1. Generate All Paranthesis II:

 INTERVIEW BIT PROBLEM

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses of length 2\*n.

For example, given n = 3, a solution set is:

"((()))", "(()())", "(())()", "()(())", "()()()"

**Make sure the returned list of strings are sorted.**

Approach :

So lets think how can we generate parenthesis nothing else no balancing only generating permutations of brackets.

So to do that what we could do is that for each step we either to add a opening bracket or closing bracket so how our tree might look like would be:

So as you can see this tree will go on so what could be the base case , well according to question it will be when the string size will be 6 as for now as balancing is still needed to be done.

So first thing is clear we have to approach question as each step we either add ‘(’ or ‘)’ ,

But to balance the parenthesis lets think about what is against balanced brackets

 We need to make ensure that at some point no opening brackets are still left while all closed brackets are over as it means we have overused them like shown

So the main checks are

1. Base Condition :  When both the remaining open and close brackets are zero it means we have created the string
2. When open>0 then only we can add ‘(’ to existing string
3. When close>open :why bcz the remaining closing brackets to balance the parenthesis are still left then only add ‘)’ add prevent preuse.

SOLUTION:

void gp(int open,int close,vector<string> &v , string s){

    if(open==0 && close==0){

        v.push\_back(s);

    }

    if(open>0){

        gp(open-1,close,v,s+'(');

    }

    if(open<close){

        gp(open,close-1,v,s+')');

    }

}

vector<string> Solution::generateParenthesis(int A) {

    vector<string> v;

    string s="(";

    gp(A-1,A,v,s);

    return v;

}

1. Largest Number with k swaps:

Asket In: Amazon , MakeMyTrip, Walmart

Given a number **K**and string **S**of digits denoting a positive integer, build the largest number possible by performing swap operations on the digits of **S**atmost **K**times.

**Input**:  
The first line of input contains an integer **T**denoting the number of test cases. Then **T**test cases follow. The first line of each test case contains a positive integer **K**. The second line of each test case contains a string of digits denoting a positive integer.

**Output**:  
You have to compute result and store in "**max**" variable. The result will be printed by driver code.

**User task:**  
Since this is a functional problem you don't have to worry about input, you just have to complete the function **findMaximumNum().**There will be three arguments which will be provided. They are as such.  
**For Cpp**: arguments - findMaximumNum(str, k, max) (str: string input, k: number, max: will contain result)  
**For Java**: arguments - findMaximumNum(ar[], k, max) (ar[]: character array of input, k: number, max: will contain result).

**Constraints**:  
1 <= **T**<= 100  
1 <= **S**<= 30  
1 <= **K**<= 10

**Examples :**  
**Input**:  
3  
4  
1234567  
3  
3435335  
2  
1034

**Output**:  
7654321  
5543333  
4301

**Explanation:  
Testcase 1:** Three swaps can make the input 1234567 to 7654321, swapping 1 with 7, 2 with 6 and finally 3 with 5.

**Approach:**

To create the largest number what simply we can think is something like we make all

permutation for that number with k swaps and have a variable max to store the max number out of

all.

So how can we implement is like

The more efficient solution would be when we swap only maximum number with each starting index for k times as that will give largest number among all permutations so

Solution:

 void maxNum(string str,int k,string &max){

     if(k==0){

        return;

     }

    for(int j=0;j<str.size();j++){

    for(int i=0;i<str.size();i++){

        if(str[i]!=str[j]){

             swap(str[i],str[j]);

             max=str.compare(max)>0?str:max;

             maxNum(str,k-1,max);

             swap(str[i],str[j]);

        }

        }

    }

 }

Efficient

 void maxNum(string str,int k,string &max,int ctr){

     if(k==0){

        return;

     }

    int maxv=str[ctr];

    for(int i=ctr+1;i<str.size();i++){

        if(maxv<str[i]){

            maxv=str[i];

        }

    }

    if(maxv!=str[ctr]){

        k--;

    }

    for(int i=ctr;i<str.size();i++){

        if(str[i]==maxv){

             swap(str[i],str[ctr]);

             max=str.compare(max)>0?str:max;

           maxNum(str,k,max,ctr+1);

           swap(str[i],str[ctr]);

        }

    }

 }

void findMaximumNum(string str, int k, string& max) {

    // Your code here

   maxNum(str,k,max,0);

}

1. **Partition Array to K equal subsets**

Given an integer array A[] of N elements, the task is to complete the function which returns true if the array A[] could be divided into K non-empty subsets such that the sum of elements in every subset is same.  
  
**Note:** All elements of this array should be part of exactly one partition.  
  
Examples:  
Input : A[] = [2, 1, 4, 5, 6], K = 3  
Output : 1, as we can divide above array into 3 parts with equal sum as (2, 4), (1, 5), (6)

Input : A[] = [2, 1, 5, 5, 6], K = 3  
Output : 0, as it is not possible to divide above array into 3 parts with equal sum

**Input:**  
The first line of input contains an integer T denoting the no of test cases. Then T test cases follow. Each test case contains 3 lines. The first line of each test case contains an integer denoting the size of the array. In the next line are N space separated values of the array A[ ]. In the next line is an integer K.  
  
**Output:**  
The output for each test case will be 1 if the array could be divided into k subsets else 0 .  
  
**Constraints:**  
1<=T<=100  
1<=N,K<=100  
1<=A[ ] <=100  
  
**Example(To be used only for expected output):  
Input:**  
2  
5  
2 1 4 5 6  
3  
5  
2 1 5 5 6  
3  
**Output:**  
1  
0

**Approach:**

**Solution:**

bool partition(int A[], int N,int start,int targetSum,int cursum,int k,bool visited[]){

      if(k==1){

         return true;

     }

  if(cursum>targetSum){

      return false;

  }

     if(cursum==targetSum){

         cursum=0;

         start=0;

         k--;

     }

     for(int i=start;i<N;i++){

         if(visited[i]==false){

             visited[i]=true;

          bool val= partition(A,N,start+1,targetSum,cursum+A[i],k,visited);

          if(val==true){

              return true;

          }

             visited[i]=false;

         }

     }

return false;

}

bool checkPartion(int arr[],int n,int k){

    int totalsum=0;

    for(int i=0;i<n;i++){

        totalsum+=arr[i];

    }

    if(k==0 || (totalsum%k)!=0){

        return false;

    }else{

     bool visited[n];

     memset(visited,false,sizeof(visited));

        return partition(arr,n,0,(totalsum/k),0,k,visited);

    }

}

bool isKPartitionPossible(int A[], int N, int K)

{

     //Your code here

    return checkPartion(A,N,K);

}

1. Power of N

Find x raised to power N:

**Solution:**

#include<iostream>

**using** **namespace** std;

**class** powern

{

/\* Function to calculate x raised to the power y \*/

**public**:

**int** power(**int** x, unsigned **int** y)

{

**if** (y == 0)

**return** 1;

**else** **if** (y % 2 == 0)

**return** power(x, y / 2) \* power(x, y / 2);

**else**

**return** x \* power(x, y / 2) \* power(x, y / 2);

}

};

/\* Driver code \*/

**int** main()

{

    Powern g;

**int** x = 2;

    unsigned **int** y = 3;

    cout << g.power(x, y);

**return** 0;

}

**Return Key Pad Code**

Given an integer n, using phone keypad find out all the possible strings that can be made using digits of input n.

Return empty string for numbers 0 and 1.

Note : 1. The order of strings are not important.

2. Input and output has already been managed for you. You just have to populate the output array and return the count of elements populated in the output array.

**Input Format :**

Integer n

**Output Format :**

All possible strings in different lines

**Constraints :**

1 <= n <= 10^6

**Sample Input:**

23

**Sample Output:**

ad

ae

af

bd

be

bf

cd

ce

cf

#include <string>

using namespace std;

int keypad(int num, string output[]){

    /\* Insert all the possible combinations of the integer number into the output string array. You do not need to

    print anything, just return the number of strings inserted into the array.

    \*/

    if(num==0){

        output[0]="";

        return 1;

    }

  int size;

    if(num%10==2){

      size = keypad(num/10,output);

        for(int i=0;i<size;i++){

             string value=output[i];

            output[i]=value+"a";

            output[i+size]=value+"b";

             output[i+(2\*size)]=value+"c";

        }

        return (size\*3);

    }else if(num%10==3){

        size = keypad(num/10,output);

        for(int i=0;i<size;i++){

             string value=output[i];

            output[i]=value+"d";

            output[i+size]=value+"e";

             output[i+(2\*size)]=value+"f";

        }

        return (size\*3);

    }else if(num%10==4){

           size = keypad(num/10,output);

        for(int i=0;i<size;i++){

             string value=output[i];

            output[i]=value+"g";

            output[i+size]=value+"h";

             output[i+(2\*size)]=value+"i";

        }

        return (size\*3);

    }else if(num%10==5){

           size = keypad(num/10,output);

        for(int i=0;i<size;i++){

             string value=output[i];

            output[i]=value+"j";

            output[i+size]=value+"k";

             output[i+(2\*size)]=value+"l";

        }

        return (size\*3);

    }else if(num%10==6){

           size = keypad(num/10,output);

        for(int i=0;i<size;i++){

             string value=output[i];

            output[i]=value+"m";

            output[i+size]=value+"n";

             output[i+(2\*size)]=value+"o";

        }

        return (size\*3);

    }else if(num%10==7){

           size = keypad(num/10,output);

        for(int i=0;i<size;i++){

             string value=output[i];

            output[i]=value+"p";

            output[i+size]=value+"q";

             output[i+(2\*size)]=value+"r";

             output[i+(3\*size)]=value+"s";

        }

        return (size\*4);

    }else if(num%10==8){

           size = keypad(num/10,output);

        for(int i=0;i<size;i++){

        string value=output[i];

            output[i]=value+"t";

            output[i+size]=value+"u";

             output[i+(2\*size)]=value+"v";

        }

        return (size\*3);

    }else if(num%10==9){

           size = keypad(num/10,output);

        for(int i=0;i<size;i++){

             string value=output[i];

            output[i]=value+"w";

            output[i+size]=value+"x";

             output[i+(2\*size)]=value+"y";

               output[i+(3\*size)]=value+"z";

        }

        return (size\*4);

    }

}

***6. Inversion Count:***

Let A[0 ... n-1] be an array of n distinct positive integers. If i < j and A[i] > A[j] then the pair (i, j) is called an inversion of A (where i and j are indexes of A). Given an integer

array A, your task is to find the number of inversions in A. Input format :

Line 1 : n, array size  
Line 2 : Array elements (separated by space). Output format :

Count of inversions

Constraints :

1 <= n <= 10^5

1 <= A[i] <= 10^9 Sample Input 1 :

3  
32 1  
Sample Output 1 : 3  
Sample Input 2 : 5  
2513 4  
Sample Output 1 : 4

***Approach: Now in this question we know that we need to count the I<j and a[I]>j. So first we need to think that how sorting could help , so how sorting could help is that for any sorted array if at any I<j a[I]>a[j] then this relation will also be correct for j to n no which is count+=(start of right array-i) this was the main catch. So this is implemented along with the merge sort in because we divide it into two half where we know the left part will always have i< right part j thus above approach can be implemented correctly.***

**Solution:**  
void mergeArr(int arr[],int start1,int end1,int start2,int end2,long long &count){

int i=start1;  
int j=start2;  
int k=0;  
int temp[end2-start1+1];

while(i<=end1 && j<=end2){

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

temp[k++]=arr[i++]; }

else{

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

 count+=start2-i;

} }

while(i<=end1){ temp[k++]=arr[i++];

}

while(j<=end2){ temp[k++]=arr[j++];

}

for(int i=0;i<end2-start1+1;i++){ arr[i+start1]=temp[i];

} }

void mergeSort(int arr[],int start,int end,long long &count){ if(start<end){

int mid=start+(end-start)/2; mergeSort(arr,start,mid,count); mergeSort(arr,mid+1,end,count); mergeArr(arr,start,mid,mid+1,end,count);

} }

long long solve(int A[], int n) { long long count=0;

mergeSort(A,0,n-1,count); return count;

 }

**Stair Case Problem**

A child is running up a staircase with n steps and can hop either 1 step, 2 steps or 3 steps at a time. Implement a method to count how many possible ways the child can run up to the stairs. You need to return all possible number of ways.

Time complexity of your code should be O(n).

**Input format :**

Integer n (No. of steps)

**Constraints :**

n <= 70

**Sample Input 1:**

4

**Sample Output 1:**

7

Solution:

#include <bits/stdc++.h>

**using** **namespace** std;

// A simple recursive program to

// find N'th fibonacci number

**int** fib(**int** n)

{

**if** (n <= 1)

**return** n;

**return** fib(n - 1) + fib(n - 2);

}

// Returns number of ways to

// reach s'th stair

**int** countWays(**int** s)

{

**return** fib(s + 1);

}

// Driver C

**int** main()

{

**int** s = 4;

    cout << "Number of ways = " << countWays(s);

**return** 0;

}

**There’s a picture below do refer…….**

