# Two Sum

Given an array of integers nums and an integer target, return *indices of the two numbers such that they add up to target*.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

**Example 1:**

**Input:** nums = [2,7,11,15], target = 9

**Output:** [0,1]

**Explanation:** Because nums[0] + nums[1] == 9, we return [0, 1].

**Example 2:**

**Input:** nums = [3,2,4], target = 6

**Output:** [1,2]

**Example 3:**

**Input:** nums = [3,3], target = 6

**Output:** [0,1]

**Constraints:**

* 2 <= nums.length <= 104
* -109 <= nums[i] <= 109
* -109 <= target <= 109
* **Only one valid answer exists.**

**Follow-up:**Can you come up with an algorithm that is less than O(n2) time complexity?

# Answer:

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

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

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

int num=nums[i];

if(dictNum.TryGetValue(target-num, out int index)){

return new int[] {index,i};

}

dictNum[num]=i;

}

return null;

}

# 56 Merge Intervals

Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

**Example 1:**

**Input:** intervals = [[1,3],[2,6],[8,10],[15,18]]

**Output:** [[1,6],[8,10],[15,18]]

**Explanation:** Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].

**Example 2:**

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

**Output:** [[1,5]]

**Explanation:** Intervals [1,4] and [4,5] are considered overlapping.

**Constraints:**

* 1 <= intervals.length <= 104
* intervals[i].length == 2
* 0 <= starti <= endi <= 104

# Answer:

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

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

return new int[][]{};

}

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

foreach(var item in intervals.OrderBy(x => x[0]).ToArray())

{

if(result.Count == 0 || result[result.Count-1][1] < item[0]){

result.Add(item);

}

else{

result[result.Count-1][1]= Math.Max(result[result.Count-1][1], item[1]);

}

}

return result.ToArray();

}

# 42 Trapping Rain Water

Given  n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

**Example 1:**



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

**Output:** 6

**Explanation:** The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

**Example 2:**

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

**Output:** 9

**Constraints:**

* n == height.length
* 1 <= n <= 2 \* 104
* 0 <= height[i] <= 105

# Answer:

public int Trap(int[] height) {

int n=height.Length;

int[] L = new int[n]; int water=0;

int[] R = new int[n];

if(height.Length > 0){

L[0]= height[0];

R[n-1]= height[n-1];

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

L[i]= Math.Max(L[i-1], height[i]);

}

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

R[i]= Math.Max(R[i+1], height[i]);

}

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

water+= Math.Min(L[i], R[i]) -height[i];

}

}

return water;

}

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

Let's define a function countUniqueChars(s) that returns the number of unique characters on s.

* For example, calling countUniqueChars(s) if s = "LEETCODE" then "L", "T", "C", "O", "D" are the unique characters since they appear only once in s, therefore countUniqueChars(s) = 5.

Given a string s, return the sum of countUniqueChars(t) where t is a substring of s.

Notice that some substrings can be repeated so in this case you have to count the repeated ones too.

**Example 1:**

**Input:** s = "ABC"

**Output:** 10

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

Every substring is composed with only unique letters.

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

**Example 2:**

**Input:** s = "ABA"

**Output:** 8

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

**Example 3:**

**Input:** s = "LEETCODE"

**Output:** 92

**Constraints:**

* 1 <= s.length <= 105
* s consists of uppercase English letters only.

# Answer:

public class Solution {

public int UniqueLetterString(string s) {

int result=0;

int N= s.Length;

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

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

if(!map.ContainsKey(s[i]))

{

map.Add(s[i], new List<int>(){i});

}

else

{

var list= map[s[i]];

list.Add(i);

}

}

foreach(var item in map.Values){

for(int i=0; i < item.Count; i++)

{

int left= (i==0) ? 1+ item[i]: item[i]-item[i-1];

int right = (i==item.Count-1) ? N-item[i]: item[i+1] -item[i];

result +=(left\*right);

}

}

return result;

}

}

# 973 K Closest Points to Origin

Given an array of points where points[i] = [xi, yi] represents a point on the **X-Y** plane and an integer k, return the k closest points to the origin (0, 0).

The distance between two points on the **X-Y** plane is the Euclidean distance (i.e., √(x1 - x2)2 + (y1 - y2)2).

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

**Example 1:**

Chart, line chart

Description automatically generated

**Input:** points = [[1,3],[-2,2]], k = 1

**Output:** [[-2,2]]

**Explanation:**

The distance between (1, 3) and the origin is sqrt(10).

The distance between (-2, 2) and the origin is sqrt(8).

Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.

We only want the closest k = 1 points from the origin, so the answer is just [[-2,2]].

**Example 2:**

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

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

**Explanation:** The answer [[-2,4],[3,3]] would also be accepted.

**Constraints:**

* 1 <= k <= points.length <= 104
* -104 < xi, yi < 104

# Answer:

public int[][] KClosest(int[][] points, int k) {

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

return new int[][] { };

return points.OrderBy(x => Math.Pow(x[0], 2) + Math.Pow(x[1], 2)).Take(k).ToArray();

}

# 1152 Analyze User Website Visit Pattern

You are given two string arrays username and website and an integer array timestamp. All the given arrays are of the same length and the tuple [username[i], website[i], timestamp[i]] indicates that the user username[i] visited the website website[i] at time timestamp[i].

A **pattern** is a list of three websites (not necessarily distinct).

* For example, ["home", "away", "love"], ["leetcode", "love", "leetcode"], and ["luffy", "luffy", "luffy"] are all patterns.

The **score** of a **pattern** is the number of users that visited all the websites in the pattern in the same order they appeared in the pattern.

* For example, if the pattern is ["home", "away", "love"], the score is the number of users x such that x visited "home" then visited "away" and visited "love" after that.
* Similarly, if the pattern is ["leetcode", "love", "leetcode"], the score is the number of users x such that x visited "leetcode" then visited "love" and visited "leetcode" **one more time** after that.
* Also, if the pattern is ["luffy", "luffy", "luffy"], the score is the number of users x such that x visited "luffy" three different times at different timestamps.

Return *the****pattern****with the largest****score***. If there is more than one pattern with the same largest score, return the lexicographically smallest such pattern.

**Example 1:**

**Input:** username = ["joe","joe","joe","james","james","james","james","mary","mary","mary"], timestamp = [1,2,3,4,5,6,7,8,9,10], website = ["home","about","career","home","cart","maps","home","home","about","career"]

**Output:** ["home","about","career"]

**Explanation:** The tuples in this example are:

["joe","home",1],["joe","about",2],["joe","career",3],["james","home",4],["james","cart",5],["james","maps",6],["james","home",7],["mary","home",8],["mary","about",9], and ["mary","career",10].

The pattern ("home", "about", "career") has score 2 (joe and mary).

The pattern ("home", "cart", "maps") has score 1 (james).

The pattern ("home", "cart", "home") has score 1 (james).

The pattern ("home", "maps", "home") has score 1 (james).

The pattern ("cart", "maps", "home") has score 1 (james).

The pattern ("home", "home", "home") has score 0 (no user visited home 3 times).

**Example 2:**

**Input:** username = ["ua","ua","ua","ub","ub","ub"], timestamp = [1,2,3,4,5,6], website = ["a","b","a","a","b","c"]

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

**Constraints:**

* 3 <= username.length <= 50
* 1 <= username[i].length <= 10
* timestamp.length == username.length
* 1 <= timestamp[i] <= 109
* website.length == username.length
* 1 <= website[i].length <= 10
* username[i] and website[i] consist of lowercase English letters.
* It is guaranteed that there is at least one user who visited at least three websites.
* All the tuples [username[i], timestamp[i], website[i]] are **unique**.

# Answer:

public IList<string> MostVisitedPattern(string[] username, int[] timestamp, string[] website)

{

int len = username.Length;

int[] indices = Enumerable.Range(0, len).ToArray();

Array.Sort(indices, (i1, i2) => timestamp[i1].CompareTo(timestamp[i2]));

List<(string u,string w)> table = new List<(string u,string w)>();

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

table.Add((username[indices[i]],website[indices[i]]));

var websListPerUser = table.GroupBy(x=>x.u)

.Select(grp => {

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

foreach(var tup in grp)

websites.Add(tup.w);

return websites;

});

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

foreach(var a in websListPerUser)

{

var allSequencesPerUser=GetAllCombination(a);

allSequences.AddRange(allSequencesPerUser);

}

var groupedSequences = allSequences.GroupBy(x=>x).Select(grp=>new {Key = grp.Key, Count = grp.Count()});

int max = groupedSequences.Max(x=>x.Count);

var rr = groupedSequences.Where(x=>x.Count==max).OrderBy(x=>x.Key,new MyComparer()).First();

return rr.Key.Split('-');

}

private List<string> GetAllCombination(List<string> li)

{

HashSet<string> ans = new HashSet<string>();

int cnt = li.Count;

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

{

for(int j=i+1;j<cnt;j++)

{

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

ans.Add(li[i]+"-"+li[j]+"-"+li[k]);

}

}

return ans.ToList();

}

class MyComparer : IComparer<string>

{

public int Compare(string x, string y)

{

string[] arr1 = x.Split('-');

string[] arr2 = y.Split('-');

for(int i=0; i<3; i++)

{

int cmp = arr1[i].CompareTo(arr2[i]);

if (cmp != 0)

return cmp;

}

return 0;

}

}

# 2104 Sum of Subarray Ranges

You are given an integer array nums. The **range** of a subarray of nums is the difference between the largest and smallest element in the subarray.

Return *the****sum of all****subarray ranges of*nums*.*

A subarray is a contiguous **non-empty** sequence of elements within an array.

**Example 1:**

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

**Output:** 4

**Explanation:** The 6 subarrays of nums are the following:

[1], range = largest - smallest = 1 - 1 = 0

[2], range = 2 - 2 = 0

[3], range = 3 - 3 = 0

[1,2], range = 2 - 1 = 1

[2,3], range = 3 - 2 = 1

[1,2,3], range = 3 - 1 = 2

So the sum of all ranges is 0 + 0 + 0 + 1 + 1 + 2 = 4.

**Example 2:**

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

**Output:** 4

**Explanation:** The 6 subarrays of nums are the following:

[1], range = largest - smallest = 1 - 1 = 0

[3], range = 3 - 3 = 0

[3], range = 3 - 3 = 0

[1,3], range = 3 - 1 = 2

[3,3], range = 3 - 3 = 0

[1,3,3], range = 3 - 1 = 2

So the sum of all ranges is 0 + 0 + 0 + 2 + 0 + 2 = 4.

**Example 3:**

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

**Output:** 59

**Explanation:** The sum of all subarray ranges of nums is 59.

**Constraints:**

* 1 <= nums.length <= 1000
* -109 <= nums[i] <= 109

**Follow-up:** Could you find a solution with O(n) time complexity?

# Answer:

public long SubArrayRanges(int[] nums) {

long result=0;

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

int min = nums[i];

int max = nums[i];

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

{

min= Math.Min(min,nums[j]) ;

max= Math.Max(max,nums[j]) ;

result += (max-min);

}

}

return result;

}

# 1268 Search Suggestions System

You are given an array of strings products and a string searchWord.

Design a system that suggests at most three product names from products after each character of searchWord is typed. Suggested products should have common prefix with searchWord. If there are more than three products with a common prefix return the three lexicographically minimums products.

Return *a list of lists of the suggested products after each character of*searchWord*is typed*.

**Example 1:**

**Input:** products = ["mobile","mouse","moneypot","monitor","mousepad"], searchWord = "mouse"

**Output:** [

["mobile","moneypot","monitor"],

["mobile","moneypot","monitor"],

["mouse","mousepad"],

["mouse","mousepad"],

["mouse","mousepad"]

]

**Explanation:** products sorted lexicographically = ["mobile","moneypot","monitor","mouse","mousepad"]

After typing m and mo all products match and we show user ["mobile","moneypot","monitor"]

After typing mou, mous and mouse the system suggests ["mouse","mousepad"]

**Example 2:**

**Input:** products = ["havana"], searchWord = "havana"

**Output:** [["havana"],["havana"],["havana"],["havana"],["havana"],["havana"]]

**Example 3:**

**Input:** products = ["bags","baggage","banner","box","cloths"], searchWord = "bags"

**Output:** [["baggage","bags","banner"],["baggage","bags","banner"],["baggage","bags"],["bags"]]

**Constraints:**

* 1 <= products.length <= 1000
* 1 <= products[i].length <= 3000
* 1 <= sum(products[i].length) <= 2 \* 104
* All the strings of products are **unique**.
* products[i] consists of lowercase English letters.
* 1 <= searchWord.length <= 1000
* searchWord consists of lowercase English letters.

# Answer:

public IList<IList<string>> SuggestedProducts(string[] products, string searchWord) {

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

Array.Sort(products);

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

{

string partWord= searchWord.Substring(0, i+1);

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

for(int j=0; j<products.Length; j++){

if(innerList.Count < 3 && products[j].Length > i)

{

string s = products[j].Substring(0,i+1);

if(s.Equals(partWord) && s.Length== partWord.Length){

innerList.Add( products[j]);}

}

}

list.Add(innerList);

}

return list;

}

# 212 Word Search II

Given an m x n board of characters and a list of strings words, return *all words on the board*.

Each word must be constructed from letters of sequentially adjacent cells, where **adjacent cells** are horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

**Example 1:**

Calendar

Description automatically generated

**Input:** board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","pea","eat","rain"]

**Output:** ["eat","oath"]

**Example 2:**

Calendar

Description automatically generated

**Input:** board = [["a","b"],["c","d"]], words = ["abcb"]

**Output:** []

**Constraints:**

* m == board.length
* n == board[i].length
* 1 <= m, n <= 12
* board[i][j] is a lowercase English letter.
* 1 <= words.length <= 3 \* 104
* 1 <= words[i].length <= 10
* words[i] consists of lowercase English letters.
* All the strings of words are unique.

# Answer:

public class Solution

{

public IList<string> FindWords(char[][] board, string[] words)

{

if(board==null || words==null)

return new List<string>();

Trie trie = new Trie();

foreach(var w in words)

trie.Add(w);

int[,] visited = new int[board.Length,board[0].Length];

StringBuilder builder = new StringBuilder();

HashSet<string> hash = new HashSet<string>();

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

{

for(int j=0; j<board[0].Length; j++)

{

Dfs(board,i,j,builder,trie,visited,hash);

Array.Clear(visited,0,visited.Length);

builder.Clear();

}

}

return new List<string>(hash);

}

private void Dfs(char[][] board, int i, int j, StringBuilder builder, Trie trie, int[,] visited, HashSet<string> hash)

{

if(i<0 || i>=board.Length || j<0 || j>=board[0].Length || visited[i,j]==1)

return;

if(!trie.Prefix(builder.ToString()))

return;

visited[i,j]=1;

builder.Append(board[i][j]);

string word = builder.ToString();

if(trie.Contains(word))

hash.Add(word);

Dfs(board,i-1,j,builder,trie,visited,hash);

Dfs(board,i+1,j,builder,trie,visited,hash);

Dfs(board,i,j-1,builder,trie,visited,hash);

Dfs(board,i,j+1,builder,trie,visited,hash);

builder.Remove(builder.Length-1,1);

visited[i,j]=0;

}

}

public class TrieNode

{

public Dictionary<char,TrieNode> Children = new Dictionary<char,TrieNode>();

public bool IsEnd;

}

public class Trie

{

TrieNode root = new TrieNode();

public void Add(string word)

{

TrieNode cur = root;

foreach(char c in word)

{

if(!cur.Children.ContainsKey(c))

cur.Children.Add(c,new TrieNode());

cur = cur.Children[c];

}

cur.IsEnd = true;

}

public bool Contains(string word)

{

TrieNode cur = root;

foreach(char c in word)

{

if(!cur.Children.ContainsKey(c))

return false;

cur = cur.Children[c];

}

return cur.IsEnd;

}

public bool Prefix(string word)

{

TrieNode cur = root;

foreach(char c in word)

{

if(!cur.Children.ContainsKey(c))

return false;

cur = cur.Children[c];

}

return true;

}

}

# 735 Asteroid Collision

We are given an array asteroids of integers representing asteroids in a row.

For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

**Example 1:**

**Input:** asteroids = [5,10,-5]

**Output:** [5,10]

**Explanation:** The 10 and -5 collide resulting in 10. The 5 and 10 never collide.

**Example 2:**

**Input:** asteroids = [8,-8]

**Output:** []

**Explanation:** The 8 and -8 collide exploding each other.

**Example 3:**

**Input:** asteroids = [10,2,-5]

**Output:** [10]

**Explanation:** The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting in 10.

**Constraints:**

* 2 <= asteroids.length <= 104
* -1000 <= asteroids[i] <= 1000
* asteroids[i] != 0

# Answer:

public static int[] AsteroidCollision(int[] asteroids)

{

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

return null;

var stack = new Stack<int>();

stack.Push(asteroids[0]);

for (int i = 1; i < asteroids.Length; i++)

{

while (true)

{

if (stack.Count > 0 && stack.Peek() > 0 && asteroids[i] < 0)

{

if (Math.Abs(stack.Peek()) == Math.Abs(asteroids[i]))

{

stack.Pop();

break;

}

else if (Math.Abs(stack.Peek()) < Math.Abs(asteroids[i]))

stack.Pop();

else break;

}

else

{

stack.Push(asteroids[i]);

break;

}

}

}

return stack.Reverse().ToArray();

}

# 239 Sliding Window Maximum

You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return *the max sliding window*.

**Example 1:**

**Input:** nums = [1,3,-1,-3,5,3,6,7], k = 3

**Output:** [3,3,5,5,6,7]

**Explanation:**

Window position Max

--------------- -----

[1 3 -1] -3 5 3 6 7 **3**

1 [3 -1 -3] 5 3 6 7 **3**

1 3 [-1 -3 5] 3 6 7  **5**

1 3 -1 [-3 5 3] 6 7 **5**

1 3 -1 -3 [5 3 6] 7 **6**

1 3 -1 -3 5 [3 6 7] **7**

**Example 2:**

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

**Output:** [1]

**Constraints:**

* 1 <= nums.length <= 105
* -104 <= nums[i] <= 104
* 1 <= k <= nums.length

# Answer:

public static int[] MaxSlidingWindow(int[] nums, int k)

{

if (k == 0) return nums;

int len = nums.Length;

int maxArrayLen = len - k + 1;

int[] ans = new int[maxArrayLen];

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

// Queue stores indices of array, and

// values are in decreasing order.

// So, the first node in queue is the max in window

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

{

// 1. remove element from head until first number within window

if (q.Count > 0 && q.First.Value + k <= i)

{

q.RemoveFirst();

}

// 2. before inserting i into queue, remove from the tail of the

// queue indices with smaller value they array[i]

while (q.Count > 0 && nums[q.Last.Value] <= nums[i])

{

q.RemoveLast();

}

q.AddLast(i);

// 3. set the max value in the window (always the top number in

// queue)

int index = i + 1 - k;

if (index >= 0)

{

ans[index] = nums[q.First.Value];

}

}

return ans;

}

}

# 937 Reorder Data in Log Files

You are given an array of logs. Each log is a space-delimited string of words, where the first word is the **identifier**.

There are two types of logs:

* **Letter-logs**: All words (except the identifier) consist of lowercase English letters.
* **Digit-logs**: All words (except the identifier) consist of digits.

Reorder these logs so that:

1. The **letter-logs** come before all **digit-logs**.
2. The **letter-logs** are sorted lexicographically by their contents. If their contents are the same, then sort them lexicographically by their identifiers.
3. The **digit-logs** maintain their relative ordering.

Return *the final order of the logs*.

**Example 1:**

**Input:** logs = ["dig1 8 1 5 1","let1 art can","dig2 3 6","let2 own kit dig","let3 art zero"]

**Output:** ["let1 art can","let3 art zero","let2 own kit dig","dig1 8 1 5 1","dig2 3 6"]

**Explanation:**

The letter-log contents are all different, so their ordering is "art can", "art zero", "own kit dig".

The digit-logs have a relative order of "dig1 8 1 5 1", "dig2 3 6".

**Example 2:**

**Input:** logs = ["a1 9 2 3 1","g1 act car","zo4 4 7","ab1 off key dog","a8 act zoo"]

**Output:** ["g1 act car","a8 act zoo","ab1 off key dog","a1 9 2 3 1","zo4 4 7"]

**Constraints:**

* 1 <= logs.length <= 100
* 3 <= logs[i].length <= 100
* All the tokens of logs[i] are separated by a **single** space.
* logs[i] is guaranteed to have an identifier and at least one word after the identifier.

# Answer:

public string[] ReorderLogFiles(string[] logs)

{

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

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

foreach (var log in logs)

{

if (char.IsDigit(log[log.IndexOf(' ') + 1]))

{

digitLogs.Add(log);

}

else

{

letterLogs.Add(log);

}

}

letterLogs.Sort((a, b) =>

{

string a\_substr = a.Substring(a.IndexOf(' ') + 1);

string b\_substr = b.Substring(b.IndexOf(' ') + 1);

var result = a\_substr.CompareTo(b\_substr);

if (result == 0)

{

result = a.Substring(0, a.IndexOf(' ')).CompareTo(b.Substring(0, b.IndexOf(' ')));

}

return result;

});

letterLogs.AddRange(digitLogs);

return letterLogs.ToArray();

}

# 12 Integer to Roman

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

For example, 2 is written as II in Roman numeral, just two one's added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given an integer, convert it to a roman numeral.

**Example 1:**

**Input:** num = 3

**Output:** "III"

**Explanation:** 3 is represented as 3 ones.

**Example 2:**

**Input:** num = 58

**Output:** "LVIII"

**Explanation:** L = 50, V = 5, III = 3.

**Example 3:**

**Input:** num = 1994

**Output:** "MCMXCIV"

**Explanation:** M = 1000, CM = 900, XC = 90 and IV = 4.

**Constraints:**

* 1 <= num <= 3999

# Answer:

public string IntToRoman(int num) {

string[] M = {"", "M", "MM", "MMM"};

string[] C = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};

string[] X = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};

string[] I = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};

return M[num/1000] + C[(num%1000)/100] + X[(num%100)/10] + I[num%10];

}

# 696 Count Binary Substrings

Given a binary string s, return the number of non-empty substrings that have the same number of 0's and 1's, and all the 0's and all the 1's in these substrings are grouped consecutively.

Substrings that occur multiple times are counted the number of times they occur.

**Example 1:**

**Input:** s = "00110011"

**Output:** 6

**Explanation:** There are 6 substrings that have equal number of consecutive 1's and 0's: "0011", "01", "1100", "10", "0011", and "01".

Notice that some of these substrings repeat and are counted the number of times they occur.

Also, "00110011" is not a valid substring because all the 0's (and 1's) are not grouped together.

**Example 2:**

**Input:** s = "10101"

**Output:** 4

**Explanation:** There are 4 substrings: "10", "01", "10", "01" that have equal number of consecutive 1's and 0's.

**Constraints:**

* 1 <= s.length <= 105
* s[i] is either '0' or '1'.

# Answer:

public int CountBinarySubstrings(string s) {

var groups = new List<int>();

int count = 1;

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

if (s[i] == s[i-1]) {

count++;

} else {

groups.Add(count);

count = 1;

}

}

groups.Add(count);

int cnt = 0;

for (int i = 1; i < groups.Count; i++) {

cnt += Math.Min(groups[i-1], groups[i]);

}

return cnt;

}

}

# 772 Basic Calculator III

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only non-negative integers, '+', '-', '\*', '/' operators, and open '(' and closing parentheses ')'. 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 = "1+1"

**Output:** 2

**Example 2:**

**Input:** s = "6-4/2"

**Output:** 4

**Example 3:**

**Input:** s = "2\*(5+5\*2)/3+(6/2+8)"

**Output:** 21

**Constraints:**

* 1 <= s <= 104
* s consists of digits, '+', '-', '\*', '/', '(', and ')'.
* s is a **valid** expression.

# Answer:

public int Calculate(string s) {

var stack = new Stack<string>();

for(int i = s.Length-1; i >= 0; i--)

{

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

{

stack.Push(Eval(stack));

}

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

{

var num = s[i].ToString();

if(char.IsNumber(s[i]))

{

while( i-1 >= 0 && char.IsNumber(s[i-1]))

{

num = s[i-1]+num;

i--;

}

}

stack.Push(num);

}

}

return int.Parse(Eval(stack));

}

public string Eval(Stack<string> stack)

{

var op = " ";

var result = 0;

var tempStack = new Stack<int>();

while(stack.Count > 0 && stack.Peek() != ")")

{

int x;

var e = stack.Pop();

bool success = int.TryParse(e, out x);

if(success)

{

switch(op)

{

case "/":

x = tempStack.Pop() / x;

break;

case "\*":

x = tempStack.Pop() \* x;

break;

case "-":

x = -1\*x;

break;

}

tempStack.Push(x);

}

else

{

op = e;

if(e.Length > 1 && x == 0)

tempStack.Push(int.MinValue);

}

}

if(stack.Count > 0)

stack.Pop();

return tempStack.Sum().ToString();

}

# 139 Word Break

Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.

**Note** that the same word in the dictionary may be reused multiple times in the segmentation.

**Example 1:**

**Input:** s = "leetcode", wordDict = ["leet","code"]

**Output:** true

**Explanation:** Return true because "leetcode" can be segmented as "leet code".

**Example 2:**

**Input:** s = "applepenapple", wordDict = ["apple","pen"]

**Output:** true

**Explanation:** Return true because "applepenapple" can be segmented as "apple pen apple".

Note that you are allowed to reuse a dictionary word.

**Example 3:**

**Input:** s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]

**Output:** false

**Constraints:**

* 1 <= s.length <= 300
* 1 <= wordDict.length <= 1000
* 1 <= wordDict[i].length <= 20
* s and wordDict[i] consist of only lowercase English letters.
* All the strings of wordDict are **unique**.

# Answer:

public bool WordBreak(string s, IList<string> wordDict) {

var n = s.Length;

var dp = new bool[n + 1];

dp[0] = true;

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

var sIndex = i - 1;

foreach (var word in wordDict) {

var wordLen = word.Length;

var lastChar = word.Last();

if (lastChar == s[sIndex] && sIndex + 1 >= wordLen) {

if (s.Substring(sIndex - wordLen + 1, wordLen) == word) {

dp[i] |= dp[i - wordLen];

}

}

}

}

return dp[n];

}

# 863 All Nodes Distance K in Binary Tree

Given the root of a binary tree, the value of a target node target, and an integer k, return *an array of the values of all nodes that have a distance*k*from the target node.*

You can return the answer in **any order**.

**Example 1:**

A picture containing clock

Description automatically generated

**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, k = 2

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

Explanation: The nodes that are a distance 2 from the target node (with value 5) have values 7, 4, and 1.

**Example 2:**

**Input:** root = [1], target = 1, k = 3

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [1, 500].
* 0 <= Node.val <= 500
* All the values Node.val are **unique**.
* target is the value of one of the nodes in the tree.
* 0 <= k <= 1000

# Answer:

public IList<int> DistanceK(TreeNode root, TreeNode target, int K) {

if (root == null)

return new List<int>();

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

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

Dictionary<TreeNode, List<TreeNode>> dict = new Dictionary<TreeNode, List<TreeNode>>();

HashSet<TreeNode> set = new HashSet<TreeNode>();

int n = 0;

q.Enqueue(root);

while (q.Count > 0)

{

TreeNode cur = q.Dequeue();

if (!dict.ContainsKey(cur))

dict.Add(cur, new List<TreeNode>());

if (cur.left != null)

{

dict[cur].Add(cur.left);

if (!dict.ContainsKey(cur.left))

dict.Add(cur.left, new List<TreeNode>());

dict[cur.left].Add(cur);

q.Enqueue(cur.left);

}

if (cur.right != null)

{

dict[cur].Add(cur.right);

if (!dict.ContainsKey(cur.right))

dict.Add(cur.right, new List<TreeNode>());

dict[cur.right].Add(cur);

q.Enqueue(cur.right);

}

}

q.Enqueue(target);

set.Add(target);

while (q.Count > 0)

{

int count = q.Count;

while (count > 0)

{

TreeNode cur = q.Dequeue();

if (n == K)

res.Add(cur.val);

else

foreach (var node in dict[cur])

if (!set.Contains(node))

{

q.Enqueue(node);

set.Add(node);

}

count--;

}

if (n == K)

return res;

n++;

}

return res;

}

# 767 Reorganize String

Given a string s, rearrange the characters of s so that any two adjacent characters are not the same.

Return *any possible rearrangement of* s *or return* "" *if not possible*.

**Example 1:**

**Input:** s = "aab"

**Output:** "aba"

**Example 2:**

**Input:** s = "aaab"

**Output:** ""

**Constraints:**

* 1 <= s.length <= 500
* s consists of lowercase English letters.

# Answer:

public string ReorganizeString(string s) {

if(string.IsNullOrEmpty(s))

return "";

int[] freq = new int[26];

int maxFreq = 0;

char maxChar = ' ';

foreach(var c in s)

{

freq[c - 'a']++;

if(freq[c - 'a'] > maxFreq)

{

maxFreq = freq[c - 'a'];

maxChar = c;

}

}

if(maxFreq > (s.Length + 1) / 2)

return "";

char[] arr = new char[s.Length];

int i = 0;

while(freq[maxChar - 'a'] > 0)

{

arr[i] = maxChar;

i += 2;

freq[maxChar - 'a']--;

}

for(int j = 0; j < freq.Length; j++)

{

if(freq[j] > 0)

{

while(freq[j] > 0)

{

if(i >= arr.Length)

i = 1;

arr[i] = Convert.ToChar(j + 'a');

i += 2;

freq[j]--;

}

}

}

return new string(arr);

}

# 472 Concatenated Words

Given an array of strings words (**without duplicates**), return *all the****concatenated words****in the given list of* words.

A **concatenated word** is defined as a string that is comprised entirely of at least two shorter words in the given array.

**Example 1:**

**Input:** words = ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratcatdogcat"]

**Output:** ["catsdogcats","dogcatsdog","ratcatdogcat"]

**Explanation:** "catsdogcats" can be concatenated by "cats", "dog" and "cats";

"dogcatsdog" can be concatenated by "dog", "cats" and "dog";

"ratcatdogcat" can be concatenated by "rat", "cat", "dog" and "cat".

**Example 2:**

**Input:** words = ["cat","dog","catdog"]

**Output:** ["catdog"]

**Constraints:**

* 1 <= words.length <= 104
* 0 <= words[i].length <= 30
* words[i] consists of only lowercase English letters.
* 0 <= sum(words[i].length) <= 105

# Answer:

public IList<string> FindAllConcatenatedWordsInADict(string[] words)

{

var res = new List<string>();

var memo = new HashSet<string>(words);

foreach (var word in words)

{

if(Helper(word, memo))

{

res.Add(word);

}

}

return res;

}

public bool Helper(string word, HashSet<string> memo)

{

for (var i = 1 ; i < word.Length; i++) {

var prefix = word.Substring(0 , i);

var suffix = word.Substring(i , word.Length - i);

if((memo.Contains(prefix)) &&

(memo.Contains(suffix) || Helper(suffix, memo))

) {

return true;

}

}

return false;

}

# 926 Flip String to Monotone Increasing

A binary string is monotone increasing if it consists of some number of 0's (possibly none), followed by some number of 1's (also possibly none).

You are given a binary string s. You can flip s[i] changing it from 0 to 1 or from 1 to 0.

Return *the minimum number of flips to make*s*monotone increasing*.

**Example 1:**

**Input:** s = "00110"

**Output:** 1

**Explanation:** We flip the last digit to get 00111.

**Example 2:**

**Input:** s = "010110"

**Output:** 2

**Explanation:** We flip to get 011111, or alternatively 000111.

**Example 3:**

**Input:** s = "00011000"

**Output:** 2

**Explanation:** We flip to get 00000000.

**Constraints:**

* 1 <= s.length <= 105
* s[i] is either '0' or '1'.

# Answer:

public int MinFlipsMonoIncr(string S)

{

var totalOnes = S.Count(x => x == '1');

if (totalOnes == S.Length || totalOnes == 0)

return 0;

// calculate number of ones up to centain index

var preOnes = new int[S.Length + 1];

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

preOnes[i + 1] = S[i] == '1' ? preOnes[i] + 1 : preOnes[i];

var ans = int.MaxValue;

for (var i = 0; i < S.Length + 1; i++)

{

var onesLeft = preOnes[i];

var onesRight = totalOnes - onesLeft;

// for every left and right substring:

// answer is the sum of ones on the left (to be converted to zeros)

// plus the sum of zeros on the right (to be converted to ones)

// in order to make is monotone increasing.

ans = Math.Min(ans,

onesLeft + (S.Length - i - onesRight));

}

return ans;

}

# 103 Binary Tree Zigzag Level Order Traversal

Given the root of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

**Example 1:**

A picture containing text, clock, clipart

Description automatically generated

**Input:** root = [3,9,20,null,null,15,7]

**Output:** [[3],[20,9],[15,7]]

**Example 2:**

**Input:** root = [1]

**Output:** [[1]]

**Example 3:**

**Input:** root = []

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 2000].
* -100 <= Node.val <= 100

# Answer:

public IList<IList<int>> ZigzagLevelOrder(TreeNode root)

{

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

if (root == null)

{

return result;

}

Stack<TreeNode> s1 = new Stack<TreeNode>();

Stack<TreeNode> s2 = new Stack<TreeNode>();

s1.Push(root);

while (s1.Count != 0 || s2.Count != 0)

{

int n1 = s1.Count;

int n2 = s2.Count;

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

for (int i = 0; i < n1; i++)

{

root = s1.Pop();

list.Add(root.val);

if (root.left != null)

{

s2.Push(root.left);

}

if (root.right != null)

{

s2.Push(root.right);

}

}

for (int i = 0; i < n2; i++)

{

root = s2.Pop();

list.Add(root.val);

if (root.right != null)

{

s1.Push(root.right);

}

if (root.left != null)

{

s1.Push(root.left);

}

}

result.Add(list);

}

return result;

}

# 994 Rotting Oranges

You are given an m x n grid where each cell can have one of three values:

* 0 representing an empty cell,
* 1 representing a fresh orange, or
* 2 representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return *the minimum number of minutes that must elapse until no cell has a fresh orange*. If *this is impossible, return* -1.

**Example 1:**

A picture containing scatter chart

Description automatically generated

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

**Output:** 4

**Example 2:**

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

**Output:** -1

**Explanation:** The orange in the bottom left corner (row 2, column 0) is never rotten, because rotting only happens 4-directionally.

**Example 3:**

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

**Output:** 0

**Explanation:** Since there are already no fresh oranges at minute 0, the answer is just 0.

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 10
* grid[i][j] is 0, 1, or 2.

# Answer:

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

if(grid.Length==0) return 0;

int minutes=0,fresh=0;

Queue<int[]> rotten=new Queue<int[]>();

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

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

if(grid[i][j]==2){

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

}

else if(grid[i][j]==1) fresh++;

}

}

if(fresh==0)return 0;

List<int[]> dirs=new List<int[]>{

new int[]{1,0},

new int[]{-1,0},

new int[]{0,1},

new int[]{0,-1}

};

while(rotten.Any()){

minutes++;

int count=rotten.Count;

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

int[] point=rotten.Dequeue();

foreach(int[] dir in dirs){

int x=point[0]+dir[0];

int y=point[1]+dir[1];

if(x<0||y<0||x==grid.Length||y==grid[0].Length||grid[x][y]==2||grid[x][y]==0) continue;

grid[x][y]=2;

rotten.Enqueue(new int[]{x,y});

fresh--;

}

}

}

if(fresh>0)return -1;

else return minutes-1;

}

# 1710 Maximum Units on a Truck

You are assigned to put some amount of boxes onto **one truck**. You are given a 2D array boxTypes, where boxTypes[i] = [numberOfBoxesi, numberOfUnitsPerBoxi]:

* numberOfBoxesi is the number of boxes of type i.
* numberOfUnitsPerBoxiis the number of units in each box of the type i.

You are also given an integer truckSize, which is the **maximum** number of **boxes** that can be put on the truck. You can choose any boxes to put on the truck as long as the number of boxes does not exceed truckSize.

Return *the****maximum****total number of****units****that can be put on the truck.*

**Example 1:**

**Input:** boxTypes = [[1,3],[2,2],[3,1]], truckSize = 4

**Output:** 8

**Explanation:** There are:

- 1 box of the first type that contains 3 units.

- 2 boxes of the second type that contain 2 units each.

- 3 boxes of the third type that contain 1 unit each.

You can take all the boxes of the first and second types, and one box of the third type.

The total number of units will be = (1 \* 3) + (2 \* 2) + (1 \* 1) = 8.

**Example 2:**

**Input:** boxTypes = [[5,10],[2,5],[4,7],[3,9]], truckSize = 10

**Output:** 91

**Constraints:**

* 1 <= boxTypes.length <= 1000
* 1 <= numberOfBoxesi, numberOfUnitsPerBoxi <= 1000
* 1 <= truckSize <= 106

# Answer:

public int MaximumUnits(int[][] boxTypes, int truckSize) {

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

return 0;

int res = 0,

size = 0;

boxTypes = boxTypes.OrderByDescending(x => x[1]).ToArray();

foreach (var item in boxTypes)

{

int cur = item[0];

while (cur > 0)

{

if (size == truckSize)

return res;

res += item[1];

size++;

cur--;

}

}

return res;

}

# 1567 Maximum Length of Subarray With Positive Product

Given an array of integers nums, find the maximum length of a subarray where the product of all its elements is positive.

A subarray of an array is a consecutive sequence of zero or more values taken out of that array.

Return *the maximum length of a subarray with positive product*.

**Example 1:**

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

**Output:** 4

**Explanation:** The array nums already has a positive product of 24.

**Example 2:**

**Input:** nums = [0,1,-2,-3,-4]

**Output:** 3

**Explanation:** The longest subarray with positive product is [1,-2,-3] which has a product of 6.

Notice that we cannot include 0 in the subarray since that'll make the product 0 which is not positive.

**Example 3:**

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

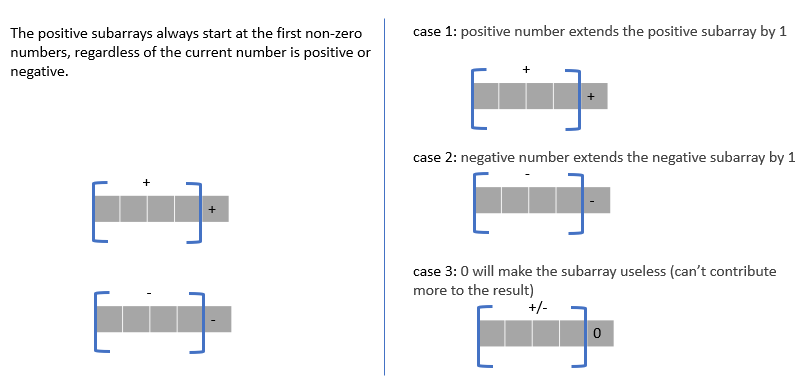
**Output:** 2

**Explanation:** The longest subarray with positive product is [-1,-2] or [-2,-3].

**Constraints:**

* 1 <= nums.length <= 105
* -109 <= nums[i] <= 109

# Answer:



**Complexity**

* Time: O(n)
* Space: O(1)

**Implementation**

public class Solution {

public int GetMaxLen(int[] nums) {

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

return 0;

int positive = 0, negative = 0, res = 0;

foreach(var num in nums)

{

if(num == 0)

{

positive = 0;

negative = 0;

}

else if(num > 0)

{

positive++;

negative = negative == 0? 0 : negative + 1;

}

else

{

int prevPositive = positive;

positive = negative == 0? 0 : negative + 1;

negative = prevPositive + 1;

}

res = Math.Max(res, positive);

}

return res;

}

# 907 Sum of Subarray Minimums

Given an array of integers arr, find the sum of min(b), where b ranges over every (contiguous) subarray of arr. Since the answer may be large, return the answer **modulo** 109 + 7.

**Example 1:**

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

**Output:** 17

**Explanation:**

Subarrays are [3], [1], [2], [4], [3,1], [1,2], [2,4], [3,1,2], [1,2,4], [3,1,2,4].

Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1, 1.

Sum is 17.

**Example 2:**

**Input:** arr = [11,81,94,43,3]

**Output:** 444

**Constraints:**

* 1 <= arr.length <= 3 \* 104
* 1 <= arr[i] <= 3 \* 104

# Answer:

public int SumSubarrayMins(int[] A) {

int len = A.Length, mod = 1000000007, res = 0;

int[] bf = new int[len], af = new int[len];

for (var i = 0; i < len; i++)

{

var s = i - 1;

while (s >= 0 && A[s] >= A[i]) s -= bf[s];

bf[i] = i - s;

}

for (var i = len - 1; i >= 0; i--)

{

var s = i + 1;

while (s < len && A[s] > A[i]) s += af[s];

af[i] = s - i;

res = (res + bf[i] \* af[i] \* A[i]) % mod;

}

return res;

}

# 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);

}

}

# 1044 Longest Duplicate Substring

Given a string s, consider all *duplicated substrings*: (contiguous) substrings of s that occur 2 or more times. The occurrences may overlap.

Return **any** duplicated substring that has the longest possible length. If s does not have a duplicated substring, the answer is "".

**Example 1:**

**Input:** s = "banana"

**Output:** "ana"

**Example 2:**

**Input:** s = "abcd"

**Output:** ""

**Constraints:**

* 2 <= s.length <= 3 \* 104
* s consists of lowercase English letters.

# Answer:

public string LongestDupSubstring(string s)

{

int left = 1;

int right = s.Length - 1;

string lastFoundString = "";

while (left <= right) // Binary Search

{

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

var a = Contains(s, mid);

if (a != "")

{

lastFoundString = a;

left = mid + 1;

}

else

right = mid - 1;

}

return lastFoundString;

}

string Contains(string s, int length)

{

int q = 100007; // prime number

int h = 1;

var d = 26; // number of letters in alphabet

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

h = (h \* d) % q;

int hash = 0;

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

hash = (d \* hash + s[i] - 'a') % q;

if (hash < 0) hash += q;

var dictHashToPos = new Dictionary<int, List<int>>(); // Hash -> List of start positions of string with this hash

dictHashToPos[hash] = new List<int>() { 0 };

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

{

hash = (d \* (hash - (s[i] - 'a') \* h) + s[i + length] - 'a') % q;

if (hash < 0) hash += q;

if (dictHashToPos.TryGetValue(hash, out var listOfStartPositions))

{

foreach (var startPosition in listOfStartPositions)

{

// works much faster than (string.Compare(s, startPosition, s, i + 1, length) == 0)

bool equal = true;

for (int k = 0; k < length; k++)

{

if (s[startPosition + k] != s[i + 1 + k])

{

equal = false;

break;

}

}

if (equal)

return s.Substring(i + 1, length);

}

listOfStartPositions.Add(i + 1);

}

else

dictHashToPos[hash] = new List<int>() { i + 1 };

}

return "";

}

# 370 Range Addition

You are given an integer length and an array updates where updates[i] = [startIdxi, endIdxi, inci].

You have an array arr of length length with all zeros, and you have some operation to apply on arr. In the ith operation, you should increment all the elements arr[startIdxi], arr[startIdxi + 1], ..., arr[endIdxi] by inci.

Return arr *after applying all the* updates.

**Example 1:**

A screenshot of a game

Description automatically generated with low confidence

**Input:** length = 5, updates = [[1,3,2],[2,4,3],[0,2,-2]]

**Output:** [-2,0,3,5,3]

**Example 2:**

**Input:** length = 10, updates = [[2,4,6],[5,6,8],[1,9,-4]]

**Output:** [0,-4,2,2,2,4,4,-4,-4,-4]

**Constraints:**

* 1 <= length <= 105
* 0 <= updates.length <= 104
* 0 <= startIdxi <= endIdxi < length
* -1000 <= inci <= 1000

# Answer:

public int[] GetModifiedArray(int length, int[][] updates)

{

int[] diffs = new int[length];

int[] res = new int[length];

foreach (var update in updates)

{

diffs[update[0]] += update[2];

if (update[1] < length - 1)

{

diffs[update[1] + 1] -= update[2];

}

}

int diff = 0;

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

{

diff += diffs[i];

res[i] += diff;

}

return res;

}

# 909 Snakes and Ladders

You are given an n x n integer matrix board where the cells are labeled from 1 to n2 in a [**Boustrophedon style**](https://en.wikipedia.org/wiki/Boustrophedon) starting from the bottom left of the board (i.e. board[n - 1][0]) and alternating direction each row.

You start on square 1 of the board. In each move, starting from square curr, do the following:

* Choose a destination square next with a label in the range [curr + 1, min(curr + 6, n2)].
  + This choice simulates the result of a standard **6-sided die roll**: i.e., there are always at most 6 destinations, regardless of the size of the board.
* If next has a snake or ladder, you **must** move to the destination of that snake or ladder. Otherwise, you move to next.
* The game ends when you reach the square n2.

A board square on row r and column c has a snake or ladder if board[r][c] != -1. The destination of that snake or ladder is board[r][c]. Squares 1 and n2 do not have a snake or ladder.

Note that you only take a snake or ladder at most once per move. If the destination to a snake or ladder is the start of another snake or ladder, you do **not** follow the subsequent snake or ladder.

* For example, suppose the board is [[-1,4],[-1,3]], and on the first move, your destination square is 2. You follow the ladder to square 3, but do **not** follow the subsequent ladder to 4.

Return *the least number of moves required to reach the square*n2*. If it is not possible to reach the square, return*-1.

**Example 1:**

Chart

Description automatically generated

**Input:** board = [[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,35,-1,-1,13,-1],[-1,-1,-1,-1,-1,-1],[-1,15,-1,-1,-1,-1]]

**Output:** 4

**Explanation:**

In the beginning, you start at square 1 (at row 5, column 0).

You decide to move to square 2 and must take the ladder to square 15.

You then decide to move to square 17 and must take the snake to square 13.

You then decide to move to square 14 and must take the ladder to square 35.

You then decide to move to square 36, ending the game.

This is the lowest possible number of moves to reach the last square, so return 4.

**Example 2:**

**Input:** board = [[-1,-1],[-1,3]]

**Output:** 1

**Constraints:**

* n == board.length == board[i].length
* 2 <= n <= 20
* grid[i][j] is either -1 or in the range [1, n2].
* The squares labeled 1 and n2 do not have any ladders or snakes.

# Answer:

public int SnakesAndLadders(int[][] board)

{

int n = board.Length;

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

bfs.Enqueue(1);

if (board[0][0] != -1)

{

return -1;

}

ISet<int> visited = new HashSet<int>();

visited.Add(1);

int res = 0;

while (bfs.Count > 0)

{

int count = bfs.Count;

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

{

var current = bfs.Dequeue();

if (current == n \* n)

{

return res;

}

for (int j = 1; j <= 6; j++)

{

int next = current + j;

if (next > n \* n)

{

break;

}

int shiftedIdx = next - 1;

int row = shiftedIdx / n;

int col = shiftedIdx % n;

if (row % 2 == 1)

{

col = (n - 1) - col;

}

row = (n - 1) - row;

if (board[row][col] != -1)

{

next = board[row][col];

}

if (visited.Add(next))

{

bfs.Enqueue(next);

}

}

}

res++;

}

return -1;

}

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

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

**Example 1:**

**Input:** nums = [8,2,4,7], limit = 4

**Output:** 2

**Explanation:** All subarrays are:

[8] with maximum absolute diff |8-8| = 0 <= 4.

[8,2] with maximum absolute diff |8-2| = 6 > 4.

[8,2,4] with maximum absolute diff |8-2| = 6 > 4.

[8,2,4,7] with maximum absolute diff |8-2| = 6 > 4.

[2] with maximum absolute diff |2-2| = 0 <= 4.

[2,4] with maximum absolute diff |2-4| = 2 <= 4.

[2,4,7] with maximum absolute diff |2-7| = 5 > 4.

[4] with maximum absolute diff |4-4| = 0 <= 4.

[4,7] with maximum absolute diff |4-7| = 3 <= 4.

[7] with maximum absolute diff |7-7| = 0 <= 4.

Therefore, the size of the longest subarray is 2.

**Example 2:**

**Input:** nums = [10,1,2,4,7,2], limit = 5

**Output:** 4

**Explanation:** The subarray [2,4,7,2] is the longest since the maximum absolute diff is |2-7| = 5 <= 5.

**Example 3:**

**Input:** nums = [4,2,2,2,4,4,2,2], limit = 0

**Output:** 3

**Constraints:**

* 1 <= nums.length <= 105
* 1 <= nums[i] <= 109
* 0 <= limit <= 109

# Answer:

public int LongestSubarray(int[] nums, int limit) {

if (nums == null || nums.Length == 0) return 0;

int l=0, r=0;

int ans = 0;

var dict = new Dictionary<int, int>();

while (r < nums.Length) {

// moving r

if (!dict.ContainsKey(nums[r])) {

dict[nums[r]] = 1;

} else {

dict[nums[r]]++;

}

r++;

// shrink

int min = dict.Keys.Min();

int max = dict.Keys.Max();

while (max-min > limit) {

dict[nums[l]]--;

if (dict[nums[l]] == 0) {

dict.Remove(nums[l]);

}

min = dict.Keys.Min();

max = dict.Keys.Max();

l++;

}

// check

ans = Math.Max(ans, r-l);

}

return ans;

}

# 2130 Maximum Twin Sum of a Linked List

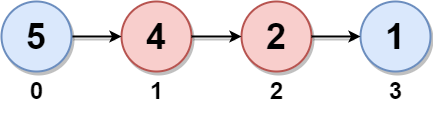
In a linked list of size n, where n is **even**, the ith node (**0-indexed**) of the linked list is known as the **twin** of the (n-1-i)th node, if 0 <= i <= (n / 2) - 1.

* For example, if n = 4, then node 0 is the twin of node 3, and node 1 is the twin of node 2. These are the only nodes with twins for n = 4.

The **twin sum**is defined as the sum of a node and its twin.

Given the head of a linked list with even length, return *the****maximum twin sum****of the linked list*.

**Example 1:**



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

**Output:** 6

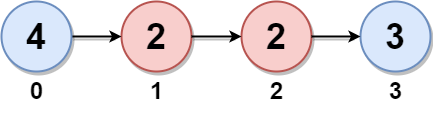
**Explanation:**

Nodes 0 and 1 are the twins of nodes 3 and 2, respectively. All have twin sum = 6.

There are no other nodes with twins in the linked list.

Thus, the maximum twin sum of the linked list is 6.

**Example 2:**



**Input:** head = [4,2,2,3]

**Output:** 7

**Explanation:**

The nodes with twins present in this linked list are:

- Node 0 is the twin of node 3 having a twin sum of 4 + 3 = 7.

- Node 1 is the twin of node 2 having a twin sum of 2 + 2 = 4.

Thus, the maximum twin sum of the linked list is max(7, 4) = 7.

**Example 3:**

A picture containing icon

Description automatically generated

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

**Output:** 100001

**Explanation:**

There is only one node with a twin in the linked list having twin sum of 1 + 100000 = 100001.

**Constraints:**

* The number of nodes in the list is an **even** integer in the range [2, 105].
* 1 <= Node.val <= 105

# Answer:

public int PairSum(ListNode head) {

Stack<int> st = new();

var node = head;

while(node!=null){

st.Push(node.val);

node = node.next;

}

int length = st.Count;

int max = 0;

node = head;

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

var twin = st.Pop();

max = Math.Max(max, node.val+twin);

node = node.next;

}

return max;

}

# 1492 The kth Factor of n

You are given two positive integers n and k. A factor of an integer n is defined as an integer i where n % i == 0.

Consider a list of all factors of n sorted in **ascending order**, return *the*kth*factor* in this list or return -1 if n has less than k factors.

**Example 1:**

**Input:** n = 12, k = 3

**Output:** 3

**Explanation:** Factors list is [1, 2, 3, 4, 6, 12], the 3rd factor is 3.

**Example 2:**

**Input:** n = 7, k = 2

**Output:** 7

**Explanation:** Factors list is [1, 7], the 2nd factor is 7.

**Example 3:**

**Input:** n = 4, k = 4

**Output:** -1

**Explanation:** Factors list is [1, 2, 4], there is only 3 factors. We should return -1.

**Constraints:**

* 1 <= k <= n <= 1000

**Follow up:**

Could you solve this problem in less than O(n) complexity?

# Answer:

public int KthFactor(int n, int k) {

// seach for factors that < sqrt(n)

for(int factor = 1; factor < Math.Sqrt(n); factor++)

{

if(n % factor == 0)

{

k--;

if(k == 0)

return factor;

}

}

// search for factors that >= sqrt(n)

for(int factor = (int)Math.Sqrt(n); factor >= 1; factor--)

{

if(n % factor == 0)

{

k--;

if(k == 0)

return n / factor;

}

}

return -1;

}

# 1730 Shortest Path to Get Food

You are starving and you want to eat food as quickly as possible. You want to find the shortest path to arrive at any food cell.

You are given an m x n character matrix, grid, of these different types of cells:

* '\*' is your location. There is **exactly one**'\*' cell.
* '#' is a food cell. There may be **multiple** food cells.
* 'O' is free space, and you can travel through these cells.
* 'X' is an obstacle, and you cannot travel through these cells.

You can travel to any adjacent cell north, east, south, or west of your current location if there is not an obstacle.

Return *the****length****of the shortest path for you to reach****any****food cell*. If there is no path for you to reach food, return -1.

**Example 1:**

A clock on a brick wall

Description automatically generated with medium confidence

**Input:** grid = [["X","X","X","X","X","X"],["X","\*","O","O","O","X"],["X","O","O","#","O","X"],["X","X","X","X","X","X"]]

**Output:** 3

**Explanation:** It takes 3 steps to reach the food.

**Example 2:**

A clock on a brick wall

Description automatically generated with medium confidence

**Input:** grid = [["X","X","X","X","X"],["X","\*","X","O","X"],["X","O","X","#","X"],["X","X","X","X","X"]]

**Output:** -1

**Explanation:** It is not possible to reach the food.

**Example 3:**

Clocks on a brick building

Description automatically generated with medium confidence

**Input:** grid = [["X","X","X","X","X","X","X","X"],["X","\*","O","X","O","#","O","X"],["X","O","O","X","O","O","X","X"],["X","O","O","O","O","#","O","X"],["X","X","X","X","X","X","X","X"]]

**Output:** 6

**Explanation:** There can be multiple food cells. It only takes 6 steps to reach the bottom food.

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 200
* grid[row][col] is '\*', 'X', 'O', or '#'.
* The grid contains **exactly one** '\*'.

# Answer:

public int GetFood(char[][] grid)

{

int rows = grid.Length;

int columns = grid[0].Length;

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

for(int row = 0; row < rows; row++)

{

for(int column = 0; column < columns; column++)

{

if(grid[row][column] == '\*')

{

queue.Enqueue((row, column));

grid[row][column] = '-'; //Mark as visited

break;

}

}

}

int steps = 0;

(int, int)[] directions = new (int, int)[] { (-1, 0), (1, 0), (0, -1), (0, 1)};

while(queue.Count > 0)

{

int size = queue.Count;

steps++;

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

{

(int row, int column) = queue.Dequeue();

foreach(var direction in directions)

{

int nextRow = row + direction.Item1;

int nextColumn = column + direction.Item2;

if(nextRow < 0 || nextRow >= rows || nextColumn < 0 || nextColumn >= columns ||

grid[nextRow][nextColumn] == 'X' || grid[nextRow][nextColumn] == '-')

continue;

if(grid[nextRow][nextColumn]=='#')

return steps;

grid[nextRow][nextColumn] = '-'; //Mark as visited

queue.Enqueue((nextRow, nextColumn));

}

}

}

return -1;

}

# 1597 Build Binary Expression Tree From Infix Expression

A [**binary expression tree**](https://en.wikipedia.org/wiki/Binary_expression_tree) is a kind of binary tree used to represent arithmetic expressions. Each node of a binary expression tree has either zero or two children. Leaf nodes (nodes with 0 children) correspond to operands (numbers), and internal nodes (nodes with 2 children) correspond to the operators '+' (addition), '-' (subtraction), '\*' (multiplication), and '/' (division).

For each internal node with operator o, the [**infix expression**](https://en.wikipedia.org/wiki/Infix_notation) it represents is (A o B), where A is the expression the left subtree represents and B is the expression the right subtree represents.

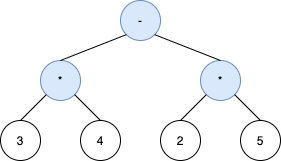
You are given a string s, an **infix expression** containing operands, the operators described above, and parentheses '(' and ')'.

Return *any valid****binary expression tree****, whose*[***in-order traversal***](https://en.wikipedia.org/wiki/Tree_traversal#In-order_(LNR))*reproduces*s *after omitting the parenthesis from it.*

**Please note that order of operations applies in**s**.** That is, expressions in parentheses are evaluated first, and multiplication and division happen before addition and subtraction.

Operands must also appear in the **same order** in both s and the in-order traversal of the tree.

**Example 1:**



**Input:** s = "3\*4-2\*5"

**Output:** [-,\*,\*,3,4,2,5]

**Explanation:** The tree above is the only valid tree whose inorder traversal produces s.

**Example 2:**

Background pattern

Description automatically generated

**Input:** s = "2-3/(5\*2)+1"

**Output:** [+,-,1,2,/,null,null,null,null,3,\*,null,null,5,2]

**Explanation:** The inorder traversal of the tree above is 2-3/5\*2+1 which is the same as s without the parenthesis. The tree also produces the correct result and its operands are in the same order as they appear in s.

The tree below is also a valid binary expression tree with the same inorder traversal as s, but it not a valid answer because it does not evaluate to the same value.

Background pattern

Description automatically generated

The third tree below is also not valid. Although it produces the same result and is equivalent to the above trees, its inorder traversal does not produce s and its operands are not in the same order as s.

Background pattern

Description automatically generated

**Example 3:**

**Input:** s = "1+2+3+4+5"

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

**Explanation:** The tree [+,+,5,+,+,null,null,1,2,3,4] is also one of many other valid trees.

**Constraints:**

* 1 <= s.length <= 100
* s consists of digits and the characters '+', '-', '\*', and '/'.
* Operands in s are **exactly** 1 digit.
* It is guaranteed that s is a valid expression.

# Answer:

public class Solution {

public Node ExpTree(string s) {

if(string.IsNullOrEmpty(s))

return null;

var postFix = ConvertToPostfix(s);

// Console.WriteLine($"{s} -> {postFix}");

var st = new Stack<Node>();

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

{

var cur = postFix[i];

if(char.IsDigit(cur))

{

st.Push(new Node(cur));

}

else

{

var right = st.Pop();

var left = st.Pop();

var root = new Node(cur);

root.left = left;

root.right = right;

st.Push(root);

}

}

return st.Pop();

}

private string ConvertToPostfix(string s)

{

var sb = new StringBuilder();

var st = new Stack<char>();

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

{

var ch = s[i];

if(ch == '(')

{

st.Push(ch);

}

else if(char.IsLetterOrDigit(ch))

{

sb.Append(ch);

}

else if(ch == ')')

{

while(st.Count > 0 && st.Peek() != '(')

sb.Append(st.Pop());

st.Pop();

}

else

{

while(st.Count > 0 && GetRank(ch) <= GetRank(st.Peek()))

sb.Append(st.Pop());

st.Push(ch);

}

}

while(st.Count > 0)

sb.Append(st.Pop());

return sb.ToString();

}

private int GetRank(char ch)

{

switch (ch)

{

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

case '^':

return 3;

}

return -1;

}

}

# 1151 Minimum Swaps to Group All 1's Together

Given a binary array data, return the minimum number of swaps required to group all 1’s present in the array together in **any place** in the array.

**Example 1:**

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

**Output:** 1

**Explanation:** There are 3 ways to group all 1's together:

[1,1,1,0,0] using 1 swap.

[0,1,1,1,0] using 2 swaps.

[0,0,1,1,1] using 1 swap.

The minimum is 1.

**Example 2:**

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

**Output:** 0

**Explanation:** Since there is only one 1 in the array, no swaps are needed.

**Example 3:**

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

**Output:** 3

**Explanation:** One possible solution that uses 3 swaps is [0,0,0,0,0,1,1,1,1,1,1].

**Constraints:**

* 1 <= data.length <= 105
* data[i] is either 0 or 1.

# Answer:

public class Solution {

public int MinSwaps(int[] data) {

int numOfOne = 0;

foreach(int d in data)

{

if(d == 1)

numOfOne++;

}

int min = Int32.MaxValue, left = 0, right = 0, numOfZero = 0;

while(right < data.Length)

{

if(data[right] == 0)

numOfZero++;

right++;

if(right - left == numOfOne)

{

min = Math.Min(min, numOfZero);

if(data[left] == 0)

numOfZero--;

left++;

}

}

return min == Int32.MaxValue? 0 : min;

}

}

# 1628 Design an Expression Tree With Evaluate Function

Given the postfix tokens of an arithmetic expression, build and return *the binary expression tree that represents this expression.*

**Postfix** notation is a notation for writing arithmetic expressions in which the operands (numbers) appear before their operators. For example, the postfix tokens of the expression 4\*(5-(7+2)) are represented in the array postfix = ["4","5","7","2","+","-","\*"].

The class Node is an interface you should use to implement the binary expression tree. The returned tree will be tested using the evaluate function, which is supposed to evaluate the tree's value. You should not remove the Node class; however, you can modify it as you wish, and you can define other classes to implement it if needed.

A [**binary expression tree**](https://en.wikipedia.org/wiki/Binary_expression_tree) is a kind of binary tree used to represent arithmetic expressions. Each node of a binary expression tree has either zero or two children. Leaf nodes (nodes with 0 children) correspond to operands (numbers), and internal nodes (nodes with two children) correspond to the operators '+' (addition), '-' (subtraction), '\*' (multiplication), and '/' (division).

It's guaranteed that no subtree will yield a value that exceeds 109 in absolute value, and all the operations are valid (i.e., no division by zero).

**Follow up:** Could you design the expression tree such that it is more modular? For example, is your design able to support additional operators without making changes to your existing evaluate implementation?

**Example 1:**

Shape

Description automatically generated

**Input:** s = ["3","4","+","2","\*","7","/"]

**Output:** 2

**Explanation:** this expression evaluates to the above binary tree with expression ((3+4)\*2)/7) = 14/7 = 2.

**Example 2:**

Diagram

Description automatically generated

**Input:** s = ["4","5","2","7","+","-","\*"]

**Output:** -16

**Explanation:** this expression evaluates to the above binary tree with expression 4\*(5-(2+7)) = 4\*(-4) = -16.

**Constraints:**

* 1 <= s.length < 100
* s.length is odd.
* s consists of numbers and the characters '+', '-', '\*', and '/'.
* If s[i] is a number, its integer representation is no more than 105.
* It is guaranteed that s is a valid expression.
* The absolute value of the result and intermediate values will not exceed 109.
* It is guaranteed that no expression will include division by zero.

# Answer:

public abstract class Node {

public abstract int evaluate();

};

public class TreeNode: Node {

public string Value;

public TreeNode Left;

public TreeNode Right;

public TreeNode(string data= null,TreeNode left=null,TreeNode right=null){

this.Value = data;

this.Left = left;

Right = right;

}

public override int evaluate(){

return dfs(this);

}

private int dfs(TreeNode node){

if(node.Left == null && node.Right == null){

return Convert.ToInt32(node.Value);

}

int left = dfs(node.Left);

int right = dfs(node.Right);

int res = 0;

if(node.Value == "+")

res = left + right;

else if(node.Value == "-")

res = left - right;

else if(node.Value == "/")

res = left / right;

else if(node.Value == "\*")

res = left \* right;

return res;

}

}

/\*\*

\* This is the TreeBuilder class.

\* You can treat it as the driver code that takes the postinfix input

\* and returns the expression tree represnting it as a Node.

\*/

public class TreeBuilder {

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

List<string> operators = new List<string>{"+","-","/","\*"};

public Node buildTree(string[] postfix) {

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

if(!operators.Contains(postfix[i])){

treeStack.Push(new TreeNode(postfix[i]));

}else{

var right = treeStack.Pop();

var left = treeStack.Pop();

var node = new TreeNode(postfix[i],left,right);

treeStack.Push(node);

}

}

return treeStack.Pop();

}

}

# 582 Kill Process

You have n processes forming a rooted tree structure. You are given two integer arrays pid and ppid, where pid[i] is the ID of the ith process and ppid[i] is the ID of the ith process's parent process.

Each process has only **one parent process** but may have multiple children processes. Only one process has ppid[i] = 0, which means this process has **no parent process** (the root of the tree).

When a process is **killed**, all of its children processes will also be killed.

Given an integer kill representing the ID of a process you want to kill, return *a list of the IDs of the processes that will be killed. You may return the answer in****any order****.*

**Example 1:**

A picture containing text, clipart

Description automatically generated

**Input:** pid = [1,3,10,5], ppid = [3,0,5,3], kill = 5

**Output:** [5,10]

**Explanation:** The processes colored in red are the processes that should be killed.

**Example 2:**

**Input:** pid = [1], ppid = [0], kill = 1

**Output:** [1]

**Constraints:**

* n == pid.length
* n == ppid.length
* 1 <= n <= 5 \* 104
* 1 <= pid[i] <= 5 \* 104
* 0 <= ppid[i] <= 5 \* 104
* Only one process has no parent.
* All the values of pid are **unique**.
* kill is **guaranteed** to be in pid.

# Answer:

public IList<int> KillProcess(IList<int> pid, IList<int> ppid, int kill)

{

var list = new List<int>();

if (!pid.Contains(kill)) return list;

var dict = new Dictionary<int, List<int>>();

for(var i = 0; i < ppid.Count; ++i)

{

if (!dict.ContainsKey(ppid[i])) dict[ppid[i]] = new List<int>();

dict[ppid[i]].Add(pid[i]);

}

var s = new Stack<int>();

s.Push(kill);

while (s.Count > 0)

{

var kp = s.Pop();

list.Add(kp);

if (dict.ContainsKey(kp))

{

foreach(var p in dict[kp])

{

s.Push(p);

}

}

}

return list;

}

# 2214 Minimum Health to Beat Game

You are playing a game that has n levels numbered from 0 to n - 1. You are given a **0-indexed** integer array damage where damage[i] is the amount of health you will lose to complete the ith level.

You are also given an integer armor. You may use your armor ability **at most once** during the game on **any** level which will protect you from **at most** armor damage.

You must complete the levels in order and your health must be **greater than** 0 at all times to beat the game.

Return *the****minimum****health you need to start with to beat the game.*

**Example 1:**

**Input:** damage = [2,7,4,3], armor = 4

**Output:** 13

**Explanation:** One optimal way to beat the game starting at 13 health is:

On round 1, take 2 damage. You have 13 - 2 = 11 health.

On round 2, take 7 damage. You have 11 - 7 = 4 health.

On round 3, use your armor to protect you from 4 damage. You have 4 - 0 = 4 health.

On round 4, take 3 damage. You have 4 - 3 = 1 health.

Note that 13 is the minimum health you need to start with to beat the game.

**Example 2:**

**Input:** damage = [2,5,3,4], armor = 7

**Output:** 10

**Explanation:** One optimal way to beat the game starting at 10 health is:

On round 1, take 2 damage. You have 10 - 2 = 8 health.

On round 2, use your armor to protect you from 5 damage. You have 8 - 0 = 8 health.

On round 3, take 3 damage. You have 8 - 3 = 5 health.

On round 4, take 4 damage. You have 5 - 4 = 1 health.

Note that 10 is the minimum health you need to start with to beat the game.

**Example 3:**

**Input:** damage = [3,3,3], armor = 0

**Output:** 10

**Explanation:** One optimal way to beat the game starting at 10 health is:

On round 1, take 3 damage. You have 10 - 3 = 7 health.

On round 2, take 3 damage. You have 7 - 3 = 4 health.

On round 3, take 3 damage. You have 4 - 3 = 1 health.

Note that you did not use your armor ability.

**Constraints:**

* n == damage.length
* 1 <= n <= 105
* 0 <= damage[i] <= 105
* 0 <= armor <= 105

# Answer:

public long MinimumHealth(int[] damage, int armor) {

long totalDamage = 0, maxDamage = 0;

foreach(var d in damage)

{

totalDamage += d;

maxDamage = Math.Max(maxDamage, d);

}

return 1 + totalDamage - Math.Min(armor, maxDamage);

}

# 2102 Sequentially Ordinal Rank Tracker

A scenic location is represented by its name and attractiveness score, where name is a **unique** string among all locations and score is an integer. Locations can be ranked from the best to the worst. The **higher** the score, the better the location. If the scores of two locations are equal, then the location with the **lexicographically smaller** name is better.

You are building a system that tracks the ranking of locations with the system initially starting with no locations. It supports:

* **Adding** scenic locations, **one at a time**.
* **Querying** the ith **best** location of **all locations already added**, where i is the number of times the system has been queried (including the current query).
  + For example, when the system is queried for the 4th time, it returns the 4th best location of all locations already added.

Note that the test data are generated so that **at any time**, the number of queries **does not exceed** the number of locations added to the system.

Implement the SORTracker class:

* SORTracker() Initializes the tracker system.
* void add(string name, int score) Adds a scenic location with name and score to the system.
* string get() Queries and returns the ith best location, where i is the number of times this method has been invoked (including this invocation).

**Example 1:**

**Input**

["SORTracker", "add", "add", "get", "add", "get", "add", "get", "add", "get", "add", "get", "get"]

[[], ["bradford", 2], ["branford", 3], [], ["alps", 2], [], ["orland", 2], [], ["orlando", 3], [], ["alpine", 2], [], []]

**Output**

[null, null, null, "branford", null, "alps", null, "bradford", null, "bradford", null, "bradford", "orland"]

**Explanation**

SORTracker tracker = new SORTracker(); // Initialize the tracker system.

tracker.add("bradford", 2); // Add location with name="bradford" and score=2 to the system.

tracker.add("branford", 3); // Add location with name="branford" and score=3 to the system.

tracker.get(); // The sorted locations, from best to worst, are: branford, bradford.

// Note that branford precedes bradford due to its **higher score** (3 > 2).

// This is the 1st time get() is called, so return the best location: "branford".

tracker.add("alps", 2); // Add location with name="alps" and score=2 to the system.

tracker.get(); // Sorted locations: branford, alps, bradford.

// Note that alps precedes bradford even though they have the same score (2).

// This is because "alps" is **lexicographically smaller** than "bradford".

// Return the 2nd best location "alps", as it is the 2nd time get() is called.

tracker.add("orland", 2); // Add location with name="orland" and score=2 to the system.

tracker.get(); // Sorted locations: branford, alps, bradford, orland.

// Return "bradford", as it is the 3rd time get() is called.

tracker.add("orlando", 3); // Add location with name="orlando" and score=3 to the system.

tracker.get(); // Sorted locations: branford, orlando, alps, bradford, orland.

// Return "bradford".

tracker.add("alpine", 2); // Add location with name="alpine" and score=2 to the system.

tracker.get(); // Sorted locations: branford, orlando, alpine, alps, bradford, orland.

// Return "bradford".

tracker.get(); // Sorted locations: branford, orlando, alpine, alps, bradford, orland.

// Return "orland".

**Constraints:**

* name consists of lowercase English letters, and is unique among all locations.
* 1 <= name.length <= 10
* 1 <= score <= 105
* At any time, the number of calls to get does not exceed the number of calls to add.
* At most 4 \* 104 calls **in total** will be made to add and get.

# Answer:

public class SORTracker {

SortedList<int, SortedList<string, int>> sl;

int qi = 1;

public SORTracker() {

sl = new SortedList<int, SortedList<string, int>>();

}

public void Add(string name, int score) {

if (!sl.ContainsKey(score)) {

sl.Add(score, new SortedList<string, int>());

sl[score].Add(name, 1);

} else {

sl[score].Add(name, 1);

}

}

public string Get() {

int ith = qi++;

string res = null;

foreach (KeyValuePair<int, SortedList<string, int>> e in sl.OrderByDescending(x => x.Key)) {

if (sl[e.Key].Count < ith) {

ith -= e.Value.Count;

} else {

res = e.Value.Keys[ith - 1];

break;

}

}

return res;

}

}