/\*1706\*/

1. Find duplicate in an array of 1 to N numbers only.

a. Brute Force: For each element check whether another element exist or not.

b. Using extra space : Use map or set.

c. Without extra space: O(nlogn): Sort the array, check i and i+1

d. O(N) solution: For each i, mark a[i] as -1, if the same element occurs again a[j] will be

already -1.

class Solution {

public:

int findDuplicate(vector<int>& nums) {

int i=0,n=nums.size();

while(i<n)

{

if (nums[i] == -1)return i;

int cur = i;

i = nums[i] ;

nums[cur] = -1;

}

return 0;

}

};

Simply revisiting the similar index does the trick.

e. From above point, we can assume that there is a cycle in the linked list. We can then use hare and tortoise technique to detect the entry point of the cycle. Approach is still O(N) but doesn’t modify the array.

1a. What if more than one element is duplicate ?

[4,3,2,7,8,2,3,1]

a. Instead of using -1, mark each element as its negative value. If you find that the number is being revisited then add it to the answer. To check it, the value is already -ve when you visit it.

class Solution {

public:

vector<int> findDuplicates(vector<int>& nums) {

vector<int> ans;

for(int i=0,n=nums.size();i<n;i++)

{

int num = abs(nums[i]) -1 ;

if(nums[num]<0){

ans.push\_back(num+1);

}

nums[num]\*=-1;

}

return ans;

}

};

2. Sort an Array of 0’s , 1’s and 2’s

a. Using count sort will do the trick in O(N) but will cost extra space.

b. Count 0 and 1 to get exact positions of where 1 and 2 must start. Now if we encounter 0 in 1 & 2’s region, swap them until that position is not claimed by a 1 or 2 whatever. Do the same for 1 and 2 too. It is a two pass approach.

class Solution {

public:

void sortColors(vector<int>& nums) {

int i=0,r,b,w,wc=0,bc=0,n = nums.size(),temp;

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

{

if(nums[i] == 0)++wc;

else if(nums[i] == 1) ++bc;

}

bc+=wc;

r=0;

w=wc;

b = bc;

for(i=0;i<n;)

{

if(nums[i] == 0 && i>=wc)

{

temp = nums[i];

nums[i] = nums[r];

nums[r] = temp;

++r;

}

else if(nums[i] ==1 && (i<wc || i>=bc))

{

temp = nums[i];

nums[i] = nums[w];

nums[w] = temp;

++w;

}

else if(nums[i] == 2 && i<bc)

{

temp = nums[i];

nums[i] = nums[b];

nums[b] = temp;

++b;

}

else{

i++;

}

}

}

};

c. Let zero = 0 and two = N-1.

Now swap each 0 with zero++ if i>zero

and swap each 2 with two-- if i<two. This is a single pass approach.

class Solution {

public:

void sortColors(vector<int>& nums) {

int n = nums.size(),second=n-1, zero=0;

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

while (nums[i]==2 && i<second) swap(nums[i], nums[second--]);

while (nums[i]==0 && i>zero) swap(nums[i], nums[zero++]);

}

}

};

3. Find the missing Number.

Given an array with numbers 0 to N, find the missing number from the array.

a. Expected sum = n\*(n+1)/2 , lets say sum = S

Missing number is n\*(n+1)/2 -S

// Find missing number

class Solution {

public:

int missingNumber(vector<int>& nums) {

int n=nums.size(),sumshouldbe = n\*(n+1)/2,sum=0;

for(int i=0;i<n;i++)sum+=nums[i];

return sumshouldbe - sum;

}

};

b. Initialize X = 0,

for each i, X ^= (i^arr[i]) ; ^ = XOR

At last X^=N

each number will appear even number of time,so effect will be 0 except for the missing number. Hence X is the answer.

c. If array is sorted Use Binary search O(log N)

4. Merge two sorted arrays arr1 and arr2 given that arr1 has enough space left to store arr2.

a. Use merge step of merge sort. Costs Extra O(N) space.

b. To avoid using extra space, simply start from the last ie n+m-1th position and use merge function. It can be proved that arr1 will not be overwritten before that element is pushed in the array at some other position. Let’s say N’th element is being overwritten, that means all M elements are already pushed. Hence it won’t get touched. It may happen that all elements of arr1 are greater than arr2 . In that case the elements of arr2 will remain. Otherwise arr1 will be sorted in place in the other case.

// merge the sorted arrays

class Solution {

public:

void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {

int i=m-1;

int j=n-1;

int k = m+n-1;

while(i >=0 && j>=0)

{

if(nums1[i] > nums2[j])

nums1[k--] = nums1[i--];

else

nums1[k--] = nums2[j--];

}

while(j>=0)

nums1[k--] = nums2[j--];

}

};

5. Max sum subarray with negative elements. Find subarray with atleast one element.

a. Use Kadane’s algorithm. Make sure to initialise maxsofar = INT\_MIN. That way we will consider the negative elements too. Otherwise if all the elements are positive or 0, initialize it as 0.

// MAX subarray

class Solution {

public:

int maxSubArray(vector<int>& nums) {

int i,n=nums.size(), sum=0,maxsum=INT\_MIN;

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

{

sum+=nums[i];

if(sum>maxsum){

maxsum=sum;

}

if(sum<0)sum=0;

}

return maxsum;

}

};

6. Merge the overlapping intervals.

Given N intervals in the form (Li , Ri ), merge those which are overlapping with each other.

a. Simply sort them using left index. And whatever keeps on connecting, skip that, and restart the same where a disconnected interval is found.

Start with {L1, \_}

Keep skipping while Rk >= Lk+1.. Keep on updating R to the max avaialble.

Add {L0, Rk } to the answer; make another {Lk+1,\_} and repeat the same.

At last add {Lm, Rn}

// Merge overlapping intervals

class Solution {

public:

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

vector<vector<int>> res;

int i,n=intervals.size();

if(n==0)return res;

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

vector<int> overlap;

overlap.push\_back(intervals[0][0]);

int r = intervals[0][1];

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

{

if(intervals[i][0]<=r)

{

r = max(r,intervals[i][1]);

}else{

overlap.push\_back(r);

res.push\_back(overlap);

overlap.clear();

overlap.push\_back(intervals[i][0]);

r = intervals[i][1];

}

}

overlap.push\_back(r);

res.push\_back(overlap);

return res;

}

};

7. Set matrix Zeroes

Given a matrix of all 0’s and 1’s. Set all the rows and colums =0 Where a 0 is present in any cell

eg

5 10 15 5 0 15

1 0 3 0 0 0

11 2 1 11 0 1

a. Use two sets to store the rows and cells seperately where there is a 0 in those cells. After that update them in place afterwards. It will take O(m+n) extra space.

b. Instead of using extra sets, mark the first row and column as 0 where a cell contains zero.

Start from second row and second column, and do this process. Note that if there is a 0 in first row, matrix[0][0] is updated but not for first column. So check the column seperately.

Mark the internal (M-1)x(N-1) matrix first then, check the cell [0][0] to mark the first row. Atlast if first column contained 0 , mark it also.This approach does everything inplace with O(1) space.

// set matrix zero.

class Solution {

public:

void setZeroes(vector<vector<int>>& matrix) {

int i=0,j=0,n=matrix.size(),m=matrix[0].size();

bool c=false;

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

{

if(matrix[i][0] == 0)

{

c=true;

}

for(j=1;j<m;j++)

{

if(!matrix[i][j])

{

matrix[i][0] = matrix[0][j] = 0;

}

}

}

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

{

for(j=1;j<m;j++)

{

if(matrix[i][0] == 0 || matrix[0][j]==0){

matrix[i][j] = 0;

}

}

}

if(matrix[0][0]==0)

{

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

matrix[0][j] = 0;

}

if(c){

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

{

matrix[i][0]=0;

}

}

}

};

8. Generate the pascal’s triangle.

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

...

Simply create a triangle of only 1’s

The pattern can be observed easily.

For each row,starting from 3rd ie 2nd index, each element is sum of its upper and diagonally left element. At last a 1 is present which is not a matter of concern.

// Pascal Triangle

class Solution:

def generate(self, numRows: int) -> List[List[int]]:

l = []

for i in range(numRows):

l.append([1]\*(i+1))

for i in range(2,numRows):

for j in range(1,len(l[i-1])):

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

return l

9. Next permutation.

Given an array of numbers or a string, return the next permutaion in sorted sequence order... if no such permutaion exist, return the sorted array.

ie 1 3 2 => 2 1 3

and “abcd” => “abdc”

a. Use c++ stl next\_permutation(arr.begin(),arr.end()) (Not preferred)

b. Use proper algorithm,

i. From right to left, Find the first element that isn’t in descending order.

ii. Swap it with the closest larger element in its right.

iii. Sort the rest array from that index (excluding itself.)

// Next Permutation

class Solution {

public:

void nextPermutation(vector<int>& nums) {

int i=0,n=nums.size(),k=-1;

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

{

if(nums[i]<nums[i+1]){

k = i;

break;

}

}

if(k==-1)

{

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

}

else{

int j,diff=INT\_MAX;

for(i=n-1;i>k;i--)

{

if(nums[i]>nums[k] and nums[i]-nums[k]<diff)

{

diff = nums[i] - nums[k];

j = i;

}

}

swap(nums[k],nums[j]);

sort(nums.begin()+k+1,nums.end());

}

}

};

10. Inversion Count using merge sort.

Given an array, inversion count is the number of elements such that for i<j , A[i]>A[j].

a. In the merge step, there are two arrays, L and R. Both are sorted, For each element in L such that L[i] > R[i] , it means that all elements from L[i] to last index of L are greater that R[i] , so inversion count can be increased with mid – i + 1 where mid is the last index of L.

// Inversion Count using merge sort

class Solution {

public:

int globalinversion=0;

void merge(vector<int>& A , int l , int mid , int r)

{

vector<int> temp(r-l+1);

int i=l,j=mid+1,k=0;

while(i<=mid and j<=r){

if(A[i]<=A[j]){

temp[k++] = A[i++];

}else{

temp[k++] = A[j++];

globalinversion += mid-i+1;

}

}

while(i<=mid)temp[k++]=A[i++];

while(j<=r)temp[k++]=A[j++];

for(k=0;k<r-l+1;k++){

A[l+k] = temp[k];

}

}

void mergeSort(vector<int>& A , int l , int r)

{

if (l==r)return;

int mid = (l+r)/2;

mergeSort(A,l,mid);

mergeSort(A,mid+1,r);

merge(A,l,mid,r);

}

bool isIdealPermutation(vector<int>& A) {

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

for(i=0;i<n-1;i++)if(A[i]>A[i+1])localinversion++;

mergeSort(A,0,n-1);

return (localinversion == globalinversion);

}

};

10a. Lets say array contains the elements between 0 to N-1 only.

a. Merge sort : NlogN

b. BIT : NlogN , but won’t work if A[i] is too large. Since A[i]<=N-1, we can use it here.

10b. For a permutation of {0,1,...N-1} lets define:

local inversion : A[i] > A[i+1] for each i

global inversion : A[i] > A[j] if i<j.

Note that local inversion is also a global inversion but reverse is not true.

Check whether local == global or not.

a. Use the previous approaches to count both and check if they are equal or not.

b. For each A[i]>A[i+1], simply swap them and check whether the array is now sorted or not. If it is sorted now, global = local.

c. Extending to last approach, if the elements A[i] at are at A[i]th A[i+1]th or A[i-1]th position, they can be sorted using previous apprach, so simply check for | A[i] – i|>1 for any element means that it can’t be sorted and hence local ≠ global.

// Inversion Count for Array [0,1...N-1]

class Solution {

public:

bool isIdealPermutation(vector<int>& A) {

for(int i=0,n=A.size();i<n;i++)

if(A[i]!=i && A[i]!=i+1 && A[i]!=i-1)return false; // Can also use condition => [abs(A[i]-i)>1]

return true;

}

};

11. Stock Buy and sell:

Given a list of integers, ith element is the price of the stock on ith day. You can buy a stock once and sell it once, Find the maximum possible profit.

a. Idea is to find two numbers A[i] & A[j] ; i<j, with A[j] – A[i] having the maximum value.

From left to right keep an eye on the minimum element, because A[j] – A[i] is maximum if A[i] is minimum; A[j] is maximum. For each price, check the value of A[j] – minsofar, and whatever gives the max at last, print that.

// Buy and Sell stock

class Solution {

public:

int maxProfit(vector<int>& prices) {

int i,n=prices.size(),minsofar=INT\_MAX,ans=0;

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

{

if(prices[i]<minsofar){

minsofar = prices[i];

}

ans = max(ans,prices[i] - minsofar);

}

return ans;

}

};

11b. What if we are allowed to buy and sell multiple times?

a. We can use the same logic as before but consider the example below:

[7 1 5 3 6 4]

instead of taking 6 – 1 = 5, We will take (5 – 1) + (6 – 3) = 7.

Idea is simple, breaking the range like this will give us more profit because it is a straight slope,from 1 to 6. If we add a hinderence or a valley the difference will either remain the same or it will increase.

Eg [1 4 6] gives (4-1) + (6-4) = 5

But [1 4 3 6] gives (4-1 ) + (6 – 3) = 6

So find the contiguous increasing subarray , add the difference of endpoints in the answer.

Don’t forget to handle the last index seperately.

Again if we observe the graph carefully, we find that answer is nothing but ΣA[i] – A[i-1] (>0) ie only positive differences.

// Buy and Sell stock with multiple transactions

class Solution {

public:

int maxProfit(vector<int>& prices) {

int i,n=prices.size(),ans=0,start = 0;

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

{

if(prices[i]<prices[i-1])

{

ans += prices[i-1] - prices[start];

start = i;

}

if(i==n-1)

{

ans+=max(0,prices[i] - prices[start]);

}

}

return ans;

}

};

12. Rotate matrix.

Given an NxN matrix, rotate the matrix 90 degree.

a. take transpose, (swap upper and lower triangle )

b . To rotate 90 degree, Take mirror image about mid column. To rotate -90 degree take mirror image about mid row.

Why this works:

consider:

1 2 3 7 4 1

4 5 6 => 8 5 2

7 8 9 9 6 3

We need to convert the row into columns and columns into rows for rotating, and that is the definition of transpose.

// Rotate NxN matrix by 90 degree.

class Solution {

public:

// I already know the solution. We just need to

//transpose the matrix and find the mirror image

// about the mid Y axis

void rotate(vector<vector<int>>& matrix) {

int i=0,j=0,n=matrix.size();

// Taking transpose of this matrix

// ie swap the upper and lower triangle

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

{

for(j=0;j<i;++j)

{

swap(matrix[i][j],matrix[j][i]);

}

}

// Now simply take the mirror image

// ie go upto mid, swap first,last & second,

// second last.....upto mid

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

for(j=0;j<n/2;++j)

{

swap(matrix[i][j],matrix[i][n-j-1]);

}

}

}

};

13. Excel sheet column number

Given a string s, find the serial number of this string in lexcographically orders dictionary of [A-Z] from ‘A’ to ‘ZZZZZZZZZ...Z’

a. Consider a number 2136. It is at 2136th position in number list. How? It is formed by skipping 2\*103 numbers, then 1\*102 numbers , 3\*101 numbers then 6 numbers.

Similarly in this case, A = 1, B = 2 , .. Z = 26. So

“AAB” will be formed by skipping 1\*262 + 1\*261 + 2 strings.

// Excel Column number

class Solution {

public:

// So basicaly we need to find the lexicographic index

// of the given string in [A-Z] alphabet

int titleToNumber(string s) {

//Observe this:

// 1 = 1, 2 = 2 , .. 100 = 100, 248 = 248

// How do we get here?

// 248 = 2\*10^2 + 4\*10^1 + 8\*10^0

// What is 10 here? Its the base.

// For [A-Z], base is 26

// Hence replace 10 by 26 in above logic

int ans=0;

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

{

ans += pow(26,n-i-1)\*(s[i]-64);

}

return ans;

}

};

14. Find pow(X, N) in logN time. N can be negative or zero.

a. Brute force with O(N)

b. Use recursion, pow(X,N) = [pow(X,N/2)] 2  ; if N is even

= X \* [pow(X,N/2)] 2 ; if N is odd

// pow x^n in log(n) time

class Solution {

public:

double myPow(double x, long long n) {

if(n==0)return 1.0;

if(n<0)return 1.0/myPow(x,-n);

if(n==1){

return x;

}

if(n&1){

double xx = myPow(x,n/2);

return xx\*xx\*x;

}else{

double xx = myPow(x,n/2);

return xx\*xx;

}

}

};

15. Count trailing zeroes in N!.

a. Find N! And then count the zeroes.

b. Find the count of 2 and 5 for each number from 1 to N that divides each number. Doing it for each number will take O(N\*sqrt(N)) time. So only calculate Σ N/pow(2,i) and Σ N/pow(5,i)

// Count trailing zeroes, a naive approach

class Solution {

public:

int trailingZeroes(int n) {

int count2=0,count5=0,i;

int p = 1;

while(true){

double x = pow(2,p);

if(x<=n){

count2 += n/pow(2,p);

p++;

}else break;

}

p=1;

while(true){

double x = pow(5,p);

if(x<=n){

count5 += n/pow(5,p);

p++;

}else break;

}

return min(count2,count5);

}

};

c. Or only count Σ N/pow(5,i) because count of 2 will be always smaller.

// Count Trailing zeroes

class Solution {

public:

int trailingZeroes(int n) {

if(n==0)return 0;

return n/5 + trailingZeroes(n/5);

}

};

16. GCD in logN time:

Use Euclid algorithm which solves a\*x + b\*y = g where g = GCD of a and b.

Extended Euclid also calculates the values of x and y.

It takes O(log(min(a,b))) time.

// Euclid and Extended Euclid Algo for reference;

int gcd(int a, int b)

{

if (b == 0)

return a;

return gcd(b,a%b);

}

int gcd(int a, int b, int& x, int& y) { // extended

if (b == 0) {

x = 1;

y = 0;

return a;

}

int x1, y1;

int gcd = gcd(b, a % b, x1, y1);

x = y1;

y = x1 - y1 \* (a / b);

return gcd;

}

17. Unique Paths in a grid : Given an MxN grid, Find the number of uniques paths from TopLeft to BottomRight where only Right and Down moves are allowed.

a. We need to take exactly N- 1 moves to Right and M-1 moves to down. Consider any permutation of RRR..DDD.., we will reach to the BottomRight. If we count the number of such permutaions, we get the answer.

The answer will ne (M+N-2)! / (M-1)! / (N-1)!

We can’t calculate this value for large matrix (even if M or N is 100)

// Grid Unique Paths

class Solution:

def uniquePaths(self, m: int, n: int) -> int:

return factorial(m+n-2)//factorial(m-1)//factorial(n-1)

b. Another way is to use DP, count the number of ways to reach a cell[i][j] = number of ways to reach cell[i-1][j] + number of ways to reach cell [i][j-1]

// Append the image below

int countWays(int m , int n)

{

int arr[m][n] , i , j;

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

{

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

{

if(i==0 or j==0)arr[i][j] = 1;

else arr[i][j] = arr[i-1][j] + arr[i][j-1];

}

}

return arr[m-1][n-1];

}

18. Puzzles Link :<https://www.geeksforgeeks.org/puzzles/>

19. Two Sum

Given an array of integers, return indices of the two numbers such that they add up to a specific target.You may assume that each input would have exactly one solution, and you may not use the same element twice.

Example:

Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9,

return [0, 1].

a. Idea is to use hashing. Use map or set to store each element. For arr[i] check whether target – arr[i] is already present in the map/set or not. Cost: O(N) time and O(N) space

class Solution {

public:

vector<int> twoSum(vector<int>& nums, int target) {

unorderd\_map<int,int> mp;

vector<int> ans;

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

if(mp.find(target-nums[i])!=mp.end() && mp[target-nums[i]]!=i){

ans.push\_back(i);

ans.push\_back(mp[target-nums[i]]);

return ans;

}

mp[nums[i]]=i;

}

return ans;

}

};

19a. What if the array is sorted already.

a. It turns out that any number can be written as sum of two numbers i and j where i<=j. So simply traverse the array using two pointers i and j, i at the start , j at the end. We don’t need extra space this way.

Consider these examples:

[1, 12, 12, 14, 14, 20] 26

[5, 10, 10, 11, 11, 14, 21, 23, 24, 25] 48

[5, 5, 6, 12, 15, 15, 19, 21] 27

If arr[i] + arr[j] > target , --j;

else if arr[i] + arr[j] < target , ++i;

else return {i,j}

pair<int,int> twosum(vector<int> arr , int target){

int i=0,j=arr.size();

while(i<j){

if(arr[i]+arr[j]<target)++i;

else if(arr[i]+arr[j]>target)--j;

else return make\_pair(i,j);

}

}

20. 4 Sum

Given an array nums of n integers and an integer target, are there elements a, b, c, and d in nums such that a + b + c + d = target? Find all unique quadruplets in the array which gives the sum of target.

The solution set must not contain duplicate quadruplets.

Example:

Given array nums = [1, 0, -1, 0, -2, 2], and target = 0.

A solution set is:

[

[-1, 0, 0, 1],

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

[-2, 0, 0, 2]

]

20a. Before jumping to the solution, let’s look at 3 sum problem.

The problem states that you have to print all the unique triplets that sum up to zero.

a. A naive solution is to fix any one number, and find a pair that sums up to the negative of that number.

Eg [1,2,-3,5,-7].

If we fix 1, we have 2,-3 hence [1,2,-3]

If we fix 2, we have 5,-7 hence [2,5,-7] and so on....

Now fixing the number is one thing but finding other two is simply 2 Sum problem. Here our target is negative of the fixed number. To rectify the duplicates we can use hashmap.

We can sort the array beforehand to have O(N) time to find the pairs. Overall the complexity is O(N2)

class Solution {

public:

vector<vector<int>> threeSum(vector<int>& a) {

vector<vector<int>> ans;

set<vector<int>> st;

int i,j,k,n=a.size();

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

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

{

int target = -a[i];

j= i+1;

k = n-1;

while(j<k)

{

if(a[j]+a[k]>target)k--;

else if (a[j]+a[k]<target)j++;

else{

st.insert({a[i],a[j],a[k]});

//break;

k--;

j++;

}

}

}

for(auto i:st)ans.push\_back(i);

return ans;

}

};

b. We can modify the above code, we won’t use set this time because it is taking extra space and time because afterall, we are calculating all the duplicates.

The idea is , skip all the duplicate elements after we find a valid triplet, because they will keep on adding.

class Solution {

public:

vector<vector<int>> threeSum(vector<int>& a) {

vector<vector<int>> ans;

int i,j,k,n=a.size();

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

for(i=0;i<n; )

{

int target = -a[i];

j= i+1;

k = n-1;

while(j<k)

{

if(a[j]+a[k]>target)k--;

else if (a[j]+a[k]<target)j++;

else{

ans.push\_back({a[i],a[j],a[k]});

while(j<k && a[j] == ans[ans.size()-1][1])j++;

while(j<k && a[k] == ans[ans.size()-1][2])k--;

}

}

i++;

while(i<n && a[i]==a[i-1])i++;

}

// for(auto i:st)ans.push\_back(i);

return ans;

}

};

This solution doesn’t take extra space and works in O(N2).

Now coming back to our 4 sum problem, we can use the similar approach. Fix 2 numbers, localtarget = Target – fixed1 – fixed 2, and apply the approach used for 2 sum problem.

class Solution {

public:

vector<vector<int>> fourSum(vector<int>& a, int tar) {

vector<vector<int>> ans;

int i,j,k,l,n=a.size();

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

for(l=0;l<n;){

for(i=l+1;i<n;)

{

int target = tar - a[i] - a[l];

j= i+1;

k = n-1;

while(j<k)

{

if(a[j]+a[k]>target)k--;

else if (a[j]+a[k]<target)j++;

else{

ans.push\_back({a[l],a[i],a[j],a[k]});

while(j<k && a[j] == ans[ans.size()-1][2])j++;

while(j<k && a[k] == ans[ans.size()-1][3])k--;

}

}

i++;

while(i<n && a[i]==a[i-1])i++;

}

l++;

while(l<n && a[l]==a[l-1])l++;

}

return ans;

}

};

b. We can also use recursion ie if k == 2 use 2 sum otherwise fix the current number and use an call the function for k-1. But using this approach doesn’t reduce space or time because it can take upto O(N) space.

21. Longest Consecutive subsequence: Find the length of Longest Consecutive Sequence which is defined as a[i]+1 = a[i+1] for all valid i;

a. Sort the array and count the length. Note that if we find duplicates, skip them.

class Solution {

public:

int longestConsecutive(vector<int>& a) {

int i=0,n= a.size(),ans=0,mx=1;

if(n==0)return 0;

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

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

if(a[i]+1==a[i+1])// || a[i] == a[i+1])

{

mx++;

}

else if (a[i]!=a[i+1]){

ans = max(ans,mx);

mx=1;

}

}

ans = max(ans,mx);

return ans;

}

};

b. Use a hash map to store the array. For each element x, count the size upto the point where x+1 is present in the array. After that if this chain is having greater length, make it the current answer.

In c++, set, map etc work in O(logn) so it won’t make any difference. But we can always make our own hashmap.

class Solution:

def longestConsecutive(self, nums: List[int]) -> int:

s = set(nums);ans=0

for i in s:

if i-1 not in s:

j = i+1

while j in s:

j+=1

ans = max(ans , j-i)

return ans

22. Longest subarray with sum 0.

Input: arr[] = {15, -2, 2, -8, 1, 7, 10, 23};

Output: 5

Explanation: The longest sub-array with

elements summing up-to 0 is {-2, 2, -8, 1, 7}

Input: arr[] = {1, 2, 3}

Output: 0

Explanation:There is no subarray with 0 sum

Input: arr[] = {1, 0, 3}

Output: 1

Explanation: The longest sub-array with

elements summing up-to 0 is {0}

a. Use brute force to find all ranges (i,j). This will take O(N2)time.

b. A better approach is:

Observe the prefix sum of : [15 -2 2 -8 1 7 10 23]

[15 13 15 7 8 15 25 48]

here we get a repeating sum; Why? Because the subarray between these two points is having 0 sum.

Start calculating the prefix sum.

Lets store each sum in a map of sum->index. Whenever we encounter a sum that is already present, that will mean that there is a subarray with 0 sum between this index and the index of previous sum. Hence update our answer as max(ans, differnece b/w indecies).

def longestZeroSum(l):

if len(l)==0:return 0

n = len(l)

d = dict([(l[0],0)])

ans = 0

for i in range(1,n):

l[i]+=l[i-1]

if l[i] in d:

ans = max(ans , i - d[l[i]] )

else:

d[l[i]] = i

return ans

print(longestZeroSum([1, 0, 3]))

22a. Subarray Sum Equals K

Given an array of integers and an integer k, you need to find the total number of continuous subarrays whose sum equals to k.

Example 1:

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

Output: 2 ie [1,1] and [1,1]

Input: nums = [0,0,0,0],k=0

Output: 10 ie all subarrays

a. Idea is as simple as above approach. Just like we were searching for sum = 0 ; we can also check whether there is any summation in the map that is equal to current\_sum – k. That will mean that the subarray that lies between those two points has a sum = k.

We can use this logic to check whether there exist such subarray or not. Now we need to count that how many such subarrays are present in the entire array.

Answer is simple, whenever there is sum = current\_sum – k, we can say that all those sums contribute in the answer. Hence We just need to keep an eye on the overall frequencies of all the sums, and whenever we encounter the above condition, add the frequency in our answer.

Lets take an example:

INPUT: [1,1,-1,4,2,-2] k = 1

prefix sum = [1,2,1,5,7,5]

frequency\_count = {0:1}

Note that there is key-value pair of 0->1 that is the empty subarray.

Now traverse the prefix\_sum,

i=0; sum=1, check for 1 – 1 = 0, ans+=1; fc = {0:1,1:1}

i=1, sum = 2 check for 2 – 1 = 1, ans +=1; fc = {0:1,1:1,2:1}

i=2, sum = 1 , check for 1 – 1 = 0, ans+=1; fc = {0:1,1:2,2:1}

......

i=5, sum = 5,check for 5-1 = 4, ans+=0; fc = {.....}

Hence the answer is 3.

class Solution:

def subarraySum(self, l: List[int], k: int) -> int:

if len(l)==0:return 0

if len(l) == 1: return 1 if k == l[0] else 0

n = len(l)

d = Counter([0,l[0]])

ans = 0

if k == l[0]: ans+=1

for i in range(1,n):

l[i]+=l[i-1]

if l[i] - k in d:

ans += d[l[i]-k]

d[l[i]]+=1

return ans

23. Count number of subarrays with given XOR.

a. Take two pointers i and j, generate all possible combinations => all subarrays and count the ones having the given XOR. This solution is inefficient iff N2 is too large. Complexity O(N2).

b. Just like the solution of 22a, we can take prefix XOR, and count the subarrays in the similar way.

Why it works?

Suppose the array is [a,b,c,d,e,f], and we are at 4th position for now,

If a^b^c^d^e == e^k, we will count this subarray in our answer,

Moreover, if any of the prefix XOR is having XOR = k let’s say a^b = e^k and b^c^d = e^k, then prefix XOR upto now is a^b^c^d^e, lets replace them by k, a^e^k^e, and this value is e^k ie a^k = k^e ie from b upto 3rd postion, the subarray is having XOR = k.

A proper algorithm from GFG:

i. Initialize ans as 0.

ii. Compute xorArr, the prefix xor-sum array.

iii. Create a map mp in which we store count of

all prefixes with XOR as a particular value.

iv. Traverse xorArr and for each element in xorArr

(A) If m^xorArr[i] XOR exists in map, then

there is another previous prefix with

same XOR, i.e., there is a subarray ending

at i with XOR equal to m. We add count of

all such subarrays to result.

(B) If xorArr[i] is equal to m, increment ans by 1.

(C) Increment count of elements having XOR-sum

xorArr[i] in map by 1.

v. Return ans.

class Solution:

def subarrayXORCount(self, l: List[int], k: int) -> int:

if len(l) == 0: return 0

if len(l) == 1: return 1 if k == l[0] else 0

n = len(l)

d = Counter([0,l[0]]) # We already have XOR =0 and l[0]

ans = 0

if k == l[0]: ans+=1

for i in range(1,n):

l[i]^=l[i-1]

if l[i] ^ k in d:

ans += d[l[i]^k]

d[l[i]]+=1

if l[i] == k: ans+=1

return ans

23a. XOR Queries of a Subarray

Given the array arr of positive integers and the array queries where queries[i] = [Li, Ri], for each query i compute the XOR of elements from Li to Ri (that is, arr[Li] xor arr[Li+1] xor ... xor arr[Ri] ). Return an array containing the result for the given queries.

a. I simply used segment tree here...

class Solution {

public:

int segtree[120000];

void build(vector<int>& arr , int i,int l , int r){

if(l==r)

{

segtree[i] = arr[l];

return ;

}

int mid = (l+r)/2;

build(arr,2\*i+1,l,mid);

build(arr,2\*i+2,mid+1,r);

segtree[i] = segtree[2\*i+1]^segtree[2\*i+2];

}

int query(int i,int l,int r,int left,int right)

{

if(l>right || r<left)return 0;

if(l>=left && r<=right)return segtree[i];

int mid = (l+r)/2;

int ql = query(2\*i+1,l,mid,left,right);

int qr = query(2\*i+2,mid+1,r,left,right);

return ql^qr;

}

vector<int> xorQueries(vector<int>& arr, vector<vector<int>>& queries) {

int l,r,n=arr.size();

vector<int> res;

build(arr,0,0,n-1);

// for(int i=0;i<15;i++)cout<<segtree[i]<<" ";cout<<endl;

for(auto i:queries){

// cout<<i[0]<<" "<<i[1]<<endl;

res.push\_back(query(0,0,n-1,i[0],i[1]));

}

return res;

}

};

b. The above solution works well if we have to update the array. But there is nothing like that, so simply using a prefix array will also do the trick.

class Solution {

public:

vector<int> xorQueries(vector<int>& arr, vector<vector<int>>& queries) {

int i,j=-1,n=arr.size();

vector<int> res(queries.size());

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

for(auto &i: queries){

res[++j] = i[0]>0?arr[i[0]-1]^arr[i[1]]:arr[i[1]];

}

return res;

}

};

24. Longest Substring Without Repeating Characters

Given a string, find the length of the longest substring without repeating characters.

Example 1:

Input: "abcabcbb"

Output: 3

Explanation: The answer is "abc", with the length of 3.

Example 2:

Input: "bbbbb"

Output: 1

Explanation: The answer is "b", with the length of 1.

Example 3:

Input: "pwwkew"

Output: 3

Explanation: The answer is "wke", with the length of 3.

a. This problem is easy.

Make a hashmap to keep an eye on frequency of characters. If it is guarenteed that characters are english alphabet, use a small array else use an array of size 256, initialised to 0.

Iterate from left to right, for each character increase its frequency.

If a character has already frequency>0, put the start pointer to the next to it and decrease the frequencies of all the in between characters.

Update the answer and maxsofar accordingly.

class Solution {

public:

int lengthOfLongestSubstring(string s) {

int start = 0,end = 0, maxsofar=0, ans=0;

int arr[256]={0,};

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

{

if(arr[s[i]]){

ans = maxsofar>ans?maxsofar:ans;

while(s[start]!=s[i])

{

arr[s[start]]--;

++start;

}

++start;

maxsofar = i-start+1;

}

else{

++arr[s[i]];

maxsofar++;

}

}

ans = maxsofar>ans?maxsofar:ans;

return ans;

}

};

25. Reverse a linked list.

We have done this problem many times. But beware of empty linked list ; )

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

ListNode \*p,\*q,\*r;

if (head == NULL)return NULL;

q = NULL;

r = NULL;

p = head;

while(p!=NULL)

{

r = q;

q = p;

p = p->next;

q->next = r;

}

q->next = r;

return q;

}

};

26. Find the middle of a linked list.

Given a non-empty, singly linked list with head node head, return a middle node of linked list.

If there are two middle nodes, return the second middle node.

a. Again beware of empty linked list. Use slow and fast pointers and test it on some lists of 0,1,2,3,4,5 length.

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

ListNode\* middleNode(ListNode\* head) {

ListNode \*slow, \*fast;

if (head==NULL)return NULL;

slow = fast = head;

while(fast!=NULL && fast->next!=NULL)

{

slow = slow->next;

fast = fast->next->next;

}

return slow;

}

};

27. Merge two sorted linked lists.

a. Use the merge function of mergesort.

Again beware of empty lists.

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {

ListNode \*head,\*tail,\*p,\*q;

if(l1==NULL)return l2;

if(l2==NULL)return l1;

if(l1->val<=l2->val){

head = l1;

p = l1->next;

q = l2;

}else{

head = l2;

p = l2->next;

q = l1;

}

tail = head;

while (p!=NULL && q!= NULL)

{

if(p->val<=q->val)

{

tail->next = p;

p = p->next;

}else{

tail->next = q;

q = q->next;

}

tail = tail->next;

}

while(p!=NULL)

{

tail->next = p;

p = p->next;

tail = tail->next;

}

while(q!=NULL)

{

tail->next = q;

q = q->next;

tail = tail->next;

}

return head;

}

};

28. Remove Nth Node From End of List

Given a linked list, remove the n-th node from the end of list and return its head.

Example:

Given linked list: 1->2->3->4->5, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5.

a. Nth node from end means LIST\_SIZE – N + 1st node from start. Calculate the size of the list and using two pointers, delete the node (back->next = front->next)

Also see for the edge cases like :

what if list is empty

what if Nth node from last doesn’t exist.

What if we are removing the first node.

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

ListNode\* removeNthFromEnd(ListNode\* head, int n) {

ListNode \*p,\*q;

int size\_of\_list=0;

p = head;

while(p!=NULL)

{

p = p->next;

size\_of\_list++;

}

n = size\_of\_list - n+1;

if (size\_of\_list>n)return head;

p = head;q = NULL;

int iamat=0;

while(p!=NULL)

{

iamat++;

if(iamat==n)

{

if (q == NULL)head = head->next;

else q->next = p->next;

return head;

}

q = p;

p = p->next;

}

return NULL;

}

};

29. Given a direct pointer to a node, delete that node from the list.

Example 1:

Input: head = [4,5,1,9], node = 5

Output: [4,1,9]

Explanation: You are given the second node with value 5, the linked list should become 4 -> 1 -> 9 after calling your function.

a. We can delete the node easily if we were given the head pointer too. But since it is not given, there is one more way. Copy each node->next->value to the current value. And delete the next node.

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

NOTE: This code isn’t checking the edge cases so consider them. Also it won’t work if we are dealing with the tail.

class Solution {

public:

void deleteNode(ListNode\* node) {

if(node->next!=NULL)

{

node->val = node->next->val;

node->next = node->next->next;

}

}

};

30. Adding Two numbers as a Linked List.

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example:

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)

Output: 7 -> 0 -> 8

Explanation: 342 + 465 = 807.

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) {

ListNode \*p,\*q,\*head,\*tail;

int rem = 0;

if(l1==NULL)return l2;

if(l2==NULL)return l1;

head = new ListNode();

tail = head;

p = l1; q = l2;

while(p!=NULL && q!=NULL)

{

int x = p->val + q->val + rem;

rem = x/10;

p = p->next;

q = q->next;

tail->next = new ListNode(x%10);

tail = tail->next;

}

while(p!=NULL)

{

int x = p->val + rem;

rem = x/10;

p = p->next;

tail->next = new ListNode(x%10);

tail = tail->next;

}

while(q!=NULL)

{

int x = q->val + rem;

rem = x/10;

q = q->next;

tail->next = new ListNode(x%10);

tail = tail->next;

}

if(rem>0){

ListNode \*l3 = new ListNode(rem);

tail->next = l3;

}

return head->next;

}

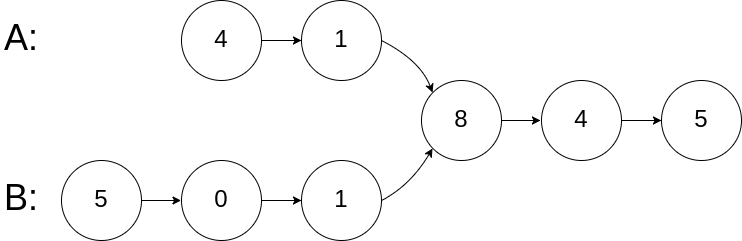
};

31. Intersection of two Linked Lists

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:

begin to intersect at node 8.



Example 1:

Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3

Output: Reference of the node with value = 8

a. A pretty clean way is to visit the list using first list’s head. Mark each node with a special value for example INT\_MIN or something... Visit the list using second list head, and the first node having this special value is the intersection point. But this approach will change the linked list data so it is not very effective.

b. Other interesting way is, consider the non-common part,

there can be two cases, either they are equal or not equal

If they are equal, running two pointers from each part will meet somewhere;

Otherwise one of them will get NULL soon.

Now check which one is NULL, reset a pointer it to other list’s head again, and run both until the other also gets NULL. Now this pointer is at a Node after which length of both lists is same. Again run two pointers and wherever they meet is the answer. If they don’t meet then the list doesn’t have an intersection point.

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

ListNode \*p,\*q,\*r,\*s;

p = r = headA;

q = s = headB;

// Make the pointers go to the last.

while(p!=NULL && q!=NULL)

{

if(p==q){ // If they meet (equal length) then return the pointer

return p;

}

p = p->next;

q = q->next;

}

// If the unmatched part is not equal in length

// then start again a pointer to equalize the length

if(p==NULL)

{

while(q!=NULL)

{

s = s->next;

q = q->next;

}

}

if(q==NULL)

{

while(p!=NULL)

{

r = r->next;

p = p->next;

}

}

// Now that the pointers have been equalized

// redo the same again. if still you don't find

// the intersection, return null

while(r!=NULL && s!=NULL)

{

if(r==s)return r;

r = r->next;

s = s->next;

}

return NULL;

}

};

32. Check if a given linked list is a palindrome or not.

Example 1: Input: 1->2 Output: false

Example 2: Input: 1->2->2->1 Output: true

a. If we can match the list from the mid point, the solution is done. Using slow and fast pointer reach to the mid point. From that point reverse the rest of the list using three pointer method. Now we can easliy match them.

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

bool isPalindrome(ListNode\* head) {

ListNode \*slow, \*fast;

slow = fast = head;

bool isodd = true;

if(head == NULL || head->next == NULL)return true;

while(true)

{

if(fast == NULL){

isodd = false;

break;

}

else if(fast->next == NULL)break;

slow = slow->next;

fast = fast->next->next;

}

if(isodd)slow = slow->next;

ListNode \*p,\*q,\*r;

p = slow;

q = NULL;

r = NULL;

while(p!=NULL)

{

r = q;

q = p;

p = p->next;

q->next = r;

}

q->next = r;

while(q!=NULL)

{

if(q->val!=head->val)return false;

q = q->next;

head = head->next;

}

return true;

}

};

33. Reverse the given linked list in K-groups

a. The solution is straight forward, we really need to do what it says in the question.

Follow these step:

1. initialise some new pointers, p,q,r to reverse the list and head, tail to keep track of the list.
2. Initially head is a new pointer, let’s call it myhead to avoid conflict.
3. Set p = head. q and r will be initialised NULL for each group.
4. Run a for loop of size K, reverse the entire list upto K nodes. Now two things may break the loop, we either hit the Kth node, or P becomes NULL.
5. If we hit the Kth node, simply put tail.next = q ie head of the reversed list and move tail to the end.
6. Otherwise we have reversed the list non intentionally so we need to restore it. To do so, again reverse the list starting from where our current head ie q is. After reversing put tail.next = new head ie q.
7. Return myhead -> next.

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

ListNode\* reverseKGroup(ListNode\* head, int k) {

ListNode \*myhead,\*tail,\*p,\*q,\*r;

if (head==NULL)return head;

myhead = new ListNode();

tail = myhead;

int i;

p = head;

while(p!=NULL){

q = NULL;

r = NULL;

for(i=0;i<k && p!=NULL;i++)

{

r = q;

q = p;

p = p->next;

q->next = r;

}

if(i==k)

{

tail->next = q;

while(tail->next!=0)

{

tail = tail->next;

}

}

else{

p = q;

q = NULL;

r = NULL;

while(p!=NULL)

{

r = q;

q = p;

p = p->next;

q->next = r;

}

q->next = r;

tail->next = q;

}

}

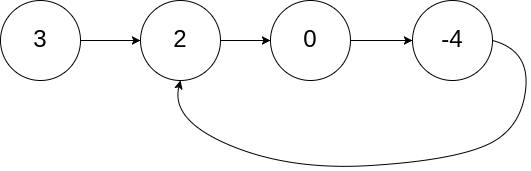
myhead = myhead->next;

return myhead;

}

};

34. Detect a loop/cycle in linkedlist.



a. We can mark each node with a special value if it is known that what are the values present in the linked list But it won’t work otherwise.. If we encounter the node again, that means there is a cycle.

This approach will modify the list so it’s not preferred.

bool hasCycle(ListNode \*head) {

ListNode \*p;

p = head;

while(p!=NULL)

{

if(p->val==INT\_MIN)return true;

p->val = INT\_MIN;

p = p->next;

}

return false;

}

b. Extending the above idea, we can actually hash the node addresses instead because addresses are unique in a machine. If we revisit a node, there is a cycle. But it will cost extra space of O(N).

c. Use Floyd’s cycle detection algorithm, using two pointers, slow and fast.

bool hasCycle(ListNode \*head) {

ListNode \*slow,\*fast;

slow = fast = head;

while(fast!=NULL && fast->next!=NULL )

{

slow = slow->next;

fast = fast->next->next;

if(slow == fast)return true;

}

return false;

}

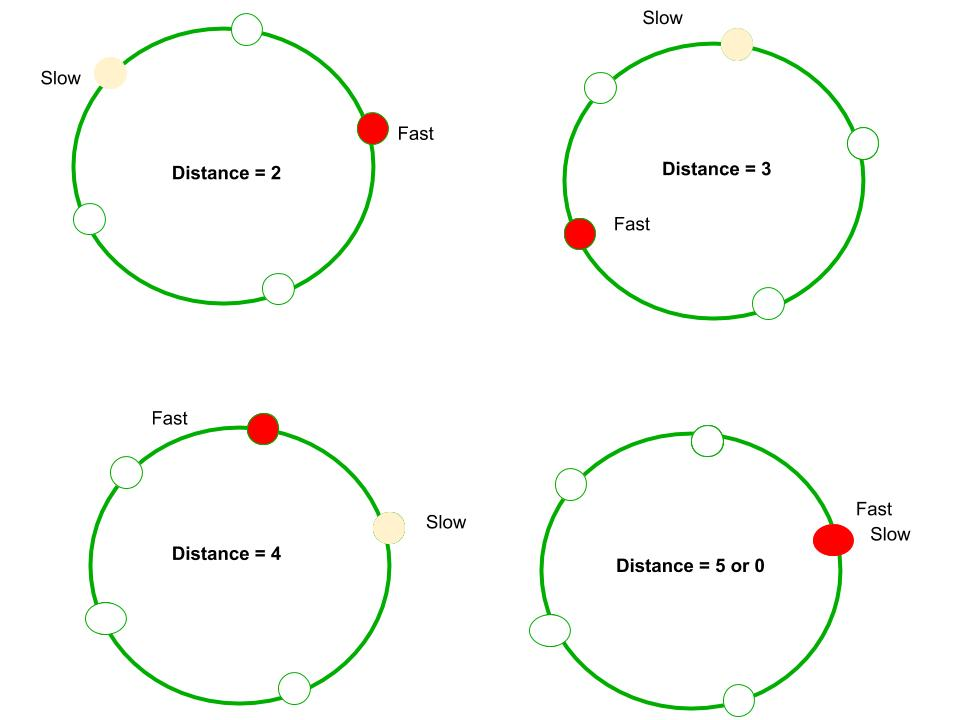
};

But why it works and how do we guarentee that the pointers will meet?

If a cycle is not present the pointers will never meet. Because faster pointer keeps on going and the slower one can’t catch upto it.

If they are in a cycle, one thing is interesting to observe:

1. Initially both pointers are at 0 distance.
2. Each iteration makes them one step apart, ie 0,1,2,...k
3. If they are in a cycle of size N, they can get atmost N step apart (in either clockwise or anticlockwise direction). Here N step in any direction means they are now at same position.
4. Since they are in same position, we can say the reverse that they are moving in a cycle, that’s all.



Refer <https://www.geeksforgeeks.org/how-does-floyds-slow-and-fast-pointers-approach-work/>

34a. Return the node from where the loop starts.

First of all check whether a cycle exists or not. If it exists, do the following:

1. put a new pointer say P on head.
2. Move slow pointer and P at same pace until slow meets fast again. This will move P to K steps where K is the length of cycle.
3. Suppose list is having N nodes and cycle is of length K. That means we need to move N-K more steps.
4. We start another pointer Q from head and move P and Q at same speed. They will meet after N-K steps; that is the start of the loop.

ListNode \*detectCycle(ListNode \*head) {

ListNode\* s , \*f, \*p,\*q;

int k=0;

if(head == NULL)return NULL;

s = f = head;

while(f!=NULL && f->next!=NULL)

{

s = s->next;

f = f->next->next;

if(s==f)break;

}

if (f==NULL || f->next==NULL)return NULL;

p = head;

do{

p = p->next;

f = f->next;

}while(s!=f);

q = head;

while(p!=q){

p=p->next;

q= q->next;

}

return p;

}

};

34b. Remove the loop in the list if it is present.

a. We can do that easily if we are following approach a or b.

b. Without extra space or without modifying the list, we have to use approach 34a and if we find a node X such that X->next = startOfLoopNode , put X->next = NULL.

35. Flattening a linked list.

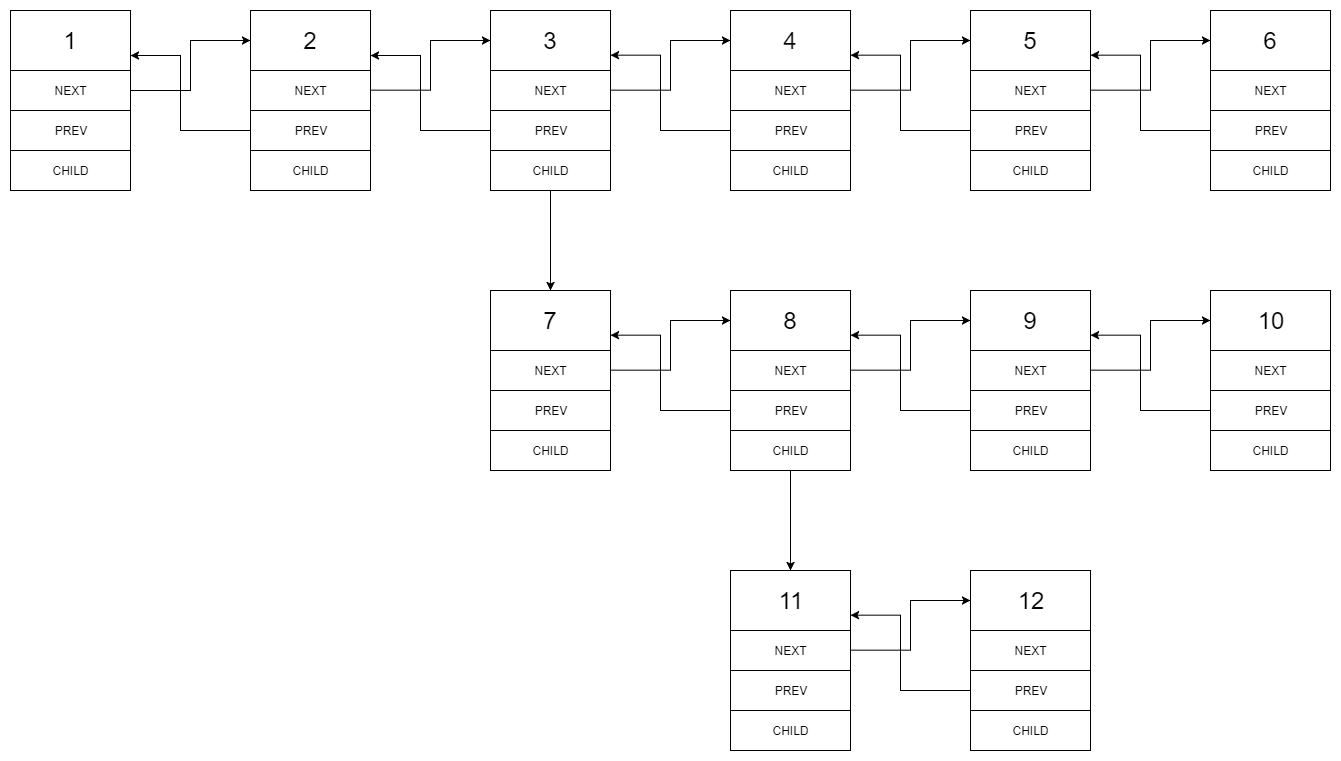
This question has multiple variants but all of them are similar in the sense that we need to append a new list (emerging out of a node) in between the current and next nodes. One variant on leetcode is about DLL where you need to maintain next and prev both pointers. The question is appended below: <https://leetcode.com/problems/flatten-a-multilevel-doubly-linked-list/>

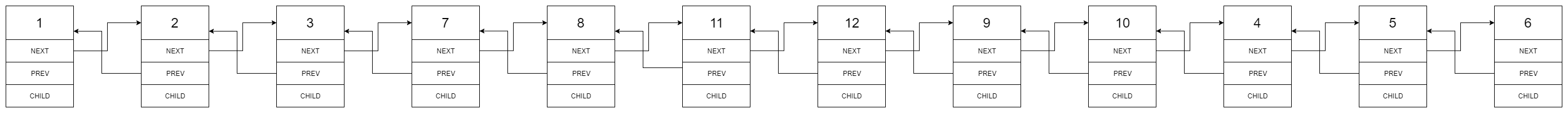
35a.Flatten a Multilevel Doubly Linked List

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

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

INPUT:

OUTPUT:



a. Use a pointer P to traverse the list. Whenever a node having a child is encountered, move a pointer to child, and one to its next. Our local goal is to merge the entire list in between these two nodes.

/\*

// Definition for a Node.

class Node {

public:

int val;

Node\* prev;

Node\* next;

Node\* child;

};

\*/

class Solution {

public:

Node\* flatten(Node\* head) {

Node \*p,\*q,\*r;

p = head;

while(p!=0)

{

if(p->child!=0)

{

q = p->next;

if(q==NULL){

p->next = p->child;

p->child->prev = p;

}

else{

r = p->child;

p->next = r;

r->prev = p;

while(r->next!=0)

{

r = r->next;

}

r->next = q;

q->prev = r;

}

p->child = 0;

}

p = p->next;

}

return head;

}

};

36. Rotate a linked list.

If you need to rotate it counter-clockwise, simply move to kth node (don’t forget to make k = n%k because rotating it n times doesn’t make any changes.) Move a pointer to kth node, one just behind it and other at the end and make proper changes.

In case you need to rotate clock wise, do the same except that move the pointer n-k times.

class Solution {

public:

ListNode\* rotateRight(ListNode\* head, int k) {

ListNode \*p,\*q,\*r;

if(head==NULL || head->next==NULL)return head;

int i,n=0;

p = head;

while(p!=NULL){

n++;

p = p->next;

}

k = k%n;

if(k==0)return head;

// This step can be done more easily so do the changes.

for(i=0,p=head;i<k-1;i++)

{

p = p->next;

}

q = head;r = 0;

while(p->next!=NULL)

{

p = p->next;

r = q;

q = q->next;

}

r->next = 0;

p->next = head;

head = q;

return head;

}

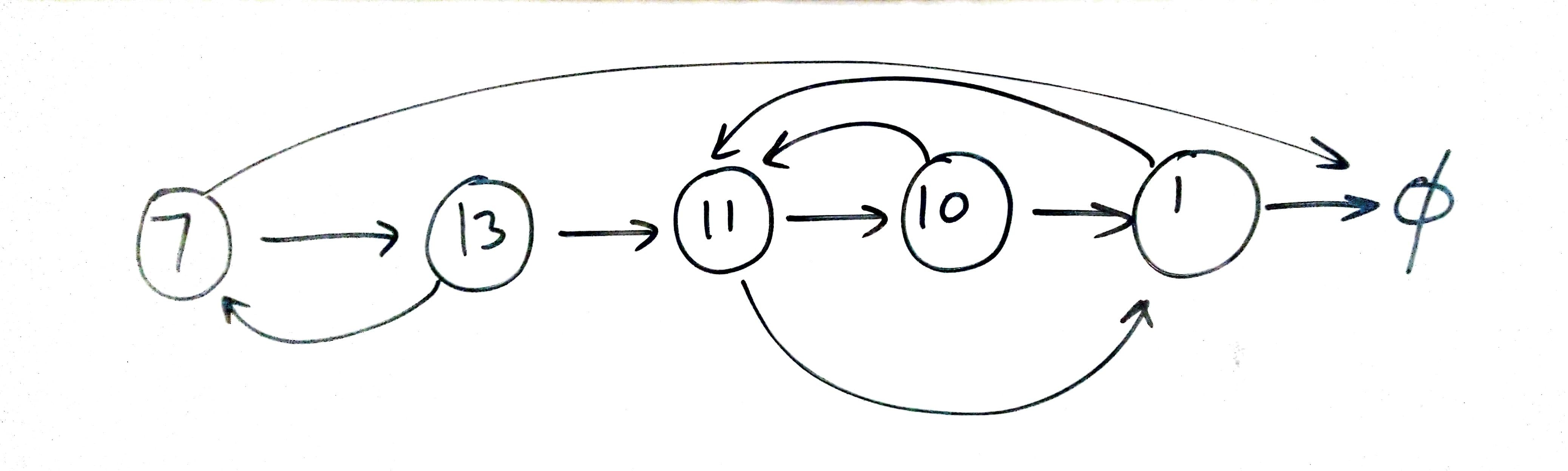
};

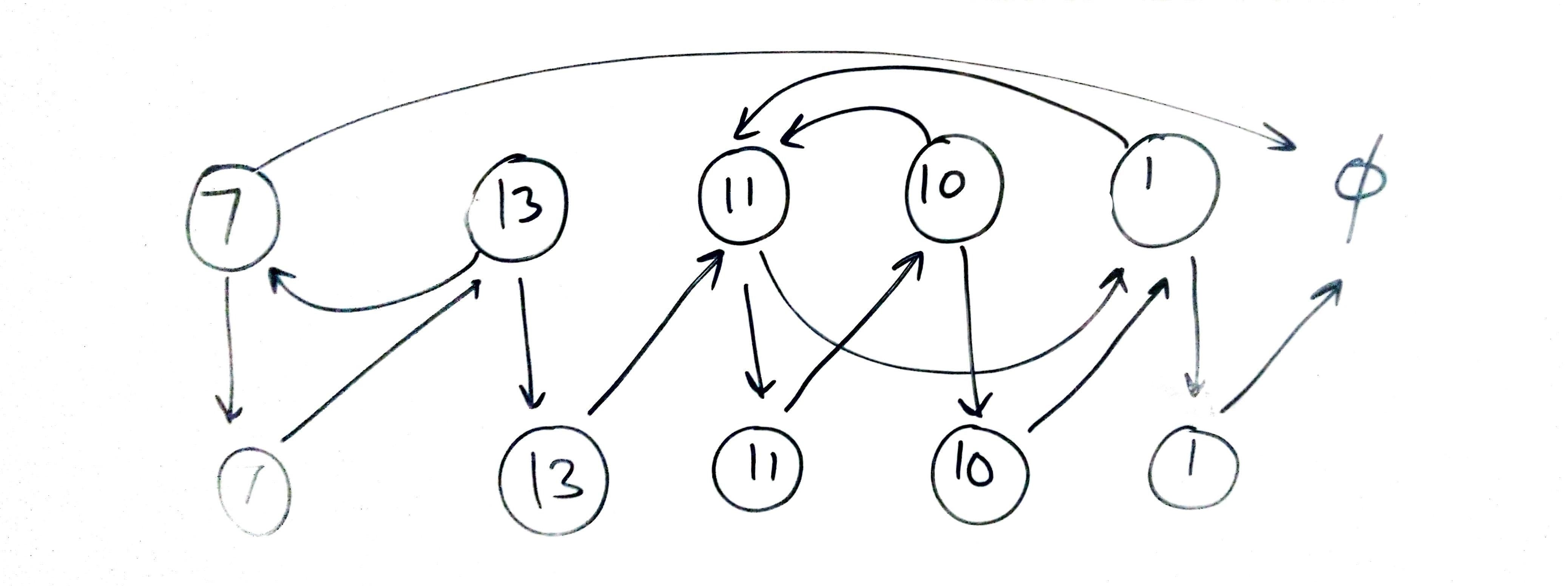
37. Clone a linked list with random and next pointer.

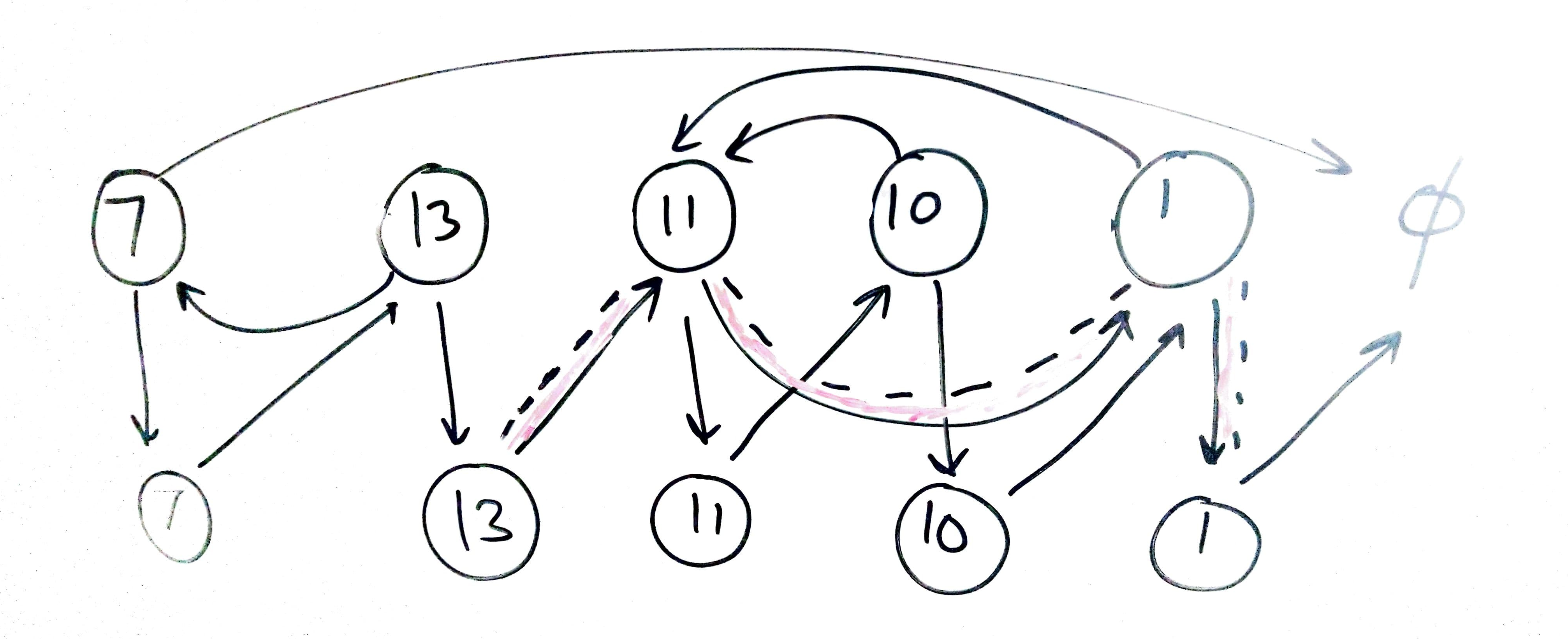
a. If we have extra space, we can use hashing to easily copy the node-node mapping.

b. The solution is as below but understanding the visualization is better.

1. Make a new list with same data. We need to put each node in A->A’->B->B’ ... pattern. Why? Because that will make it easy to store all the pointers as well as it will make it easy to access the random pointers in first list and corresponding random pointer in second list.
2. After that assign the random pointers using this interwoven list.
3. Separate this new list from the old list by deleting each node one by one and appending them to a new list.
4. Return the new list.







/\*

// Definition for a Node.

class Node {

public:

int val;

Node\* next;

Node\* random;

Node(int \_val) {

val = \_val;

next = NULL;

random = NULL;

}

};

\*/

class Solution {

public:

// This question was asked in MICROSOFT TECH SET GO round 1

// So I already know the solution....

Node\* copyRandomList(Node\* head) {

Node \* p,\*q,\*myhead,\*tail;

myhead = new Node(-1); // Initialize a new head and tail.

tail = myhead;

p = head; // initialze P

// First make a straight chain so that all nodes are copied.

while(p!=NULL){

q = new Node(p->val); // Create a node with value at P

q->next = p->next; // Save P->next reference;

p->next = q; // append this node after p

p = q->next; // move P

}

// Now nodes have been interwoven A->A'->B->B'....

// Now assign the pointers.. Visualization is the best way

p = head;

// Beware you are not trying to access NULL->next ;)

while(p!=NULL)

{

p->next->random = (p->random==NULL)?NULL:p->random->next;

p = p->next->next;

}

// Now seperate the interwoven list.

p = head;

while(p!=NULL)

{

tail->next = p->next;

tail = tail->next;

p->next = p->next->next;

p = p->next;

}

return myhead->next;

}

};

38. Merge two sorted linked lists.

39. Detect starting point of the loop.

40. 3 Sum problem.

// 38 and 39 are repeated. 40 is done with 4-Sum problem

41. Trapping Rain Water

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



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

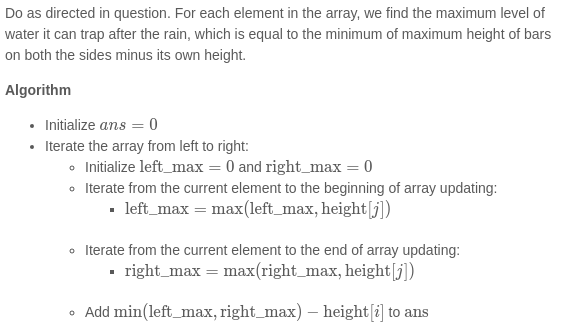
Example:

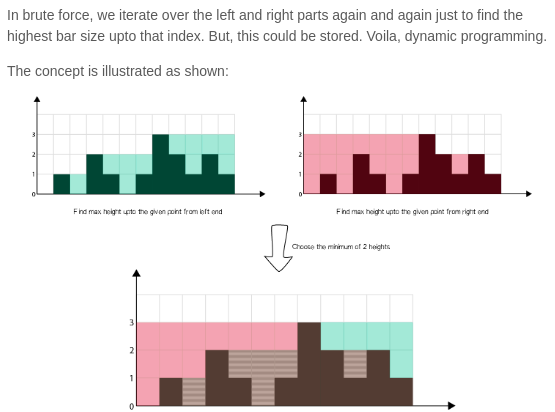
Input: [0,1,0,2,1,0,1,3,2,1,2,1]

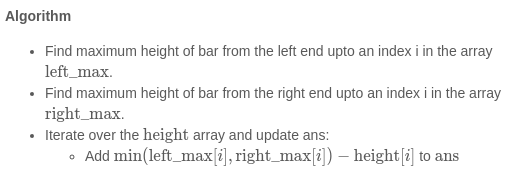
Output: 6

a. This problem is a little hard if you solve it using brute force. We have to find two walls with equal or greater height than current wall. For that we need to run two loops, for maintaining both the pointers.

The logic is we need to find the left and right walls, as left<=right wall. Suppose their are K steps from left wall to right wall. Here total area = min(left,right)\*K and if we delete the walls, whatever left is the water.



b. A better solution is :



int trap(vector<int>& height) {

int n = height.size(),i;

if(n==0)return 0;

vector<int> left(n),right(n);

left[0] = height[0];

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

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

{

left[i] = max(left[i-1],height[i]);

right[n-i-1] = max(right[n-i],height[n-i-1]);

}

int ans=0;

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

ans+= min(left[i],right[i]) - height[i] ;

}

return ans;

}

42. Remove duplicate from sorted array.

Given a sorted array A, remove the duplicates inplace and return the new length of the array.

a. Solution is simple, fix two pointers, i and j. Move j while A[i] and A[j] are equal. Let K be a third pointer that we will use for maintaining the current length of our array. Now assign this element to a pointer at the start of the array. Repeat this process and return k.

int removeDuplicates(vector<int>& a) {

int i,j,k,n = a.size();

if(n==0)return 0;

k=-1;

i=0;

while(i<n)

{

j=i+1;

while(j<n && a[j]==a[i])j++;

a[++k] = a[i];

i=j;

}

return ++k;

}

b. One more approach is :

int removeDuplicates(vector<int>& nums) {

if (nums.size() == 0) return 0;

int i = 0;

for (int j = 1; j < nums.size(); j++) {

if (nums[j] != nums[i]) {

i++;

nums[i] = nums[j];

}

}

return i + 1;

}

43. Find maximum consecutive 1’s.

a. This problem can also be solved using 2 pointers. Fix the current pointer, move the other while it is equal to 1. Update the answer with max consecutive series.

int findMaxConsecutiveOnes(vector<int>& a) {

int ans=0,i=0,j,n=a.size();

while(i<n)

{

j=i;

while(j<n && a[j]==1)j++;

ans = max(ans,j-i);

i=j+1;

}

return ans;

}

44. N meetings in one room.

I got so many variations of this problem on internet.

The main statement is, *there are N meeting times scheduled in a office, each is given in the form Si->Fi where Si is start time and Fi is finish time. Some of these times may overlap.* The variations are like these:

1. *Find the minimum number of rooms to conduct all the meetings given that a room can be reused.* [*Click Here to check out the problem*](https://www.programcreek.com/2014/05/leetcode-meeting-rooms-ii-java/)
2. *There is only one meeting room in the office, what is the maximum number of meetings that can be conducted?* [*Click here to check the problem*](https://practice.geeksforgeeks.org/problems/n-meetings-in-one-room/0)
3. *At max how many meetings can a person attend given that he can attend only one meeting at a time.*

However the solution of all these follows similar concept.

a. If we sort the meetings according to the start time, a lot of smaller meetings can be skipped in case the first meeting is too long. Hence sorting the meetings according to finish time makes sense because the earlier a meeting finishes, the shorter it is.

After that for (ii) and (iii) we need to simply find the longest non overlapping interval.

For (i) we need to find the maximum number of meetings that we have skipped, that is the number of rooms we need to conduct parellel meetings. A good data structure for fast deletion like linked list can be used in this case.

// Code for 44(ii)

#include<bits/stdc++.h>

using namespace std;

struct meeting{

int s,f,pos;

};

vector<meeting> a;

bool comp(const meeting& p1 , const meeting& p2)

{

if(p1.f==p2.f)return p1.s<p2.s;

return p1.f<p2.f;

}

vector<int > ans;

int main()

{

int t,n,i,l,r;

cin>>t;

while(t--)

{

a.clear();

cin>>n;

for(i=0;i<n;i++) // input provides start times in one line

{ // and finish times in other line.

cin>>l;

a.push\_back({l,0,i+1});

}

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

{

cin>>r;

a[i].f= r;

}

sort(a.begin(),a.end(),comp); // Sort the meetings.

ans.clear();

r = a[0].f;

ans.push\_back(a[0].pos); // We will attend the first

for(i=1;i<a.size();++i) // meeting

{

if(r<=a[i].s){ // After that each meeting that doesn’t

ans.push\_back(a[i].pos);// Overlap must be pushed.

r = a[i].f; // Update the new boundary

}

}

for(i=0;i<ans.size();++i)cout<<ans[i]<<" ";

cout<<endl;

}

}

45. Activity selection problem.

The problem statement differs but the concept is what we used in problem 44(ii).

Problem Statement is: *You are given N activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.*

46. Greedy way to exchange minimum number of coins.

Given a set of available denominations, you need to find the minimum number of coins to exchange the given amount.

a. We can solve this problem by sorting the denominations in decreasing order and fulfilling the amount using largest denomination first.

int coinChange(vector<int>& coins, int amount) {

int i,n=coins.size(),ans=0;

if(n==0)return -1;

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

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

{ //cout<<amount<<" "<<coins[i]<<endl;

if(amount>=coins[i]){

ans+=amount/coins[i];

amount -= coins[i] \* (amount/coins[i]);

}

if(amount==0)return ans;

}

return -1;

}

Although this solution is proved to be wrong for many test cases because greedy algorithm is a poor choice for this problem.

47. Fractional Knapsack problem.

*Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.*

The optimal solution is to take the maximum possible item having largest value/weight ratio. If this whole item is taken, move to the other one having the highest value.

struct item{

int value,weight;

};

bool comp(const item& a , const item& b){return 1.0\*a.value/a.weight>1.0\*b.value/b.weight;}

double fractionalKnapsack(vector<item>& a , int W)

{

int i,n=a.size();

double ans=0;

sort(a.begin(),a.end(),comp);

//for(i=0;i<n;i++)cout<<a[i].value<<" "<<a[i].weight<<" ";cout<<endl;

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

if(a[i].weight<=W) {

ans+=a[i].value;

W-=a[i].weight;

} else {

ans += (double)W \* ((double)a[i].value/(double)a[i].weight);

W = 0;

}

if(W==0)return ans;

}

return ans;

}

48. Minimum number of platforms required for a railway.

Given arrival and departure times of all trains that reach a railway station, find the minimum number of platforms required for the railway station so that no train waits. We are given two arrays which represent arrival and departure times of trains that stop.

Examples:

Input: arr[] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}

dep[] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}

Output: 3

There are at-most three trains at a time

(time between 11:00 to 11:20)

a. Suppose the times are given in string format. If they were in integer form, there is no problem otherwise we can convert them to an equivalent integer.

A. If array size is too big convert 9:00 to 900, 11:20 to 1120 ... etc

B. Otherwise sort the array and from the beginning use mapping like 9:00 =1 , 9:10 = 2 and so on..

We can use prefix sum for ranges to solve the problem, the time range at which most of the trains arrive, is our answer.

1. Compute the prefix array by putting MAP[arr[i]] += 1 and MAP[dep[i]+1] -= 1.
2. Compute the prefix sum.
3. Find the maximum and that’s the answer.

49. Job sequencing problem.

Given a set of N jobs where each job i has a deadline and profit associated to it. Each job takes 1 unit of time to complete and only one job can be scheduled at a time. We earn the profit if and only if the job is completed by its deadline. The task is to find the maximum profit and the number of jobs done.

a. Using greedy strategy, we sort the Jobs on the basis of their profit. After that we will try to place each of them at the extreme end(near to their deadline). If two jobs have same profit, we will put the job having nearer deadline first.

struct Job // from gfg

{

char id; // Job Id

int dead; // Deadline of job

int profit; // Profit if job is over before or on deadline

};

// This function is used for sorting all jobs according to profit

bool comparison(Job a, Job b)

{

return (a.profit > b.profit);

}

// Returns minimum number of platforms reqquired

void printJobScheduling(Job arr[], int n)

{

// Sort all jobs according to decreasing order of profit

sort(arr, arr+n, comparison);

int result[n]; // To store result (Sequence of jobs)

bool slot[n]; // To keep track of free time slots

// Initialize all slots to be free

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

slot[i] = false;

// Iterate through all given jobs

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

{

// Find a free slot for this job (Note that we start

// from the last possible slot)

for (int j=min(n, arr[i].dead)-1; j>=0; j--)

{

// Free slot found

if (slot[j]==false)

{

result[j] = i; // Add this job to result

slot[j] = true; // Make this slot occupied

break;

}

}

}

// Print the result

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

if (slot[i])

cout << arr[result[i]].id << " ";

}

50. N Queens Problem.

*You are given an NxN chessboard and N Queens. Your task is to place exactly N Queens on the chess board in a way that no Queen can kill any other Queen in one move.*

a.

1. A Queen can attack in 8 directions, N,E,S,W,NE,NW,SE,SW.
2. We must place each Queen one by one, checking for where it is violating the above conditions.
3. If any of the conditions is violated, we must backtrack.

See the code below, it’s enough to understand the solution.

class Solution {

public:

bool isSafe(vector<string>& board, int row, int col){

// Check whether placing a Queen here is safe or not?

int i,j;

// Check in N direction

for(i=row-1;i>=0;i--)

if(board[i][col]=='Q')

return false;

// Check in NE direction

for(i=row-1,j=col-1;i>=0&&j>=0;i--,j--)

if(board[i][j]=='Q')

return false;

// Check in NW direction

for(i=row-1,j=col+1;i>=0&&j<board.size();i--,j++)

if(board[i][j]=='Q')

return false;

// If all clear, board is safe

return true;

}

void solve(vector<string>& board, int i,int n,vector<vector<string>>& ans){

// If we filled n rows, push the current board

// and backtrack for more solutions

if(i>=n){

ans.push\_back(board);

return;

}

// Now for the current row ie 'i', try to place

// a Queen in a column if it is safe.

// After that remove it because we can't place more

// than one Queen in a row.

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

{

if(isSafe(board,i,j)){ // Is it safe?

board[i][j] = 'Q'; // If yes, place a queen

solve(board,i+1,n,ans); // Now check for next row

board[i][j] = '.'; // remove the Queen

}

}

}

vector<vector<string>> solveNQueens(int n) {

// We have to return every possible solution of this problem

// Answer is a 3-D grid having multiple 2-D grids

// Each row is represented by a string

// (.) means blank and Q means there is a queen

vector<vector<string>> ans;

// Prepare the board

vector<string> board;

string s ="";

int i,j;

for(i=0;i<n;i++)s+=".";

for(i=0;i<n;i++)board.push\_back(s);

// Board prepared! Now solve the problem,

// here I am passing the ans vector so that

// I can push each board whenever I reach

// to a solution.

solve(board,0,n,ans);

return ans;

}

};

51. Solving Sudoku using backtracking

You are given a *valid* sudoku grid, with some prefilled numbers, you have to fill the empty spaces using digits 0-9 such that 3 sudoku rules: i. Column ii. Row iii. The 3x3 grid; must contain unique digits only.

a. For each move we check whether it is safe to play or not. Go through the following code to understand the process.

class Solution {

public:

// Fantastic!! I didn't imagine that I would get this in single

// shot. Sudoku is easier than N Queen!!!

// Function to check for safe move.

bool isSafe(vector<vector<char>>& board , int r , int c , char num)

{

int i,j;

// Check for entire Row

for(i=0;i<9;i++)if(board[i][c]==num)return false;

// Check for entire column

for(i=0;i<9;i++)if(board[r][i]==num)return false;

// Shift to the start of current 3x3 grid

r = (r/3)\*3;

c = (c/3)\*3;

// Check the entire 3x3 grid

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

for(j=0;j<3;j++)

if(board[r+i][c+j]==num)

return false;

// Otherwise it is safe

return true;

}

bool solve(vector<vector<char>>& board, int r, int c)

{

// If you have successfully reached to the end

// return true

if(r==9 || c==9){ // c=9 will never occur

return true;

}

// If this row, col already has a number,

// skip it and go for the next one

if(board[r][c]!='.'){

return c==8?solve(board,r+1,0):solve(board,r,c+1);

}

// Otherwise try each digit one by one and

// check for next non empty row,col

for(int d=1;d<10;d++) // iterate over 1 to 9

{

if(isSafe(board, r, c, '0'+d)){ // Is putting this number here safe?

board[r][c] = '0'+d; // if yes, put the number

bool isright = c==8?solve(board,r+1,0):solve(board,r,c+1); // Now go for the next

if(isright)return true; // If you have solved the puzzle, congrats

board[r][c] = '.'; // Otherwise erase the number and go for other number

}

}

// Since none of the above worked, backtrack.

return false;

}

void solveSudoku(vector<vector<char>>& board) {

solve(board,0,0);

}

};

52. M Coloring Problem

*Given an undirected graph and a number m, determine if the graph can be coloured with at most m colours such that no two adjacent vertices of the graph are colored with the same color. Here coloring of a graph means the assignment of colors to all vertices.*

The solution is more simple than the Sudoku Puzzle. But before starting the coding, let’s understand few points:

1. The problem asks whether it is possible to color the graph using ‘m’ colors or not? We can also get a variation, find the smallest value of ‘m’ to color a given graph.
2. We can use adjacency list for representing the graph but it becomes complex for determining the last node of graph. So I will use adjacency matrix instead.
3. Colors can range from 1 to N, the number of nodes, so we have to determine an upper limit. Note that every graph has a solution if ‘m’ is not restricted.

I have solved this problem in such a way that if you need to tell the smallest value of M, you can find it in assignedcolors[]. If you want to determine whether it is possible to color the graph or not, you can tell that also. The problem link is [Here](https://practice.geeksforgeeks.org/problems/m-coloring-problem/0).

Below is the implementation:

#include<bits/stdc++.h>

using namespace std;

// Assign it something like N

#define MAX\_COLOR 50

// Check whether coloring currentNode with this color is safe or not?

bool isSafe(vector<vector<bool>>& graph, vector<int>& assignedcolors, int color, int V, int n){

for(int i=0;i<n;i++) // Traverse the adjacent neighbours

if(graph[V][i])

if(assignedcolors[i]==color) // Any of them has this color then it's not safe

return false;

return true;// Otherwise it is safe.

}

bool solve(vector<vector<bool>>& graph, vector<int>& assignedcolors, int currentVertex, int n)

{

if(currentVertex == n) // We have colored all nodes

{

return true;

}

for(int color = 0;color<MAX\_COLOR; color++){ // Choose a color

if(isSafe(graph, assignedcolors, color, currentVertex,n)){ // Check if it safe to fill this color

assignedcolors[currentVertex] = color; // Try to fill

bool solved = solve(graph, assignedcolors ,currentVertex+1,n); // Check is it solvalble now?

if(solved)return true; // If solved, congrats

assignedcolors[currentVertex] = -1; // Else remove this color

}

}

return false; // I guess this statement won't get executed.

}

void clearGraphAndColors(vector<vector<bool>>& graph, vector<int>& assignedcolors)

{

for(int i=0;i<50;i++){

for(int j=0;j<50;j++)

{

graph[i][j] = false;

}

}

for(int i=0;i<50;i++){

assignedcolors[i] = -1;

}

}

int main()

{

int t,n,m,i,u,v,e;

// It is given in constraints that N <=50

vector<vector<bool>> graph(50,vector<bool>(50,false));

vector<int> assignedcolors(50);

cin>>t;

while(t--){

clearGraphAndColors(graph,assignedcolors);

cin>>n; // Number of nodes

cin>>m; // Value of M

cin>>e; // Number of edges

for(i=0;i<e;i++){

cin>>u>>v;

--u;--v;

graph[u][v] = graph[v][u] = true;

}

bool ans = solve(graph, assignedcolors, 0,n);

if(ans==false){

cout<<0<<endl;

}else{

int mx=-1;

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

mx = max(mx,assignedcolors[i]);

}

if(mx<m){

cout<<1<<endl;

}else{

cout<<0<<endl;

}

}

}

return 0;

}

53. Rat in the Maze Problem

Consider a rat placed at (0, 0) in a square matrix m[ ][ ] of order n and has to reach the destination at (n-1, n-1). The task is to find a sorted array of strings denoting all the possible directions which the rat can take to reach the destination at (n-1, n-1). The directions in which the rat can move are ‘U'(up), ‘D'(down), ‘L’ (left), ‘R’ (right).

In this problem, we need to check for the safe move as: a. Don’t visit the already visited cells and b. this cell shouldn’t be a wall.

// I will add the code later

54. Print all the permutations of a string/ array.

a. Recursively call for each index. Try each element at that index.

void solve(vector<int>& a, int index, vector<bool>& visited, vector<int>& res, vector<vector<int>>& ans){

// It may seem like I am using too much arguments but I am only using references to the vectors

// Plus I am using 0 global variables i.e. the code is reusable and readable.

// Here if I use global variables, I will need only index ie one variable.

if(index == a.size()) { // Congrats you reached at the end.

ans.push\_back(res); // Push the current permutation

return;

}

// Start using each number of the array at current index

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

if(visited[i]==false){

res[index] = a[i]; // Put this element

visited[i] = true; // mark it visited

solve(a, index+1, visited, res, ans); // Solve for next index

res[index] = INT\_MIN; // Erase the element

visited[i] = false; // Mark it unvisited

}

}

}

vector<vector<int>> permute(vector<int>& a) {

int i,n =a.size();

vector<vector<int>> ans;

vector<bool> visited(n,false);

vector<int> res(n,INT\_MIN);

solve(a,0,visited,res,ans);

return ans;

}

55. Word Break.

This problem is interesting. You are given a non empty word S and a dictionary of other words D.

Your task is to determine whether it is possible to form the string S using the words of Dictionary D or not? Or in other words, is it possible to break the string S in such a way that every substring is in the dictionary?

See some examples from leetcode:

Example 1:

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

Output: true

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

Example 2:

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

Output: true

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

Note that you are allowed to reuse a dictionary word.

Example 3:

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

Output: false

a. I wrote a recursive solution with proper comments below, you should go through it:

class Solution {

public:

// I couldn't come up with a backtracking solution so I searched it on internet

// I found a video here https://www.youtube.com/watch?v=WepWFGxiwRs explaining

// the dynamic programming approach. I understood the recursive idea of his approach

// Let's take first example as it is easy.

// s = "leetcode", wordDict = ["leet", "code"]

// We need to break the string at different points and check whether both belong to a

// dictionary or not

// for example break "leetcode" as {"l eetcode", "le etcode","lee tcode"...,"leetcod e"}

// We will recursively check whether both of them exist in the dictionary or not,

// Or if they can be made up of a existing word in our dictionary

// For example we broke it as "leet code" since leet and code both are in the

// dictionary we return true

// Otherwise we will break leet and code into parts as "l eet", "le et",... and

// "c ode","co de",... and when we get a partition such that each part exist in

// dictionary, we return true.

// It takes the substring, and returns whether its parts are in dictionary or not

bool solve(string& s , int l , int r, set<string>& dict){

string S = s.substr(l,r-l+1); // The substring s[l:r+1] in Python3

if(dict.find(S)!=dict.end())return true; // If it exists return true

else if(l==r) return false; // It can't be broken anymore so sorry...

for(int i=l;i<r;i++){ // Make breaking points,

bool x = solve(s, l, i, dict) && solve(s,i+1,r,dict); // Check whether the breaking

// point breaks in two valid strings or not

if(x)

return true;

}

return false;

}

bool wordBreak(string s, vector<string>& wordDict) {

set<string> dict;

for(auto i:wordDict){

dict.insert(i);

}

return solve(s, 0,s.length()-1, dict);

}

};

b. But the above solution won’t work for larger strings because it takes too much stack space and checks for all possible strings. But it contains overlapping sub-problems( l and r are getting repeated many times) so we can turn it into dynamic programming.

The memoization approach for the above approach is as following, so go through the changes:

bool solve(string& s , int l , int r, set<string>& dict,vector<vector<int>>& dp){

if(dp[l][r]!=0)return dp[l][r]==1?true:false;

string S = s.substr(l,r-l+1); // The substring s[l:r+1] in Python3

if(dict.find(S)!=dict.end()){

dp[l][r] = 1;

return true;} // If it exists return true

else if(l==r) {

dp[l][r] = 2;

return false;} // It can't be broken anymore so sorry...

for(int i=l;i<r;i++){ // Make breaking points,

bool x = solve(s, l, i, dict,dp) && solve(s,i+1,r,dict,dp); // Check whether the breaking

// point breaks in two valid strings or not

if(x) {

dp[l][r] = 1;

return true;

}

}

dp[l][r] = 2;

return false;

}

bool wordBreak(string s, vector<string>& wordDict) {

set<string> dict;

vector<vector<int>> dp(s.length(), vector<int>(s.length(), 0));

for(auto i:wordDict){

dict.insert(i);

}

return solve(s, 0,s.length()-1, dict,dp);

}

One interesting thing that this solution got accepted on leetcode with 10% and 20% (time& space) which is greater than the dynamic programming solution.

c. Now convert it to pure dynamic programming:

bool wordBreak(string s, vector<string>& wordDict) {

set<string> dict;

vector<vector<bool>> dp(s.length(), vector<bool>(s.length(), false));

for(auto i:wordDict){

dict.insert(i);

}

int i,j,n=s.length();

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

if(dict.find(s.substr(i,1))!=dict.end()){

dp[i][i] = true;

}

}

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

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

if(dict.find(s.substr(j,i+1))!=dict.end()) {

dp[j][j+i] = true;

}else{

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

if(dp[j][j+k]&dp[j+k+1][j+i]){

dp[j][j+i]= true;

break;

}

}

}

}

}

return dp[0][n-1];

}

55a. Modify the above solution and print all words that build the string. Also print all possible ways.

// Will do soon.

56. Combination Sum-I

Given a set of candidate numbers (candidates) (without duplicates) and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target.

The same repeated number may be chosen from candidates unlimited number of times

Note:

All numbers (including target) will be positive integers.

The solution set must not contain duplicate combinations.

Example 1:

Input: candidates = [2,3,6,7], target = 7,

A solution set is:

[

[7],

[2,2,3]

]

Example 2:

Input: candidates = [2,3,5], target = 8,

A solution set is:

[

[2,2,2,2],

[2,3,3],

[3,5]

]

a. Below is the implementation:

class Solution {

public:

// Backtracking is so easy... I just had to do some practice.

// last argument 'i' is taken to avoid using duplicate sets.

// This way we will only take new elements in single direction.

void solve(int target, int current\_sum, vector<int>& temp, vector<vector<int>>& ans,vector<int>& candidates, int i){

if(current\_sum == target){ // Target achieved

ans.push\_back(temp);

return;

}

for(;i<candidates.size();i++) // Start with current element, take each candidate into consideration

{

if(current\_sum+candidates[i]<=target){ // Safe move condition

temp.push\_back(candidates[i]); // push this element

solve(target, current\_sum+candidates[i], temp,ans,candidates,i);

temp.pop\_back(); // erase it

}

}

}

vector<vector<int>> combinationSum(vector<int>& candidates, int target) {

vector<vector<int>> ans;

vector<int> temp;

solve(target,0,temp,ans,candidates,0);

return ans;

}

};

57. Combination Sum II

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target.

Each number in candidates may only be used once in the combination.

Note:

All numbers (including target) will be positive integers.

The solution set must not contain duplicate combinations.

Example 1:

Input: candidates = [10,1,2,7,6,1,5], target = 8,

A solution set is:

[

[1, 7],

[1, 2, 5],

[2, 6],

[1, 1, 6]

]

Example 2:

Input: candidates = [2,5,2,1,2], target = 5,

A solution set is:

[

[1,2,2],

[5]

]

a. This question differs from the previous one because we can’t use an element more than once. To avoid that, in solve() function, I have passed the next index of the candidate array instead of the current element.

Now one more thing that there can be duplicates now. In the previous question, we were allowed to use the cuurent element as many time as we wanted and so the candidates were duplicate. But now we can’t use the same element. Eg candidates are [1,7,7,1] and target is 8, with previous approach, we will get [1,7][1,7][7,1][7,1] and all four are duplicate. Using set is a bad idea for vectors. Hence we must use something different. Do you remember how we removed duplicates in 3-sum and 4-sum problem? Yes we will use that approach here too. First sort the candidates and then use each candidate only once at a position.

Again take the candidates = [1,1,7,7] and target = 8

push 1, push 1, push 7 is invalid [1,1], backtrack

push 1, push 7 is valid [1,7], backtrack

now remove 7 [1]

now skip the elements till they are 7

since no more elements are there, we can not push, so backtrack

remove 1 []

skip all 1’s...

push 7 [7]

push 7 is invalid move so backtrack...

We can’t generate any more combinations, so answer is [1,7]

Below is the implementation:

class Solution {

public:

// Backtracking is so easy... I just had to do some practice.

void solve(int target, int current\_sum, vector<int>& temp, vector<vector<int>>& ans,vector<int>& candidates, int i){

if(current\_sum == target){

ans.push\_back(temp);

return;

}

for(;i<candidates.size();i++)

{

if(current\_sum+candidates[i]<=target){

temp.push\_back(candidates[i]);

solve(target, current\_sum+candidates[i], temp,ans,candidates,i+1);

temp.pop\_back();

}

while(i+1<candidates.size() && candidates[i]==candidates[i+1])++i;

}

}

vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {

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

vector<vector<int>> ans;

vector<int> temp;

solve(target,0,temp,ans,candidates,0);

return ans;

}

};

58. Palindrome Partitioning

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

Example:

Input: "aab"

Output:

[

["aa","b"],

["a","a","b"]

]

a. Since we need to print all possibilities, backtracking is again a good choice. However, one more variation of this question is about finding the minimum number of partitions to get all palindrome substrings. That can be done using a recursive strategy similar to chain matrix multiplication.

In this question, the concept is, we will try to use the substrings of length 1, 2, 3 ... |S|. If these strings are palindrome, we will recursively check for the next portion of the string. Whenever we see that a partition perfectly fits, we need to print that solution and backtrack. Also if a partition overfits, we need to terminate and backtrack.

class Solution {

public:

bool isPalindrome(string& s, int l, int r){

while(l<=r){

if(s[l++]!=s[r--])

return false;

}

return true;

}

void solve(string& s, int l,vector<string>& temp, vector<vector<string>>& ans){

if(l==s.length()){

ans.push\_back(temp);

return;

}else if(l>s.length()){

return;

}

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

if(l+i-1<s.length() && isPalindrome(s,l,l+i-1)){

temp.push\_back(s.substr(l,i));

solve(s,l+i,temp,ans);

temp.pop\_back();

}

}

}

vector<vector<string>> partition(string s) {

vector<vector<string>> ans;

vector<string> temp;

solve(s,0,temp,ans);

return ans;

}

};

59. Subset Sum

Given a non-empty array containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

a. Use backtracking as follows:

class Solution {

public:

// here i is the index of next element

// we are taking the element and erasing it while backtracking

bool solve(int& target, int sum, vector<int>& a, int i){

if(target==sum){

return true;

}

else if(sum>target) return false;

for(;i<a.size();i++){

if(sum+a[i]<=target){

bool x = solve(target,sum+a[i],a,i+1);

if(x)

return true;

}

}

return false;

}

bool canPartition(vector<int>& a) {

int target=0;

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

target+=a[i];

}

if (target&1)return false;

target = target/2;

return solve(target, 0, a, 0);

}

};

b. Another backtracking solution which is more efficient is as follows:

class Solution {

public:

bool solve(vector<int>& a, int sum, int i) {

if(i==a.size())return false;

if(sum == a[i]) return true;

if(sum < a[i]) return false;

return solve(a,sum-a[i],i+1) || solve(a,sum,i+1);

}

bool canPartition(vector<int>& a) {

int target=0;

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

target+=a[i];

}

if (target&1)return false;

sort(a.rbegin(),a.rend());

target = target/2;

return solve(a,target,0);

}

};

c. // Try using 0-1 Knapsack approach // Will update soon

60. K-th permutation sequence

The set [1,2,3,...,n] contains a total of n! unique permutations.

By listing and labeling all of the permutations in order, we get the following sequence for n = 3:

"123"

"132"

"213"

"231"

"312"

"321"

Given n and k, return the kth permutation sequence.

Note:

Given n will be between 1 and 9 inclusive.

Given k will be between 1 and n! Inclusive.

a. We can generate all possible permutations and return the kth.

class Solution:

def getPermutation(self, n: int, k: int) -> str:

return "".join(map(str,list(permutations(range(1,n+1)))[k-1]))

b. Remember “Excel sheet column number” problem? Our solution will be similar to that.

Observe the following permutations for n=4

0 (1, 2, 3, 4)

1 (1, 2, 4, 3)

2 (1, 3, 2, 4)

3 (1, 3, 4, 2)

4 (1, 4, 2, 3)

5 (1, 4, 3, 2)

6 (2, 1, 3, 4)

7 (2, 1, 4, 3)

8 (2, 3, 1, 4)

9 (2, 3, 4, 1)

10 (2, 4, 1, 3)

11 (2, 4, 3, 1)

12 (3, 1, 2, 4)

13 (3, 1, 4, 2)

14 (3, 2, 1, 4)

15 (3, 2, 4, 1)

16 (3, 4, 1, 2)

17 (3, 4, 2, 1)

18 (4, 1, 2, 3)

19 (4, 1, 3, 2)

20 (4, 2, 1, 3)

21 (4, 2, 3, 1)

22 (4, 3, 1, 2)

23 (4, 3, 2, 1)

First position takes each integer 6 times, where 6 is 3!. If we fix the first position, the first position of rest array takes each integer 2 times that is 2!, and similarly 1! for 3rd position.

Suppose we want to know the 17th permutation.

So initially A = [1,2,3,4]

element at first position(0th index) = 17//(3!) = 2nd element ie 3.

So we put 3 at front. A = [3,1,2,4]

Again, we have rest of the array as [1,2,4] and we have generated the 12th permutation .

Now about [1,2,4] we need to generate 17-12 (or 17%(3!))= 5th permutation.

So 5/(2!) = 2, ie element at 2 index = 4

So array becomes [3,4,1,2].

Now for rest [1,2] we need to generate 5%(2!) = 1st permutation

1/(1!) = 1. Hence third postion goes to element at 1st index ie 2 (in [1,2] 1 is at 0th index, 2 is at 1st index)

So array becomes [3,4,2,1], that’s the answer.

Below is the implementation in python3:

def getPermutation(self, n: int, k: int) -> str:

k-=1

l = [str(i) for i in range(1,n+1)]

f = factorial(n-1)

ans = ""

while k>0:

ans += str(l[k//f])

l.remove(l[k//f])

k = k%f

f//=len(l)

for i in l:

ans+=str(i)

return ans

61. Nth root of a number.

This problem has many variations, like:

1. Find nth root of x if root(x,n) is a whole number. Else return -1
2. Find the closest integer of root(x,n). ie root(4,2) = 2 and root(8,2) = 2.
3. Given an error measure E, find the root (x,n) given that atmost ±E error is tolerable.

The idea is to use Binary search over an interval [0,x/2].

Code for First:

int root(int x, int n){

if(x==0 || x==1) return x;

int start=0,end = x/2,mid;

double m;

while(start<=end){

mid = (start+end)/2;

m = pow(mid , n);

if(m==x){

return mid;

}else if(m>x){

end = mid-1;

}else{

start = mid+1;

}

}

return -1;

}

But if we need to return the closest integer, we return the value of ‘end’ instead of -1.

int root(int x, int n){

if(x==0 || x==1) return x;

int start=0,end = x/2,mid;

double m;

while(start<=end){

mid = (start+end)/2;

m = pow(mid , n);

if(m==x){

return mid;

}else if(m>x){

end = mid-1;

}else{

start = mid+1;

}

}

return end;

}

For the third variation, we need to use double/float datatype.

We have to make some changes in the binary search function too:

def root(x,n,e):

if(x==0 or x==1 or n==1):return x

start = 0

end = x/2

while start<end:

mid = (start+end)/2

m = mid\*\*n

if x-e <= m<= x+e:

return mid

elif m>x:

end = mid # Not mid-1 because we don’t know what fraction to reduce

else:

start = mid

return start # loop breaks when start <= end, that is the most accurate answer

62. Find median in Matrix

Given a matrix with N rows and M columns, with each row sorted in non decreasing order. Your task is to find the median of all elements in this matrix. It is given that N\*M is odd(ie median is an integer, not average of two middle elements.)

Example:

Input : 1 3 5

2 6 9

3 6 9

Output : Median is 5

If we put all the values in a sorted

array A[] = 1 2 3 3 5 6 6 9 9, middle element is 5

Input: 1 3 4

2 5 6

7 8 9

Output: Median is 5

a. The idea is that for a number to be median there should be exactly (n/2) numbers which are less than this number.

So we need to find the max and min of this matrix and apply binary search for a number such that there are exactly r\*c/2 numbers smaller than it. Below is the code:

# Function to find median in the matrix

def binaryMedian(m, r, d):

mi = m[0][0]

mx = 0

for i in range(r):

if m[i][0] < mi:

mi = m[i][0]

if m[i][d-1] > mx :

mx = m[i][d-1]

desired = (r \* d + 1) // 2

while (mi < mx):

mid = mi + (mx - mi) // 2

place = [0];

# Find count of elements smaller than mid

for i in range(r):

j = bisect\_right(m[i], mid) // j = position of m[i] means there are j smaller elements

place[0] = place[0] + j

if place[0] < desired:

mi = mid + 1

else:

mx = mid

return mi

62a. Given a matrix of size n x m , let’s say you convert the 2-D matrix in linear array. What is the element at index i of this linear array.

a. The matrix is n rows of m elements. So i-th index will point at i/m th row and i%m th column.

If we want to search an element in this matrix using binary search,we can do that using the code below:

bool searchMatrix(vector<vector<int>>& a, int target) {

if(a.size()==0 || a[0].size()==0)return false;

int n = a.size(), m = a[0].size(),i,j,start=0,end = m\*n-1,mid;

while(start<=end){

mid = (start+end)/2;

if(a[mid/m][mid%m] == target)return true;

else if(a[mid/m][mid%m] > target){

end = mid-1;

}

else{

start = mid+1;

}

}

return false;

}

63. Single Element in a Sorted Array

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once. Find this single element that appears only once.

Follow up: Your solution should run in O(log n) time and O(1) space.

Example 1:

Input: nums = [1,1,2,3,3,4,4,8,8]

Output: 2

Example 2:

Input: nums = [3,3,7,7,10,11,11]

Output: 10

a.Since array is sorted, that means the duplicate values will be together. We can linearly search for the first element that has no neighbours equal to itself. This will take O(N) time.

b. Since exactly one element has no duplicate, there will be odd number of elements in the array. We can use binary search in this problem. In each half if we find the element at mid, congrats, otherwise we will go to the next half where there are odd number of elements(excluding the duplicate element).

Eg [1,1,2,3,3,4,4,8,8], mid = 4th index ie 3, having a dupllicate on left.

We see that left subarray has 3 elements, so our target must be there.

Now [1,1,2,....], mid = 1st index ie 1, having a dupllicate on right.

We see that in left 0 elements are there. But in right subarray, 1 element is present and 1 is odd. So we move to right this time.

[..1,2,3,...], mid = 2nd index ie 2 and it has no duplicate neighbours, so it is the answer.

int singleNonDuplicate(vector<int>& a) {

int n = a.size(), start = 0, end = n,mid;

while(start<=end){

mid = start + (end - start)/2;

if(mid>0 && a[mid]==a[mid-1]){

if((mid-start-1)&1){

end = mid-2;

}else{

start = mid+1;

}

}else if (mid<n-1 && a[mid] == a[mid+1]){

if((mid-start)&1){

end = mid-1;

}else{

start = mid+2;

}

}else{

return a[mid];

}

}

return 0;

}

64. Search Element in sorted and rotated array.

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., [0,1,2,4,5,6,7] might become [4,5,6,7,0,1,2]).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

Your algorithm's runtime complexity must be in the order of O(log n).

Example 1:

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

Output: 4

Example 2:

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

Output: -1

a. If the given array is rotated, the first element must be greater than last element and this applies to all subarrays.

To do this in O(logN) time, we can check at middle points whether in the subarray [start:mid], the first element if greater than mid or not. If it is, the array is rotated,but it implies that the rest of the array is sorted. So we can use binary search on the rest of the array. Lets understand with an example:

Initially A = [4,5,6,7,0,1,2] and target = 0

since first > last, array is rotated. So we break it in two parts:

[4,5,6,7] and [0,1,2]. Since first element is smaller than last in [4,5,6,7] , we can apply binary search. Also [0,1,2] is sorted, so applying binary search will return the answer.

Take one more example:

[14,18,2,3,5,7,8,11] target = 8

first>last, so break in two parts.

[14,18,2,3] and [5,7,8,11]. Here first array is rotated, that means second is sorted. So we will apply binary search in that array.

For first array, break it. [14,18] and [2,3] Both are sorted so use binary search.

class Solution {

public:

int bsearch(vector<int>& a, int l, int r, int target){

while(l<=r){

int mid = (l+r)/2;

if(a[mid]==target)return mid;

else if(a[mid]>target) r = mid-1;

else l = mid+1;

}

return -1;

}

int find(vector<int>&a, int l, int r,int target){

if(l>r)return -1;

if(a[l]>a[r]){

int mid = (l+r)/2;

int x = find(a,l,mid, target);

if(x!=-1)return x;

x = find(a,mid+1,r, target);

return x;

}else{

return bsearch(a,l,r,target);

}

}

int search(vector<int>& a, int target) {

return find(a,0,a.size()-1,target);

}

};

b. If we can find the minimum element in array in logN time, we can Use two binary searches to solve this problem more easily, without any recursion.

We can actually find the smallest element using binary search.

Let’s observe the array [14,18,2,3,5,7,8,11].

The smallest one will be on right of the rotated part. So lets start the binary search:

mid = 3rd index ie 3. Since 3<11 (the right end), that means(array is strictly increasing) the smallest element is in left subarray(including mid), so we will search in [14,18,2,3].

Ths time middle one is 18.

Since 18>3 that means(the rotated part is also there) smallest one is on the right side, so we search in [2,3]

mid is 2 now and 2<3 so go for left subarray.

[2] Now only one element is left so this is our smallest element.

class Solution {

public:

int bsearch(vector<int>& a, int l, int r, int target){

while(l<=r){

int mid = (l+r)/2;

if(a[mid]==target)return mid;

else if(a[mid]>target) r = mid-1;

else l = mid+1;

}

return -1;

}

int search(vector<int>& a, int target) {

int lo,hi,mid;

lo = 0;

hi = a.size()-1;

while(lo<hi){ // Find the smallest element

mid = (lo+hi)/2;

if(a[mid]>a[hi])lo = mid+1;

else hi = mid;

}

int x = bsearch(a,lo,a.size()-1,target);

if(x<0){

return bsearch(a,0,lo-1,target);

}

return x;

}

};

65. Kth largest element in two sorted arrays.

Given two sorted arrays nums1 and nums2 of size m and n respectively and an int k. Find the k-th largest element of these arrays.

Note that it is the k-th largest element in the sorted order, not the k-th distinct element.

Example 1:

Input: nums1 = [-2, -1, 3, 5, 6, 8], nums2 = [0, 1, 2, 5, 9], k = 4

Output: 5

Explanation: Union of above arrays will be [-2, -1, 0, 1, 2, 3, 5, 5, 6, 8, 9] and the 4th largest element is 5.

Example 2:

Input: nums1 = [2, 4], nums2 = [6], k = 1

Output: 6

Explanation: union of above arrays will be [2, 4, 6] and the 1st largest element is 6.

You may assume k is always valid, 1 ≤ k ≤ m + n.

a. One straight forward solution is to merge the arrays and return the kth largest element directly.

b. Before solving this one, lets see a similar question 65a.

65a. Find median of two sorted arrays. The size of the arrays may differ.

a. Merge the arrays and return the median. We don’t need to merge the arrays physically but keep a count of how many elements we have merged (Or skipped) while traversing using 2 pointers. When we have skipped exactly (m+n+1)/2 elements, we are at the mid.

b. We can use binary search for the solution. <https://www.youtube.com/watch?v=LPFhl65R7ww>

Suppose we have two arrays A and B, having size m and n. Here m<=n, if not, we swap the arrays.

We will search for the answer in smaller array, considering m available partitions. Suppose we are at midA in any iteration. Since we want exactly (n+m)/2 elements in the left of our partition, we must choose midB = (m+n+1)/2 – midA as the index in array B so that we always balance the arrays.

Why (m+n+1)/2 and not (m+n)/2? Actually it doesn’t matter, using (m+n+1)/2 will keep 1 extra element in left array in case (m+n) is odd, that’s all. Even if we use (m+n)/2 – midA, we will need to check at midB +1st index that will be a little inconvenient.

Lets say A = a b c d

and B = e f g h i j k

If we are at ‘b’ in array A, that means we partitioned it as [a] [b,c,d].

Hence we must choose (m+n+1)/2 – midA = (4+7+1)/2 – 1 = 5 ie ‘j’ as the partition point in B.

That means B is partitioned as [e,f,g,h,i] [j,k].

That is [a,e,f,g,h,i] in left part and [b,c,d,j,k] in right. We can see that left part is having one extra element since (m+n) is odd.

Lets rewrite it:

[a] [b,c,d]

[e,f,g,h,i] [j,k]

This partition is valid if a<=j and i<=b ie the merged result will is sorted.

In case (m+n) is odd, [a,e,f,g,**h,i**,b,c,d,j,k] has two mid candidates, h and i, so answer is max(h,i) ie max of what left partitions have at their ends.

Lets make a quick example for even (m+n).

[a ] [b c d]

[g h] [ i j]

If (m+n) is even, merging these will result in [**a**,g,**h , *b*** *,* c , d , ***i****,* j]. Here a,h are candidates for left partition’s rightmost element and b,i can be right partition’s leftmost elements. So the answer is avg(max(a,h),min(b,i)).

One edge case we must keep in mind: What if midA/midB is at 0? That means there is nothing in left side, so we must assume that midA-1 points to -∞. Similarly, if midA/midB are at right of array(ie m,m+1,.. and n,n+1,...) that means nothing is present at right so assume that value to be ∞.

Below is the code:

class Solution {

public:

double findMedianSortedArrays(vector<int>& a, vector<int>& b) {

int i,j,m = a.size(),n = b.size();

if(m>n){

return findMedianSortedArrays(b,a);

}

bool odd = (m+n)&1;

if(m==0){

if(odd)return b[n/2];

return (double)(b[n/2]+b[n/2-1])/2.0;

}

if(n==0){

if(odd)return a[m/2];

return (double)(a[m/2]+a[m/2-1])/2.0;

}

int start = 0, end = m, midx,midy; // end = m because there can be right placed partition

while(start<=end){

midx = (start+end)/2; // You land at the first point of the second half

midy = (m+n+1)/2 - midx; // Here too. So consider left partition upto the left index ie mid-1 .

int a\_left = midx<=0?INT\_MIN:a[midx-1],b\_left = midy<=0?INT\_MIN:b[midy-1], a\_right = midx>=m?INT\_MAX:a[midx], b\_right = midy>=n?INT\_MAX:b[midy] ;

if(a\_left<=b\_right && b\_left<=a\_right){

if(odd){

return (double)max(a\_left,b\_left);

}else{

return (double)(max(a\_left,b\_left)+min(a\_right, b\_right))/2.0;

}

}

else if(a\_left>b\_right){ // If midx-1 is larger, shift to left

end = midx-1;

}else{

start = midx+1;

}

}

return 0;

}

};

Using the logic discussed above we can find kth largest and kth smallest elements in logN time.

**Talking about medians, it’s very irritating to write different median formulas for even and odd size arrays. So here is a trick for Python lists(or any other language that support -ve index), *Use (A[N/2] + A[~(N/2)])/2* for both even and odd arrays.**

**Below is the demo how it works:**

**[0,1,2,3,4] is odd sized, N = 5. So N/2 = 2 and ~2 = -3 both point to same element.**

**[0,1,2,3,4,5] is even sized, N = 6. N/2 = 3 and ~3 = -4 , both point to desired elements.**

66. Sliding window median.

// Will update soon

67. Given a number N, check whether N is a power of 2 or not.

a. Check the binary representaion of N. If there is only 1 set bit, return True (O(logN))

b. Right shift N in a loop and check for its parity. The moment it is odd and not 1, return False. (O(logN)).

c. Count the bits in the number using c = log2(N). If we can achieve the same number again using 1<<c, return True. (O(1))

d. Or a better way, if log2(N) is a whole number, return True. If fraction, return False.

e. One more, we can remove the rightmost set bit using N&(N-1), so if doing this results in 0, return True. (O(1)).

For a number of type X = 1000 ie (1)(0)\*, X-1 will be 111 ie 1\* with one less bit. If we take their bitwise & , the result is 0.

Similarly, lets take a number 110100100.

We will apply N &= (N-1) again and again

110100100 & 110100011 = 110100000 (The rightmost set bit is gone)

110100000 & 110011111 = 110000000

...

Eventually it will become 0

68. Count number of set bits in the given number.

To make it more interesting, you have to count number of set bits for all numbers in range [0,N].

(Note: The time complexities are for single element in the given range)

a. Use the binary representation to count the bits. (O(logN))

b. Right shift the bits and count the rightmost bits (N&1) (O(logN))

c. Use the method above (N&(N-1)). (O(logN))

d. In C++, we can use \_\_builtin\_popcount(N) to directly get the answer.

e. Or a clever way to use method c:

class Solution {

public:

vector<int> countBits(int n) {

vector<int> ans(n+1,0);

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

ans[i] = ans[i&(i-1)]+1;

}

return ans;

}

};

69. Divide two numbers A/B without \*, /, or % operator. (return Quotient)

a. Keep subtracting the number and count the number of times you subtracted. (-.-)

Just for fun, suppose \* operator was allowed, we can keep multiplying B with i{1to A} and the moment it becomes larger, return i;

b. We cannot use multiplication but we can simulate a similar operation using left shift operator.

Suppose A = 35, B = 6. A/B = 5

In binary, 35 = 100011 and 6 = 110

We will leftshift 6 using ‘i’, and look for the larget ‘i’ for which 6<<i is less than 35.

i=0 35>6

i=1 35>12

i=2 35>24

i=3 35<48 current = 1<<2 = 4

So that means 6\*4 = 24 is the current checkpoint. Now we need to make 35-24 = 9 using 6.

i=0 9>6

i=1 9<12 current = 1<<0 = 1

So answer is 4 + 1 = 5.

def divide(a,b):

sign = False # ie negative

sign = ((a<0 and b<0) or (a>0 and b>0)) # If true, makes it positive

## Make them positive

a = abs(a)

b = abs(b)

q = 0

while a>=b: # while a is dividable because we

# can't divide the smaller number

for i in range(a):

if (b<<i) > a:

q += 1<<(i-1) # means previous 'i' was samller

a -= b<<(i-1)

break

# Here for negative numbers I am adding an extra -1

# because answer doesn't match in python3 but with other

# languages (like c++), it should match.

return (1 if sign else -1)\*q + (0 if sign else -1)

c. (From GFG) As every number can be represented in base 2(0 or 1), represent the quotient in binary form by using shift operator as given below :

Determine the most significant bit in the quotient. This can easily be calculated by iterating on the bit position i from 31 to 1.

Find the first bit for which divisor << i is less than dividend and keep updating the ith bit position for which it is true.

Add the result in temp variable for checking the next position such that (temp + (divisor << i) ) is less than dividend.

Return the final answer of quotient after updating with corresponding sign.

def divide(dividend, divisor):

# Calculate sign of divisor

# i.e., sign will be negative

# either one of them is negative

# only iff otherwise it will be

# positive

sign = (-1 if((dividend < 0) ^

(divisor < 0)) else 1);

# remove sign of operands

dividend = abs(dividend);

divisor = abs(divisor);

# Initialize

# the quotient

quotient = 0;

temp = 0;

# test down from the highest

# bit and accumulate the

# tentative value for valid bit

for i in range(31, -1, -1):

if (temp + (divisor << i) <= dividend):

temp += divisor << i;

quotient |= 1 << i;

return sign \* quotient;

70. Power set.

Given a set S, print all the sets in the power set of S.

Eg Input: S = {1,2,3}

Output: PS = {},{1},{2},{3},{1,2},{2,3},{1,3},{1,2,3}

a. We can use the backtracking solution of subset sum here. Print every possibility, either take the current element or leave it.

// Append a tree here.

b. Since this question is from bit manipulation section, lets add the solution using bits. Observe the pattern of binary numbers from 0 to 7:

000

001

010

011

100

101

110

111

Each number represents a set of power set. If poistion is 1, take that element, else don’t take.

n = int(input())

a = [i+1 for i in range(n)]

ans = []

for i in range(1<<n):

x = bin(i)[2:]

x = (n-len(x))\*’0’ + x

l=[]

for j in range(len(x)):

if x[j]=='1':

l.append(a[j])

ans.append(l)

print(ans)

71. Find MSB of a given number.

a. Use the binary representation. O(logN)

b. Use log2n to count total bits. Return the count or 1<<count whatever needed. O(1)

c. Set all of the unset bits, add 1 and return ans/2. O(logN) or O(1)

Eg the number is 1001011 in binary

To set all bits for an 8 byte integer:

n|=n>>1

n|=n>>2

n|=n>>4

n|=n>>8

...

n|=n>>32.

Convert it to 1111111,

add 1, make it 10000000,

divide by 2 or right shift, make it 1000000. This is the answer.

72. Find the square of a number without using \* or / operator.

a. Use ‘+’ to simulate multiplication and multiply N and N by adding N to itself N times. O(N)

b.

If n is even, it can be written as

n = 2\*x

n2 = (2\*x)2 = 4\*x2

If n is odd, it can be written as

n = 2\*x + 1

n2 = (2\*x + 1)2 = 4\*x2 + 4\*x + 1

We don’t need to multiply with 4, or divide by 2 because we can do that using left and right shift operators.

We can recursively find these values easily.

def square(n):

if n==0:return 0

if n<0:n = -n

x = n>>1

if n&1:

return (square(x)<<2) + (x<<2) + 1

else:

return (square(x)<<2)

**// Next target...Short notes.**

**C++ STL**

Iterator lower\_bound (Iterator first, Iterator last, const val)

Iterator upper\_bound (Iterator first, Iterator last, const val)

lower\_bound returns an iterator pointing to the first element in the range [first,last) which has a value not less than ‘val’.

upper\_bound returns an iterator pointing to the first element in the range [first,last) which has a value greater than ‘val’.

priority\_queue:

In C++ STL, there is priority\_queue that can directly be used to implement Max Heap.

Supports push(), pop(), top()

priority\_queue supports a constructor that requires two extra arguments to make it min heap.

priority\_queue <Type, vector<Type>, ComparisonType > min\_heap;

eg priority\_queue<int, vector<int>, greater<int>()>;