1. Two Sum

revised on 24/7/20

2. Add Two Numbers

/**The digits are stored in reverse order */.

https://www.geeksforgeeks.org/add-two-numbers-represented-by-linked-lists/ (https://www.geeksforgeeks.org/add-two-numbers-represented-by-linked-lists/)

4. Median of Two Sorted Arrays

//revised public class MedianOfTwoSortedArrayOfDifferentLength {

```
public double findMedianSortedArrays(int input1[], int input2[]) {
    //if input1 length is greater than switch them so that input1 is smaller than input2.
    if (input1.length > input2.length) {
        return findMedianSortedArrays(input2, input1);
    }
    int x = input1.length;
    int y = input2.length;

    int low = 0;
    int high = x;
    while (low <= high) {
        int partitionX = (low + high)/2;
        int partitionY = (x + y + 1)/2 - partitionX;

        //if partitionX is 0 it means nothing is there on left side. Use -INF for maxLeftX
        //if partitionX is length of input then there is nothing on right side. Use +INF for minRightX
        int maxLeftX = (partitionX == 0) ? Integer.MIN_VALUE : input1[partitionX - 1];
        int minRightX = (partitionX == x) ? Integer.MAX_VALUE : input1[partitionX];

        int maxLeftY = (partitionY == 0) ? Integer.MIN_VALUE : input2[partitionY - 1];
        int minRightY = (partitionY == y) ? Integer.MAX_VALUE : input2[partitionY];

        if (maxLeftX <= minRightY && maxLeftY <= minRightX) {
            //We have partitioned array at correct place
            // Now get max of left elements and min of right elements to get the median in case of even length combined array
size
            // or get max of left for odd length combined array size.
            if ((x + y) % 2 == 0) {
                return ((double)Math.max(maxLeftX, maxLeftY) + Math.min(minRightX, minRightY))/2;
            } else {
                return (double)Math.max(maxLeftX, maxLeftY);
            }
        } else if (maxLeftX > minRightY) { //we are too far on right side for partitionX. Go on left side.
            high = partitionX - 1;
        } else { //we are too far on left side for partitionX. Go on right side.
            low = partitionX + 1;
        }
    }

    //Only we we can come here is if input arrays were not sorted. Throw in that scenario.
    throw new IllegalArgumentException();
}

public static void main(String[] args) {
    int[] x = {1, 3, 8, 9, 15};
    int[] y = {7, 11, 19, 21, 18, 25};

    MedianOfTwoSortedArrayOfDifferentLength mm = new MedianOfTwoSortedArrayOfDifferentLength();
    mm.findMedianSortedArrays(x, y);
}
```

}

5. Longest Palindromic Substring

revised

10. Regular Expression Matching

https://leetcode.com/problems/regular-expression-matching/discuss/510884/C%2B%2B-DP-solution-with-explanation (https://leetcode.com/problems/regular-expression-matching/discuss/510884/C%2B%2B-DP-solution-with-explanation) // Dynamic programming // table[i][j] tells whether s[0:i-1] and p[0:j-1] matches // table[0][0] = true; // Case 1: p[j-1] != '', *table[i][j] = (i>0) && (p[j-1] == s[i-1] || p[j-1] == '.') && table[i-1][j-1]; (p cannot match empty, i.e. i > 0) // Case 2: p[j-1] ==* '', denote 'x' = p[j-2]. // Case 2.1: 'x' matches 0 times in the tail of s: table[i][j] = table[i][j-2] (omit "x*" in p); // Case 2.2: 'x' matches at least 1 times in the tail of s: table[i][j] =(i > 0)&&(s[i-1] == p[j-2] || p[j-2] == '.') && table[i-1][j] (omit s[i-1] in s);

15. 3Sum

311 / 313 test cases passed. We process each value from left to right. For value v, we need to find all pairs whose sum is equal -v. To find such pairs, we apply the Two Sum: One-pass Hash Table approach to the rest of the array. To ensure unique triplets, we use a hash set found as described above.

Because hashmap operations could be expensive, the solution below may be too slow. We'll add some optimizations in the next section.

vector<vector> threeSum(vector& nums) { vector<vector> res;

```
set<pair<int, int>> found; // or define hash<pair<int, int>> and use unordered_set.
for (int i = 0; i < nums.size(); ++i) {
    unordered_set<int> seen;
    for (int j = i + 1; j < nums.size(); ++j) {
        int complement = -nums[i] - nums[j];
        if (seen.count(complement)) {
            int v1 = min(nums[i], min(complement, nums[j]));
            int v2 = max(nums[i], max(complement, nums[j]));
            if (found.insert({v1, v2}).second)
                res.push_back({nums[i], complement, nums[j]});
        }
        seen.insert(nums[j]);
    }
}
return res;
```

}

19. Remove Nth Node From End of List

https://www.geeksforgeeks.org/remove-nth-node-from-end-of-the-linked-list/ (https://www.geeksforgeeks.org/remove-nth-node-from-end-of-the-linked-list/)

20. Valid Parentheses

https://leetcode.com/problems/valid-parentheses/discuss/728304/C%2B%2B-Easiest-solution-fully-Explained-takes-0ms-beats-100-49 (https://leetcode.com/problems/valid-parentheses/discuss/728304/C%2B%2B-Easiest-solution-fully-Explained-takes-0ms-beats-100-49)

21. Merge Two Sorted Lists

https://www.geeksforgeeks.org/merge-two-sorted-linked-lists/ (https://www.geeksforgeeks.org/merge-two-sorted-linked-lists/)

23. Merge k Sorted Lists

revised

24. Swap Nodes in Pairs

https://www.geeksforgeeks.org/pairwise-swap-elements-of-a-given-linked-list/ (https://www.geeksforgeeks.org/pairwise-swap-elements-of-a-given-linked-list/)

25. Reverse Nodes in k-Group

https://www.geeksforgeeks.org/reverse-a-list-in-groups-of-given-size/ (https://www.geeksforgeeks.org/reverse-a-list-in-groups-of-given-size/)

29. Divide Two Integers

Approach 2: Repeated Exponential Searches Intuition

Linear Search is too slow because at each step, we only subtract one copy of the divisor from the dividend. A better way would be to try and subtract multiple copies of the divisor each time.

One way of quickly increasing numbers, without using multiplication, is to double them repeatedly. So let's try doubling the divisor until it no longer fits into the dividend.

It'll be easiest to understand with an example, so let's say we have a dividend of 93706 and a divisor of 157. We'll now just see what happens when we repeatedly double 157 until it's bigger than 93706.

157 314 628 1256 2512 5024 10048 20096 40192 80384 160768 # Too big From this, we know that we can fit 80384 into 93706, and that 80384 must be a multiple of 157. But how many copies of 157 is this?

Well, each time we double a number we also double the amount of copies of the original number. So because we doubled 157 nine times, we must have had 2^9 copies of 157. Indeed, $2^9 \cdot 157 = 80384$. Yay!

But, we still have some left over—in fact we have 93706 - 80384 = 13322 left over! That's still a lot of copies of 157 we haven't counted! So what could we do about this? Well, if we work out how many times 157 fits into 13322, we could just add that to 512 to get our result.

How can we work out how many times 157 fits into 13322? Well, we just repeat the same process, adding to the result as we go, until there's nothing left for 157 to fit into.

If we do this, we'll find that $157 \cdot 2^6 = 10048$ is the highest power that fits into 13322, leaving us with 13322 - 10048 = 3274 and a quotient so far of $2^6 + 2^9 = 576$ (if you noticed that 10048 looks very familiar, well done. We'll be looking at this in approach 3).

We repeat this process until the dividend is less than 157.

Here is the algorithm in code (for this example we're pretending the numbers are positive, and we're ignoring the "overflow" case. In the actual code, we use negatives numbers to prevent the overflow).

int quotient = 0; /* Once the divisor is bigger than the current dividend,

- we can't fit any more copies of the divisor into it. */ while (dividend >= divisor) { /* Now that we're in the loop, we know it'll fit at least once as
  - divivend >= divisor */ int powerOfTwo = 1; int value = quotient; /* Check if double the current value is too big. If not, continue doubling.
  - If it is too big, stop doubling and continue with the next step */ while (value + value < dividend) { value += value; powerOfTwo += powerOfTwo; } // We have been able to subtract divisor another powerOfTwo times. quotient += powerOfTwo; // Remove value so far so that we can continue the process with remainder. dividend -= value; }

return quotient; This algorithm is known as exponential search and is commonly used for searching sorted spaces of unknown size for the first value that past a particular condition. It it a lot like binary search, having the same time complexity of O(\log , n)O(logn). I believe this is why this question is tagged as binary search (there is technically a way of using binary search, but it is a lot more complicated and gives no real efficiency gain, and so we won't be talking about it in this article.)

31. Next Permutation

https://leetcode.com/problems/next-permutation/discuss/13878/C%2B%2BJava-Clean-Code-with-Really-Simple-Explanation (https://leetcode.com/problems/next-permutation/discuss/13878/C%2B%2BJava-Clean-Code-with-Really-Simple-Explanation)

To understand this approach, we need to define 2 concept with a 5 digit example 12543:

NCR - No Capacity Range : the number in that range cannot be bigger, Like: "543" DWC - Digit With Capacity : A digit bring capacity to the range after it. Like: "[2]543" Let's get some hint from finding the Next Decimal:

Next Decimal [5](9 9 9) <- NCR (No Capacity Range) ^ DWC (Digit With Capacity) => [6](0 0 0) Imagine how you increase this number 5999 to 6000: Because the first 3 digits from right is a No Capacity Range, you have to flip the 4th digits (5, the first Digit With Capacity.), to what - The next smallest digits greater than 5, which is 6, then make the rest 3 digits the smallest combination, which is "000", that's how you get this "6000";

In an permutation though, the next smallest number for DWC(Digit With Capacity) would be whatever is available in that following NCR(No Capacity Range)

Next Permutation 1[2](5 4 3) <- NCR (No Capacity Range) ^ DWC (Digit With Capacity) => 1[3](2 4 5) So we have found the pattern is always to find the first DWC(Digit With Capacity), then minimize the NCR after it.

To summarize:

Find the first DWC, which is 2; 1[2](5 4 3) ^ Reverse the NCR after it to make the NCR in increasing order(because it is already sorted in descending order). 1[2](3 4 5) ^ Swap the DWC with the 1st digit in the reversed range that is slightly bigger than it. 1[3](2 4 5) ^--^ DONE!

C++

class Solution { public: void nextPermutation(vector& a) { if (a.size() <= 1) return; int dwc = -1; // Digit With Capacity for (int i = a.size() - 2; i >= 0; i--) { if (a[i] < a[i + 1]) { dwc = i; break; } } // if dwc is not found, means this is a max array, reverse whole array and return; reverse(a.begin() + dwc + 1, a.end()); if (dwc != -1) return; for (int i = dwc + 1; i < a.size(); i++) { if (a[i] > a[dwc]) { swap(a[i], a[dwc]); break; } } } }

33. Search in Rotated Sorted Array

*

The idea is to find the pivot point, divide the array in two sub-arrays and perform binary search.

The main idea for finding pivot is – for a sorted (in increasing order) and pivoted array, pivot element is the only element for which next element to it is smaller than it.

Using the above statement and binary search pivot can be found.

After the pivot is found out divide the array in two sub-arrays.

Now the individual sub – arrays are sorted so the element can be searched using Binary Search.

Implementation:

Input arr[] = {3, 4, 5, 1, 2}

Element to Search = 1

1) Find out pivot point and divide the array in two

sub-arrays. (pivot = 2) /Index of 5/

2) Now call binary search for one of the two sub-arrays.

(a) If element is greater than 0th element then

search in left array

(b) Else Search in right array

(1 will go in else as 1 < 0th element(3))

3) If element is found in selected sub-array then return index

Else return -1.*

36. Valid Sudoku

A Sudoku board (partially filled) could be valid but is not necessarily solvable. Only the filled cells need to be validated according to the mentioned rules.

39. Combination Sum

//geeks:https://www.geeksforgeeks.org/combinational-sum/ (https://www.geeksforgeeks.org/combinational-sum/) 2. 1. Sort the array(non-decreasing). 2. First remove all the duplicates from array. 3. Then use recursion and backtracking to solve the problem. (A) If at any time sub-problem sum == 0 then add that array to the result (vector of vectors). (B) Else if sum if negative then ignore that sub-problem. (C) Else insert the present array in that index to the current vector and call the function with sum = sum-ar[index] and index = index, then pop that element from current index (backtrack) and call the function with sum = sum and index = index+1

42. Trapping Rain Water

revised

49. Group Anagrams

editorial:

Two strings are anagrams if and only if their sorted strings are equal.

Algorithm

Maintain a map ans : {String -> List} where each key \text{K}K is a sorted string, and each value is the list of strings from the initial input that when sorted, are equal to \text{K}K.

In Java, we will store the key as a string, eg. code. In Python, we will store the key as a hashable tuple, eg. ('c', 'o', 'd', 'e').

50. Pow(x, n)

editotial class Solution { public double myPow(double x, int n) { long N = n; if (N < 0) { x = 1 / x; N = -N; } double ans = 1; double current_product = x; for (long i = N; i > 0; i /= 2) { if ((i % 2) == 1) { ans = ans * current_product; } current_product = current_product * current_product; } return ans; } };

53. Maximum Subarray

revised

61. Rotate List

same steps: https://leetcode.com/articles/rotate-list/ (https://leetcode.com/articles/rotate-list/)

65. Valid Number

https://www.geeksforgeeks.org/check-given-string-valid-number-integer-floating-point/ (https://www.geeksforgeeks.org/check-given-string-valid-number-integer-floating-point/)

67. Add Binary

https://www.geeksforgeeks.org/program-to-add-two-binary-strings/ (https://www.geeksforgeeks.org/program-to-add-two-binary-strings/)

74. Search a 2D Matrix

geeks for geeks

75. Sort Colors

While curr <= p2 :

If nums[curr] = 0 : swap currth and p0th elements and move both pointers to the right.

If nums[curr] = 2 : swap currth and p2th elements. Move pointer p2 to the left.

If nums[curr] = 1 : move pointer curr to the right.

class Solution { public: /* Dutch National Flag problem solution. */ void sortColors(vector& nums) { // for all idx < p0 : nums[idx < p0] = 0 // curr is an index of element under consideration int p0 = 0, curr = 0; // for all idx > p2 : nums[idx > p2] = 2 int p2 = nums.size() - 1;

```
while (curr <= p2) {
  if (nums[curr] == 0) {
    swap(nums[curr++], nums[p0++]);
  }
  else if (nums[curr] == 2) {
    swap(nums[curr], nums[p2--]);
  }
  else curr++;
}
```

} };

76. Minimum Window Substring

//https://leetcode.com/problems/minimum-window-substring/discuss/617888/C%2B%2B-Solution-68ms-explanation (https://leetcode.com/problems/minimum-window-substring/discuss/617888/C%2B%2B-Solution-68ms-explanation) We are going to use a two pointer approach to solve this.

The idea here is that

1. We will store the characters of t in a map lets say mapt.
2. We will have two pointers l and r.
3. Whille we traverse through s we check if the character is found in mapt If so we will store the character into another hash map lets say maps.
4. If the mapped charcter freq of s is less than or equal to mapt we increment a letter counter variable that will let us know when we have reached the t string size.
5. We try to find the smallest substring which contains all chracters in t using a while loop.

   S = "ADOBECODEBANC" and T = "ABC"

   mapt A -> 1 B -> 1 C -> 1

   We keep l=0 and traverse S with r.

   "ADOBECODEBANC" ^ | r

as A is present in t we use another map for s to store A into the hashmap

maps A->1 we have another variable lettercounter that keeps track of the size. lettercounter 1

"ADOBECODEBANC" ^ | r As D isnt present in t we just continue traversing.

"ADOBECODEBANC" ^ | r Same as the above step.

"ADOBECODEBANC" ^ | r
As B is present in map t

maps A->1 B->1

lettercounter 2

"ADOBECODEBANC" ^ | r "ADOBECODEBANC" ^ | r C is present in mapt

maps A->1 B->1 C->1

lettercounter 3 as lettercount is equal to t.size We will try shortening the substring

As there is only 1 A in s and t and A is the first character we cant reduce the size.

So out best bet currently would the substring ans= "ADOBEC"

We continue traversing and checking the above steps.

79. Word Search

// for(int r=0;r<4;r++) // { // res=res|findmatch(arr,str,i+dir[r][0],j+dir[r][1],level+1); // }

```
                  ye  loop use karne par wrong ans de raha hai
                   87/89  pass test case
```

https://www.geeksforgeeks.org/check-if-a-word-exists-in-a-grid-or-not/ (https://www.geeksforgeeks.org/check-if-a-word-exists-in-a-grid-or-not/) Approach: The idea used here is described in the steps below:

Check every cell, if the cell has first character, then recur one by one and try all 4 directions from that cell for a match. Mark the position in the grid as visited and recur in the 4 possible directions. After recurring, again mark the position as unvisited. Once all the letters in the word is matched, return true. Below is the implementation of the above approach:

81. Search in Rotated Sorted Array II

geeks for geeks

86. Partition List

https://leetcode.com/articles/partition-list/ (https://leetcode.com/articles/partition-list/)

92. Reverse Linked List II

https://www.geeksforgeeks.org/reverse-sublist-linked-list/ (https://www.geeksforgeeks.org/reverse-sublist-linked-list/)

95. Unique Binary Search Trees II

inspired by editorial https://leetcode.com/problems/unique-binary-search-trees-ii/discuss/617865/simple-cpp-Dp-Matrix-chain-Multiplication (https://leetcode.com/problems/unique-binary-search-trees-ii/discuss/617865/simple-cpp-Dp-Matrix-chain-Multiplication) class Solution { public LinkedList generate_trees(int start, int end) { LinkedList all_trees = new LinkedList(); if (start > end) { all_trees.add(null); return all_trees; }

```
// pick up a root
for (int i = start; i <= end; i++) {
  // all possible left subtrees if i is choosen to be a root
  LinkedList<TreeNode> left_trees = generate_trees(start, i - 1);

  // all possible right subtrees if i is choosen to be a root
  LinkedList<TreeNode> right_trees = generate_trees(i + 1, end);

  // connect left and right trees to the root i
  for (TreeNode l : left_trees) {
    for (TreeNode r : right_trees) {
      TreeNode current_tree = new TreeNode(i);
      current_tree.left = l;
      current_tree.right = r;
      all_trees.add(current_tree);
    }
  }
}
return all_trees;
```

}

public List generateTrees(int n) { if (n == 0) { return new LinkedList(); } return generate_trees(1, n); } }

101. Symmetric Tree

https://www.interviewbit.com/problems/symmetric-binary-tree/ (https://www.interviewbit.com/problems/symmetric-binary-tree/)

102. Binary Tree Level Order Traversal

interview bit

103. Binary Tree Zigzag Level Order Traversal

kiya hua hai https://www.interviewbit.com/problems/zigzag-level-order-traversal-bt/ (https://www.interviewbit.com/problems/zigzag-level-order-traversal-bt/)

107. Binary Tree Level Order Traversal II

https://www.geeksforgeeks.org/reverse-level-order-traversal/ (https://www.geeksforgeeks.org/reverse-level-order-traversal/)

109. Convert Sorted List to Binary Search Tree

Does we see pattern here ? There is one common thing in all balanced binary search tree is Root node is middle element of sorted array. Consider Example one given array is: 1 2 3 4 5 6 7 8 9 therefore middle elment of array is 5, 1 2 3 4 5 6 7 8 9 now, this array is divided into two parts one is left subarray(1 2 3 4) and second is right subarray (6 7 8 9) the root node for left subtree is the middle element of left subarray and root node right subtree array is middle element of right subarray, (shown in Fig 1).

Algorithm:

Initialize start=0, end = lenght of the array-1. create a tree node with mid as root (lets call its A). Recursively do following steps: Calculate mid of left subarray and make it root of left subtree of A. Calculate mid of right subarray and make it root of right subtree of A.

113. Path Sum II

https://www.interviewbit.com/problems/root-to-leaf-paths-with-sum/ (https://www.interviewbit.com/problems/root-to-leaf-paths-with-sum/)
https://leetcode.com/problems/path-sum-ii/discuss/415716/C%2B%2B-Beats-92-Simple-(with-Explanation) (https://leetcode.com/problems/path-sum-ii/discuss/415716/C%2B%2B-Beats-92-Simple-(with-Explanation))

116. Populating Next Right Pointers in Each Node

interviewbit

121. Best Time to Buy and Sell Stock

https://leetcode.com/problems/best-time-to-buy-and-sell-stock/discuss/39039/Sharing-my-simple-and-clear-C%2B%2B-solution (https://leetcode.com/problems/best-time-to-buy-and-sell-stock/discuss/39039/Sharing-my-simple-and-clear-C%2B%2B-solution)

124. Binary Tree Maximum Path Sum

kiya hua hai

129. Sum Root to Leaf Numbers

https://www.interviewbit.com/problems/sum-root-to-leaf-numbers/ (https://www.interviewbit.com/problems/sum-root-to-leaf-numbers/)

137. Single Number II

That's already great, so one could detect the bit which appears once, and the bit which appears three times. The problem is to distinguish between these two situations.

AND and NOT

To separate number that appears once from a number that appears three times let's use two bitmasks instead of one: seen_once and seen_twice.

The idea is to

change seen_once only if seen_twice is unchanged

change seen_twice only if seen_once is unchanged

// first appearence: // add num to seen_once // don't add to seen_twice because of presence in seen_once

```
  // second appearance:
  // remove num from seen_once
  // add num to seen_twice

  // third appearance:
  // don't add to seen_once because of presence in seen_twice
  // remove num from seen_twice
```

139. Word Break

https://leetcode.com/problems/word-break/discuss/43814/C%2B%2B-Dynamic-Programming-simple-and-fast-solution-(4ms)-with-optimization (https://leetcode.com/problems/word-break/discuss/43814/C%2B%2B-Dynamic-Programming-simple-and-fast-solution-(4ms)-with-optimization)

We use a boolean vector dp[]. dp[i] is set to true if a valid word (word sequence) ends there. The optimization is to look from current position i back and only substring and do dictionary look up in case the preceding position j with dp[j] == true is found.

bool wordBreak(string s, unordered_set &dict) { if(dict.size()==0) return false;

```
    vector<bool> dp(s.size()+1,false);
    dp[0]=true;

    for(int i=1;i<=s.size();i++)
    {
        for(int j=i-1;j>=0;j--)
        {
            if(dp[j])
            {
                string word = s.substr(j,i-j);
                if(dict.find(word)!= dict.end())
                {
                    dp[i]=true;
                    break; //next i
                }
            }
        }
    }

    return dp[s.size()];
}
```

142. Linked List Cycle II

***https://www.geeksforgeeks.org/find-first-node-of-loop-in-a-linked-list/ (https://www.geeksforgeeks.org/find-first-node-of-loop-in-a-linked-list/)

143. Reorder List

https://leetcode.com/articles/reorder-list/ (https://leetcode.com/articles/reorder-list/)

146. LRU Cache

https://leetcode.com/problems/lru-cache/discuss/572417/C%2B%2B-map-%2B-double-linked-list-(simple-code-but-very-long) (https://leetcode.com/problems/lru-cache/discuss/572417/C%2B%2B-map-%2B-double-linked-list-(simple-code-but-very-long))

using doubly linked list and hash_map

147. Insertion Sort List

https://leetcode.com/problems/insertion-sort-list/discuss/681839/C%2B%2B-Simple-O(N2)-solution.-Detailed-Explanation (https://leetcode.com/problems/insertion-sort-list/discuss/681839/C%2B%2B-Simple-O(N2)-solution.-Detailed-Explanation)

148. Sort List

https://leetcode.com/problems/sort-list/discuss/705163/Cpp-Clean-code-with-Analysis (https://leetcode.com/problems/sort-list/discuss/705163/Cpp-Clean-code-with-Analysis)

To sort a linkedlist with constant space, we can use merge sort (for array it will be O(N) space).

Let's recap what to do if we are sorting an array:

define base case: there is only one element find mid and split array into left - mid and mid+1 - right recurse left part and right part use a temporary array to store the sorted array in range of current left - right reassign back to the original array from the temporary array Convert to linkedlist:

define base case: only one node (listnode) use slow-fast pointers to find mid set mid -> next = NULL so it will split left - mid -> NULL and mid + 1 - right recurse left and right create a dummy list node and use two pointers to append append the one with smaller value first since we didn't create a new list, we can just return now

150. Evaluate Reverse Polish Notation

While traversing the vector A, we encounter two types of string.

A[i] is an integer A[i] is a operation. Case 1: Whenever we encounter integer, we push it into our stack Case 2: We apply the operation to the last two elements added to our stack. Lets call the previous two elements as temp1 and temp2. We store the result inside temp1 and push it again to stack.

Successively doing this will result in stack containing finally only one element and that my friends would be the answer to the question!

int Solution::evalRPN(vector &A) {

```
stack <int> stk;
for(auto s : A){
    if(s.size()>1 || isdigit(s[0]))stk.push(stoi(s));
    else{
        auto temp2 = stk.top(); stk.pop();
        auto temp1 = stk.top(); stk.pop();
        switch(s[0]){
            case '+': temp1+=temp2; break;
            case '-': temp1-=temp2; break;
            case '*': temp1*=temp2; break;
            case '/': temp1/=temp2; break;
        }
        stk.push(temp1);
    }
}
return stk.top();
```

}

161. One Edit Distance

determine if they are both one edit distance apart. agar s==t return false;

164. Maximum Gap

https://www.youtube.com/watch?v=VT-6zwGKYwA (https://www.youtube.com/watch?v=VT-6zwGKYwA)

169. Majority Element

moore voting algo

https://www.geeksforgeeks.org/majority-element/ (https://www.geeksforgeeks.org/majority-element/)

171. Excel Sheet Column Number

https://www.interviewbit.com/problems/excel-column-number/ (https://www.interviewbit.com/problems/excel-column-number/)

https://www.geeksforgeeks.org/find-excel-column-number-column-title/ (https://www.geeksforgeeks.org/find-excel-column-number-column-title/)

column column number A -> 1 B -> 2 C -> 3 ... Z -> 26 AA -> 27 AB -> 28

200. Number of Islands

revised

206. Reverse Linked List

https://www.geeksforgeeks.org/reverse-a-linked-list/ (https://www.geeksforgeeks.org/reverse-a-linked-list/)

210. Course Schedule II

Approach 2: Using Node Indegree Intuition

This approach is much easier to think about intuitively as will be clear from the following point/fact about topological ordering.

The first node in the topological ordering will be the node that doesn't have any incoming edges. Essentially, any node that has an in-degree of 0 can start the topologically sorted order. If there are multiple such nodes, their relative order doesn't matter and they can appear in any order.

Our current algorithm is based on this idea. We first process all the nodes/course with 0 in-degree implying no prerequisite courses required. If we remove all these courses from the graph, along with their outgoing edges, we can find out the courses/nodes that should be processed next. These would again be the nodes with 0 in-degree. We can continuously do this until all the courses have been accounted for.

Algorithm

Initialize a queue, Q to keep a track of all the nodes in the graph with 0 in-degree. Iterate over all the edges in the input and create an adjacency list and also a map of node v/s in-degree. Add all the nodes with 0 in-degree to Q. The following steps are to be done until the Q becomes empty. Pop a node from the Q. Let's call this node, N. For all the neighbors of this node, N, reduce their in-degree by 1. If any of the nodes' in-degree reaches 0, add it to the Q. Add the node N to the list maintaining topologically sorted order. Continue from step 4.1. Let us now look at an animation depicting this algorithm and then we will get to the implementations.

213. House Robber II

Since you cannot rob both the first and last house, just create two separate vectors, one excluding the first house, and another excluding the last house. The best solution generated from these two vectors using the original House Robber DP algorithm is the optimal one.

222. Count Complete Tree Nodes

https://www.geeksforgeeks.org/count-full-nodes-binary-tree-iterative-recursive/ (https://www.geeksforgeeks.org/count-full-nodes-binary-tree-iterative-recursive/)

224. Basic Calculator

https://leetcode.com/problems/basic-calculator/discuss/564036/C%2B%2B-solution-using-stack (https://leetcode.com/problems/basic-calculator/discuss/564036/C%2B%2B-solution-using-stack)

class Solution { public: int calculate(string str) { long long int i,j,ans=0; stackstk; // (+)sign=> 1 // (-)sign=> -1 long long int num=0; long long int res=0,sign=1;

```
    for(i=0;i<str.size();i++)
    {
         // char ch=str[i];
        // get number_part into num variable
        while(i<str.size()&&isdigit(str[i]))
        {
            num=num*10+(int)(str[i]-'0');
            i++;
        }

        res+=num*sign;
        num=0;// free num integer variable

        // decide next numerical part ka sign
        if(str[i]=='+')
        {
            sign=1;
        }
        else if(str[i]=='-')
        {
            sign=-1;
        }


        else if(str[i]=='(')// opening bracket par previous res ko push karo and sign ko bhi pus karo
        {
            stk.push(res);
            res=0;
            stk.push(sign);
            sign=1;
        }
        else if(str[i]==')')// closing bracket par res abhi tak tak update karo and pop karo
        {
            res=res*stk.top();// ye st.top()  sign hai
            stk.pop();
            res+=stk.top();// value inside this (4+5+2)
            stk.pop();// finalyy pop int
        }

    }
    return res;
}
```

};

226. Invert Binary Tree

interview bit

230. Kth Smallest Element in a BST

Breadth First Search (BFS)

We scan through the tree level by level, following the order of height, from top to bottom. The nodes on higher level would be visited before the ones with lower levels.

On the following figure the nodes are numerated in the order you visit them, please follow 1-2-3-4-5 to compare different strategies.

257. Binary Tree Paths

https://www.interviewbit.com/problems/root-to-leaf-paths-with-sum/ (https://www.interviewbit.com/problems/root-to-leaf-paths-with-sum/)

https://leetcode.com/problems/path-sum-ii/discuss/415716/C%2B%2B-Beats-92-Simple-(with-Explanation) (https://leetcode.com/problems/path-sum-ii/discuss/415716/C%2B%2B-Beats-92-Simple-(with-Explanation))

261. Graph Valid Tree

The second strategy is, instead of using a seen set, to use a seen map that also keeps track of the "parent" node that we got to a node from. We'll call this map parent. Then, when we iterate through the neighbours of a node, we ignore the "parent" node as otherwise it'll be detected as a trivial cycle (and we know that the parent node has already been visited by this point anyway). The starting node (0 in this implementation) has no "parent", so put it as -1.

At first, it's a little more difficult to understand why this strategy even works. A good way to think about it is to remember that like the first approach, we just want to avoid going along edges we've already been on (in the opposite direction). The parent links prevent that, as each node is only entered for exploration once. So, imagine you're walking through a maze, with the condition that you're not allowed to go back along any path you've already been on. If you still somehow end up somewhere you were previously, there must have been a cycle! The best strategy for this problem is probably the second one, because it doesn't require modifying the adjacency list. For more complex graph problems, the first strategy can be useful though.

Algorithm

In the example, we used an iterative depth-first search. Here is the complete code for this. Alternatively, you could use recursion, as long as you're fairly confident with it. The recursive approach is more elegant, but is considered inferior to the iterative version in some programming languages, such as Python. This is because the space used by run-time stacks vary between programming languages.

On the plus side, we can use a simple seen set and just pass a parent parameter. This makes the code a bit simpler! public boolean validTree(int n, int[][] edges) {

```java
List<List<Integer>> adjacencyList = new ArrayList<>();
for (int i = 0; i < n; i++) {
    adjacencyList.add(new ArrayList<>());
}
for (int[] edge : edges) {
    adjacencyList.get(edge[0]).add(edge[1]);
    adjacencyList.get(edge[1]).add(edge[0]);
}

Map<Integer, Integer> parent = new HashMap<>();
parent.put(0, -1);
Queue<Integer> queue = new LinkedList<>();
queue.offer(0);

while (!queue.isEmpty()) {
    int node = queue.poll();
    for (int neighbour : adjacencyList.get(node)) {
        if (parent.get(node) == neighbour) {
            continue;
        }
        if (parent.containsKey(neighbour)) {
            return false;
        }
        queue.offer(neighbour);
        parent.put(neighbour, node);
    }
}

return parent.size() == n;
```

}

268. Missing Number

We can harness the fact that XOR is its own inverse to find the missing element in linear time.

Algorithm

Because we know that nums contains nn numbers and that it is missing exactly one number on the range [0..n-1][0..n−1], we know that nn definitely replaces the missing number in nums. Therefore, if we initialize an integer to nn and XOR it with every index and value, we will be left with the missing number. Consider the following example (the values have been sorted for intuitive convenience, but need not be):

missing

=4∧(0∧0)∧(1∧1)∧(2∧3)∧(3∧4) =(4∧4)∧(0∧0)∧(1∧1)∧(3∧3)∧2 =0∧0∧0∧0∧2 =2

279. Perfect Squares

Approach 2: Dynamic Programming Intuition

The reason why it failed with the brute-force approach is simply because we re-calculate the sub-solutions over and over again. However, the formula that we derived before is still valuable. All we need is a better way to implement the formula.

\text{numSquares}(n) = \min \Big(\text{numSquares(n-k) + 1}\Big) \qquad \forall k \in {\text{square numbers}}numSquares(n)=min(numSquares(n-k) + 1)∀k∈{square numbers}

One might notice that, the problem is similar to the Fibonacci number problem, judging from the formula. And like Fibonacci number, we have several more efficient ways to calculate the solution, other than the simple recursion.

One of the ideas to solve the stack overflow issue in recursion is to apply the Dynamic Programming (DP) technique, which is built upon the idea of reusing the results of intermediate sub-solutions to calculate the final solution.

To calculate the value of \text{numSquares}(n)numSquares(n), first we need to calculate all the values before nn, i.e. \text{numSquares}(n-k) \qquad \forall{k} \in{\text{square numbers}}numSquares(n−k)∀k∈{square numbers}. If we have already kept the solution for the number n-kn−k in somewhere, we then would not need to resort to the recursive calculation which prevents the stack overflow.

Algorithm

Based on the above intuition, we could implement the DP solution in the following steps.

As for almost all DP solutions, we first create an array dp of one or multiple dimensions to hold the values of intermediate sub-solutions, as well as the final solution which is usually the last element in the array. Note that, we create a fictional element dp[0]=0 to simplify the logic, which helps in the case that the remainder (n-k) happens to be a square number. As an additional preparation step, we pre-calculate a list of square numbers (i.e. square_nums) that is less than the given number n. As the main step, we then loop from the number 1 to n, to calculate the solution for each number i (i.e. numSquares(i)). At each iteration, we keep the result of numSquares(i) in dp[i], while resuing the previous results stored in the array. At the end of the loop, we then return the last element in the array as the result of the solution. In the graph below, we illustrate how to calculate the results of numSquares(4) and numSquares(5) which correspond to the values in dp[4] and dp[5].

pic

Here are some sample implementations. In particular, the Python solution took ~3500 ms, which was faster than ~50% submissions at the time.

Note: the following python solution works for Python2 only. For some unknown reason, it takes significantly longer time for Python3 to run the same code

285. Inorder Successor in BST

kiye hai delete node in bst me kaam me aaya tha

296. Best Meeting Point

Finding the best meeting point in a 2D grid seems difficult. Let us take a step back and solve the 1D case which is much simpler. Notice that the Manhattan distance is the sum of two independent variables. Therefore, once we solve the 1D case, we can solve the 2D case as two independent 1D problems.

Let us look at some 1D examples below:

Case #1: 1-0-0-0-1

Case #2: 0-1-0-1-0 We know the best meeting point must locate somewhere between the left-most and right-most point. For the above two cases, we would select the center point at x = 2x=2 as the best meeting point. How about choosing the mean of all points as the meeting point?

Consider this case:

Case #3: 1-0-0-0-0-0-0-1-1 Using the mean gives us \bar{x} = \frac{0 + 7 + 8}{3} = 5 x ˉ = 3 0+7+8
=5 as the meeting point. The total distance is 1010.

But the best meeting point should be at x = 7x=7 and the total distance is 88.

You may argue that the mean is close to the optimal point. But imagine a larger case with many 1's congregating on the right side and just a single 1 on the left-most side. Using the mean as the meeting point would be far from optimal.

Besides mean, what is a better way to represent the distribution of points? Would median be a better representation? Indeed. In fact, the median must be the optimal meeting point.

Case #4: 1-1-0-0-1 To see why this is so, let us look at case #4 above and choose the median x = 1x=1 as our initial meeting point. Assume that the total distance traveled is d. Note that we have equal number of points distributed to its left and to its right. Now let us move one step to its right where x = 2x=2 and notice how the distance changes accordingly.

Since there are two points to the left of x = 2x=2, we add 2 * (+1)2∗(+1) to d. And d is offset by –1 since there is one point to the right. This means the distance had overall increased by 1.

Therefore, it is clear that:

As long as there is equal number of points to the left and right of the meeting point, the total distance is minimized.

Case #5: 1-1-0-0-1-1 One may think that the optimal meeting point must fall on one of the 1's. This is true for cases with odd number of 1's, but not necessarily true when there are even number of 1's, just like case #5 does. You can choose any of the x = 1x=1 to x = 4x=4 points and the total distance is minimized. Why?

The implementation is direct. First we collect both the row and column coordinates, sort them and select their middle elements. Then we calculate the total distance as the sum of two independent 1D problems.

297. Serialize and Deserialize Binary Tree

https://www.geeksforgeeks.org/serialize-deserialize-binary-tree/ (https://www.geeksforgeeks.org/serialize-deserialize-binary-tree/)

304. Range Sum Query 2D - Immutable

https://leetcode.com/problems/range-sum-query-2d-immutable/discuss/75350/Clean-C%2B%2B-Solution-and-Explaination-O(mn)-space-with-O(1)-time (https://leetcode.com/problems/range-sum-query-2d-immutable/discuss/75350/Clean-C%2B%2B-Solution-and-Explaination-O(mn)-space-with-O(1)-time)

309. Best Time to Buy and Sell Stock with Cooldown

For a sequence of prices, denoted as \text{price}[0, 1, ..., n]price[0,1,...,n], let us first define our target function called \text{MP}(i)MP(i). The function \text{MP}(i)MP(i) gives the maximal profits that we can gain for the price subsequence starting from the index ii, i.e. \text{price}[i, i+1, ..., n]price[i,i+1,...,n].

With the final formula we derived for our target function \text{MP}(i)MP(i), we can now go ahead and translate it into any programming language.

Since the formula deals with subsequences of price that start from the last price point, we then could do an iteration over the price list in the reversed order.

We define an array MP[i] to hold the values for our target function \text{MP}(i)MP(i). We initialize the array with zeros, which correspond to the base case where the minimal profits that we can gain is zero. Note that, here we did a trick to pad the array with two additional elements, which is intended to simplify the branching conditions, as one will see later.

To calculate the value for each element MP[i], we need to look into two cases as we discussed in the previous section, namely:

Case 1). we buy the stock at the price point price[i], then we sell it at a later point. As one might notice, the initial padding on the MP[i] array saves us from getting out of boundary in the array.

Case 2). we do no transaction with the stock at the price point price[i].

At the end of each iteration, we then pick the largest value from the above two cases as the final value for MP[i].

At the end of the loop, the MP[i] array will be populated. We then return the value of MP[0], which is the desired solution for the problem.

311. Sparse Matrix Multiplication

OK, I admit the related topic icon when think about the question.

hash[key] = value key : column index of B value : non-zero row index of key class Solution { public: vector<vector> multiply(vector<vector>& A, vector<vector>& B) { /* A = mn , B = yz , AB = mz */ vector<vector> res(A.size(),vector(B[0].size(),0)); unordered_map<int,unordered_set> col_set; / build up the hash table / for(int i = 0 ; i < B[0].size() ; i++){ for(int j = 0; j < B.size() ;j++) if(B[j][i]) col_set[i].insert(j); } / calculate each output element. / for(int i = 0 ; i < res.size() ; i++){ for(int j = 0 ; j < res[0].size();j++){ int sum = 0; for(int l : col_set[j]) sum += A[i][l]B[l][j]; res[i][j] = sum; } } return res; } };

319. Bulb Switcher

https://leetcode.com/problems/bulb-switcher/discuss/77132/The-simplest-and-most-efficient-solution-well-explained (https://leetcode.com/problems/bulb-switcher/discuss/77132/The-simplest-and-most-efficient-solution-well-explained)

333. Largest BST Subtree

https://www.youtube.com/watch?v=ChZPl1Zq1os (https://www.youtube.com/watch?v=ChZPl1Zq1os)

https://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/ (https://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/)

335. Self Crossing

After drawing a few crossing cases ourselves, we can simply find out there are two basic patterns:

x[i-1]<=x[i-3] && x[i]>=x[i-2] the ending circle line cross the beginning circle line in one circle; i>=5 && x[i-1]<=x[i-3] && x[i]>=x[i-2]-x[i-4] the second line of the next circle cross the the beginning of the previous circle between two adjacent circles; But still that is not over yet, how about some special cases? How about the first line of the next circle and the previous circle? Yeah, the beginning line of the next circle can overlap the the first line of the previous circle - another two adjacent circles case:

i>=4 && x[i-1]==x[i-3] && x[i]>=x[i-2]-x[i-4] Quite straightforward. Then we can test our patterns now, however soon we will find out that the second cases is not strong enough to cover all possible situations - the second line of the next circle crossing the previous circle at the its first line

[3,3,3,2,1,1] is an example here, so x[i-2]>=x[i-4] then must be added to our conditions; [3,3,4,4,10,4,4,,3,3] is another typical example for x[i-3]<=x[i-1]+x[i-5] condition, which also should be added to make the constrained conditions stronger; At last, we make it! Bang! End of story with a very terse, clean and efficient code as follows.

Updated: 2016-09-12 For better and easier reasoning, here is the thinking thread. Suppose i is the current line, then:

i and i-3 can cross i and i-4 can cross i and i-5 can cross no more or no less just exactly the right combination.

Now it's time for us to restrict the conditions to make them just happen.

i and i-3

i>=i-2 && i-1<=i-3

i and i-4

i+i-4>=i-2 && i-1==i-3

i and i-5

i+i-4>=i-2 && i-2>=i-4 && i-1+i-5>=i-3 && i-1<=i-3

337. House Robber III

https://leetcode.com/problems/house-robber-iii/discuss/79330/Step-by-step-tackling-of-the-problem (https://leetcode.com/problems/house-robber-iii/discuss/79330/Step-by-step-tackling-of-the-problem) Step I -- Think naively

At first glance, the problem exhibits the feature of "optimal substructure": if we want to rob maximum amount of money from current binary tree (rooted at root), we surely hope that we can do the same to its left and right subtrees.

So going along this line, let's define the function rob(root) which will return the maximum amount of money that we can rob for the binary tree rooted at root; the key now is to construct the solution to the original problem from solutions to its subproblems, i.e., how to get rob(root) from rob(root.left), rob(root.right), ... etc.

Apparently the analyses above suggest a recursive solution. And for recursion, it's always worthwhile figuring out the following two properties:

Termination condition: when do we know the answer to rob(root) without any calculation? Of course when the tree is empty ---- we've got nothing to rob so the amount of money is zero.

Recurrence relation: i.e., how to get rob(root) from rob(root.left), rob(root.right), ... etc. From the point of view of the tree root, there are only two scenarios at the end: root is robbed or is not. If it is, due to the constraint that "we cannot rob any two directly-linked houses", the next level of subtrees that are available would be the four "grandchild-subtrees" (root.left.left, root.left.right, root.right.left, root.right.right). However if root is not robbed, the next level of available subtrees would just be the two "child-subtrees" (root.left, root.right). We only need to choose the scenario which yields the larger amount of money.

Here is the program for the ideas above:

public int rob(TreeNode root) { if (root == null) return 0;

```
int val = 0;

if (root.left != null) {
    val += rob(root.left.left) + rob(root.left.right);
}

if (root.right != null) {
    val += rob(root.right.left) + rob(root.right.right);
}

return Math.max(val + root.val, rob(root.left) + rob(root.right));
```

} However the solution runs very slowly (1186 ms) and barely got accepted (the time complexity turns out to be exponential, see my comments below).

Step II -- Think one step further

In step I, we only considered the aspect of "optimal substructure", but think little about the possibilities of overlapping of the subproblems. For example, to obtain rob(root), we need rob(root.left), rob(root.right), rob(root.left.left), rob(root.left.right), rob(root.right.left), rob(root.right.right); but to get rob(root.left), we also need rob(root.left.left), rob(root.left.right), similarly for rob(root.right). The naive solution above computed these subproblems repeatedly, which resulted in bad time performance. Now if you recall the two conditions for dynamic programming (DP): "optimal substructure" + "overlapping of subproblems", we actually have a DP problem. A naive way to implement DP here is to use a hash map to record the results for visited subtrees.

And here is the improved solution:

public int rob(TreeNode root) { return robSub(root, new HashMap<>()); }

private int robSub(TreeNode root, Map<TreeNode, Integer> map) { if (root == null) return 0; if (map.containsKey(root)) return map.get(root);

```
int val = 0;

if (root.left != null) {
    val += robSub(root.left.left, map) + robSub(root.left.right, map);
}

if (root.right != null) {
    val += robSub(root.right.left, map) + robSub(root.right.right, map);
}

val = Math.max(val + root.val, robSub(root.left, map) + robSub(root.right, map));
map.put(root, val);

return val;
```

} The runtime is sharply reduced to 9 ms, at the expense of O(n) space cost (n is the total number of nodes; stack cost for recursion is not counted).

Step III -- Think one step back

In step I, we defined our problem as rob(root), which will yield the maximum amount of money that can be robbed of the binary tree rooted at root. This leads to the DP problem summarized in step II.

Now let's take one step back and ask why we have overlapping subproblems. If you trace all the way back to the beginning, you'll find the answer lies in the way how we have defined rob(root). As I mentioned, for each tree root, there are two scenarios: it is robbed or is not. rob(root) does not distinguish between these two cases, so "information is lost as the recursion goes deeper and deeper", which results in repeated subproblems.

If we were able to maintain the information about the two scenarios for each tree root, let's see how it plays out. Redefine rob(root) as a new function which will return an array of two elements, the first element of which denotes the maximum amount of money that can be robbed if root is not robbed, while the second element signifies the maximum amount of money robbed if it is robbed.

Let's relate rob(root) to rob(root.left) and rob(root.right)..., etc. For the 1st element of rob(root), we only need to sum up the larger elements of rob(root.left) and rob(root.right), respectively, since root is not robbed and we are free to rob its left and right subtrees. For the 2nd element of rob(root), however, we only need to add up the 1st elements of rob(root.left) and rob(root.right), respectively, plus the value robbed from root itself, since in this case it's guaranteed that we cannot rob the nodes of root.left and root.right.

As you can see, by keeping track of the information of both scenarios, we decoupled the subproblems and the solution essentially boiled down to a greedy one. Here is the program:

public int rob(TreeNode root) { int[] res = robSub(root); return Math.max(res[0], res[1]); }

private int[] robSub(TreeNode root) { if (root == null) return new int[2];

```
int[] left = robSub(root.left);
int[] right = robSub(root.right);
int[] res = new int[2];

res[0] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);
res[1] = root.val + left[0] + right[0];

return res;
```

357. Count Numbers with Unique Digits

when n is 0, it is clear that there is just one number 0.

when n is 1, it is trivial that there are 10 numbers: 0,1,2...9.

when n is 2, the range is [0, 99]. The total unique digits is divided to two part: just one digit or two digit.

```
      **dp[2]  = dp[1] + the combination with  two digits.**
```

'0' could only be at unit digit. so when '0' is at unit digit, there are 9 kinds of. when there is no '0', there are 9 kinds of numbers at tens digit, and 8 kinds of numbers at unit digit. So the combination with two digits are: 9 + 9*8 is equal to

```
                            9 * (1+8) = 9 * 9.
```

when n is 3, the range is [0, 999]. The total unique digits is divided to two part: less than 3 digit or 3 digit.

```
      **dp[3]  = dp[2] + the combination with  3 digits.**
```

'0' could only be at unit digit and ten's digit.

When '0' is at unit digit, there are 9 * 8 kind of numbers(9 is the kind of numbers at hundred's digit, 8 is the kind of numbers at ten's digit);

When '0' is at ten's digit, there are 9 * 8 kind of numbers(9 is the kind of numbers at hundred's digit, 8 is the kind of numbers at unit digit);

When there is no '0', there are 9 * 8 * 7 kinds of numbers(9 is the kind of numbers at hundred's digit, 8 is the kind of numbers at ten's digit, 7 is the kind of numbers at unit digit).

So there are:

```
        9 * 8  + 9 * 8 + 9 * 8 * 7  = 9 * 8 * (1 + 1 + 8 ) = 9 * 9 * 8
```

kinds of combinations with three digits.

....

Than it is easy to understand the DP solution of this problem.

359. Logger Rate Limiter

https://leetcode.com/problems/logger-rate-limiter/discuss/318165/Intuitive-C%2B%2B-solution (https://leetcode.com/problems/logger-rate-limiter/discuss/318165/Intuitive-C%2B%2B-solution)

361. Bomb Enemy

https://leetcode.com/problems/bomb-enemy/discuss/83405/Python-Brute-Force-O((mn)*(m%2Bn))-to-DP-O(mn) (https://leetcode.com/problems/bomb-enemy/discuss/83405/Python-Brute-Force-O((mn)*(m%2Bn))-to-DP-O(mn)) The brute force solution is very intuitive.. just count 'E's in rows and cols for each 0 in the matrix and return the maximum. This takes O((mn)*(m+n)) as you have to traverse up, down, left, and right for every i,j.

We can optimize on this by doing 4 passes and adding the number of E's seen so far, and reset if we see a 'W'. From left to right then right to left for E's seen in each row. And then from up to down and down to up for each E seen so far in column.

Total time -> O(4*mn) --> O(mn)

class Solution(object): def maxKilledEnemies(self, grid): if len(grid) == 0: return 0 max_hits = 0 nums = [[0 for i in range(len(grid[0]))] for j in range(len(grid))]

```
    #From Left
    for i in range(len(grid)):
        row_hits = 0
        for j in range(len(grid[0])):
            if grid[i][j] == 'E':
                row_hits += 1
            elif grid[i][j] == 'W':
                row_hits = 0
            else:
                nums[i][j] = row_hits

    #From Right
    for i in range(len(grid)):
        row_hits = 0
        for j in range(len(grid[0])-1, -1, -1):
            if grid[i][j] == 'W':
                row_hits = 0
            elif grid[i][j] == 'E':
                row_hits +=1
            else:
                nums[i][j] += row_hits

    for i in range(len(nums[0])):
        col_hits = 0
        for col in range(len(nums)):
            if grid[col][i] == 'E':
                col_hits += 1
            elif grid[col][i] == 'W':
                col_hits = 0
            else:
                nums[col][i] += col_hits

    for i in range(len(nums[0])):
        col_hits = 0
        for col in range(len(nums)-1, -1, -1):
            if grid[col][i] == 'E':
                col_hits +=1
            elif grid[col][i] == 'W':
                col_hits = 0
            else:
                nums[col][i] += col_hits
                max_hits = max(max_hits, nums[col][i])


    return max_hits
```

366. Find Leaves of Binary Tree

editorials: The order in which the elements (nodes) will be collected in the final answer depends on the "height" of these nodes. The height of a node is the number of edges from the node to the deepest leaf. The nodes that are located in the ith height will be appear in the ith collection in the final answer. For any given node in the binary tree, the height is obtained by adding 1 to the maximum height of any children. Formally, for a given node of the binary tree \text{root}root, it's height can be represented as

\text{height(root)} = \text{1} + \text{max(height(root.left), height(root.right))}height(root)=1+max(height(root.left), height(root.right))

Where \text{root.left}root.left and \text{root.right}root.right are left and right children of the root respectively

Algorithm

In our first approach, we'll simply traverse the tree recursively in a height first search manner using the function int getHeight(node), which will return the height of the given node in the binary tree. Since height of any node depends on the height of it's children node, hence we traverse the tree in a post-order manner (i.e. height of the childrens are calculated first before calculating the height of the given node). Additionally, whenever we encounter a null node, we simply return -1 as it's height.

Next, we'll store the pair (height, val) for all the nodes which will be sorted later to obtain the final answer. The sorting will be done in increasing order considering the height first and then the val. Hence we'll obtain all the pairs in the increasing order of their height in the given binary tree.

368. Largest Divisible Subset

https://leetcode.com/problems/largest-divisible-subset/discuss/83998/C%2B%2B-Solution-with-Explanations (https://leetcode.com/problems/largest-divisible-subset/discuss/83998/C%2B%2B-Solution-with-Explanations) The key concept here is: Given a set of integers that satisfies the property that each pair of integers inside the set are mutually divisible, for a new integer S, S can be placed into the set as long as it can divide the smallest number of the set or is divisible by the largest number of the set.

For example, let's say we have a set P = { 4, 8, 16 }, P satisfies the divisible condition. Now consider a new number 2, it can divide the smallest number 4, so it can be placed into the set; similarly, 32 can be divided by 16, the biggest number in P, it can also placed into P.

Next, let's define:

EDIT: For clarification, the following definitions try to enlarge candidate solutions by appending a larger element at the end of each potential set, while my implementation below is prefixing a smaller element at the front of a set. Conceptually they are equivalent but by adding smaller elements at the front saves the trouble for keeping the correct increasing order for the final answer. Please refer to comments in code for more details.

For an increasingly sorted array of integers a[1 .. n]

T[n] = the length of the largest divisible subset whose largest number is a[n]

T[n+1] = max{ 1 + T[i] if a[n+1] mod a[i] == 0 else 1 }

Now, deducting T[n] becomes straight forward with a DP trick. For the final result we will need to maintain a backtrace array for the answer.

Implementation in C++:

class Solution { public: vector largestDivisibleSubset(vector& nums) { sort(nums.begin(), nums.end());

```
    vector<int> T(nums.size(), 0);
    vector<int> parent(nums.size(), 0);

    int m = 0;
    int mi = 0;

    // for(int i = 0; i < nums.size(); ++i) // if extending by larger elements
    for(int i = nums.size() - 1; i >= 0; --i) // iterate from end to start since it's easier to track the answer index
    {
        // for(int j = i; j >=0; --j) // if extending by larger elements
        for(int j = i; j < nums.size(); ++j)
        {
            // if(nums[i] % nums[j] == 0 && T[i] < 1 + T[j]) // if extending by larger elements
            // check every a[j] that is larger than a[i]
            if(nums[j] % nums[i] == 0 && T[i] < 1 + T[j])
            {
                // if a[j] mod a[i] == 0, it means T[j] can form a larger subset by putting a[i] into T[j]
                T[i] = 1 + T[j];
                parent[i] = j;

                if(T[i] > m)
                {
                    m = T[i];
                    mi = i;
                }
            }
        }
    }

    vector<int> ret;

    for(int i = 0; i < m; ++i)
    {
        ret.push_back(nums[mi]);
        mi = parent[mi];
    }

    // sort(ret.begin(), ret.end()); // if we go by extending larger ends, the largest "answer" element will come first since
the candidate element we observe will become larger and larger as i increases in the outermost "for" loop above.
   // alternatively, we can sort nums in decreasing order obviously.

    return ret;
}
```

};

376. Wiggle Subsequence

Approach #2 Dynamic Programming [Accepted] Algorithm

To understand this approach, take two arrays for dp named upup and downdown.

Whenever we pick up any element of the array to be a part of the wiggle subsequence, that element could be a part of a rising wiggle or a falling wiggle depending upon which element we have taken prior to it.

up[i]up[i] refers to the length of the longest wiggle subsequence obtained so far considering i^{th}i th element as the last element of the wiggle subsequence and ending with a rising wiggle.

Similarly, down[i]down[i] refers to the length of the longest wiggle subsequence obtained so far considering i^{th}i th element as the last element of the wiggle subsequence and ending with a falling wiggle.

up[i]up[i] will be updated every time we find a rising wiggle ending with the i^{th}i th element. Now, to find up[i]up[i], we need to consider the maximum out of all the previous wiggle subsequences ending with a falling wiggle i.e. down[j]down[j], for every j<ij<i and nums[i]>nums[j]nums[i]>nums[j]. Similarly, down[i]down[i] will be updated.

380. Insert Delete GetRandom O(1)

This problem requires us to design a data structure that supports insert, remove, and getRandom in constant time O(1).

This solution uses two data structures, a Hash Table and a resizable array. The Hash Table maps the values to their respective indices in the array. https://leetcode.com/problems/insert-delete-getrandom-o1/discuss/551158/C%2B%2B-O(1)-Solution-using-array-and-hashmap-(with-explanation) (https://leetcode.com/problems/insert-delete-getrandom-o1/discuss/551158/C%2B%2B-O(1)-Solution-using-array-and-hashmap-(with-explanation)) Insert Operation This operation can be easily supported by inserting the new value to the end of the resizable array and updating its index in the Hash Table.

Remove Operation This operation is a little tricky. We can easily remove any element from the Hash Table in constant time as long as we know it's value. However, removing an element from the middle of the array is a costly operation. Fortunately, removing the last element of the array can be done in constant time (using pop_back()). We will swap the value we want to remove with the last element of the array and then remove the last element of the array.

getRandom Operation This operation can be easily done by using C++ inbuilt rand() on the array.

The code is as follows.

394. Decode String

https://leetcode.com/problems/decode-string/discuss/354083/Beats-100-cpp-solutions.-Very-easy-to-understand (https://leetcode.com/problems/decode-string/discuss/354083/Beats-100-cpp-solutions.-Very-easy-to-understand).

406. Queue Reconstruction by Height

recording

416. Partition Equal Subset Sum

Dynamic Programming Solution The problem can be solved using dynamic programming when the sum of the elements is not too big. We can create a 2D array part[][] of size (sum/2 + 1)*(n+1). And we can construct the solution in bottom up manner such that every filled entry has following property

part[i][j] = true if a subset of {arr[0], arr[1], ..arr[j-1]} has sum equal to i, otherwise false

417. Pacific Atlantic Water Flow

https://leetcode.com/problems/pacific-atlantic-water-flow/discuss/324852/CPP-Solution-(2-ways)-DFS-BFS-with-Explanation (https://leetcode.com/problems/pacific-atlantic-water-flow/discuss/324852/CPP-Solution-(2-ways)-DFS-BFS-with-Explanation) Second Solution: Solution class (2) The second approach is much more efficient as mentioned before. We need to record the 'map' of where the pacific and atlantic ocean can spill into. So we need two separate maps one for the atlantic and one for the pacific. Once done, we just compare to see where the common areas where both the atlantic and pacific water can 'spill upwards' from the edges. By spill upwards I mean imagine the water from the edges flowing up hill, hence we are looking for bigger numbers. This is just the reverse logic from (1).

class Solution { private:

```
int rowSize, colSize;
vector<int> offset = {0, 1, 0, -1, 0};

bool outOfBounds(int row, int col){
    return row < 0 || col < 0 || row >= rowSize || col >= colSize;
}

void flowToPacificOrAtlantic(vector<vector<int>>& matrix, int row, int col, vector<vector<bool>>& visited){
    visited[row][col] = true;

    int currWaterLevel = matrix[row][col];
    for (int i = 1; i<offset.size(); i++){ // Go to left right up down using the offset --> (+1, +0) (+0, +1), (-1, +0), (+0,
-1)
        int r = row + offset[i], c = col + offset[i-1];
        if(!outOfBounds(r , c) &&
           currWaterLevel <= matrix[r][c] && // !!! current water level is lower than the surrounding water !!!
           !visited[r][c]){

            flowToPacificOrAtlantic(matrix, r, c, visited);
        }
    }
}
```

public: vector<vector> pacificAtlantic(vector<vector>& matrix) { vector<vector> result; rowSize = matrix.size(); if (rowSize == 0) return result; colSize = matrix[0].size();

```
    // initialize the maps to 0
    vector<vector<bool>> pacific(rowSize, vector<bool> (colSize, 0));
    vector<vector<bool>> atlantic(rowSize, vector<bool> (colSize, 0));

    for (int row = 0; row < rowSize; row++){
        flowToPacificOrAtlantic(matrix, row, 0, pacific); // column 0 (pacific)
        flowToPacificOrAtlantic(matrix, row, colSize -1, atlantic); // column end (atlantic)
    }
    for (int col = 0; col < colSize; col++){
         flowToPacificOrAtlantic(matrix, 0, col, pacific); // row 0 (pacific)
         flowToPacificOrAtlantic(matrix, rowSize -1, col, atlantic); // row end (atlantic)
    }

    // Compare the two 'maps' and find where they have a common 1
    for (int row = 0; row<rowSize; row++){
        for(int col = 0; col<colSize; col++){
            if (pacific[row][col] == 1 && atlantic[row][col] == 1){
                result.push_back({row, col});
            }
        }
    }

    return result;
}
```

};

Edit: I am adding BFS rather than the DFS for finding the visited matrix. This code is very similar to DFS except it uses a queue which prevents additional space incurred by use of the call stack. Just change flowToPacificOrAtlantic to BFS from solution 2 and everything should work.

void BFS(vector<vector> & matrix, int row, int col, vector<vector> &visited){ queue<pair<int, int>> todo; todo.push(make_pair(row, col)); visited[row][col] = 1; while (!todo.empty()){ pair<int, int> p = todo.front(); int currWaterLevel = matrix[p.first][p.second]; todo.pop(); // check neighbours of todo for (int i = 1; i<offset.size(); i++){ int r = p.first + offset[i], c = p.second + offset[i-1]; if (!outOfBounds(r, c) && currWaterLevel <= matrix[r][c] && !visited[r][c]){ todo.push(make_pair(r, c)); visited[r][c] = 1; } } } }

421. Maximum XOR of Two Numbers in an Array

Wow...I've been staring at this solution for 2 hours, I think I understand it now... Hope this comment can help someone instead of confusing you guys even more.

First of all...Great solution by the way~

I just need to dumb down the problem for myself. For me, I understand the problem like this, the question is asking you to find the max XOR of any 2 numbers, here's an example of the biggest XOR, my understanding is...make the result all 1s..eg,

1111 ^ 0000 = 1111 (15 ^ 0 = 15)

The part confused me the most is the, int tmp = max | (1 << i); and if(set.contains(tmp ^ prefix))

I treat tmp as the current "guessed new max" (not yet confirmed), which is the "old max which we have confirmed before" OR-ing "1 << i", this basically means "let's enable 1 more bit from the "confirmed max we found from last time".

Since we made a guess of the "new max" denoted as tmp. Then what do we need to do with it? I see people commenting, 0^0=0 [0^1=1] [1^0=1] 1^1=0 I KNOW THAT, but, what does it have to do with the question? SOOOO, basically what they are trying to say is...TO MAXIMIZE your solution, you want the result to be "1". In the case of tmp which we enabled the new bit to be "1", now all we need to do is to find a number in the set that has "0" on this bit, then we can confirm that there are at least 2 numbers in the set who's XOR result can achieve our "guessed new max". If those numbers exist, then we have CONFIRMED that our "guessed new max" IS the current max.

To summarize 2) OG used "tmp ^ prefix", this means that "if if we enable 1 more bit for max (guessed max), is there a number out there that has a 0 on this bit?" If this exist, then that means our "guess of the new max" is achievable.

435. Non-overlapping Intervals

Approach #4 Using Greedy Approach based on starting points [Accepted] Algorithm

If we sort the given intervals based on starting points, the greedy approach works very well. While considering the intervals in the ascending order of starting points, we make use of a pointer prevprev pointer to keep track of the interval just included in the final list. While traversing, we can encounter 3 possibilities as shown in the figure:

Non_overlapping

Case 1:

The two intervals currently considered are non-overlapping:

In this case, we need not remove any interval and we can continue by simply assigning the prevprev pointer to the later interval and the count of intervals removed remains unchanged.

Case 2:

The two intervals currently considered are overlapping and the end point of the later interval falls before the end point of the previous interval:

In this case, we can simply take the later interval. The choice is obvious since choosing an interval of smaller width will lead to more available space labelled as AA and BB, in which more intervals can be accommodated. Hence, the prevprev pointer is updated to current interval and the count of intervals removed is incremented by 1.

Case 3:

The two intervals currently considered are overlapping and the end point of the later interval falls after the end point of the previous interval:

In this case, we can work in a greedy manner and directly remove the later interval. To understand why this greedy approach works, we need to see the figure below, which includes all the subcases possible. It is clear from the figures that we choosing interval 1 always leads to a better solution in the future. Thus, the prevprev pointer remains unchanged and the count of intervals removed is incremented by 1.

438. Find All Anagrams in a String

Approach 2: Sliding Window with Array Algorithm

Hashmap is quite complex structure, with known performance issues in Java. Let's implement approach 1 using 26-elements array instead of hashmap:

Element number 0 contains count of letter a.

Element number 1 contains count of letter b.

...

Element number 26 contains count of letter z.

Algorithm

Build reference array pCount for string p.

Move sliding window along the string s:

Recompute sliding window array sCount at each step by adding one letter on the right and removing one letter on the left.

If sCount == pCount, update the output list.

Return output list.

450. Delete Node in a BST

https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/ (https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/)

451. Sort Characters By Frequency

This might not be the best answer but sure is easy to understand. And it works!

Approach: I can have a map(unordered), that stores character and it's corresponding count. store the count of each occuring character by traversing through the string once. Now as the question says, we need characters in decreasing order of frequency.

So next we need to sort the map on the basis of their value. Now we know we can't do that directly, so we copy the map contents in vector, and then sort it in the required way.

We then form an output string ans made from the counted characters.

class Solution { public:

//comaprision function to pass to sort() static bool cmp(pair<char,int>&a, pair<char,int>&b) { return a.second > b.second; }

```
string frequencySort(string s) {

    if(s.length()==0 || s.length()==1 || s.length()==2) return s;

    string ans="";
    unordered_map<char,int>m;

    vector<pair<char,int>>vm;

    for(char c: s)
    {
        // if(c != ' ')
            m[c]++;
    }

    for(auto it : m)
        vm.push_back(make_pair(it.first,it.second));

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

    for(auto& i : vm)
    {
        while(i.second > 0)
        {
            ans+= i.first;
            --i.second;
        }
    }

    return ans;
}
```

};

453. Minimum Moves to Equal Array Elements

he problem gets simplified if we sort the given array in order to obtain a sorted array aa. Now, similar to Approach 2,we use the difference diff=max-mindiff=max−min to update the elements of the array, but we need not traverse the whole array to find the maximum and minimum element every time, since if the array is sorted, we can make use of this property to find the maximum and minimum element after updation in O(1)O(1) time. Further, we need not actually update all the elements of the array. To understand how this works, we'll go in a stepwise manner.

Firstly, assume that we are updating the elements of the sorted array after every step of calculating the difference diffdiff. We'll see how to find the maximum and minimum element without traversing the array. In the first step, the last element is the largest element. Therefore, diff=a[n-1]-a[0]diff=a[n−1]−a[0]. We add diffdiff to all the elements except the last one i.e. a[n-1]a[n−1]. Now, the updated element at index 0 ,a'[0]a ′ [0] will be a[0]+diff=a[n-1]a[0]+diff=a[n−1]. Thus, the smallest element a'[0]a ′ [0] is now equal to the previous largest element a[n-1]a[n−1]. Since, the elements of the array are sorted, the elements upto index i-2i−2 satisfy the property a[j]>=a[j-1]a[j]>=a[j−1]. Thus, after updation, the element a'[n-2]a ′ [n−2] will become the largest element, which is obvious due to the sorted array property. Also, a[0] is still the smallest element.

Thus, for the second updation, we consider the difference diffdiff as diff=a[n-2]-a[0]diff=a[n−2]−a[0]. After updation, a''[0]a ′′ [0] will become equal to a'[n-2]a ′ [n−2] similar to the first iteration. Further, since a'[0]a ′ [0] and a'[n-1]a ′ [n−1] were equal. After the second updation, we get a''[0]=a''[n-1]=a'[n-2]a ′′ [0]=a ′′ [n−1]=a ′ [n−2]. Thus, now the largest element will be a[n-3]a[n−3]. Thus, we can continue in this fashion, and keep on incrementing the number of moves with the difference found at every step.

Now, let's come to step 2. In the first step, we assumed that we are updating the elements of the array aa at every step, but we need not do this. This is because, even after updating the elements the difference which we consider to add to the number of moves required remains the same because both the elements maxmax and minmin required to find the diffdiff get updated by the same amount everytime.

Thus, we can simply sort the given array once and use moves=\sum_{i=1}^{n-1} (a[i]-a[0])moves=∑ i=1 n−1
(a[i]−a[0]).

454. 4Sum II

Approach 1: Hashmap A brute force solution will be to enumerate all combinations of elements using four nested loops, which will result in \mathcal{O}(n^4)O(n 4) time complexity. A faster approach is to use three nested loops, and, for each sum a + b + c, search for a complementary value d == -(a + b + c) in the fourth array. We can do the search in \mathcal{O}(1)O(1) if we populate the fourth array into a hashmap.

Note that we need to keep track of the frequency of each element in the fourth array. If an element is repeated multiple times, it will form multiple quadruples. Therefore, we will use values in our hashmap to count each element.

Building further on this idea, we can observe that a + b == -(c + d). First, we will count sums of elements a + b from the first two arrays using a hashmap. Then, we will enumerate elements from the third and fourth arrays, and search for a complementary sum a + b == -(c + d) in the hashmap.

Current 8 / 8 Algorithm

For each a in A.

For each b in B. If a + b exists in the hashmap m, increment the value. Else add a new key a + b with the value 1. For each c in C.

For each d in D. Lookup key -(c + d) in the hashmap m. Add its value to the count cnt. Return the count cnt.

Complexity Analysis

Time Complexity: \mathcal{O}(n^2)O(n 2). We have 2 nested loops to count sums, and another 2 nested loops to find complements.

Space Complexity: \mathcal{O}(n^2)O(n 2) for the hashmap. There could be up to \mathcal{O}(n^2)O(n 2) distinct a + b keys.

457. Circular Array Loop

class Solution { // next function makes one move int next(vector& nums, int i){ int n = nums.size(); return (n+nums[i]+i)%n; } public: bool circularArrayLoop(vector& nums) { int n = nums.size(); // visited array for making sure we visit every element just once vector visited(n, 0); // steps of size greater than n circle back, so taking remainder for(int i=0; i<n; i++) nums[i]=nums[i]%n; for(int i=0;i<n;i++){ // initializing slow and fast int slow = i, fast = i; // already visited, no point running slow-fast algorithm again if(visited[slow]) continue; // condition inside ensures that all elements are positive or all negative // 2 negatives / 2 positives multilplied will give +ve as output // we need to ensure all elements slow, next(fast), next(next(fast)) are same sign // we don't need to check next(slow) because it is some previous fast (so already checked) while(nums[slow]*nums[next(nums,fast)]>0 &&
nums[slow]*nums[next(nums,next(nums,fast))]>0){ // one step for slow slow = next(nums,slow); // two steps for fast fast = next(nums,next(nums,fast)); // if already visited break if(visited[slow]) break; visited[slow]=1; if(slow==fast){ if(slow==next(nums,slow)) // single length return false; return true; // found a loop } } } return false; //no loop found } };

464. Can I Win

https://leetcode.com/problems/can-i-win/discuss/95320/Clean-C%2B%2B-beat-98.4-DFS-with-early-termination-check-(detailed-explanation) (https://leetcode.com/problems/can-i-win/discuss/95320/Clean-C%2B%2B-beat-98.4-DFS-with-early-termination-check-(detailed-explanation)) For short notation, let M = maxChoosableInteger and T = desiredTotal.

Key Observation: the state of the game is completely determined by currently available numbers to pick in the common pool.

State of Game: initially, we have all M numbers [1, M] available in the pool. Each number may or may not be picked at a state of the game later on, so we have maximum 2^M different states. Note that M <= 20, so int range is enough to cover it. For memorization, we define int k as the key for a game state, where

the i-th bit of k, i.e., k&(1<<i) represents the availability of number i+1 (1: picked; 0: not picked). At state k, the current player could pick any unpicked number from the pool, so state k can only go to one of the valid next states k':

if i-th bit of k is 0, set it to be 1, i.e., next state k' = k|(1<<i). Recursion: apparently

the current player can win at state k iff opponent can't win at some valid next state k'. Memorization: to speed up the recursion, we can use a vector m of size 2^M to memorize calculated results m[k] for state key k:

0 : not calculated yet; 1 : current player can win; -1: current player can't win. Initial State Check: There are several checks to be done at initial state k = 0 for early termination so we won't waste our time for DFS process:

if T < 2, obviously, the first player wins by simply picking 1. if the sum of entire pool S = M*(M+1)/2 is less than T, of course, nobody can reach T. if the sum S == T, the order to pick numbers from the pool is irrelevant. Whoever picks the last will reach T. So the first player can win iff M is odd. bool canIWin(int M, int T) { int sum = M*(M+1)/2; // sum of entire choosable pool

```
// I just pick 1 to win
if (T < 2) return true;

// Total is too large, nobody can win
else if (sum < T) return false;

// Total happens to match sum, whoever picks at odd times wins
else if (sum == T) return M%2;

// Non-trivial case: do DFS
// Initial total: T
// Initial game state: k = 0 (all numbers are not picked)
else return dfs(M, T, 0);
```

}

// DFS to check if I can win // k: current game state // T: remaining total to reach bool dfs(int M, int T, int k) { // memorized if (mem[k] != 0) return mem[k] > 0;

```
// total is already reached by opponent, so I lose
if (T <= 0) return false;

// try all currently available numbers
for (int i = 0; i < M; ++i)
  // if (i+1) is available to pick and my opponent can't win after I picked, I win!
  if (!(k&(1<<i)) && !dfs(M, T-i-1, k|(1<<i))) {
    mem[k] = 1;
    return true;
  }

// Otherwise, I will lose
mem[k] = -1;
return false;
```

}

// m[key]: memorized game result when pool state = key // 0: un-computed; 1: I win; -1: I lose int mem[1<<20] = {};

523. Continuous Subarray Sum

Approach #2 Better Brute Force [Accepted] Algorithm

We can optimize the brute force approach to some extent, if we make use of an array sumsum that stores the cumulative sum of the elements of the array, such that sum[i]sum[i] stores the sum of the elements upto the i^{th}i th element of the array.

Thus, now as before, we consider every possible subarray for checking its sum. But, instead of iterating over a new subarray everytime to determine its sum, we make use of the cumulative sum array. Thus, to determine the sum of elements from the i^{th}i th index to the j^{th}j th index, including both the corners, we can use: sum[j] - sum[i] + nums[i]sum[j]−sum[i]+nums[i].

525. Contiguous Array

class Solution { public: int findMaxLength(vector& arr) { int n=arr.size(),i,j,k; unordered_map<int,int>mp; int max_len=0; int cnt=0; mp[0]=-1; for(i=0;i<n;i++) { if(arr[i]>0) cnt++; else cnt--; if(mp.count(cnt)>0) { max_len=max(max_len,i-mp[cnt]); } else mp[cnt]=i; } return max_len; } }; /* Algorithm In this approach, we make use of a countcount variable, which is used to store the relative number of ones and zeros encountered so far while traversing the array. The countcount variable is incremented by one for every \text{1}1 encountered and the same is decremented by one for every \text{0}0 encountered.

We start traversing the array from the beginning. If at any moment, the countcount becomes zero, it implies that we've encountered equal number of zeros and ones from the beginning till the current index of the array(ii). Not only this, another point to be noted is that if we encounter the same countcount twice while traversing the array, it means that the number of zeros and ones are equal between the indices corresponding to the equal countcount values. The following figure illustrates the observation for the sequence [0 0 1 0 0 0 1 1]:

Contiguous_Array

In the above figure, the subarrays between (A,B), (B,C) and (A,C) (lying between indices corresponing to count = 2count=2) have equal number of zeros and ones.

Another point to be noted is that the largest subarray is the one between the points (A, C). Thus, if we keep a track of the indices corresponding to the same countcount values that lie farthest apart, we can determine the size of the largest subarray with equal no. of zeros and ones easily.

Now, the countcount values can range between \text{-n}-n to \text{+n}+n, with the extreme points corresponding to the complete array being filled with all 0's and all 1's respectively. Thus, we make use of an array arrarr(of size \text{2n+1}2n+1to keep a track of the various countcount's encountered so far. We make an entry containing the current element's index (ii) in the arrarr for a new countcount encountered everytime. Whenever, we come across the same countcount value later while traversing the array, we determine the length of the subarray lying between the indices corresponding to the same countcount values.

This approach relies on the same premise as the previous approach. But, we need not use an array of size \text{2n+1}2n+1, since it isn't necessary that we'll encounter all the countcount values possible. Thus, we make use of a HashMap mapmap to store the entries in the form of (index, count)(index,count). We make an entry for a countcount in the mapmap whenever the countcount is encountered first, and later on use the correspoding index to find the length of the largest subarray with equal no. of zeros and ones when the same countcount is encountered again.

map<count ,index>mp;

*/

529. Minesweeper

https://leetcode.com/problems/minesweeper/discuss/99885/C%2B%2B-16-lines-BFS (https://leetcode.com/problems/minesweeper/discuss/99885/C%2B%2B-16-lines-BFS)

542. 01 Matrix

Intuition

A better brute force: Looking over the entire matrix appears wasteful and hence, we can use Breadth First Search(BFS) to limit the search to the nearest 0 found for each 1. As soon as a 0 appears during the BFS, we know that the 0 is nearest, and hence, we move to the next 1.

Think again: But, in this approach, we will only be able to update the distance of one 1 using one BFS, which could in fact, result in slightly higher complexity than the Approach #1 brute force. But hey,this could be optimised if we start the BFS from 0s and thereby, updating the distances of all the 1s in the path.

Algorithm

For our BFS routine, we keep a queue, q to maintain the queue of cells to be examined next. We start by adding all the cells with 0s to q. Intially, distance for each 0 cell is 0 and distance for each 1 is INT_MAX, which is updated during the BFS. Pop the cell from queue, and examine its neighbours. If the new calculated distance for neighbour {i,j} is smaller, we add {i,j} to q and update dist[i][j].

544. Output Contest Matches

Approach #1: Simulation [Accepted] Intuition

Let team[i] be the correct team string of the i-th strongest team for that round. We will maintain these correctly as the rounds progress.

Algorithm

In each round, the i-th team becomes "(" + team[i] + "," + team[n-1-i] + ")", and then there are half as many teams.

545. Boundary of Binary Tree

kiya hua hai https://www.geeksforgeeks.org/boundary-traversal-of-binary-tree/ (https://www.geeksforgeeks.org/boundary-traversal-of-binary-tree/)

553. Optimal Division

// in order to get maximum sum the denominator should be made minimum.So to get denominator minimum we need to perform division serial wise from second int to end.Also add bracket to make sum maximum.

572. Subtree of Another Tree

https://www.geeksforgeeks.org/check-if-a-binary-tree-is-subtree-of-another-binary-tree/ (https://www.geeksforgeeks.org/check-if-a-binary-tree-is-subtree-of-another-binary-tree/)

587. Erect the Fence

https://www.youtube.com/watch?v=Vu84lmMzP2o (https://www.youtube.com/watch?v=Vu84lmMzP2o)

https://leetcode.com/problems/erect-the-fence/discuss/103317/C%2B%2B-O(nlogn)-Graham-Scan-Handling-the-Co-linear-Case (https://leetcode.com/problems/erect-the-fence/discuss/103317/C%2B%2B-O(nlogn)-Graham-Scan-Handling-the-Co-linear-Case)

588. Design In-Memory File System

https://leetcode.com/problems/design-in-memory-file-system/discuss/103329/C%2B%2B-Trie-Solution (https://leetcode.com/problems/design-in-memory-file-system/discuss/103329/C%2B%2B-Trie-Solution)

594. Longest Harmonious Subsequence

https://leetcode.com/problems/longest-harmonious-subsequence/discuss/103499/Three-C%2B%2B-Solution-run-time-with-explanation (https://leetcode.com/problems/longest-harmonious-subsequence/discuss/103499/Three-C%2B%2B-Solution-run-time-with-explanation) First loop through all elements and count each number appearance. Then loop through unordered_map, to find if the key - 1 is in the unordered map(To avoid counting twice, we only find if key - 1 in the map instead of finding key + 1 and key -1). If key - 1 and key both in the map, update the result int findLHS(vector& nums) { unordered_map<int,int>m; for(auto i: nums) m[i]++; int res = 0; for(auto it:m) if(m.count(it.first-1)>0) res = max(res, it.second+m[it.first-1]); return res; }

611. Valid Triangle Number

class Solution { public: int triangleNumber(vector& arr) { long int n=arr.size(),i,j,k,count=0; sort(arr.begin(),arr.end()); ***traingle sides arr[i] < arr[j] < arr[k]** *fixing arr[i] and arr[j] find all possible arr[k] (its easirs as data is sorted)* for(i=0;i<n-2;i++)// { k=i+2; for(j=i+1;j<n-1;j++) { //The important thing to note here is, we // use the previous value of k. If value of // arr[i] + arr[j-1] was greater than arr[k], // then arr[i] + arr[j] must be greater than k, // because the array is sorted. while(k<n&&arr[i]+arr[j]>arr[k]) { k++; }

```
            if(k>j)
                count+=((k-1)-(j+1)+1);// since k is incremented 1 extra time
        }
    }
    return count;

}
```

};

616. Add Bold Tag in String

1>>It means not found.

It is usually defined like so:

static const size_t npos = -1; It is better to compare to npos instead of -1 because the code is more legible.

```
2>>            it = temp.find(dict[i],it+1);// now it will search after start_indexof previos occurance if exit so it will no
t fall in infinte loop
```

617. Merge Two Binary Trees

https://www.youtube.com/watch?v=PbnNL7bbHe8 (https://www.youtube.com/watch?v=PbnNL7bbHe8)

636. Exclusive Time of Functions

https://leetcode.com/problems/exclusive-time-of-functions/discuss/105103/C%2B%2B-O(n)-stack-with-explaination (https://leetcode.com/problems/exclusive-time-of-functions/discuss/105103/C%2B%2B-O(n)-stack-with-explaination)

646. Maximum Length of Pair Chain

Intuition

If a chain of length k ends at some pairs[i], and pairs[i][1] < pairs[j][0], we can extend this chain to a chain of length k+1.

Algorithm

Sort the pairs by first coordinate, and let dp[i] be the length of the longest chain ending at pairs[i]. When i < j and pairs[i][1] < pairs[j][0], we can extend the chain, and so we have the candidate answer dp[j] = max(dp[j], dp[i] + 1).

659. Split Array into Consecutive Subsequences

1.count=0 =>> for each sequence initilay count++ ,when its len_of_sequence reaches 3 count-- ensure previosly created sequence is SATIESFILED . 2.read question carefully as saare number ko lete hue split karna hai subsequence me split karna hai eg:[1,2,3,4,4,5] return false becaus 1,2,3 is subsequence hai but 4,4,5 is not

/// The Algorithm

Maintain a set of consecutive sequences, call this set s. s begins as an empty set of consecutive sequences.

Now, iterate through each num in nums. For each iteration, if there exists a consecutive sequence in s that ends with element num-1, then append num to the end of the shortest such sequence; otherwise, create a new sequence that begins with num.

The problem has a solution (i.e. the array can be split into consecutive subsequences such that each subsequence consists of at least 3 consecutive integers) if and only if each sequence in s has size greater than or equal to 3.

Proof of Algorithm

Why does this algorithm work? It was intuitive to me, but I could not indisputably prove that it was correct. Hopefully, someone else can prove it.

Implementation

We don't need to actually store each sequence. Instead, we just need to know (1) the number of sequences that end at a particular element, and (2) the size of each of those sequences. To implement this, we can have an unordered map backs to represent the sequences: backs[key] returns a priority queue (smallest value at top) of the sizes of all sequences that end with element key. Now that we have (1) and (2), we can implement the algorithm above without knowing each particular sequence.

For each num in nums, if there exists any sequence that ends with num-1 (i.e. if backs[num-1] is a non-empty priority queue), then find such a sequence with the smallest possible size (get the smallest value from the priority queue at backs[num-1]). Now, the sequence will be extended by 1 since we will add num to it. So pop the smallest value count from the priority queue at backs[num-1], and add a new value count+1 to the priority queue at backs[num].

If no sequence was found that ends in num-1 (i.e. backs[num-1] is empty), then create a new sequence. In other words, add 1 to the priority queue at backs[num].

The Code

class Solution { public: bool isPossible(vector& nums) { unordered_map<int, priority_queue<int, vector, std::greater>> backs;

```
    // Keep track of the number of sequences with size < 3
    int need_more = 0;

    for (int num : nums)
    {
        if (! backs[num - 1].empty())
        {    // There exists a sequence that ends in num-1
            // Append 'num' to this sequence
            // Remove the existing sequence
            // Add a new sequence ending in 'num' with size incremented by 1
            int count = backs[num - 1].top();
            backs[num - 1].pop();
            backs[num].push(++count);

            if (count == 3)
                need_more--;
        }
        else
        {    // There is no sequence that ends in num-1
            // Create a new sequence with size 1 that ends with 'num'
            backs[num].push(1);
            need_more++;
        }
    }
    return need_more == 0;
}
```

};

662. Maximum Width of Binary Tree

https://www.geeksforgeeks.org/maximum-width-of-a-binary-tree/#:~:text=Width%20of%20a%20tree%20is,consider%20the%20below%20example%20tree.&text=width%20of%20level%204%20is,of%20the%20tree%20is%203 (https://www.geeksforgeeks.org/maximum-width-of-a-binary-tree/#:~:text=Width%20of%20a%20tree%20is,consider%20the%20below%20example%20tree.&text=width%20of%20level%204%20is,of%20the%20tree%20is%203).

667. Beautiful Arrangement II

https://leetcode.com/problems/beautiful-arrangement-ii/discuss/306337/Arrange-the-first-k-elements-easy-java-solution-with-explanation (https://leetcode.com/problems/beautiful-arrangement-ii/discuss/306337/Arrange-the-first-k-elements-easy-java-solution-with-explanation) We can arrange the first k elements that there difference is n-1, n-2, etc, and there are k-1 unique difference now, then we can make the k-th difference as 1. eg. n = 7, k = 4 (1) make the first k elements starting from 1 (beacuase only 1 and n can form 'n-1' difference) 1, 7, 2, 6 (2) now there are 3 unique difference (6,5,4), we then set all other difference as 1 1, 7, 2, 6, 5, 4, 3

class Solution { public int[] constructArray(int n, int k) { int[] res = new int[n]; boolean[] visited = new boolean[n + 1]; //ensure the first k number has k-1 unique difference int count = k; int diff = n - 1; res[0] = 1; visited[1] = true; for (int i = 1; i < k; i++) { if (res[i-1] + diff > 0 && res[i-1] + diff <= n && !visited[res[i-1]+diff]) res[i] = res[i-1] + diff; else res[i] = res[i-1] - diff; diff--; visited[res[i]] = true; } //for the rest numbers, let the difference as 1 for (int i = k; i < n; i++) { if (res[i-1] + 1 > n || visited[res[i-1] + 1]) res[i] = res[i-1] - 1; else res[i] = res[i-1] + 1; visited[res[i]] = true; } return res; } }

673. Number of Longest Increasing Subsequence

Suppose for sequences ending at nums[i], we knew the length length[i] of the longest sequence, and the number count[i] of such sequences with that length.

For every i < j with A[i] < A[j], we might append A[j] to a longest subsequence ending at A[i]. It means that we have demonstrated count[i] subsequences of length length[i] + 1.

Now, if those sequences are longer than length[j], then we know we have count[i] sequences of this length. If these sequences are equal in length to length[j], then we know that there are now count[i] additional sequences to be counted of that length (ie. count[j] += count[i]).

Complexity Analysis class Solution { public int findNumberOfLIS(int[] nums) { int N = nums.length; if (N <= 1) return N; int[] lengths = new int[N]; //lengths[i] = length of longest ending in nums[i] int[] counts = new int[N]; //count[i] = number of longest ending in nums[i] Arrays.fill(counts, 1);

```
    for (int j = 0; j < N; ++j) {
        for (int i = 0; i < j; ++i) if (nums[i] < nums[j]) {
            if (lengths[i] >= lengths[j]) {
                lengths[j] = lengths[i] + 1;
                counts[j] = counts[i];
            } else if (lengths[i] + 1 == lengths[j]) {
                counts[j] += counts[i];
            }
        }
    }

    int longest = 0, ans = 0;
    for (int length: lengths) {
        longest = Math.max(longest, length);
    }
    for (int i = 0; i < N; ++i) {
        if (lengths[i] == longest) {
            ans += counts[i];
        }
    }
    return ans;
}
```

}

675. Cut Off Trees for Golf Event

https://leetcode.com/problems/cut-off-trees-for-golf-event/discuss/107403/C%2B%2B-Sort-%2B-BFS-with-explanation (https://leetcode.com/problems/cut-off-trees-for-golf-event/discuss/107403/C%2B%2B-Sort-%2B-BFS-with-explanation)

688. Knight Probability in Chessboard

https://leetcode.com/problems/knight-probability-in-chessboard/discuss/108214/My-easy-understand-dp-solution (https://leetcode.com/problems/knight-probability-in-chessboard/discuss/108214/My-easy-understand-dp-solution)

690. Employee Importance

https://leetcode.com/problems/employee-importance/discuss/656964/using-map-bfs-solution-beats-97-time-100-space-cpp-solution-with-comments (https://leetcode.com/problems/employee-importance/discuss/656964/using-map-bfs-solution-beats-97-time-100-space-cpp-solution-with-comments) class Solution { public: int getImportance(vector<Employee*> employees, int id) { int n = employees.size(); if(n == 0){ return 0; } int importance = 0; unordered_map<int,int> u; for(int i = 0; i < n; i++){ // to keep track on which index of employee with id 'X' is in vector employees u[employees[i]->id] = i; } queue q; q.push(u[id]); while(q.size()){ int size = q.size(); while(size--){ int front = q.front(); q.pop(); importance += employees[front]->importance; for(int i = 0; i < employees[front]->subordinates.size(); i++){ //push the index of the subordinates for adding importance q.push(u[employees[front]->subordinates[i]]); } } } return importance; } };

692. Top K Frequent Words

https://www.fluentcpp.com/2018/03/20/heaps-and-priority-queues-in-c-part-3-queues-and-priority-queues/ (https://www.fluentcpp.com/2018/03/20/heaps-and-priority-queues-in-c-part-3-queues-and-priority-queues/)

https://leetcode.com/problems/top-k-frequent-words/discuss/628431/C%2B%2B-Simple-heap-solution (https://leetcode.com/problems/top-k-frequent-words/discuss/628431/C%2B%2B-Simple-heap-solution)

Sure. Firstly, the compare operator is used to make the priority_queue sort on the basis of frequency of the word from high to low, in case the frequency of the words is equal then we sort on the basis of the word from low to high(lexicographically).

Coming to the function, we first store the frequency of each word in a map mp. Then we store the values the map inside a vector of pair by taking the frequency of words as the first value of the pair and the word itself as the second value.

So now the priority_queue is sorted in such a way that the most occurred word is at the root and if two words have the same frequency, then the one with least lexicographic value is above the one with low. Now we have to get the top k frequent items, that means we will run a loop for k times and keep on taking the top value of the priority_queue and storing the word(second value of the returned top item, which is a pair) inside the ans vector and then popping out the top value.

698. Partition to K Equal Sum Subsets

Put n items into k bucket so each bucket has same total item value.

For each bucket, try all possible content O(k*2^n) -- no good. For each item, try all possible destined bucket O(n^k) -- doable.

712. Minimum ASCII Delete Sum for Two Strings

Let dp[i][j] be the answer to the problem for the strings s1[i:], s2[j:].

When one of the input strings is empty, the answer is the ASCII-sum of the other string. We can calculate this cumulatively using code like dp[i][s2.length()] = dp[i+1][s2.length()] + s1.codePointAt(i).

When s1[i] == s2[j], we have dp[i][j] = dp[i+1][j+1] as we can ignore these two characters.

When s1[i] != s2[j], we will have to delete at least one of them. We'll have dp[i][j] as the minimum of the answers after both deletion options.

The solutions presented will use bottom-up dynamic programming.

714. Best Time to Buy and Sell Stock with Transaction Fee

Approach #1: Dynamic Programming [Accepted] Intuition and Algorithm

At the end of the i-th day, we maintain cash, the maximum profit we could have if we did not have a share of stock, and hold, the maximum profit we could have if we owned a share of stock.

To transition from the i-th day to the i+1-th day, we either sell our stock cash = max(cash, hold + prices[i] - fee) or buy a stock hold = max(hold, cash - prices[i]). At the end, we want to return cash. We can transform cash first without using temporary variables because selling and buying on the same day can't be better than just continuing to hold the stock.

723. Candy Crush

// here i1,j1 goes 1 step ahead slly i0,j0 goes i step back than requered if(i1-i0>3||j1-j0>3) vp.push_back({i,j});

739. Daily Temperatures

https://leetcode.com/problems/daily-temperatures/discuss/736683/next-greater-element-question (https://leetcode.com/problems/daily-temperatures/discuss/736683/next-greater-element-question)

743. Network Delay Time

https://www.geeksforgeeks.org/bellman-ford-algorithm-simple-implementation/ (https://www.geeksforgeeks.org/bellman-ford-algorithm-simple-implementation/) class Solution { public: int networkDelayTime(vector<vector>& times, int N, int src) { // N=number of vertex int n=times.size(),i,j;// number of edges if(n==0) return -1; // srcource node form where signal broadcasted vectorcost(N+1,INT_MAX); cost[src]=0; for(i=1;i<=N-1;i++)// relaxing each edges N-1 times { for(j=0;j<n;j++)//all edges { int u=times[j][0],v=times[j][1],wt=times[j][2]; if(cost[u]!=INT_MAX&&cost[u]+wt<cost[v]) { cost[v]=cost[u]+wt; } } }

```
    long int res=0;
    for(i=1;i<=N;i++)// cost[0] is always INT_MAX as vertex are 1,2,....,N (0 not included)
        res=max(res,cost[i]);

    if(res!=INT_MAX)
        return res;
    return -1;
}
```

};

750. Number Of Corner Rectangles

pproach #1: Count Corners [Accepted] Intuition

We ask the question: for each additional row, how many more rectangles are added?

For each pair of 1s in the new row (say at new_row[i] and new_row[j]), we could create more rectangles where that pair forms the base. The number of new rectangles is the number of times some previous row had row[i] = row[j] = 1.

Algorithm

Let's maintain a count count[i, j], the number of times we saw row[i] = row[j] = 1. When we process a new row, for every pair new_row[i] = new_row[j] = 1, we add count[i, j] to the answer, then we increment count[i, j].

752. Open the Lock

queue input likes:

1. 0000|| $||1000||9000||0100||0900||0010||0090||0001||0009|| curr=0000; ......

   2. $||1000||9000||0100||0900||0010||0090||0001||0009|| curr=$;

   3 1000||9000||0100||0900||0010||0090||0001||0009|| $ curr=1000;

Intuition

We can think of this problem as a shortest path problem on a graph: there are 10000 nodes (strings '0000' to '9999'), and there is an edge between two nodes if they differ in one digit, that digit differs by 1 (wrapping around, so '0' and '9' differ by 1), and if both nodes are not in deadends.

Algorithm

To solve a shortest path problem, we use a breadth-first search. The basic structure uses a Queue queue plus a Set seen that records if a node has ever been enqueued. This pushes all the work to the neighbors function - in our Python implementation, all the code after while queue: is template code, except for if node in dead: continue.

As for the neighbors function, for each position in the lock i = 0, 1, 2, 3, for each of the turns d = -1, 1, we determine the value of the lock after the i-th wheel has been turned in the direction d.

Care should be taken in our algorithm, as the graph does not have an edge unless both nodes are not in deadends. If our neighbors function checks only the target for being in deadends, we also need to check whether '0000' is in deadends at the beginning. In our implementation, we check at the visitor level so as to neatly handle this problem in all cases.

In Java, our implementation also inlines the neighbors function for convenience, and uses null inputs in the queue to represent a layer change. When the layer changes, we depth++ our global counter, and queue.peek() != null checks if there are still nodes enqueued.

426. Convert Binary Search Tree to Sorted Doubly Linked List

https://www.youtube.com/watch?v=cgsGrbRPH6I (https://www.youtube.com/watch?v=cgsGrbRPH6I)

759. Employee Free Time

https://leetcode.com/problems/employee-free-time/discuss/119418/C%2B%2B-using-Set-(-EASY-to-understand-)-with-explanation (https://leetcode.com/problems/employee-free-time/discuss/119418/C%2B%2B-using-Set-(-EASY-to-understand-)-with-explanation)

761. Special Binary String

This solution is inspired by @lee215. Here is my key points on why it works (they can all be derived from the definition of special string):

Remove a special string prefix from a special string (if possible), the rest must a special string. (This implies that a special string can be split into one or more special substrings) A non-splitable special string must start with '1' and end with '0'. Remove the leading '1' and ending '0' from a non-splitable special string, the rest must be a special string. (this implies that we can recursively arrange on non-splitable special string to get largest string) These properties shouldn't be hard to prove. But please let me know if you see any of them is non-trivial.

So the algorithm will be straightforward:

Split given special string s into special substrings specials. Convert each special substring special recursively to largest lexicographical string. Sort all maximized special substrings specials in greater lexicographical order. Join all special substrings specials to form the final result res.

428. Serialize and Deserialize N-ary Tree

paata hai

https://www.geeksforgeeks.org/serialize-deserialize-n-ary-tree/ (https://www.geeksforgeeks.org/serialize-deserialize-n-ary-tree/)

767. Reorganize String

https://www.geeksforgeeks.org/rearrange-characters-string-no-two-adjacent/ (https://www.geeksforgeeks.org/rearrange-characters-string-no-two-adjacent/)

702. Search in a Sorted Array of Unknown Size

Algorithm

Define boundaries:

Initiate left = 0 and right = 1.

While target is on the right to the right boundary: reader.get(right) < target:

Set left boundary equal to the right one: left = right.

Extend right boundary: right *= 2. To speed up, use right shift instead of multiplication: right <<= 1.

Now the target is between left and right boundaries.

Binary Search:

While left <= right:

Pick a pivot index in the middle: pivot = (left + right) / 2. To avoid overflow, use the form pivot = left + ((right - left) >> 1) instead of straightforward expression above.

Retrieve the element at this index: num = reader.get(pivot).

Compare middle element num to the target value.

If the middle element is the target num == target: return pivot.

If the target is not yet found:

If num > target, continue to search on the left right = pivot - 1.

Else continue to search on the right left = pivot + 1.

We're here because target is not found. Return -1.

Implementation

773. Sliding Puzzle

https://leetcode.com/problems/sliding-puzzle/discuss/113613/C%2B%2B-9-lines-DFS-and-BFS (https://leetcode.com/problems/sliding-puzzle/discuss/113613/C%2B%2B-9-lines-DFS-and-BFS) It is easier for me to think about the board as a string (e.g. "123450" is the solution). I have a map that tells me positions I can move zero from the current one. I also keep track of the zero position, so I do not have to search for it every time.

unordered_map<int, vector> moves{{0,{1,3}},{1,{0,2,4}},{2,{1,5}},{3,{0,4}},{4,{3,5,1}},{5,{4,2}}};

DFS I am running DFS with memo to store the string and the smallest number of moves to achieve this permutation. Also, I am pruning all searches that go deeper that the best solution found so far.

787. Cheapest Flights Within K Stops

https://leetcode.com/problems/cheapest-flights-within-k-stops/discuss/128217/Three-C%2B%2B-solutions-BFS-DFS-and-BF (https://leetcode.com/problems/cheapest-flights-within-k-stops/discuss/128217/Three-C%2B%2B-solutions-BFS-DFS-and-BF)

int findCheapestPrice(int n, vector<vector>& flights, int s, int d, int K) { const int INF = 1e9; vector<vector> dp(K + 2, vector(n, INF)); dp[0][s] = 0;
for (int i = 1; i <= K + 1; ++i) { dp[i][s] = 0; for (const auto& x : flights) dp[i][x[1]] = min(dp[i][x[1]], dp[i-1][x[0]] + x[2]);
} return dp[K + 1][d] >= INF ? -1 : dp[K + 1][d];

```
    }
```

790. Domino and Tromino Tiling

C:\Users\Satish Kumar\Downloads\WhatsApp Image 2020-05-28 at 5.14.17 PM C:\Users\Satish Kumar\Downloads\WhatsApp Image 2020-05-28 at 5.14.16 PM

797. All Paths From Source to Target

https://www.geeksforgeeks.org/find-paths-given-source-destination/ (https://www.geeksforgeeks.org/find-paths-given-source-destination/)

801. Minimum Swaps To Make Sequences Increasing

@Acker help explain: https://leetcode.com/problems/minimum-swaps-to-make-sequences-increasing/discuss/119879/C%2B%2BJavaPython-DP-O(N)-Solution (https://leetcode.com/problems/minimum-swaps-to-make-sequences-increasing/discuss/119879/C%2B%2BJavaPython-DP-O(N)-Solution) Explanation swap[n] means the minimum swaps to make the A[i] and B[i] sequences increasing for 0 <= i <= n, in condition that we swap A[n] and B[n] not_swap[n] is the same with A[n] and B[n] not swapped.

A[i - 1] < A[i] && B[i - 1] < B[i]. In this case, if we want to keep A and B increasing before the index i, can only have two choices. -> 1.1 don't swap at (i-1) and don't swap at i, we can get not_swap[i] = not_swap[i-1] -> 1.2 swap at (i-1) and swap at i, we can get swap[i] = swap[i-1]+1 if swap at (i-1) and do not swap at i, we can not guarantee A and B increasing.

A[i-1] < B[i] && B[i-1] < A[i] In this case, if we want to keep A and B increasing before the index i, can only have two choices. -> 2.1 swap at (i-1) and do not swap at i, we can get notswap[i] = Math.min(swap[i-1], notswap[i]) -> 2.2 do not swap at (i-1) and swap at i, we can get swap[i]=Math.min(notswap[i-1]+1, swap[i])

813. Largest Sum of Averages

https://leetcode.com/problems/largest-sum-of-averages/discuss/237913/C%2B%2B-easily-understand-dp-method (https://leetcode.com/problems/largest-sum-of-averages/discuss/237913/C%2B%2B-easily-understand-dp-method)

827. Making A Large Island

Approach #2: Component Sizes [Accepted] Intuition

As in the previous solution, we check every 0. However, we also store the size of each group, so that we do not have to use depth-first search to repeatedly calculate the same size.

However, this idea fails when the 0 touches the same group. For example, consider grid = [[0,1],[1,1]]. The answer is 4, not 1 + 3 + 3, since the right neighbor and the bottom neighbor of the 0 belong to the same group.

We can remedy this problem by keeping track of a group id (or index), that is unique for each group. Then, we'll only add areas of neighboring groups with different ids.

Algorithm

For each group, fill it with value index and remember it's size as area[index] = dfs(...).

Then for each 0, look at the neighboring group ids seen and add the area of those groups, plus 1 for the 0 we are toggling. This gives us a candidate answer, and we take the maximum.

To solve the issue of having potentially no 0, we take the maximum of the previously calculated areas.

834. Sum of Distances in Tree

https://www.youtube.com/watch?v=Agsuftu_cwg (https://www.youtube.com/watch?v=Agsuftu_cwg)

837. New 21 Game

/** * dp[i] represents the total probability to get points i * dp[i] = dp[1] * 1/W + dp[2] * 1/W + dp[3] * 1/W + ... dp[i-2] * 1/W + dp[i-1] * 1/W * So dp[i] = (dp[1] + dp[2] + ... + dp[i - 1]) / W = Wsum / W * Conditional probability: keep a window with size K (assume K = 10), the probability of getting point i is the sum * of probability from point i - 10 to i, point i - 9 to i, ... , i -1 to i. Since every card has equal probability, * the probability to get any one of cards is 1/10. So the total probability of dp[i] is sum of all conditional * probability. * Once i is over than or equal to K, we can accumulate probability to final result * * （翻译）条件概率：dp[i]表示可以到达i分数的概率总和。假设我们的K为10的话，抽到每张牌的概率都是1/10。那么我们只需要从dp[i-10] * 开始加就可以了，所以就是维持一个size为K的window。比如12这个数，我们可以由抽到2的概率（dp[2]）乘以1/10或者抽到3的概率（dp[3]） * 乘以1/10得来...一直到dp[11] * 1/10，那么把他们相加就是可以到达point i的总概率了（就是dp[i]的值）。相当于是最基本的条件概率。 * 那么总概率就是 dp[12] = dp[2] * 1/10 + dp[3] * 1/10 + dp[4] * 1/10 + ... + dp[11] * 1/10. * 再详细剖析：因为从2到11这10个数都有可能通过只抽一张牌就到达12分。以此类推，只要是point没到K，之前的数都有可能到达当前的数，size超过K的时候， * 我们就维持一个size为K的window，再通过条件概率的公式累加和就可以了。 * * 现在我们已经有公式 dp[i] = dp[1] * 1/W + dp[2] * 1/W + dp[3] * 1/W + ... dp[i-2] * 1/W + dp[i-1] * 1/W， * 通过这个公式，我们可以换算成 (dp[1] + dp[2] + ... + dp[i - 1]) / W，这里的dp[1] + dp[2] + ... + dp[i - 1] 就是Wsum，所以dp[i] = Wsum / W。 * 换句话来说Wsum就是通过一次抽排就能到当前分数的概率之和。当然，我们这个example公式的Wsum是当i没有超过W时候的值， * 如果i超过了W，那就不是从dp[1]开始加了，而是从dp[i - W]开始加，相当于向右挪动window， 最多只能是从point i - W * 开始才能通过只抽一张牌就到达point i * */ public double new21Game(int N, int K, int W) { if (K == 0 || N >= K + W) { return 1; }

```
    double result = 0;
    double Wsum = 1;
    double dp[] = new double[N + 1];
    dp[0] = 1;

    for (int i = 1; i <= N; i++) {
        dp[i] = Wsum / W;
        // when points is less than K, all previous card could go to current i by only drawing one card
        if (i < K) {
            Wsum += dp[i];
        }
        // when points is equal to or more than K, all probability will be accumulated to results
        else {
            result += dp[i];
        }

        // when i is over than W, then we need to move the window
        if (i - W >= 0) {
            Wsum -= dp[i - W];
        }
    }
    return result;
}
```

838. Push Dominoes

Intuition

We can calculate the net force applied on every domino. The forces we care about are how close a domino is to a leftward 'R', and to a rightward 'L': the closer we are, the stronger the force.

Algorithm

Scanning from left to right, our force decays by 1 every iteration, and resets to N if we meet an 'R', so that force[i] is higher (than force[j]) if and only if dominoes[i] is closer (looking leftward) to 'R' (than dominoes[j]).

Similarly, scanning from right to left, we can find the force going rightward (closeness to 'L').

For some domino answer[i], if the forces are equal, then the answer is '.'. Otherwise, the answer is implied by whichever force is stronger.

Example

Here is a worked example on the string S = 'R.R...L': We find the force going from left to right is [7, 6, 7, 6, 5, 4, 0]. The force going from right to left is [0, 0, 0, -4, -5, -6, -7]. Combining them (taking their vector addition), the combined force is [7, 6, 7, 2, 0, -2, -7], for a final answer of RRRR.LL.

856. Score of Parentheses

When travel through S, only meets ')', we need to calculate the score of this pair of parenthese. If we know the score of inner parentheses, such as 3, we can double it and pass to the outer parenthese. But the question is how do we know the score of inner parentheses? Using stack.

explain with "(( ) (( )))"

stack: 0-> string_traveled :"" When start only 0 in stack, this int will store the total score

stack: 0->0->0-> string_traveled:"((" Meet two '(', push two zeros to the stack

stack: 0->1-> string_traveled: "(( )" First time meets ')', it balance the last '(', so pop the stack. But 0 indicates no inner parentheses exists, so just pass 1 to parent parenthese.

stack: 0->1->0->0-> string_traveled: "(( ) ((" Keep pushing zeros

stack 0->1->1-> string_traveled: "(( ) (( )" Balance one '(', and still no inner parenthese, so pass 1 to parent

stack 0->3-> string_traveled: "(( ) (( ))" Balance another '(', but the inner is not zero, so double it and add to parent's score

stack 6-> string_traveled: "(( ) (( )))" Same as last step, double the inner score and add to parent's

int scoreOfParentheses(string S) { stack m_stack; m_stack.push(0); // to keep the total score for(char c:S){ if(c=='(') m_stack.push(0); //When meets '(', just push a zero to stack else{ int tmp=m_stack.top(); // balance the last '(', it stored the score of inner parentheses m_stack.pop(); int val=0; if(tmp>0) // not zero means inner parentheses exists and double it val=tmp*2; else // zero means no inner parentheses, just using 1 val=1; m_stack.top()+=val; // pass the score of this level to parent parenthese }
} return m_stack.top(); }

861. Score After Flipping Matrix

https://leetcode.com/problems/score-after-flipping-matrix/discuss/143812/C%2B%2BJava-From-Intuition-Un-optimized-code-to-Optimized-Code-with-Detailed-Explanation (https://leetcode.com/problems/score-after-flipping-matrix/discuss/143812/C%2B%2BJava-From-Intuition-Un-optimized-code-to-Optimized-Code-with-Detailed-Explanation) First of all, thanks to @lee215 for providing an optimized code. My optimized code is an exact replica of his solution. But, I would like to offer more explanation to people who might find it difficult to understand. Hence, in this post, I start from the thinking process, implementing the thought-process to an un-optimized code and then improving it.

First of all, after looking at this question, the first thing which comes in mind is simulation. But wait, the question doesn't state any restrictions, because you can toggle as many rows and columns as you want and however times as you want. Then it means, its practically impossible to simulate all situations. So, what do we do next. Lets understand the structure of the binary number. A binary number, for example is something like

01010 Now, think for a moment, how can we maximize it? You guessed it right probably, by maximizing the number of 1's, which are as left as possible. Now, see below example,

0111 and 1000 The first number is 7 and the second is 8. So, overall we decreased the number of 1's, but the value has increased as we tried to get the 1 as left as possible. Now you got it? Exactly, if the first bit is zero, flip the binary. So, for our problem, we use this property (call it property 1).First thing which we will do is try to flip all rows, whose first column is zero. Great, now lets look at another optimization. For a particular column, it would be great if we could maximize the number of 1's. Why? Because, if you think carefully, all are at same bit position. Hence, if the number of1's increase, the total sum would increase. Hence, we use this second property(call it property 2). We scan through each column, and check if the number of 1's in that column are less than or equal to half the number of rows. If yes, then we flip the column to maximize the number of 1's in that column. However, be careful to do this for all columns except first, else you would break the first property. That's it. Now, I hope you would easily understand the below unoptimized code which is written in C++.

872. Leaf-Similar Trees

https://leetcode.com/problems/leaf-similar-trees/discuss/767694/C%2B%2B-DFS-100-Time-90-Space-Solution-Explained (https://leetcode.com/problems/leaf-similar-trees/discuss/767694/C%2B%2B-DFS-100-Time-90-Space-Solution-Explained)

873. Length of Longest Fibonacci Subsequence

public int lenLongestFibSubseq(int[] A) { int N = A.length; Map<Integer, Integer> index = new HashMap(); for (int i = 0; i < N; ++i) index.put(A[i], i); Intuition

Think of two consecutive terms A[i], A[j] in a fibonacci-like subsequence as a single node (i, j), and the entire subsequence is a path between these consecutive nodes. For example, with the fibonacci-like subsequence (A[1] = 2, A[2] = 3, A[4] = 5, A[7] = 8, A[10] = 13), we have the path between nodes (1, 2) <-> (2, 4) <-> (4, 7) <-> (7, 10).

The motivation for this is that two nodes (i, j) and (j, k) are connected if and only if A[i] + A[j] == A[k], and we needed this amount of information to know about this connection. Now we have a problem similar to Longest Increasing Subsequence.

Algorithm

Let longest[i, j] be the longest path ending in [i, j]. Then longest[j, k] = longest[i, j] + 1 if (i, j) and (j, k) are connected. Since i is uniquely determined as A.index(A[k] - A[j]), this is efficient: we check for each j < k what i is potentially, and update longest[j, k] accordingly. class Solution { public: int lenLongestFibSubseq(vector& A) { int N = A.size(); unordered_map<int, int> index; for (int i = 0; i < N; ++i) index[A[i]] = i;

```
    unordered_map<int, int> longest;
    int ans = 0;
    for (int k = 0; k < N; ++k)
        for (int j = 0; j < k; ++j) {
            if (A[k] - A[j] < A[j] && index.count(A[k] - A[j])) {
                int i = index[A[k] - A[j]];
                longest[j * N + k] = longest[i * N + j] + 1;
                ans = max(ans, longest[j * N + k] + 2);
            }
        }

    return ans >= 3 ? ans : 0;
}
```

};

////////////////////////////
Map<Integer, Integer> longest = new HashMap(); int [][] dp = new int[N][N]; int ans = 0;

```
for (int k = 0; k < N; ++k)
    for (int j = 0; j < k; ++j) {
        int i = index.getOrDefault(A[k] - A[j], -1);
        if (i >= 0 && i < j) {
            int cand = (dp[i][j] > 2 ? dp[i][j] : 2) + 1;
            dp[j][k] = cand;
            ans = Math.max(ans, cand);
        }
    }

return ans >= 3 ? ans : 0;
```

}

874. Walking Robot Simulation

We simulate the path of the robot step by step. Since there are at most 90000 steps, this is efficient enough to pass the given input limits.

Algorithm

We store the robot's position and direction. If we get a turning command, we update the direction; otherwise we walk the specified number of steps in the given direction.

Care must be made to use a Set data structure for the obstacles, so that we can check efficiently if our next step is obstructed. If we don't, our check is point in obstacles could be ~10,000 times slower.

In some languages, we need to encode the coordinates of each obstacle as a long integer so that it is a hashable key that we can put into a Set data structure. Alternatively, we could also encode the coordinates as a string

875. Koko Eating Bananas

Koko needs at least piles.length hours. (She cannot move onto a new pile even if she completes her current one within the hour). This is why we have H >= piles.length in the description.

Looking at the examples given, if H = piles.length, we get only one hour to finish each pile. In this case, the rate will depend only on the size of the largest pile. Say piles = [30,11,23,4,20] and H = 5, Koko needs to go at a rate >=30.

For any list of pile sizes, eating at rate K=max(piles) will ensure that each pile takes only one hour, and the total time taken will be piles.length which is <=H according to the description. This makes max(piles) a rate which is always able to finish.

Any answer that we report should fall in the closed interval [1, max(piles)]

The problem is basically asking us how much slower Koko can eat, and still finish the piles within H hours. To get the optimal answer, we should probably try to use as many of the hours available as possible, so the divisions below use H.

What is the slowest she could go?

There are B bananas across the piles, and Koko eats atmost K every hour. Therefore, in H hours, she can finish H*K bananas at the most. So we have H*K >= B or K >= ceil(B/H). If she goes any slower, she won't be able to finish.

So, we know that the answer is going to lie in the interval [ceil(B/H), max(piles)]. Binary search in this interval and find the least number which lets Koko finish. Verifying whether a particular rate works can be done like this:

piles = [piles[0] piles[1] . . . piles[n-1]] First pile takes ceil(piles[0] / rate), second takes ceil(piles[1]/rate) , etc. Add them all up and check if the total <= H (hours available). This is what possible does in the official solution.

If piles were sorted I think we could binary search during verification too and count the hours for all piles with # bananas <= K in one go, and then sum for the rest. But I don't think this is needed to solve the problem.

// editorial If Koko can finish eating all the bananas (within H hours) with an eating speed of K, she can finish with a larger speed too.

If we let possible(K) be true if and only if Koko can finish with an eating speed of K, then there is some X such that possible(K) = True if and only if K >= X.

For example, with piles = [3, 6, 7, 11] and H = 8, there is some X = 4 so that possible(1) = possible(2) = possible(3) = False, and possible(4) = possible(5) = ... = True.

Algorithm

We can binary search on the values of possible(K) to find the first X such that possible(X) is True: that will be our answer. Our loop invariant will be that possible(hi) is always True, and lo is always less than or equal to the answer. For more information on binary search, please visit [LeetCode Explore - Binary Search].

To find the value of possible(K), (ie. whether Koko with an eating speed of K can eat all bananas in H hours), we simulate it. For each pile of size p > 0, we can deduce that Koko finishes it in Math.ceil(p / K) = ((p-1) // K) + 1 hours, and we add these times across all piles and compare it to H.

885. Spiral Matrix III

https://www.geeksforgeeks.org/print-a-given-matrix-in-spiral-form/ (https://www.geeksforgeeks.org/print-a-given-matrix-in-spiral-form/)

898. Bitwise ORs of Subarrays

https://leetcode.com/problems/bitwise-ors-of-subarrays/discuss/165881/C%2B%2BJavaPython-O(30N) (https://leetcode.com/problems/bitwise-ors-of-subarrays/discuss/165881/C%2B%2BJavaPython-O(30N)) Assume B[i][j] = A[i] | A[i+1] | ... | A[j] Hash set cur stores all wise B[0][i], B[1][i], B[2][i], B[i][i].

When we handle the A[i+1], we want to update cur So we need operate bitwise OR on all elements in cur. Also we need to add A[i+1] to cur.

In each turn, we add all elements in cur to res.

Complexity: Time O(30N)

Normally this part is easy. But for this problem, time complexity matters a lot.

The solution is straight forward, while you may worry about the time complexity up to O(N^2) However, it's not the fact. This solution has only O(30N)

The reason is that, B[0][i] >= B[1][i] >= ... >= B[i][i]. B[0][i] covers all bits of B[1][i] B[1][i] covers all bits of B[2][i] ....

There are at most 30 bits for a positive number 0 <= A[i] <= 10^9. So there are at most 30 different values for B[0][i], B[1][i], B[2][i], ..., B[i][i]. Finally cur.size() <= 30 and res.size() <= 30 * A.length()

In a worst case, A = {1,2,4,8,16,..., 2 ^ 29} And all B[i][j] are different and res.size() == 30 * A.length()

C++:

```
int subarrayBitwiseORs(vector<int> A) {
    unordered_set<int> res, cur, cur2;
    for (int i: A) {
        cur2 = {i};
        for (int j: cur) cur2.insert(i|j);
        for (int j: cur = cur2) res.insert(j);
    }
    return res.size();
}
```

Java:

```
public int subarrayBitwiseORs(int[] A) {
    Set<Integer> res = new HashSet<>(), cur = new HashSet<>(), cur2;
    for (Integer i: A) {
        cur2 = new HashSet<>();
        cur2.add(i);
        for (Integer j: cur) cur2.add(i|j);
        res.addAll(cur = cur2);
    }
    return res.size();
          }
```

911. Online Election

https://leetcode.com/problems/online-election/discuss/173382/C%2B%2BJavaPython-Binary-Search-in-Times (https://leetcode.com/problems/online-election/discuss/173382/C%2B%2BJavaPython-Binary-Search-in-Times) Initialization part In the order of time, we count the number of votes for each person. Also, we update the current lead of votes for each time point. if (count[person] >= count[lead]) lead = person

Time Complexity: O(N)

Query part Binary search t in times, find out the latest time point no later than t. Return the lead of votes at that time point.

Time Complexity: O(logN)

916. Word Subsets

For each word b in B, we use function counter to count occurrence of each letter. We take the maximum occurrences of counts, and use it as a filter of A.

C++:

934. Shortest Bridge

https://www.youtube.com/watch?v=B64JXbZsTOE&feature=youtu.be (https://www.youtube.com/watch?v=B64JXbZsTOE&feature=youtu.be)

using flood fill

https://leetcode.com/problems/shortest-bridge/discuss/189293/C%2B%2B-BFS-Island-Expansion-%2B-UF-Bonus (https://leetcode.com/problems/shortest-bridge/discuss/189293/C%2B%2B-BFS-Island-Expansion-%2B-UF-Bonus) iske niche commnet me hai

935. Knight Dialer

// similar type just change in DP table wushangzhen 371 Last Edit: September 20, 2019 1:56 AM

Read More Solution's code is hard to understand O(N) space solution

By hand or otherwise, have a way to query what moves are available at each square. This implies the exact recursion for f. For example, from 1 we can move to 6, 8, so f(1, n) = f(6, n-1) + f(8, n-1).

After, let's keep track of dp[start] = f(start, n), and update it for each n from 1, 2, ..., N.

At the end, the answer is f(0, N) + f(1, N) + ... + f(9, N) = sum(dp).

937. Reorder Data in Log Files

revised

938. Range Sum of BST

https://www.youtube.com/watch?v=SIdrJwWp3H0 (https://www.youtube.com/watch?v=SIdrJwWp3H0)

939. Minimum Area Rectangle

Approach 2: Count by Diagonal Intuition

For each pair of points in the array, consider them to be the long diagonal of a potential rectangle. We can check if all 4 points are there using a Set.

For example, if the points are (1, 1) and (5, 5), we check if we also have (1, 5) and (5, 1). If we do, we have a candidate rectangle.

Algorithm

Put all the points in a set. For each pair of points, if the associated rectangle are 4 distinct points all in the set, then take the area of this rectangle as a candidate answer.

947. Most Stones Removed with Same Row or Column

Input: stones = [[0,0],[0,1],[1,0],[1,2],[2,1],[2,2]] Output: 5

0-[1,2] 1-[4] 2-[3] 3-[5] 4-[5]

vis[x] ussi time mark kar do jab identify hua to inseret duplicate

963. Minimum Area Rectangle II

https://leetcode.com/problems/minimum-area-rectangle-ii/solution/ (https://leetcode.com/problems/minimum-area-rectangle-ii/solution/) Say the first 3 points are p1, p2, p3, and that p2 and p3 are opposite corners of the final rectangle. The 4th point must be p4 = p2 + p3 - p1 (using vector notation) because p1, p2, p4, p3 must form a parallelogram, and p1 + (p2 - p1) + (p3 - p1) = p4.

If this point exists in our collection (we can use a HashSet to check), then we should check that the angles of this parallelogram are 90 degrees. The easiest way is to check the dot product of the two vectors (p2 - p1) and (p3 - p1). (Another way is we could normalize the vectors to length 1, and check that one equals the other rotated by 90 degrees.)

967. Numbers With Same Consecutive Differences

We initial the current result with all 1-digit numbers, like cur = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].

Each turn, for each x in cur, we get its last digit y = x % 10. If y + K < 10, we add x * 10 + y + K to the new list. If y - K >= 0, we add x * 10 + y - K to the new list.

We repeat this step N - 1 times and return the final result.

vector numsSameConsecDiff(int N, int K) { vector cur = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}; for (int i = 2; i <= N; ++i) { vector cur2; for (int x : cur) { int y = x % 10; if (x > 0 && y + K < 10) cur2.push_back(x * 10 + y + K); if (x > 0 && K > 0 && y - K >= 0) cur2.push_back(x * 10 + y - K); } cur = cur2; } return cur;

974. Subarray Sums Divisible by K

https://leetcode.com/problems/subarray-sums-divisible-by-k/discuss/413234/DETAILED-WHITEBOARD!-BEATS-100-(Do-you-really-want-to-understand-It) (https://leetcode.com/problems/subarray-sums-divisible-by-k/discuss/413234/DETAILED-WHITEBOARD!-BEATS-100-(Do-you-really-want-to-understand-It))

979. Distribute Coins in Binary Tree

https://www.geeksforgeeks.org/distribute-candies-in-a-binary-tree/draw (https://www.geeksforgeeks.org/distribute-candies-in-a-binary-tree/draw) recursive stack The idea is that, if the son have more than one coin, give extra coins to parent, or if the son have less than one coin, borrow some coins from parent. The total amount of transferred coins between each parent and son is the answer. The key is that we always let sons have only one coin. Specifically, for current node, we will firstly let its sons to have only one coin. Then the current node may have more than one coin, or less than one coin (because its sons transfer or borrow money to or from it). Then let current node transfer or borrow some coins to or from its parent.

Why the total amount of transferred coins is the answer?

Let's see it from bottom to up. For every subtree, the number of coins must be equal to the number of nodes in the subtree, so for the root of the subtree, it can only and must obtain coins or give extra coins from or to its parent. If we sum up all the transferred coins between each parent and son, it should be the minimum cost of move. The dfs function actually does is to make current subtree have correct number of coins, then let the parent of the root to handle lacked coins or extra coins.

Some one may feel confused that the node can obtain some money from its brothers. It is true, but this process could be divided into two parts: 1. the brother give extra money to the parent. 2. the node borrow money from the parent. So its basically the same thing.

void dfs(TreeNode* root, TreeNode* parent, int& ans) { if(root -> left != NULL) dfs(root -> left, root, ans); if(root -> right != NULL) dfs(root -> right, root, ans); if(root -> val != 1) { int move = root -> val - 1; ans += abs(move); if(parent != NULL) parent -> val += move; } return;

```
}
int distributeCoins(TreeNode* root) {
    int ans = 0;
    dfs(root, NULL, ans);
    return ans;
}
```

981. Time Based Key-Value Store

// minus for sorted in ascending order

Here is a clear c++ solution for this problem, and this solution only use map and unorder_map data structure in STL.

First, we use a unordered_map to save Key and 'Value'(not truely value). And then, we use a map to save Time and turely value in map's value, so that we can easy use the lower_bound api in map to finish search in time.

class TimeMap { public:

```
TimeMap(){

}

void set(string key, string value, int timestamp) {
    data[key][timestamp] = value;
}

string get(string key, int timestamp) {
    auto it1 = data.find(key);
    if (it1 != data.end()) {
        auto it2 = it1->second.lower_bound(timestamp);
        if (it2 != it1->second.end()) {
            return it2->second;
        }
    }
    return "";
}

unordered_map<string, map<int, string, greater<int> >> data;
```

};

991. Broken Calculator

Approach 1: Work Backwards Intuition

Instead of multiplying by 2 or subtracting 1 from X, we could divide by 2 (when Y is even) or add 1 to Y.

The motivation for this is that it turns out we always greedily divide by 2:

If say Y is even, then if we perform 2 additions and one division, we could instead perform one division and one addition for less operations [(Y+2) / 2 vs Y/2 + 1].

If say Y is odd, then if we perform 3 additions and one division, we could instead perform 1 addition, 1 division, and 1 addition for less operations [(Y+3) / 2 vs (Y+1) / 2 + 1].

Algorithm

While Y is larger than X, add 1 if it is odd, else divide by 2. After, we need to do X - Y additions to reach X.

1007. Minimum Domino Rotations For Equal Row

1.st find all potentail candidate from 1 to 6 2.for each potentail candidate run a loop through all elelment in both array A[],B[] 3.dp[i][0]= minimum number of rotations so that all the values in A till ith index are the same(equal to x); 4.dp[i][1] = minimum number of rotations so that all the values in B till ith index are the same(equal to x); 5.finally keep updating ans for each poteitial candidate

1008. Construct Binary Search Tree from Preorder Traversal

https://www.interviewbit.com/problems/construct-binary-tree-from-inorder-and-preorder/ (https://www.interviewbit.com/problems/construct-binary-tree-from-inorder-and-preorder/)

1011. Capacity To Ship Packages Within D Days

Intuition The largest capacity (r) we may even need is the sum of weights of all packages. The smallest capacity (l) is the weight of the largest package. Optimization: the smallest capacity cannot be less than r / D, which reduces the search interval if we have a lot of small packages (and D is small). Our result is within this interval. Linearithmic Solution We use binary search to find the minimum capacity. For each capacity we analyze, we count the number of days required to ship all packages.

We decrease the capacity if it takes less days than D, and increase otherwise. Note that, when the number of days equals D, this algorithm keeps decreasing the capacity while it can, therefore finding the smallest capacity required.

1062. Longest Repeating Substring

https://leetcode.com/problems/longest-repeating-substring/discuss/473116/C%2B%2B-DP (https://leetcode.com/problems/longest-repeating-substring/discuss/473116/C%2B%2B-DP)

1020. Number of Enclaves

We flood-fill the land (change 1 to 0) from the boundary of the grid. Then, we count the remaining land.

https://leetcode.com/problems/number-of-enclaves/discuss/265555/C%2B%2B-with-picture-DFS-and-BFS (https://leetcode.com/problems/number-of-enclaves/discuss/265555/C%2B%2B-with-picture-DFS-and-BFS)

1021. Remove Outermost Parentheses

https://www.youtube.com/watch?v=yZRipR0CFwU&list=PLZS6qsEL17o5gusw6aXPS75e-Iwrj1Rz2&index=13&t=0s (https://www.youtube.com/watch?v=yZRipR0CFwU&list=PLZS6qsEL17o5gusw6aXPS75e-Iwrj1Rz2&index=13&t=0s)

1024. Video Stitching

https://leetcode.com/problems/video-stitching/discuss/821273/DP-Explained-or-beats-92-or-C%2B%2B (https://leetcode.com/problems/video-stitching/discuss/821273/DP-Explained-or-beats-92-or-C%2B%2B)

https://leetcode.com/problems/video-stitching/discuss/270036/JavaC%2B%2BPython-Greedy-Solution-O(1)-Space (https://leetcode.com/problems/video-stitching/discuss/270036/JavaC%2B%2BPython-Greedy-Solution-O(1)-Space)

[[8,10],[17,39],[18,19],[8,16],[13,35],[33,39],[11,19],[18,35]] 20

1026. Maximum Difference Between Node and Ancestor

1029. Two City Scheduling

Intuition

Let's figure out how to sort the input here. The input should be sorted by a parameter which indicates a money lost for the company.

The company would pay anyway : price_A to send a person to the city A, or price_B to send a person to the city B. By sending the person to the city A, the company would lose price_A - price_B, which could negative or positive.

bla

To optimize the total costs, let's sort the persons by price_A - price_B and then send the first n persons to the city A, and the others to the city B, because this way the company costs are minimal.

Algorithm

Now the algorithm is straightforward :

Sort the persons in the ascending order by price_A - price_B parameter, which indicates the company additional costs.

To minimise the costs, send n persons with the smallest price_A - price_B to the city A, and the others to the city B.

1031. Maximum Sum of Two Non-Overlapping Subarrays

1>> class Solution { public: int maxSumTwoNoOverlap(vector& A, int L, int M) { vector prefixSums; int sum = 0; prefixSums.push_back(sum); for (auto i : A) { sum += i; prefixSums.push_back(sum); }

```
    int result = -1;
    for (int i = L; i < prefixSums.size(); i++) {
        // subarray sum of len L:
        auto lsum = prefixSums[i] - prefixSums[i-L];
        // for this prefix array of [i,i+L-1], find the prefix array of size
        // [j,j+M-1] where j is before and after i without overlap

        // 1. after:
        int msum = -1;
        for (int j = i + M; j < prefixSums.size(); j++) {
            // add subarray sum of len M to sum:
            msum = max(msum, prefixSums[j] - prefixSums[j-M]);
        }

        // 2. before:
        for (int j = M; j < i - L + 1; j++) {
            // add subarray sum of len M to sum:
            msum = max(msum, prefixSums[j] - prefixSums[j-M]);
        }
        result = max(result, lsum + msum);
    }

    return result;

}
```

};

2>> The solution is amazing. It took me a while to understand its correctness. Basically it can be broken it into 2 cases: L is always before M vs M is always before L. L is always before M, we maintain a Lmax to keep track of the max sum of L subarray, and sliding the window of M from left to right to cover all the M subarray. The same for the case where M is before L. Rewrite it and add some comments, hope it helps someone who is confused at first.

class Solution { public int maxSumTwoNoOverlap(int[] A, int L, int M) { if (A == null || A.length == 0) { return 0; } int n = A.length; int[] preSum = new int[n + 1]; for (int i = 0; i < n; i++) { preSum[i + 1] = A[i] + preSum[i]; } int lMax = preSum[L], mMax = preSum[M]; int res = preSum[L + M]; for (int i = L + M; i <= n; i++) { //case 1: L subarray is always before M subarray lMax = Math.max(lMax, preSum[i - M] - preSum[i - M - L]); //case 2: M subarray is always before L subarray mMax = Math.max(mMax, preSum[i - L] - preSum[i - M - L]); //compare two cases and update res res = Math.max(res, Math.max(lMax + preSum[i] - preSum[i - M], mMax + preSum[i] - preSum[i - L])); } return res; } }

1039. Minimum Score Triangulation of Polygon

https://leetcode.com/problems/minimum-score-triangulation-of-polygon/discuss/286753/C%2B%2B-with-picture (https://leetcode.com/problems/minimum-score-triangulation-of-polygon/discuss/286753/C%2B%2B-with-picture)

https://leetcode.com/problems/minimum-score-triangulation-of-polygon/discuss/641785/C%2B%2B-or-Matrix-chain-multiplication-variant-or-DP (https://leetcode.com/problems/minimum-score-triangulation-of-polygon/discuss/641785/C%2B%2B-or-Matrix-chain-multiplication-variant-or-DP)

Actually this problem looks quite troublesome at first, after we dive in, we will find out that this problem is actually a deformed version of Matrix Chain Multiplication. For example,

[1,2,3,4] can be viewed as 3 matrix multiplication (1, 2), (2, 3), (3, 4). what is the minimum operations to multiply all these three matrices.

1162. As Far from Land as Possible

https://leetcode.com/problems/as-far-from-land-as-possible/discuss/360963/C%2B%2B-with-picture-DFS-and-BFS (https://leetcode.com/problems/as-far-from-land-as-possible/discuss/360963/C%2B%2B-with-picture-DFS-and-BFS)

Intution Normally, we would run breadth-first search from each cell simultaneonly, tracking water cells we visited. Sort of like Dijkastra's algorithm. However, I wanted to try a depth-first search solution, as it seemed easier to implement at the time.

The DFS solution is accepted but has higher runtime complexity, so I then added the BFS solution to compare.

DFS Solution For each 'land' cell, start DFS and record the distance in 'water' cells.

If the distance in the 'water' cell is smaller than the current distance, we stop there. Otherwise, we update the distance to the smaller value and keep going.

So our grid will have the following values:

1 for land 0 for water

> =2 water with the recorded distance In the end, we scan the grid again and returning the largest value.

In this example, the cells contains distances after DFS is complete for each land cell. In the end, the maximum distance from the land is 3 (4 - 1). image

void dfs(vector<vector>& g, int i, int j, int dist = 1) { if (i < 0 || j < 0 || i >= g.size() || j >= g[i].size() || (g[i][j] != 0 && g[i][j] <= dist)) return; g[i][j] = dist; dfs(g, i - 1, j, dist + 1), dfs(g, i + 1, j, dist + 1), dfs(g, i, j - 1, dist + 1), dfs(g, i, j + 1, dist + 1); } int maxDistance(vector<vector>& g, int mx = -1) { for (auto i = 0; i < g.size(); ++i) for (auto j = 0; j < g[i].size(); ++j) if (g[i][j] == 1) { g[i][j] = 0; dfs(g, i, j); } for (auto i = 0; i < g.size(); ++i) for (auto j = 0; j < g[i].size(); ++j) if (g[i][j] > 1) mx = max(mx, g[i][j] - 1); return mx; } Complexity Analysis Runtime: O(m * n * n), where m is the number of land cells. Memory: O(n * n) for the recursion.

BFS Solution The solution above is slow, and BFS can help to make sure we process a single cell only once (well, twice in our case to scan for the land first).

Here, we find our land cells and put surrounding water cells in the queue. We mark water cells as visited and continue the expansion from land cells until there are no more water cells left. In the end, the number of steps in DFS is how far can we go from the land.

Using the same example as above, the picture below shows the state of the grid after each step of BFS

1041. Robot Bounded In Circle

https://leetcode.com/problems/robot-bounded-in-circle/discuss/290856/JavaC%2B%2BPython-Let-Chopper-Help-Explain (https://leetcode.com/problems/robot-bounded-in-circle/discuss/290856/JavaC%2B%2BPython-Let-Chopper-Help-Explain)

1048. Longest String Chain

217 lee215's avatar lee215 49101 Last Edit: February 20, 2020 1:59 PM lee215 solution 22.1K VIEWS

Explanation Sort the words by word's length. (also can apply bucket sort) For each word, loop on all possible previous word with 1 letter missing. If we have seen this previous word, update the longest chain for the current word. Finally return the longest word chain.

Complexity Time O(NlogN) for sorting, Time O(NSS) for the for loop, where the second S refers to the string generation and S <= 16. Space O(NS)

1182. Shortest Distance to Target Color

Define 2 DP matrix, left and right. The DP definition is left[color][index] represents the minimum distance of the color color to the left of the index index. Similarly, right[color][index] represents the minimum distance of the color color to the right of the index index. Note that we also include the current element in the DP definition.

For each of the 3 colours, we can fill both the dp matrix in linear time. Let us fix any color, say 1, and fix a dp matrix, say left. Now, the minimum distance of red from the i-th index to the left will be zero if the i-th index is of red color. If the i-th color is not zero, we can look at the minimum distance of the i-1 th index. If the index is -1, it means that no red exists in left half. Hence, the current distance would be -1. If not, the current distance would be oldDistance + 1. A similar argument can be made for right.

At the end, for each index, we can find the minimum distance to the left and right and return the minimum of both of them.

Getting the Result from the pre-computed values For any query, we are given the value of the color and the index. We find out the minimum distance of that color in the left as well as right part. If either of them is -1, the answer is the other one. If none of th

1197. Minimum Knight Moves

https://leetcode.com/problems/minimum-knight-moves/discuss/501701/Clean-and-Concise-C%2B%2B-Solution-BFS (https://leetcode.com/problems/minimum-knight-moves/discuss/501701/Clean-and-Concise-C%2B%2B-Solution-BFS)

1074. Number of Submatrices That Sum to Target

class Solution { public: int numSubmatrixSumTarget(vector<vector>&arr, int t) { int m=arr.size(),i,j,k; int n=arr[0].size(),res=0; for(i=0;i<m;i++) { for(j=1;j<n;j++) { arr[i][j]=arr[i][j]+arr[i][j-1]; } }

```
    for(i=0;i<n;i++)
    {
        for(j=i;j<n;j++)
        {
            unordered_map<int,int>mp;
            mp[0]=1;
            int curr=0;
            for(k=0;k<m;k++)
            {
               if(i!=0)
               curr+=(arr[k][j]-arr[k][i-1]);// for each k prefix sum=arr[k][i]+arr[k][i+1]+......arr[k][j]
                else
                    curr+=(arr[k][j]-0);
                // mp.count(curr-t) ka explanation
                /*
                 given t
                  let curr1=x then mp[x]=1(say freq)
                   and curr2=y then mp[y]=1(say freq)
                   if(mp.count(y-t)>0) means
                   y-t=x so t=y-x
                   means another submatix possible
                      i     j
                    0 1 2 3 4 5
       k         0  . . . . . .
                 1  . . . . . .
                 2  . . . . . .
                 3  . . . . . .
                 4  . . . . . .

                 A[0][1]+A[0][2]+.....A[0][4]=x

                 A[0][1]+A[0][2]+.....A[0][4]+
                 A[1][1]+A[1][2]+.....A[1][4]+
                 A[2][1]+A[2][2]+.....A[2][4]= y

              y-x=A[1][1]+A[1][2]+.....A[1][4]+
                  A[2][1]+A[2][2]+.....A[2][4]

                  and t=y-x so done

                */
                if(mp.count(curr-t))
                    res+=mp[curr-t];
                mp[curr]++;// ek submatrix hai jiska sum=curr mila hai (future me kaam aayega)

            }
        }
    }
    return res;
}
```

};

/////////////////////////////

https://leetcode.com/problems/number-of-submatrices-that-sum-to-target/discuss/303750/JavaC%2B%2BPython-Find-the-Subarray-with-Target-Sum (https://leetcode.com/problems/number-of-submatrices-that-sum-to-target/discuss/303750/JavaC%2B%2BPython-Find-the-Subarray-with-Target-Sum) For those who do not understand the logic behind this code at first glance as I, here is my explanation to help you understand:

1. calculate prefix sum for each row for (int i = 0; i < m; i++) for (int j = 1; j < n; j++)

   ```
   A[i][j] += A[i][j - 1];
   ```

   For this double-for loop, we are calculating the prefix sum for each row in the matrix, which will be used later

2. for every possible range between two columns, accumulate the prefix sum of submatrices that can be formed between these two columns by adding up the sum of values between these two columns for every row for (int i = 0; i < n; i++) { for (int j = i; j < n; j++) {

```
Map<Integer, Integer> counter = new HashMap<>();
counter.put(0, 1);
int cur = 0;
for (int k = 0; k < m; k++) {
    cur += A[k][j] - (i > 0 ? A[k][i - 1] : 0);
    res += counter.getOrDefault(cur - target, 0);
    counter.put(cur, counter.getOrDefault(cur, 0) + 1);
}
```

} } To understand what this triple-for loop does, let us try an example, assume i = 1 and j = 3, then for this part of code:

Map<Integer, Integer> counter = new HashMap<>(); counter.put(0, 1); int cur = 0; for (int k = 0; k < m; k++) { cur += A[k][j] - (i > 0 ? A[k][i - 1] : 0); res += counter.getOrDefault(cur - target, 0); counter.put(cur, counter.getOrDefault(cur, 0) + 1); } I will break this piece of code into two major part:

Map<Integer, Integer> counter = new HashMap<>(); counter.put(0, 1); key of this hashmap present the unique value of all possible prefix sum that we've seen so far value of this hashmap represents the count (number of appearances) of each prefix sum value we've seen so far an empty submatrix trivially has a sum of 0 for (int k = 0; k < m; k++) { cur += A[k][j] - (i > 0 ? A[k][i - 1] : 0); res += counter.getOrDefault(cur - target, 0); counter.put(cur, counter.getOrDefault(cur, 0) + 1); } Here we are actually calculating the prefix sum of submatrices which has column 1, 2, and 3, by adding up the sum of matrix[0][1...3], matrix[1][1...3] ... matrix[m-1][1...3] row by row, starting from the first row (row 0). The way of getting the number of submatrices whose sum equals to K uses the same idea of 560. Subarray Sum Equals K so I won't repeat it again.

Hope it helps

P.S. A python 3 implementation of Lee's idea

class Solution: def numSubmatrixSumTarget(self, matrix: List[List[int]], target: int) -> int: if not matrix or not matrix[0]: return 0

```
    presum = list(map(lambda row: list(itertools.accumulate(row)), matrix))

    rows, cols = len(matrix), len(matrix[0])

    count = 0
    for col_start in range(cols):
        for col_end in range(col_start, cols):
            counter = collections.Counter()
            counter[0] = 1
            curr_sum = 0
            for r in range(rows):
                curr_sum += presum[r][col_end] - (presum[r][col_start - 1] if col_start else 0)
                count += counter[curr_sum - target]
                counter[curr_sum] += 1

    return count
```

1072. Flip Columns For Maximum Number of Equal Rows

https://leetcode.com/problems/flip-columns-for-maximum-number-of-equal-rows/discuss/304494/C%2B%2B-easy-way-to-solve-it-with-explanation (https://leetcode.com/problems/flip-columns-for-maximum-number-of-equal-rows/discuss/304494/C%2B%2B-easy-way-to-solve-it-with-explanation) intuition since we don't have a care how many times we need to flip, so we can observe the behavior of the question.

like : [0,0,0,1] [0,0,1,1] [1,1,0,0] [1,0,1,0] in this case, answer is two, since [0011] and [1100] can be the same pattern after some flips. So, what we need to do is to count each row pattern.

return max(number of patternA + flip(number of patternA)) class Solution { public: int maxEqualRowsAfterFlips(vector<vector>& matrix) { map<vector,int> row_patten; for(vector v : matrix) row_patten[v]++; int res = 0; for(auto it : row_patten) res = max(res,row_patten[it.first]+row_patten[flip_func(it.first)]); return res; } /* this function is used to flip a ROW */ vector flip_func(vector v){ vector res(v.size(),0); for(int i = 0 ; i < v.size() ; i++) res[i] = !v[i]; return res; } };

1081. Smallest Subsequence of Distinct Characters

Problem is to remove duplicates and output a string with unique characters while maintaing the smallest(lexicographically) permutation of all possible subsequences of distinct characters.

We have to just keep adding the unique characters to the answer and while adding them we have to check if the character to be added is smaller than the character which we have added previuosly to the answer. If it is smaller, then we can check whether the previous character is available for use in future iterations ? If yes, then we can just remove the character we added previously. Repeat this step, until we find a previous character smaller than current character. We can now add the current character to the answer. class Solution { public: string smallestSubsequence(string text) { int last[26] = {}; int seen[26] = {}; int n = (int)text.size(); string ans=""; for(int i=0;i<n;i++) last[text[i]-'a'] = i; // In order to know the last occurence of a character(step 2 above).At any point if we want to know that a given character is available in the future iterations.

```
      for(int i=0;i<n;i++)
      {
          if(seen[text[i]-'a'])
              continue;
          seen[text[i]-'a']++; // To avoid adding duplicates
          while(!ans.empty() and ans.back()>text[i] and i<last[ans.back()-'a']) // As explained in step 2 above.
          {
              seen[ans.back()-'a'] = 0; // We have to reset this character in seen to 0 because when it comes again in futur
e, it has to be added to the answer again.
              ans.pop_back();
          }

          ans.push_back(text[i]);
      }
      return ans;
  }
```

};

1230. Toss Strange Coins

https://leetcode.com/problems/toss-strange-coins/discuss/408485/JavaC%2B%2BPython-DP (https://leetcode.com/problems/toss-strange-coins/discuss/408485/JavaC%2B%2BPython-DP) i thought of DP as well, but i used a 2D array. Love your sln. For anyone who is interested in 2D array DP.. LOL.

class Solution { public: double probabilityOfHeads(vector& prob, int target) { //F[i,j] == up until the ith position, we have j heads.. int n = prob.size(); vector<vector> F(n,vector(n+1,0)); //initialize choosing no heads at all... F[0][0] = 1-prob[0]; F[0][1] = prob[0]; for (int i = 1; i < n; ++ i) { F[i][0] = F[i-1][0] * (1-prob[i]); } //The reason I need to make sure F[0][1] is set is because //from this condition, I won't be able to set F[i-1][j], which starts from F[0][1] for (int j = 1; j <= n; ++ j) { for(int i = 1; i<n; ++i) { F[i][j] = F[i-1][j-1]*prob[i] + (1-prob[i]) * F[i-1][j]; } } return F[n-1][target]; } };

1245. Tree Diameter

https://www.geeksforgeeks.org/longest-path-undirected-tree/ (https://www.geeksforgeeks.org/longest-path-undirected-tree/)

1105. Filling Bookcase Shelves

https://leetcode.com/problems/filling-bookcase-shelves/discuss/323672/C%2B%2B-DP-4ms-solution-with-step-by-step-explanation (https://leetcode.com/problems/filling-bookcase-shelves/discuss/323672/C%2B%2B-DP-4ms-solution-with-step-by-step-explanation)

Step 1: Init DP. where dp[i] represents the min height of book up to i (1-index for padding to prevent SEGV)

Step 2: Do DP, for each new book i (for book [0, n) and dp [1, n + 1)) we have 2 chioces

Step 2-1: Put on the new layer of shelf, and the total height till book i will be dp[i - 1] + books[i][1] Step 2-2: Yet, we may try to squeeze the book as many as possible on the same layer on the condition that sum of their width < shift_width. So we go back from book i to book 0, checking all the possibilities that can be squeezed to the same layer. And if(sum > sw) the sum of width exceeds the shift width, unable to squeeze, then we quit finding the previous combinations. Analysis:

Time complexity: O(N ^ 2) Space complexity: O(N) for storing the results

class Solution { public: void print_dp(vector& dp) { for(auto &x : dp) { printf("%d ", x); } printf("\n"); } int minHeightShelves(vector<vector>& books, int sw) { int n = books.size(); // init vector dp(n + 1, 0); // min height for book up to i (starting from 0) dp[0] = 0;

```
    // doing dp
    for(int i = 0; i < n; i++)
    {
        dp[i + 1] = dp[i] + books[i][1]; // Step 2- 1: on the new layer
        int sum = 0, h = 0;
        for(int j = i; j >= 0; j--)
        {
            sum += books[j][0]; // try to put on current layer, rather than the new one
            if(sum > sw) // the sum of width exceeds the shift width, unable to squeeze
            {
                break;
            }
            else // Step 2- 2: keep squeezing
            {
                h = max(h, books[j][1]); // get the tallest book for this layer
                dp[i + 1] = min(dp[j] + h, dp[i + 1]); // for i + 1 th book it can either be the next layer, or this layer(tr
y the combination to make 'one layer' as short as possible)
                // printf("sum %d booksj_h %d h %d j %d dp[j] %d i %d dp[i + 1] %d\n", sum, books[j][1], h, j, dp[j], i, dp[i
+ 1]);
            }
        }
        // print_dp(dp);
    }
    // print_dp(dp);
    return dp[n];
}
```

};

1110. Delete Nodes And Return Forest

https://www.youtube.com/watch?v=aaSFzFfOQ0o (https://www.youtube.com/watch?v=aaSFzFfOQ0o)

1123. Lowest Common Ancestor of Deepest Leaves

ppaat nahi

1130. Minimum Cost Tree From Leaf Values

paata nahi

1129. Shortest Path with Alternating Colors

https://leetcode.com/problems/shortest-path-with-alternating-colors/discuss/656384/C%2B%2B-BFS-with-explanation (https://leetcode.com/problems/shortest-path-with-alternating-colors/discuss/656384/C%2B%2B-BFS-with-explanation)

1138. Alphabet Board Path

Approach: Step1: Map the relation of alphabet -> position Step2: Traverse the string, and calculate the needed direction, if the previsous position is the same as now, just add !, otherwise, go there and add ! For the tricky z-related test case like 'zdz', for entering z, we may first go left and down and for leaving z we may first go up then right to avoid out of board. WA

Input: "zdz" Output: "DDDDD!UUUUURRR!DDDDDLLL!" Expected: "DDDDD!UUUUURRR!DDDDLLLD!" Analysis: Time complexity: O(len(target)), since the longest path will be e to z and is fixed to a constant, so O(Constant * len(target)) = O(len(target)). Feel free to correct me if I am wrong Space complexity: O(len(target))

1329. Sort the Matrix Diagonally

xplanation A[i][j] on the same diagonal have same value of i - j For each diagonal, put its elements together, sort, and set them back.

Complexity Time O(MNlogD), where D is the length of diagonal with D = min(M,N). Space O(MN)

1192. Critical Connections in a Network

https://leetcode.com/problems/critical-connections-in-a-network/discuss/638323/C%2B%2B-clean-code-FInding-Bridges-Easy (https://leetcode.com/problems/critical-connections-in-a-network/discuss/638323/C%2B%2B-clean-code-FInding-Bridges-Easy)

https://www.hackerearth.com/practice/notes/nj/ (https://www.hackerearth.com/practice/notes/nj/)

1405. Longest Happy String

The intuition of the problem is (1) put character with larger count in the front of the string (2) avoid three consecutive repeating character

To achieve this target, priority_queue is a good choice. For avoid repeating, my answer is to record the previous two used characters, and compare them when trying to insert new one.

https://leetcode.com/problems/longest-happy-string/discuss/574645/Easy-understood-PriorityQueue-Solution-in-C%2B%2B (https://leetcode.com/problems/longest-happy-string/discuss/574645/Easy-understood-PriorityQueue-Solution-in-C%2B%2B)

1220. Count Vowels Permutation

Observation The question gives us a map of the vowels that can come after each vowel, we use this mapping to get the number of strings that can be formed by those 'ending' vowels.

a can only be added after i, e and u. Thus strings ending with a in the next step will be sum of strings ending with i, u and e in the current step. e can only be added after a and i. Thus strings ending with e in the next step will be sum of strings ending with i and a in the current step. i can only be added after o and e. Thus strings ending with i in the next step will be sum of strings ending with o and e in the current step. o can only be added after i. Thus strings ending with o in the next step will be equal to strings ending with i in the current step. u can only be added after i and o. Thus strings ending with u in the next step will be equal to strings ending with i and o in the current step. We do this for all the other vowels and repeat this N-1 times to get our answer.

Solution Bottom-up: With inversed relationships as explained above (Totally depends on how you imagine the subproblems/DP).

1234. Replace the Substring for Balanced String

Usually count the element inside sliding window, and i won't be bigger than j because nothing left in the window.

The only reason that we need a condition is in order to prevent index out of range. And how do we do that? Yes we use i < n

With i <= j + 1, the problem is index out of range when j = n - 1 and i = n. with i < n, when i = j + 1 the all() condition fails already. The while loop stops. Overall, i < n is better. Otherwise, you still have to check whether i < n

1235. Maximum Profit in Job Scheduling

https://leetcode.com/problems/maximum-profit-in-job-scheduling/discuss/411535/C%2B%2B-DP-explained-with-an-example- (https://leetcode.com/problems/maximum-profit-in-job-scheduling/discuss/411535/C%2B%2B-DP-explained-with-an-example-)...

Myntra me aaya tha

1247. Minimum Swaps to Make Strings Equal

//lets take s1=xyxy s2=xyxy //both are equal so 0 swaps //let s1=xyxy s2=yyxy //we cannot make equal -1 //let s1=xyxy s2=yxxy //swap 1: s1[0]&s2[0]....this gives...s1=yyxy s2=xxxy //swap 2: s1[0]&s2[1]....this gives....s1=xyxy s2=xyxy //so we made equal in 2 swaps when there is x-y and y-x //let s1=xxxy s2=yyxy //swap 1: s1[1]&s2[0]....this gives....s1=xyxy s2=xyxy //so we made equal in 1 swap when there is x-y ,x-y pair(or)y-x ,y-x pair int minimumSwap(string s1, string s2) { int countxy=0; int countyx=0; int i; int ans=0; for(i=0;i<s1.length();i++) { if(s1[i]=='x'&&s2[i]=='y')countxy++; if(s1[i]=='y'&&s2[i]=='x')countyx++; } //as we can make x-y ,x-y or y-x,y-x pairs equal in one swap..see that ans+=(countxy/2);//for x-y,x-y pairs ans+=(countyx/2);//for y-x,y-x pairs //if countxy,countyx are even above 2 steps is enough..if they are odd: countxy=countxy%2; countyx=countyx%2; //if countxy=1 & countyx=0...this means one x-y pair is not equal in s1,s2 //then we can't make it eual //same for countyx=1 & countxy=0 if(countxy==1&&countyx==0)return -1; if(countyx==1&&countxy==0)return -1; //if both are 1 then we can make them equal in 2 swaps as: //let s1=xyxy s2=yxxy //swap 1: s1[0]&s2[0]....this gives...s1=yyxy s2=xxxy //swap 2: s1[0]&s2[1]....this gives....s1=xyxy s2=xyxy //so we made equal in 2 swaps when there is x-y and y-x if(countxy==1&&countyx==1)ans+=2; return ans; } };

1254. Number of Closed Islands

https://leetcode.com/problems/number-of-closed-islands/discuss/425150/JavaC%2B%2B-with-picture-Number-of-Enclaves (https://leetcode.com/problems/number-of-closed-islands/discuss/425150/JavaC%2B%2B-with-picture-Number-of-Enclaves)

1262. Greatest Sum Divisible by Three

Here are some detailed explanations of the posted method. I hope them help you understand lee215's brilliant post if you have doubts.

```
Intuition:
    1.The last maximum possible sum that it is divisible by three could only depends
    on 3 kinds of "subroutines/subproblems":
        1. previous maximum possible sum that it is divisible by three
           preSum % 3 == 0       (example: preSum=12 if lastNum=3)
        2. preSum % 3 == 1       (example: preSum=13 if lastNum=2)
        3. preSum % 3 == 2       (example: preSum=14 if lastNum=1)
    2. This recusion + "subroutines" pattern hints Dynamic Programming

dp state:
    dp[i] = max sum such that the reminder == i when sum / 3
Transition:
    dp_cur[(rem + num) % 3]
        = max(dp_prev[(rem + num) % 3], dp_prev[rem]+num)
        where "rem" stands for reminder for shorter naming
    meaning:
        "Current max sum with reminder 0 or 1 or 2" could be from
        EITHER prevSum with reminder 0 or 1 or 2 consecutively
        OR     prevSum with some reminder "rem" + current number "num"

        Since (dp_prev[rem]+num) % 3 = (rem+num) % 3 = i, we are able to correctly
        update dp[i] for i = 1,2,3 each time
```

//method1: DP (disc) O(n)time O(1)space {6 ms, faster than 78.87%} public int maxSumDivThree(int[] nums) { //init: dp[i] = max sum such that the reminder == i when sum / 3 //dp[0]=0: max sum such that the reminder == 0 when 0 / 3 is 0 //dp[1]=-Inf: max sum such that the reminder == 1 when 0 / 3 does not exist //dp[2]=-Inf: max sum such that the reminder == 2 when 0 / 3 does not exist int[] dp = new int[]{0, Integer.MIN_VALUE, Integer.MIN_VALUE};

```
for(int num : nums){
    int[] temp = new int[3];
    //dp transition
    for(int reminder=0; reminder<3; reminder++){
        //updating each reminder for current "num"
        temp[(num+reminder)%3] = Math.max(dp[(num+reminder)%3], dp[reminder]+num);
    }
    //rotating array
    dp = temp;
}
//return: max sum such that the reminder == 0 when sum / 3
return dp[0];
```

}

1269. Number of Ways to Stay in the Same Place After Some Steps

Hi guys, The question is a lot easier to visualize when you think backwards. Lets set N = steps, M = arrLen You want to find how many possible ways you can reach the array[0] at Nth step. Lets ask ourselves, how do we find array[0] at N-1th step. Since we are given directions, Left, Stay, Right, we can just add the number of steps we have at N-1th step in array[0], array[1] (array[-1] isnt available) This makes sense because at N-1 th step, we can stay in array[0] or we can move left in arr[1]

0 [][][][]... 1 [][][][]... 2 [][][][]... . https://leetcode.com/problems/number-of-ways-to-stay-in-the-same-place-after-some-steps/discuss/436287/Step-by-Step-Solution-(diagrams)-(with-how-I-overcome-MemoryTime-limit-exception) (https://leetcode.com/problems/number-of-ways-to-stay-in-the-same-place-after-some-steps/discuss/436287/Step-by-Step-Solution-(diagrams)-(with-how-I-overcome-MemoryTime-limit-exception)). N-1[X][Y][][] N [Z][][][]

Using the same intuition, we find N-2 th step, for array[0] and array[1] for array[1] for N-1, we have to add array[0], array[1], array[2] together Leading us to the equation.

dp[N][M] = (dp[N-1][M-1] + dp[N-1][M] + dp[N-1][M+1]) // with boundary checks ex: steps = 4, arrLen = 4 // notice dp[0][0] will always be 1 since we start there. 0 [1][0][0][0] 1 [1][1][0][0] 2 [2][2][1][0] 3 [4][5][3][1] 4 [9][12][9][4] class Solution {

1283. Find the Smallest Divisor Given a Threshold

[962551,933661,905225,923035,990560] target=10 dry run:: mid =500000 (differ(curr_sum-tar)) =0 // here we can see even though (curr_sum-diff)==0 but number is not smallest because // number smaller than mid also satisfy all condition since we r taking ceil(x/y)

mid =250000 (differ(curr_sum-tar)) =10 mid =375000 (differ(curr_sum-tar)) =5 mid =437500 (differ(curr_sum-tar)) =5 mid =468750 (differ(curr_sum-tar)) =2 mid =484375 (differ(curr_sum-tar)) =1 mid =492188 (differ(curr_sum-tar)) =1 mid =496094 (differ(curr_sum-tar)) =0 mid =494141 (differ(curr_sum-tar)) =1 mid =495118 (differ(curr_sum-tar)) =1 mid =495606 (differ(curr_sum-tar)) =0 mid =495362 (differ(curr_sum-tar)) =0 mid =495240 (differ(curr_sum-tar)) =1 mid =495301 (differ(curr_sum-tar)) =0 mid =495271 (differ(curr_sum-tar)) =1 mid =495286 (differ(curr_sum-tar)) =0 mid =495279 (differ(curr_sum-tar)) =1 mid =495283 (differ(curr_sum-tar)) =0 mid =495281 (differ(curr_sum-tar)) =0 mid =495280 (differ(curr_sum-tar)) =0

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

1311. Get Watched Videos by Your Friends

https://leetcode.com/problems/get-watched-videos-by-your-friends/discuss/470748/BFS-%2B-Hash-Map-%2B-Set (https://leetcode.com/problems/get-watched-videos-by-your-friends/discuss/470748/BFS-%2B-Hash-Map-%2B-Set) it just has many steps:

BFS to find level-degree friends Count movies to determine frequency Sort the movies by the frequency So, we need a queue for BFS, hash map for frequency, and map to sort.

1319. Number of Operations to Make Network Connected

(Number of connected components-1) is the requied answer. Just think none of the computers are connected.Clearly n-1 is the minimum number of wires required to connect them. Now visualise each of those computers as connected components.As once we reach one of the computers in the connected component we reach everything in that component.So, (Number of connected components-1) is the requied answer.

Now twist is that to connect all computers you need n-1 wires.But so if total wires less than n-1 return -1.

See the code below for more clarity!.

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

https://leetcode.com/problems/minimum-number-of-taps-to-open-to-water-a-garden/discuss/484759/Video-Stitching (https://leetcode.com/problems/minimum-number-of-taps-to-open-to-water-a-garden/discuss/484759/Video-Stitching)

1391. Check if There is a Valid Path in a Grid

https://leetcode.com/problems/check-if-there-is-a-valid-path-in-a-grid/discuss/547225/C%2B%2B-with-picture%3A-track-direction-%2B-upscaled-grid (https://leetcode.com/problems/check-if-there-is-a-valid-path-in-a-grid/discuss/547225/C%2B%2B-with-picture%3A-track-direction-%2B-upscaled-grid)

1411. Number of Ways to Paint N × 3 Grid

https://leetcode.com/problems/number-of-ways-to-paint-n-3-grid/discuss/575287/WITH-PICTURES-and-EXPLANATION (https://leetcode.com/problems/number-of-ways-to-paint-n-3-grid/discuss/575287/WITH-PICTURES-and-EXPLANATION)

1443. Minimum Time to Collect All Apples in a Tree

Calculate distances of all nodes w.r.t. to root i.e. 0. (Using simple BFS). Also store parent for each node. Notice distance between any two nodes x, y is abs(dist[x]-dist[y]). Now start traversing the list of nodes where apple is. Preferably backwards. When you see a node with apple you travel towards root including each edge twice untill you find some already visited node or reach root obviously after which you can't go any further. Why till a visited node? Because if it is already visited it implies that path from that node to root is already included and since we want shortest time we need not visit it again.

Your solution gives wrong answer for:

4 [[0,2],[0,3],[1,2]] [false,true,false,false]

1 Hide 2 replies Reply Share Report Don_Corleone's avatar Don_Corleone 13 June 2, 2020 9:59 PM

Read More Yeah I checked, you are right. I didn't bothered in the contest as it got accepted. Thanks by the way for noting. I'll try it again.

1462. Course Schedule IV

https://leetcode.com/problems/course-schedule-iv/discuss/660509/JavaPython-FloydWarshall-Algorithm-Clean-code-O(n3) (https://leetcode.com/problems/course-schedule-iv/discuss/660509/JavaPython-FloydWarshall-Algorithm-Clean-code-O(n3))

1463. Cherry Pickup II

code: https://leetcode.com/problems/cherry-pickup-ii/discuss/663451/3D-dp-Bottom-up-or-easy-to-understand-or-clean-code (https://leetcode.com/problems/cherry-pickup-ii/discuss/663451/3D-dp-Bottom-up-or-easy-to-understand-or-clean-code)

explanation: https://leetcode.com/problems/cherry-pickup-ii/discuss/688218/C%2B%2B-DP-Bottom-up-solution-memory-efficient-(explained) (https://leetcode.com/problems/cherry-pickup-ii/discuss/688218/C%2B%2B-DP-Bottom-up-solution-memory-efficient-(explained))

1466. Reorder Routes to Make All Paths Lead to the City Zero

You need to understand first that for running BFS ,you need few tools ,namely: 1)Graph (To know the children of every node) .We have used a map(map1) for this purpose(Just for fun).Generally adjacency list or matrix is used. 2)visited array (To perform code only once on nodes)We have used array (possible) here. 3)Queue(To maintain a sequencial order and traversing all nodes). We have used a queue(q1)here. Additionaly for this question we have used a set (set1) for knowing whether a particular edge is present or not.

1493. Longest Subarray of 1's After Deleting One Element

https://leetcode.com/problems/longest-subarray-of-1s-after-deleting-one-element/discuss/708109/Python-O(n)-dynamic-programming-detailed-explanation (https://leetcode.com/problems/longest-subarray-of-1s-after-deleting-one-element/discuss/708109/Python-O(n)-dynamic-programming-detailed-explanation)

1494. Parallel Courses II

passed during contest due to weak test case: https://leetcode.com/problems/parallel-courses-ii/discuss/708416/C%2B%2B-BFS-%2B-Priority_Queue-with-Explanation-and-Comments (https://leetcode.com/problems/parallel-courses-ii/discuss/708416/C%2B%2B-BFS-%2B-Priority_Queue-with-Explanation-and-Comments)

1508. Range Sum of Sorted Subarray Sums

https://leetcode.com/problems/range-sum-of-sorted-subarray-sums/discuss/730665/C%2B%2B-simple-straightforward-with-prefix-sums (https://leetcode.com/problems/range-sum-of-sorted-subarray-sums/discuss/730665/C%2B%2B-simple-straightforward-with-prefix-sums)

1509. Minimum Difference Between Largest and Smallest Value in Three Moves

https://leetcode.com/problems/minimum-difference-between-largest-and-smallest-value-in-three-moves/discuss/751091/Detailed-Explanation-with-Code (https://leetcode.com/problems/minimum-difference-between-largest-and-smallest-value-in-three-moves/discuss/751091/Detailed-Explanation-with-Code)

1497. Check If Array Pairs Are Divisible by k

https://leetcode.com/problems/check-if-array-pairs-are-divisible-by-k/discuss/709212/Short-C%2B%2B-Solution-explained-with-comments (https://leetcode.com/problems/check-if-array-pairs-are-divisible-by-k/discuss/709212/Short-C%2B%2B-Solution-explained-with-comments)

1524. Number of Sub-arrays With Odd Sum

https://leetcode.com/problems/number-of-sub-arrays-with-odd-sum/discuss/754870/JAVAC%2B%2B-SImple-Mathematical-approach-in-O(n) (https://leetcode.com/problems/number-of-sub-arrays-with-odd-sum/discuss/754870/JAVAC%2B%2B-SImple-Mathematical-approach-in-O(n))

1525. Number of Good Ways to Split a String

https://leetcode.com/problems/number-of-good-ways-to-split-a-string/discuss/863462/C%2B%2B-Easy-to-understand-solution-using-Hashmap-with-comments (https://leetcode.com/problems/number-of-good-ways-to-split-a-string/discuss/863462/C%2B%2B-Easy-to-understand-solution-using-Hashmap-with-comments)

1557. Minimum Number of Vertices to Reach All Nodes

https://www.youtube.com/watch?v=ZVm9iVZxsA0&t=367s&ab_channel=ChhaviBansal (https://www.youtube.com/watch?v=ZVm9iVZxsA0&t=367s&ab_channel=ChhaviBansal)

1558. Minimum Numbers of Function Calls to Make Target Array

https://leetcode.com/problems/minimum-numbers-of-function-calls-to-make-target-array/discuss/805819/C%2B%2B-Well-commented-code-or-Observation-or-Video-explanation (https://leetcode.com/problems/minimum-numbers-of-function-calls-to-make-target-array/discuss/805819/C%2B%2B-Well-commented-code-or-Observation-or-Video-explanation)

1559. Detect Cycles in 2D Grid

https://leetcode.com/problems/detect-cycles-in-2d-grid/discuss/805673/C%2B%2B-BFS (https://leetcode.com/problems/detect-cycles-in-2d-grid/discuss/805673/C%2B%2B-BFS)

1574. Shortest Subarray to be Removed to Make Array Sorted

https://leetcode.com/problems/shortest-subarray-to-be-removed-to-make-array-sorted/discuss/830480/C%2B%2B-O(N)-Sliding-window-Explanation-with-Illustrations (https://leetcode.com/problems/shortest-subarray-to-be-removed-to-make-array-sorted/discuss/830480/C%2B%2B-O(N)-Sliding-window-Explanation-with-Illustrations)

1594. Maximum Non Negative Product in a Matrix

https://leetcode.com/problems/maximum-non-negative-product-in-a-matrix/discuss/855742/C%2B%2B-min-max-dp-solution-100-speed-with-picture-explanation-and-similar-problems (https://leetcode.com/problems/maximum-non-negative-product-in-a-matrix/discuss/855742/C%2B%2B-min-max-dp-solution-100-speed-with-picture-explanation-and-similar-problems)

https://leetcode.com/problems/maximum-non-negative-product-in-a-matrix/discuss/865451/C%2B%2B-DP-With-Comments-and-Video-Explanation (https://leetcode.com/problems/maximum-non-negative-product-in-a-matrix/discuss/865451/C%2B%2B-DP-With-Comments-and-Video-Explanation)