct answer in this case.

**Constraints:**

* 0 <= a, b, c <= 100
* a + b + c > 0

# Answer:

public string LongestDiverseString(int a, int b, int c)

{

var sb = new StringBuilder();

var pq = new PriorityQueue<char, int>(Comparer<int>.Create((a, b) => b.CompareTo(a)));

if (a != 0) pq.Enqueue('a', a);

if (b != 0) pq.Enqueue('b', b);

if (c != 0) pq.Enqueue('c', c);

while (pq.Count != 0)

{

pq.TryDequeue(out var currChar, out var currCount);

if (sb.Length >= 2 && sb[sb.Length - 1] == sb[sb.Length - 2] && sb[sb.Length - 1] == currChar)

{

if (!pq.TryDequeue(out var nextChar, out var nextCount))

break;

sb.Append(nextChar);

nextCount--;

if (nextCount != 0)

pq.Enqueue(nextChar, nextCount);

}

else

{

currCount--;

sb.Append(currChar);

}

if (currCount != 0)

pq.Enqueue(currChar, currCount);

}

return sb.ToString();

}

# 295. Find Median from Data Stream

The **median** is the middle value in an ordered integer list. If the size of the list is even, there is no middle value and the median is the mean of the two middle values.

* For example, for arr = [2,3,4], the median is 3.
* For example, for arr = [2,3], the median is (2 + 3) / 2 = 2.5.

Implement the MedianFinder class:

* MedianFinder() initializes the MedianFinder object.
* void addNum(int num) adds the integer num from the data stream to the data structure.
* double findMedian() returns the median of all elements so far. Answers within 10-5 of the actual answer will be accepted.

**Example 1:**

**Input**

["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]

[[], [1], [2], [], [3], []]

**Output**

[null, null, null, 1.5, null, 2.0]

**Explanation**

MedianFinder medianFinder = new MedianFinder();

medianFinder.addNum(1); // arr = [1]

medianFinder.addNum(2); // arr = [1, 2]

medianFinder.findMedian(); // return 1.5 (i.e., (1 + 2) / 2)

medianFinder.addNum(3); // arr[1, 2, 3]

medianFinder.findMedian(); // return 2.0

**Constraints:**

* -105 <= num <= 105
* There will be at least one element in the data structure before calling findMedian.
* At most 5 \* 104 calls will be made to addNum and findMedian.

**Follow up:**

* If all integer numbers from the stream are in the range [0, 100], how would you optimize your solution?
* If 99% of all integer numbers from the stream are in the range [0, 100], how would you optimize your solution?

# Answer:

List<int> Numbers;

public MedianFinder()

{

Numbers = new List<int>();

}

public void AddNum(int num)

{

// Find where to insert this num in Numbers using Binary Search

// **NOTE:** Binary Search is logarithmic time complexity O(logn)

int position = Numbers.BinarySearch(num);

// So if BinarySearch returns -1 it means we should insert at the first position

if (position < 0)

{

position = ~position; // Bitwise complement of -1 is 0

}

Numbers.Insert(position, num);

}

public double FindMedian()

{

int count = Numbers.Count;

if (count % 2 == 0)

{

// Even number of elements

return (double)((Numbers[count / 2 - 1] + Numbers[count / 2]) \* 0.5);

}

else

{

// Odd number of elements

return (double)(Numbers[count / 2]);

}

}

}

/\*\*

\* Your MedianFinder object will be instantiated and called as such:

\* MedianFinder obj = new MedianFinder();

\* obj.AddNum(num);

\* double param\_2 = obj.FindMedian();

\*/

# 348. Design Tic-Tac-Toe

Assume the following rules are for the tic-tac-toe game on an n x n board between two players:

1. A move is guaranteed to be valid and is placed on an empty block.
2. Once a winning condition is reached, no more moves are allowed.
3. A player who succeeds in placing n of their marks in a horizontal, vertical, or diagonal row wins the game.

Implement the TicTacToe class:

* TicTacToe(int n) Initializes the object the size of the board n.
* int move(int row, int col, int player) Indicates that the player with id player plays at the cell (row, col) of the board. The move is guaranteed to be a valid move.

**Example 1:**

**Input**

["TicTacToe", "move", "move", "move", "move", "move", "move", "move"]

[[3], [0, 0, 1], [0, 2, 2], [2, 2, 1], [1, 1, 2], [2, 0, 1], [1, 0, 2], [2, 1, 1]]

**Output**

[null, 0, 0, 0, 0, 0, 0, 1]

**Explanation**

TicTacToe ticTacToe = new TicTacToe(3);

Assume that player 1 is "X" and player 2 is "O" in the board.

ticTacToe.move(0, 0, 1); // return 0 (no one wins)

|X| | |

| | | | // Player 1 makes a move at (0, 0).

| | | |

ticTacToe.move(0, 2, 2); // return 0 (no one wins)

|X| |O|

| | | | // Player 2 makes a move at (0, 2).

| | | |

ticTacToe.move(2, 2, 1); // return 0 (no one wins)

|X| |O|

| | | | // Player 1 makes a move at (2, 2).

| | |X|

ticTacToe.move(1, 1, 2); // return 0 (no one wins)

|X| |O|

| |O| | // Player 2 makes a move at (1, 1).

| | |X|

ticTacToe.move(2, 0, 1); // return 0 (no one wins)

|X| |O|

| |O| | // Player 1 makes a move at (2, 0).

|X| |X|

ticTacToe.move(1, 0, 2); // return 0 (no one wins)

|X| |O|

|O|O| | // Player 2 makes a move at (1, 0).

|X| |X|

ticTacToe.move(2, 1, 1); // return 1 (player 1 wins)

|X| |O|

|O|O| | // Player 1 makes a move at (2, 1).

|X|X|X|

**Constraints:**

* 2 <= n <= 100
* player is 1 or 2.
* 0 <= row, col < n
* (row, col) are **unique** for each different call to move.
* At most n2 calls will be made to move.

**Follow-up:** Could you do better than O(n2) per move() operation?

# Answer:

public class TicTacToe {

/\*\* Initialize your data structure here. \*/

int diagonal = 0;

int antidiagonal = 0;

int[] rows;

int[] cols;

int n;

public TicTacToe(int n) {

this.n = n;

rows = new int[n];

cols = new int[n];

}

/\*\* Player {player} makes a move at ({row}, {col}).

@param row The row of the board.

@param col The column of the board.

@param player The player, can be either 1 or 2.

@return The current winning condition, can be either:

0: No one wins.

1: Player 1 wins.

2: Player 2 wins. \*/

public int Move(int row, int col, int player) {

var num = 1;

if (player == 2) {

num = -1;

}

rows[col] += num;

cols[row] += num;

if (row == col) {

diagonal += num;

}

if (row == n-col-1) {

antidiagonal += num;

}

if (rows[col] == n || cols[row] == n || diagonal == n || antidiagonal == n)

{

return 1;

}

else if (rows[col] == -n || cols[row] == -n || diagonal == -n || antidiagonal == -n)

{

return 2;

}

else {

return 0;

}

}

}

# 1275. Find Winner on a Tic Tac Toe Game

**Tic-tac-toe** is played by two players A and B on a 3 x 3 grid. The rules of Tic-Tac-Toe are:

* Players take turns placing characters into empty squares ' '.
* The first player A always places 'X' characters, while the second player B always places 'O' characters.
* 'X' and 'O' characters are always placed into empty squares, never on filled ones.
* The game ends when there are **three** of the same (non-empty) character filling any row, column, or diagonal.
* The game also ends if all squares are non-empty.
* No more moves can be played if the game is over.

Given a 2D integer array moves where moves[i] = [rowi, coli] indicates that the ith move will be played on grid[rowi][coli]. return *the winner of the game if it exists* (A or B). In case the game ends in a draw return "Draw". If there are still movements to play return "Pending".

You can assume that moves is valid (i.e., it follows the rules of **Tic-Tac-Toe**), the grid is initially empty, and A will play first.

**Example 1:**

A picture containing shoji, crossword puzzle, clock

Description automatically generated

**Input:** moves = [[0,0],[2,0],[1,1],[2,1],[2,2]]

**Output:** "A"

**Explanation:** A wins, they always play first.

**Example 2:**

A picture containing crossword puzzle, shoji

Description automatically generated

**Input:** moves = [[0,0],[1,1],[0,1],[0,2],[1,0],[2,0]]

**Output:** "B"

**Explanation:** B wins.

**Example 3:**

A picture containing crossword puzzle, shoji, clock, clipart

Description automatically generated

**Input:** moves = [[0,0],[1,1],[2,0],[1,0],[1,2],[2,1],[0,1],[0,2],[2,2]]

**Output:** "Draw"

**Explanation:** The game ends in a draw since there are no moves to make.

**Constraints:**

* 1 <= moves.length <= 9
* moves[i].length == 2
* 0 <= rowi, coli <= 2
* There are no repeated elements on moves.
* moves follow the rules of tic tac toe.

# Answer:

public string Tictactoe(int[][] moves)

{

int[] row = new int[3], col = new int[3];

int diag1 = 0, diag2 = 0;

for(int i = 0; i < moves.Length; i++)

{

int x = moves[i][0], y = moves[i][1];

var inc = i % 2 == 0 ? 1 : -1;

row[x] += inc;

col[y] += inc;

if(x == y) diag1 += inc;

if(x + y == 2) diag2 += inc;

if(Math.Abs(row[x]) == 3 || Math.Abs(col[y]) == 3 || Math.Abs(diag1) == 3 || Math.Abs(diag2) == 3)

return i % 2 == 0 ? "A" : "B";

}

return moves.Length == 9 ? "Draw" : "Pending";

}

# 438. Find All Anagrams in a String

Given two strings s and p, return *an array of all the start indices of*p*'s anagrams in*s. You may return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Example 1:**

**Input:** s = "cbaebabacd", p = "abc"

**Output:** [0,6]

**Explanation:**

The substring with start index = 0 is "cba", which is an anagram of "abc".

The substring with start index = 6 is "bac", which is an anagram of "abc".

**Example 2:**

**Input:** s = "abab", p = "ab"

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

**Explanation:**

The substring with start index = 0 is "ab", which is an anagram of "ab".

The substring with start index = 1 is "ba", which is an anagram of "ab".

The substring with start index = 2 is "ab", which is an anagram of "ab".

**Constraints:**

* 1 <= s.length, p.length <= 3 \* 104
* s and p consist of lowercase English letters.

# Answer:

public IList<int> FindAnagrams(string s, string p)

{

int[] freq1 = new int[26], freq2 = new int[26];

var result = new List<int>();

foreach(var ch in p)

freq1[ch - 'a']++;

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

{

freq2[s[i] - 'a']++;

if(i - p.Length >= 0)

freq2[s[i - p.Length] -'a']--;

if(i >= p.Length - 1 && IsEqual(freq1, freq2))

result.Add(i - p.Length + 1);

}

return result;

}

private bool IsEqual(int[] freq1, int[] freq2)

{

for(int i = 0; i < freq1.Length; i++)

if(freq1[i] != freq2[i])

return false;

return true;

}

# 25. Reverse Nodes in k-Group

Given the head of a linked list, reverse the nodes of the list k at a time, and return *the modified list*.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.

You may not alter the values in the list's nodes, only nodes themselves may be changed.

**Example 1:**

Diagram

Description automatically generated

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

**Output:** [2,1,4,3,5]

**Example 2:**

Diagram

Description automatically generated

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

**Output:** [3,2,1,4,5]

**Constraints:**

* The number of nodes in the list is n.
* 1 <= k <= n <= 5000
* 0 <= Node.val <= 1000

**Follow-up:** Can you solve the problem in O(1) extra memory space?

# Answer:

public ListNode ReverseKGroup(ListNode head, int k) {

//find the length of list

if(head == null || head.next == null || k == 1)

{

return head;

}

int length = LengthOfList(head);

int groupsToReverse = length/k;

ListNode prevListTail = head;

ListNode curr = head;

ListNode resultHead = null;

while(groupsToReverse > 0)

{

ListNode prev = null;

ListNode next = null;

ListNode currentListHead = curr;

int numberOfNodesVisited = 0;

while(numberOfNodesVisited < k)

{

next = curr.next;

curr.next = prev;

prev = curr;

curr = next;

numberOfNodesVisited++;

}

if(resultHead == null)

{

resultHead = prev;

}

else{

prevListTail.next = prev;

prevListTail = currentListHead;

}

groupsToReverse--;

}

prevListTail.next = curr;

return resultHead;

}

public int LengthOfList(ListNode head)

{

int length = 0;

while(head != null)

{

length++;

head = head.next;

}

return length;

}

# 1344. Angle Between Hands of a Clock

Given two numbers, hour and minutes, return *the smaller angle (in degrees) formed between the*hour*and the*minute*hand*.

Answers within 10-5 of the actual value will be accepted as correct.

**Example 1:**

A black and white clock

Description automatically generated with medium confidence

**Input:** hour = 12, minutes = 30

**Output:** 165

**Example 2:**

A black and white clock

Description automatically generated with medium confidence

**Input:** hour = 3, minutes = 30

**Output:** 75

**Example 3:**

A black and white clock

Description automatically generated with medium confidence

**Input:** hour = 3, minutes = 15

**Output:** 7.5

**Constraints:**

* 1 <= hour <= 12
* 0 <= minutes <= 59

# Answer:

public double AngleClock(int hour, int minutes)

{

double hourLerp = (double) minutes / 60;

double hourWithOffset = (double)hour \* 5 + 5 \* hourLerp;

if (hourWithOffset > 60)

{

hourWithOffset -= 60;

}

double res = (Math.Abs(hourWithOffset - minutes) \* 6);

res = Math.Min(res, 360 - res);

return res;

}

# 44. Wildcard Matching

Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '\*' where:

* '?' Matches any single character.
* '\*' Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

**Example 1:**

**Input:** s = "aa", p = "a"

**Output:** false

**Explanation:** "a" does not match the entire string "aa".

**Example 2:**

**Input:** s = "aa", p = "\*"

**Output:** true

**Explanation:** '\*' matches any sequence.

**Example 3:**

**Input:** s = "cb", p = "?a"

**Output:** false

**Explanation:** '?' matches 'c', but the second letter is 'a', which does not match 'b'.

**Constraints:**

* 0 <= s.length, p.length <= 2000
* s contains only lowercase English letters.
* p contains only lowercase English letters, '?' or '\*'.

# Answer:

// Define other methods and classes here

public bool IsMatch(string s, string p)

{

if (s.Length == 0 && (p.Length == 0 || p.All(x => x == '\*')))

return true;

else if (s.Length == 0)

return false;

var nLastStarIndex = -1;

var nStarLastStart = -1;

var patternCharIndex = 0;

int i = 0;

do

{

while (patternCharIndex < p.Length && p[patternCharIndex] == '\*')

{

nLastStarIndex = patternCharIndex++;

nStarLastStart = i;

if (patternCharIndex == p.Length)

return true;

}

if ((nLastStarIndex >= 0) && (p.Length == patternCharIndex))

{

// Reached the end of pattern, but not string. Go back to the last wildcard and eat one more char from string

patternCharIndex = nLastStarIndex + 1;

i = ++nStarLastStart;

}

else if ((nStarLastStart >= 0) && (s.Length == i))

{

// Reached the end of string, but not pattern. This can't match?

return false;

}

else if ( (p.Length == patternCharIndex) || (s.Length == i) )

{

return false;

}

if (s[i] == p[patternCharIndex] || p[patternCharIndex] == '?')

{

// matched a char, increment both

patternCharIndex++;

i++;

}

else if (nLastStarIndex >= 0)

{

// This didn't match, just eat a char and return to star;

patternCharIndex = nLastStarIndex + 1;

i = ++nStarLastStart;

}

else

{

// This didn't match and there is no wildcard, can't match.

return false;

}

} while ((i < s.Length) || (patternCharIndex < p.Length));

return true;

}

# 151. Reverse Words in a String

Given an input string s, reverse the order of the **words**.

A **word** is defined as a sequence of non-space characters. The **words** in s will be separated by at least one space.

Return *a string of the words in reverse order concatenated by a single space.*

**Note** that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

**Example 1:**

**Input:** s = "the sky is blue"

**Output:** "blue is sky the"

**Example 2:**

**Input:** s = " hello world "

**Output:** "world hello"

**Explanation:** Your reversed string should not contain leading or trailing spaces.

**Example 3:**

**Input:** s = "a good example"

**Output:** "example good a"

**Explanation:** You need to reduce multiple spaces between two words to a single space in the reversed string.

**Constraints:**

* 1 <= s.length <= 104
* s contains English letters (upper-case and lower-case), digits, and spaces ' '.
* There is **at least one** word in s.

**Follow-up:**If the string data type is mutable in your language, can you solve it **in-place** with O(1) extra space?

# Answer:

public string ReverseWords(string s) {

List<string> words=new List<string>();

string buildWord=string.Empty;

string result=string.Empty;

s = s.Trim();

if(s==string.Empty){

return string.Empty;

}

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

if(s[i] != ' '){

buildWord+=s[i];

}

if((s[i] == ' ' && s[i-1] != ' ')|| i==s.Length-1){

words.Add(buildWord);

buildWord="";

}

}

for(int j=words.Count-1; j>=0; j--){

result += words[j] + ' ';

}

return result.Trim();

}

# 545. Boundary of Binary Tree

The **boundary** of a binary tree is the concatenation of the **root**, the **left boundary**, the **leaves** ordered from left-to-right, and the **reverse order** of the **right boundary**.

The **left boundary** is the set of nodes defined by the following:

* The root node's left child is in the left boundary. If the root does not have a left child, then the left boundary is **empty**.
* If a node in the left boundary and has a left child, then the left child is in the left boundary.
* If a node is in the left boundary, has **no** left child, but has a right child, then the right child is in the left boundary.
* The leftmost leaf is **not** in the left boundary.

The **right boundary** is similar to the **left boundary**, except it is the right side of the root's right subtree. Again, the leaf is **not** part of the **right boundary**, and the **right boundary** is empty if the root does not have a right child.

The **leaves** are nodes that do not have any children. For this problem, the root is **not** a leaf.

Given the root of a binary tree, return *the values of its****boundary***.

**Example 1:**

A picture containing text, clock, clipart

Description automatically generated

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

**Output:** [1,3,4,2]

**Explanation:**

- The left boundary is empty because the root does not have a left child.

- The right boundary follows the path starting from the root's right child 2 -> 4.

4 is a leaf, so the right boundary is [2].

- The leaves from left to right are [3,4].

Concatenating everything results in [1] + [] + [3,4] + [2] = [1,3,4,2].

**Example 2:**

Shape

Description automatically generated

**Input:** root = [1,2,3,4,5,6,null,null,null,7,8,9,10]

**Output:** [1,2,4,7,8,9,10,6,3]

**Explanation:**

- The left boundary follows the path starting from the root's left child 2 -> 4.

4 is a leaf, so the left boundary is [2].

- The right boundary follows the path starting from the root's right child 3 -> 6 -> 10.

10 is a leaf, so the right boundary is [3,6], and in reverse order is [6,3].

- The leaves from left to right are [4,7,8,9,10].

Concatenating everything results in [1] + [2] + [4,7,8,9,10] + [6,3] = [1,2,4,7,8,9,10,6,3].

**Constraints:**

* The number of nodes in the tree is in the range [1, 104].
* -1000 <= Node.val <= 1000

# Answer:

public class Solution {

private IList<int> result = new List<int>();

public IList<int> BoundaryOfBinaryTree(TreeNode root)

{

if(root == null)

return result;

// Add root to result

result.Add(root.val);

// This method will include left boundry nodes

TraverseLeftBoundary(root.left);

// This method will include leaves on left side of root

TraverseLeaves(root.left);

// This method will include leaves on right side of root

TraverseLeaves(root.right);

// This method will include right boundry nodes in reverse order

TraverseRightBoundary(root.right);

return result;

}

private void TraverseLeftBoundary(TreeNode node)

{

if(node == null) return;

// Avoiding Leaves

if(node.left == null && node.right == null) return;

if(node.left != null)

{

result.Add(node.val);

TraverseLeftBoundary(node.left);

}

else if(node.right != null)

{

result.Add(node.val);

TraverseLeftBoundary(node.right);

}

}

private void TraverseRightBoundary(TreeNode node)

{

if(node == null)

return;

// Avoiding Leaves

if(node.left == null && node.right == null)

return;

if(node.right != null)

{

// As the nodes need to be added in reverse order

TraverseRightBoundary(node.right);

result.Add(node.val);

}

else if(node.left != null)

{

// As the nodes need to be added in reverse order

TraverseRightBoundary(node.left);

result.Add(node.val);

}

}

private void TraverseLeaves(TreeNode node)

{

if(node == null)

return;

// This will include leaves in result list

if(node.left == null && node.right == null)

{

result.Add(node.val);

}

TraverseLeaves(node.left);

TraverseLeaves(node.right);

}

}

# 8. String to Integer (atoi)

Implement the myAtoi(string s) function, which converts a string to a 32-bit signed integer (similar to C/C++'s atoi function).

The algorithm for myAtoi(string s) is as follows:

1. Read in and ignore any leading whitespace.
2. Check if the next character (if not already at the end of the string) is '-' or '+'. Read this character in if it is either. This determines if the final result is negative or positive respectively. Assume the result is positive if neither is present.
3. Read in next the characters until the next non-digit character or the end of the input is reached. The rest of the string is ignored.
4. Convert these digits into an integer (i.e. "123" -> 123, "0032" -> 32). If no digits were read, then the integer is 0. Change the sign as necessary (from step 2).
5. If the integer is out of the 32-bit signed integer range [-231, 231 - 1], then clamp the integer so that it remains in the range. Specifically, integers less than -231 should be clamped to -231, and integers greater than 231 - 1 should be clamped to 231 - 1.
6. Return the integer as the final result.

**Note:**

* Only the space character ' ' is considered a whitespace character.
* **Do not ignore** any characters other than the leading whitespace or the rest of the string after the digits.

**Example 1:**

**Input:** s = "42"

**Output:** 42

**Explanation:** The underlined characters are what is read in, the caret is the current reader position.

Step 1: "42" (no characters read because there is no leading whitespace)

^

Step 2: "42" (no characters read because there is neither a '-' nor '+')

^

Step 3: "42" ("42" is read in)

^

The parsed integer is 42.

Since 42 is in the range [-231, 231 - 1], the final result is 42.

**Example 2:**

**Input:** s = " -42"

**Output:** -42

**Explanation:**

Step 1: " -42" (leading whitespace is read and ignored)

^

Step 2: " -42" ('-' is read, so the result should be negative)

^

Step 3: " -42" ("42" is read in)

^

The parsed integer is -42.

Since -42 is in the range [-231, 231 - 1], the final result is -42.

**Example 3:**

**Input:** s = "4193 with words"

**Output:** 4193

**Explanation:**

Step 1: "4193 with words" (no characters read because there is no leading whitespace)

^

Step 2: "4193 with words" (no characters read because there is neither a '-' nor '+')

^

Step 3: "4193 with words" ("4193" is read in; reading stops because the next character is a non-digit)

^

The parsed integer is 4193.

Since 4193 is in the range [-231, 231 - 1], the final result is 4193.

**Constraints:**

* 0 <= s.length <= 200
* s consists of English letters (lower-case and upper-case), digits (0-9), ' ', '+', '-', and '.'.

# Answer:

public int MyAtoi(string s) {

int index = 0, sign = 1, total = 0;

while (index < s.Length && s[index] == ' ')

{

index++;

}

sign = (index < s.Length && (s[index] == '+' || s[index] == '-')) ?

((s[index++] == '+' )? 1 : -1 ): 1;

while (index < s.Length)

{

int digit = s[index] - '0';

if (digit < 0 || 9 < digit) break;

if (int.MaxValue / 10 < total || int.MaxValue / 10 == total && int.MaxValue % 10 < digit)

return sign == -1 ? int.MinValue : int.MaxValue;

total = total \* 10 + digit;

index++;

}

return total \* sign;

}

# 1386. Cinema Seat Allocation

Shape

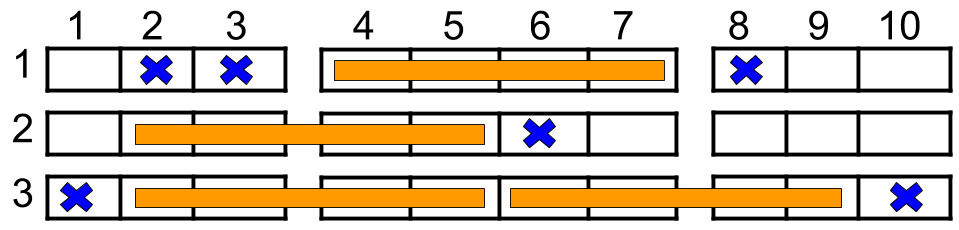
Description automatically generated with medium confidence

A cinema has n rows of seats, numbered from 1 to n and there are ten seats in each row, labelled from 1 to 10 as shown in the figure above.

Given the array reservedSeats containing the numbers of seats already reserved, for example, reservedSeats[i] = [3,8] means the seat located in row **3** and labelled with **8** is already reserved.

*Return the maximum number of four-person groups you can assign on the cinema seats.* A four-person group occupies four adjacent seats **in one single row**. Seats across an aisle (such as [3,3] and [3,4]) are not considered to be adjacent, but there is an exceptional case on which an aisle split a four-person group, in that case, the aisle split a four-person group in the middle, which means to have two people on each side.

**Example 1:**



**Input:** n = 3, reservedSeats = [[1,2],[1,3],[1,8],[2,6],[3,1],[3,10]]

**Output:** 4

**Explanation:** The figure above shows the optimal allocation for four groups, where seats mark with blue are already reserved and contiguous seats mark with orange are for one group.

**Example 2:**

**Input:** n = 2, reservedSeats = [[2,1],[1,8],[2,6]]

**Output:** 2

**Example 3:**

**Input:** n = 4, reservedSeats = [[4,3],[1,4],[4,6],[1,7]]

**Output:** 4

**Constraints:**

* 1 <= n <= 10^9
* 1 <= reservedSeats.length <= min(10\*n, 10^4)
* reservedSeats[i].length == 2
* 1 <= reservedSeats[i][0] <= n
* 1 <= reservedSeats[i][1] <= 10
* All reservedSeats[i] are distinct.

# Answer:

public int MaxNumberOfFamilies(int n, int[][] reservedSeats)

{

int result = 0;

var rows = new Dictionary<int, bool[]>();

foreach(var r in reservedSeats)

{

if(!rows.ContainsKey(r[0]))

rows[r[0]] = new bool[11];

rows[r[0]][r[1]] = true;

}

foreach(var row in rows)

result += MaxNumberOfFamilies(row.Value);

return result + 2 \* (n - rows.Count);

}

private int MaxNumberOfFamilies(bool[] row)

{

int count = 0;

if(!row[2] && !row[3] && !row[4] && !row[5])

count++;

if(!row[6] && !row[7] && !row[8] && !row[9])

count++;

if(count == 0 && !row[4] && !row[5] && !row[6] && !row[7])

count++;

return count;

}

}

# 43. Multiply Strings

Given two non-negative integers num1 and num2 represented as strings, return the product of num1 and num2, also represented as a string.

**Note:** You must not use any built-in BigInteger library or convert the inputs to integer directly.

**Example 1:**

**Input:** num1 = "2", num2 = "3"

**Output:** "6"

**Example 2:**

**Input:** num1 = "123", num2 = "456"

**Output:** "56088"

**Constraints:**

* 1 <= num1.length, num2.length <= 200
* num1 and num2 consist of digits only.
* Both num1 and num2 do not contain any leading zero, except the number 0 itself.

# Answer:

public string Multiply(string num1, string num2)

{

int n1 = num1.Length;

int n2 = num2.Length;

int[] products = new int[n1 + n2];

for (int i = n1 - 1; i >= 0; i--)

{

for (int j = n2 - 1; j >= 0; j--)

{

int p1 = i + j;

int p2 = p1 + 1;

int sum = (num1[i] - '0') \* (num2[j] - '0') + products[p2];

products[p1] += sum / 10;

products[p2] = sum % 10;

}

}

StringBuilder sb = new StringBuilder();

foreach(int num in products)

{

if (!(sb.Length == 0 && num == 0))

{

sb.Append(num);

}

}

return sb.Length == 0 ? "0" : sb.ToString();

}

# 1822. Sign of the Product of an Array

There is a function signFunc(x) that returns:

* 1 if x is positive.
* -1 if x is negative.
* 0 if x is equal to 0.

You are given an integer array nums. Let product be the product of all values in the array nums.

Return signFunc(product).

**Example 1:**

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

**Output:** 1

**Explanation:** The product of all values in the array is 144, and signFunc(144) = 1

**Example 2:**

**Input:** nums = [1,5,0,2,-3]

**Output:** 0

**Explanation:** The product of all values in the array is 0, and signFunc(0) = 0

**Example 3:**

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

**Output:** -1

**Explanation:** The product of all values in the array is -1, and signFunc(-1) = -1

**Constraints:**

* 1 <= nums.length <= 1000
* -100 <= nums[i] <= 100

# Answer:

public int ArraySign(int[] nums) {

if(nums==null)

return 0;

int result=1;

foreach(var item in nums)

{

if(item<0)

result \*= -1;

else if(item==0)

result \*= 0;

else

result \*=1;

}

return result;

}

# 1304. Find N Unique Integers Sum up to Zero

Given an integer n, return **any** array containing n **unique** integers such that they add up to 0.

**Example 1:**

**Input:** n = 5

**Output:** [-7,-1,1,3,4]

**Explanation:** These arrays also are accepted [-5,-1,1,2,3] , [-3,-1,2,-2,4].

**Example 2:**

**Input:** n = 3

**Output:** [-1,0,1]

**Example 3:**

**Input:** n = 1

**Output:** [0]

**Constraints:**

* 1 <= n <= 1000

# Answer:

public int[] SumZero(int n) {

List<int> result = new List<int>();

for(int i=1; i<=n/2;i++)

{

result.Add(i);

result.Add(-i);

}

if(n%2 == 1){

result.Add(0);

}

return result.ToArray();

}

# 285. Inorder Successor in BST

Given the root of a binary search tree and a node p in it, return *the in-order successor of that node in the BST*. If the given node has no in-order successor in the tree, return null.

The successor of a node p is the node with the smallest key greater than p.val.

**Example 1:**

Diagram

Description automatically generated

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

**Output:** 2

**Explanation:** 1's in-order successor node is 2. Note that both p and the return value is of TreeNode type.

**Example 2:**

A close-up of a stethoscope

Description automatically generated with medium confidence

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

**Output:** null

**Explanation:** There is no in-order successor of the current node, so the answer is null.

**Constraints:**

* The number of nodes in the tree is in the range [1, 104].
* -105 <= Node.val <= 105
* All Nodes will have unique values.

# Answer:

public TreeNode InorderSuccessor(TreeNode root, TreeNode p)

{

TreeNode ans = null;

while(root!=null)

{

if(p.val < root.val)

{

ans = root;

root = root.left;

}

else

root = root.right;

}

return ans;

}

# 1647. Minimum Deletions to Make Character Frequencies Unique

A string s is called **good** if there are no two different characters in s that have the same **frequency**.

Given a string s, return*the****minimum****number of characters you need to delete to make*s***good****.*

The **frequency** of a character in a string is the number of times it appears in the string. For example, in the string "aab", the **frequency** of 'a' is 2, while the **frequency** of 'b' is 1.

**Example 1:**

**Input:** s = "aab"

**Output:** 0

**Explanation:** s is already good.

**Example 2:**

**Input:** s = "aaabbbcc"

**Output:** 2

**Explanation:** You can delete two 'b's resulting in the good string "aaabcc".

Another way it to delete one 'b' and one 'c' resulting in the good string "aaabbc".

**Example 3:**

**Input:** s = "ceabaacb"

**Output:** 2

**Explanation:** You can delete both 'c's resulting in the good string "eabaab".

Note that we only care about characters that are still in the string at the end (i.e. frequency of 0 is ignored).

**Constraints:**

* 1 <= s.length <= 105
* s contains only lowercase English letters.

# Answer:

public int MinDeletions(string s) {

if (s == null || s == string.Empty)

return 0;

int res = 0,

curFreq = Int32.MaxValue;

Dictionary<char, int> dict = new Dictionary<char, int>();

foreach (var c in s)

{

if (!dict.ContainsKey(c))

dict.Add(c, 0);

dict[c] += 1;

}

foreach (var item in dict.OrderByDescending(x => x.Value).Select(x => x.Value).ToList())

if (curFreq <= item)

{

res += curFreq == 0 ? item : item - curFreq + 1;

curFreq = curFreq == 0 ? 0 : curFreq - 1;

}

else

curFreq = item;

return res;

}

# 1578. Minimum Time to Make Rope Colorful

Alice has n balloons arranged on a rope. You are given a **0-indexed** string colors where colors[i] is the color of the ith balloon.

Alice wants the rope to be **colorful**. She does not want **two consecutive balloons** to be of the same color, so she asks Bob for help. Bob can remove some balloons from the rope to make it **colorful**. You are given a **0-indexed** integer array neededTime where neededTime[i] is the time (in seconds) that Bob needs to remove the ith balloon from the rope.

Return *the****minimum time****Bob needs to make the rope****colorful***.

**Example 1:**

A group of balloons

Description automatically generated with low confidence

**Input:** colors = "abaac", neededTime = [1,2,3,4,5]

**Output:** 3

**Explanation:** In the above image, 'a' is blue, 'b' is red, and 'c' is green.

Bob can remove the blue balloon at index 2. This takes 3 seconds.

There are no longer two consecutive balloons of the same color. Total time = 3.

**Example 2:**

A picture containing diagram

Description automatically generated

**Input:** colors = "abc", neededTime = [1,2,3]

**Output:** 0

**Explanation:** The rope is already colorful. Bob does not need to remove any balloons from the rope.

**Example 3:**

A group of balloons

Description automatically generated with low confidence

**Input:** colors = "aabaa", neededTime = [1,2,3,4,1]

**Output:** 2

**Explanation:** Bob will remove the ballons at indices 0 and 4. Each ballon takes 1 second to remove.

There are no longer two consecutive balloons of the same color. Total time = 1 + 1 = 2.

**Constraints:**

* n == colors.length == neededTime.length
* 1 <= n <= 105
* 1 <= neededTime[i] <= 104
* colors contains only lowercase English letters.

# Answer:

public int MinCost(string s, int[] cost) {

if(s == null || s.Length == 0 || cost == null || cost.Length == 0)

return 0;

int res = 0, maxCost = cost[0];

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

{

if(s[i] == s[i-1])

{

// find repeating letters, remove the letter that cost less

res += Math.Min(maxCost, cost[i]);

// update the maxCost, the current letter could be duplicates with larger cost than its following letters

// e.g., s = "aaa", cost = [2,3,1]. We need to udpate the maxCost when checking the 2nd 'a'.

maxCost = Math.Max(maxCost, cost[i]);

}

else

{

// current letter is different from the previous one, but it could be duplicated with its following letters.

maxCost = cost[i];

}

}

return res;

}

# 1653. Minimum Deletions to Make String Balanced

You are given a string s consisting only of characters 'a' and 'b'​​​​.

You can delete any number of characters in s to make s **balanced**. s is **balanced** if there is no pair of indices (i,j) such that i < j and s[i] = 'b' and s[j]= 'a'.

Return *the****minimum****number of deletions needed to make*s***balanced***.

**Example 1:**

**Input:** s = "aababbab"

**Output:** 2

**Explanation:** You can either:

Delete the characters at 0-indexed positions 2 and 6 ("aababbab" -> "aaabbb"), or

Delete the characters at 0-indexed positions 3 and 6 ("aababbab" -> "aabbbb").

**Example 2:**

**Input:** s = "bbaaaaabb"

**Output:** 2

**Explanation:** The only solution is to delete the first two characters.

**Constraints:**

* 1 <= s.length <= 105
* s[i] is 'a' or 'b'​​.

# Answer:

public int MinimumDeletions(string s) {

int count = 0;

Stack<char> st = new Stack<char>();

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

if(st.Count != 0 && st.Peek() == 'b' && s[i] == 'a'){

count++;

st.Pop();

}else{

st.Push(s[i]);

}

}

return count;

}

# 1541. Minimum Insertions to Balance a Parentheses String

Given a parentheses string s containing only the characters '(' and ')'. A parentheses string is **balanced** if:

* Any left parenthesis '(' must have a corresponding two consecutive right parenthesis '))'.
* Left parenthesis '(' must go before the corresponding two consecutive right parenthesis '))'.

In other words, we treat '(' as an opening parenthesis and '))' as a closing parenthesis.

* For example, "())", "())(())))" and "(())())))" are balanced, ")()", "()))" and "(()))" are not balanced.

You can insert the characters '(' and ')' at any position of the string to balance it if needed.

Return *the minimum number of insertions* needed to make s balanced.

**Example 1:**

**Input:** s = "(()))"

**Output:** 1

**Explanation:** The second '(' has two matching '))', but the first '(' has only ')' matching. We need to add one more ')' at the end of the string to be "(())))" which is balanced.

**Example 2:**

**Input:** s = "())"

**Output:** 0

**Explanation:** The string is already balanced.

**Example 3:**

**Input:** s = "))())("

**Output:** 3

**Explanation:** Add '(' to match the first '))', Add '))' to match the last '('.

**Constraints:**

* 1 <= s.length <= 105
* s consists of '(' and ')' only.

# Answer:

public int MinInsertions(string s) {

Stack<char> stack = new Stack<char>();

int res = 0;

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

{

if(s[i] == '(')

stack.Push('(');

if(s[i] == ')')

{

if((i < s.Length-1 && s[i+1] != ')') || i == s.Length-1)

{

if(stack.Count > 0)

{

stack.Pop();

res++;

}

else

res+=2;

}

else if(i < s.Length-1 && s[i+1] == ')')

{

if(stack.Count > 0)

stack.Pop();

else

res++;

i++;

}

}

}

if(stack.Count > 0)

res += stack.Count\*2;

return res;

}

# 1615. Maximal Network Rank

There is an infrastructure of n cities with some number of roads connecting these cities. Each roads[i] = [ai, bi] indicates that there is a bidirectional road between cities ai and bi.

The **network rank**of **two different cities** is defined as the total number of **directly** connected roads to **either** city. If a road is directly connected to both cities, it is only counted **once**.

The **maximal network rank**of the infrastructure is the **maximum network rank** of all pairs of different cities.

Given the integer n and the array roads, return *the****maximal network rank****of the entire infrastructure*.

**Example 1:**

**A picture containing clock, watch

Description automatically generated**

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

**Output:** 4

**Explanation:** The network rank of cities 0 and 1 is 4 as there are 4 roads that are connected to either 0 or 1. The road between 0 and 1 is only counted once.

**Example 2:**

**A picture containing clock, watch

Description automatically generated**

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

**Output:** 5

**Explanation:** There are 5 roads that are connected to cities 1 or 2.

**Example 3:**

**Input:** n = 8, roads = [[0,1],[1,2],[2,3],[2,4],[5,6],[5,7]]

**Output:** 5

**Explanation:** The network rank of 2 and 5 is 5. Notice that all the cities do not have to be connected.

**Constraints:**

* 2 <= n <= 100
* 0 <= roads.length <= n \* (n - 1) / 2
* roads[i].length == 2
* 0 <= ai, bi <= n-1
* ai != bi
* Each pair of cities has **at most one** road connecting them.

# Answer:

public int MaximalNetworkRank(int n, int[][] roads) {

bool[,] connected = new bool[n,n];

int[] cnts = new int[n];

foreach(int[] r in roads)

{

cnts[r[0]]++;

cnts[r[1]]++;

connected[r[0],r[1]] = true;

connected[r[1],r[0]] = true;

}

int res = 0;

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

{

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

{

int tmp = cnts[i] + cnts[j] - (connected[i,j] ? 1: 0);

res = Math.Max(res,tmp);

}

}

return res;

}

# 1448. Count Good Nodes in Binary Tree

Given a binary tree root, a node *X* in the tree is named **good** if in the path from root to *X* there are no nodes with a value *greater than* X.

Return the number of **good** nodes in the binary tree.

**Example 1:**

**A close-up of a stethoscope

Description automatically generated with medium confidence**

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

**Output:** 4

**Explanation:** Nodes in blue are **good**.

Root Node (3) is always a good node.

Node 4 -> (3,4) is the maximum value in the path starting from the root.

Node 5 -> (3,4,5) is the maximum value in the path

Node 3 -> (3,1,3) is the maximum value in the path.

**Example 2:**

**A close-up of a stethoscope

Description automatically generated with medium confidence**

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

**Output:** 3

**Explanation:** Node 2 -> (3, 3, 2) is not good, because "3" is higher than it.

**Example 3:**

**Input:** root = [1]

**Output:** 1

**Explanation:** Root is considered as **good**.

**Constraints:**

* The number of nodes in the binary tree is in the range [1, 10^5].
* Each node's value is between [-10^4, 10^4].

# Answer:

private int res = 0;

public int GoodNodes(TreeNode root) {

DFS((root, Int32.MinValue));

return res;

}

private void DFS((TreeNode n, int m) p)

{

if (p.n == null)

return;

int max = p.m;

if (p.n.val >= max)

{

res++;

max = p.n.val;

}

DFS((p.n.left, max));

DFS((p.n.right, max));

}

# 1267. Count Servers that Communicate

You are given a map of a server center, represented as a m \* n integer matrix grid, where 1 means that on that cell there is a server and 0 means that it is no server. Two servers are said to communicate if they are on the same row or on the same column.  
  
Return the number of servers that communicate with any other server.

**Example 1:**

A picture containing diagram

Description automatically generated

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

**Output:** 0

**Explanation:** No servers can communicate with others.

**Example 2:**

**A picture containing text, clipart

Description automatically generated**

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

**Output:** 3

**Explanation:** All three servers can communicate with at least one other server.

**Example 3:**

A picture containing diagram

Description automatically generated

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

**Output:** 4

**Explanation:** The two servers in the first row can communicate with each other. The two servers in the third column can communicate with each other. The server at right bottom corner can't communicate with any other server.

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m <= 250
* 1 <= n <= 250
* grid[i][j] == 0 or 1

# Answer:

public int CountServers(int[][] grid)

{

int rows = grid.Length;

if (rows == 0)

{

return 0;

}

int columns = grid[0].Length;

int[] rowCounts = new int[rows];

int[] columnCount = new int[columns];

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

{

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

{

if (grid[i][j] > 0)

{

rowCounts[i]++;

columnCount[j]++;

}

}

}

int res = 0;

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

{

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

{

if (grid[i][j] > 0 && (rowCounts[i] > 1 || columnCount[j] > 1))

{

res++;

}

}

}

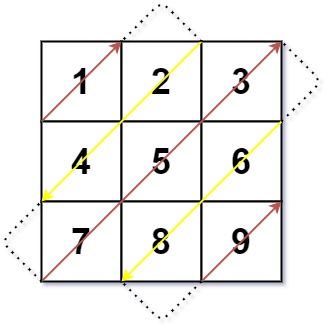
return res;

}

# 498. Diagonal Traverse

Given an m x n matrix mat, return *an array of all the elements of the array in a diagonal order*.

**Example 1:**



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

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

**Example 2:**

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

**Output:** [1,2,3,4]

**Constraints:**

* m == mat.length
* n == mat[i].length
* 1 <= m, n <= 104
* 1 <= m \* n <= 104
* -105 <= mat[i][j] <= 105

# Answer:

public int[] FindDiagonalOrder(int[][] matrix) {

var row = matrix.Length;

if (row == 0) return new int[0];

var col = matrix[0].Length;

var result = new List<int>();

var count = 0;

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

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

if (i == 0 || j == col - 1) {

var oneResult = new List<int>();

var curI = i;

var curJ = j;

while (curI < row && curJ >= 0) {

oneResult.Add(matrix[curI][curJ]);

curI++;

curJ--;

}

if (count % 2 == 0) {

oneResult.Reverse();

}

count++;

result.AddRange(oneResult);

}

}

}

return result.ToArray();

}