Backtracking

# 254. Rat in a maze Problem

Consider a rat placed at **(0, 0)** in a square matrixof order **N \* N**. It has to reach the destination at **(N - 1, N - 1)**. Find all possible paths that the rat can take to reach from source to destination. The directions in which the rat can move are **'U'(up)**, **'D'(down)**, **'L' (left)**, **'R' (right)**. Value 0 at a cell in the matrix represents that it is blocked and rat cannot move to it while value 1 at a cell in the matrix represents that rat can be travel through it.  
**Note**: In a path, no cell can be visited more than one time.

**Example 1:**

**Input**:

N = 4

m[][] = {{1, 0, 0, 0},

{1, 1, 0, 1},

{1, 1, 0, 0},

{0, 1, 1, 1}}

**Output:**

DDRDRR DRDDRR

**Explanation**:

The rat can reach the destination at

(3, 3) from (0, 0) by two paths - DRDDRR

and DDRDRR, when printed in sorted order

we get DDRDRR DRDDRR.

**Example 2:**

**Input**:

N = 2

m[][] = {{1, 0},

{1, 0}}

**Output:**

-1

**Explanation**:

No path exists and destination cell is

blocked.

**Your Task:**  
You don't need to read input or print anything. Complete the function **printPath()**which takes **N**and 2D array**m[ ][ ]**as input parameters and returns the list of paths in lexicographically increasing order.   
**Note:** In case of no path, return an empty list. The driver will output **"-1"** automatically.

**Expected Time Complexity:** O((3N^2)).  
**Expected Auxiliary Space:** O(L \* X), L = length of the path, X = number of paths.

**Constraints:**  
2 ≤ N ≤ 5  
0 ≤ m[i][j] ≤ 1

## Solution:

**Approach:**

1. Start from the initial index (i.e. (0,0)) and look for the valid moves through the adjacent cells in the order **Down->Left->Right->Up** (so as to get the sorted paths) in the grid.
2. If the move is possible, then move to that cell while storing the character corresponding to the move(D,L,R,U) and again start looking for the valid move until the last index (i.e. (n-1,n-1)) is reached.
3. Also, keep on marking the cells as visited and when we traversed all the paths possible from that cell, then unmark that cell for other different paths and remove the character from the path formed.
4. As the last index of the grid(bottom right) is reached, then store the traversed path.

Below is the implementation of the above approach:

// C++ implementation of the above approach

#include <bits/stdc++.h>

#define MAX 5

using namespace std;

// Function returns true if the

// move taken is valid else

// it will return false.

bool isSafe(int row, int col, int m[][MAX],

int n, bool visited[][MAX])

{

if (row == -1 || row == n || col == -1 ||

col == n || visited[row][col]

|| m[row][col] == 0)

return false;

return true;

}

// Function to print all the possible

// paths from (0, 0) to (n-1, n-1).

void printPathUtil(int row, int col, int m[][MAX],

int n, string& path, vector<string>&

possiblePaths, bool visited[][MAX])

{

// This will check the initial point

// (i.e. (0, 0)) to start the paths.

if (row == -1 || row == n || col == -1

|| col == n || visited[row][col]

|| m[row][col] == 0)

return;

// If reach the last cell (n-1, n-1)

// then store the path and return

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

possiblePaths.push\_back(path);

return;

}

// Mark the cell as visited

visited[row][col] = true;

// Try for all the 4 directions (down, left,

// right, up) in the given order to get the

// paths in lexicographical order

// Check if downward move is valid

if (isSafe(row + 1, col, m, n, visited))

{

path.push\_back('D');

printPathUtil(row + 1, col, m, n,

path, possiblePaths, visited);

path.pop\_back();

}

// Check if the left move is valid

if (isSafe(row, col - 1, m, n, visited))

{

path.push\_back('L');

printPathUtil(row, col - 1, m, n,

path, possiblePaths, visited);

path.pop\_back();

}

// Check if the right move is valid

if (isSafe(row, col + 1, m, n, visited))

{

path.push\_back('R');

printPathUtil(row, col + 1, m, n,

path, possiblePaths, visited);

path.pop\_back();

}

// Check if the upper move is valid

if (isSafe(row - 1, col, m, n, visited))

{

path.push\_back('U');

printPathUtil(row - 1, col, m, n,

path, possiblePaths, visited);

path.pop\_back();

}

// Mark the cell as unvisited for

// other possible paths

visited[row][col] = false;

}

// Function to store and print

// all the valid paths

void printPath(int m[MAX][MAX], int n)

{

// vector to store all the possible paths

vector<string> possiblePaths;

string path;

bool visited[n][MAX];

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

// Call the utility function to

// find the valid paths

printPathUtil(0, 0, m, n, path,

possiblePaths, visited);

// Print all possible paths

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

cout << possiblePaths[i] << " ";

}

// Driver code

int main()

{

int m[MAX][MAX] = { { 1, 0, 0, 0, 0 },

{ 1, 1, 1, 1, 1 },

{ 1, 1, 1, 0, 1 },

{ 0, 0, 0, 0, 1 },

{ 0, 0, 0, 0, 1 } };

int n = sizeof(m) / sizeof(m[0]);

printPath(m, n);

return 0;

}

**Output:**

DDRRURRDDD DDRURRRDDD DRDRURRDDD DRRRRDDD

**Complexity Analysis:**

* **Time Complexity**: O(3^(n^2)).   
  As there are N^2 cells from each cell there are 3 unvisited neighbouring cells. So the time complexity O(3^(N^2).

**My Implementation:**

class Solution{

public:

void fun(vector<string> &res, string &str, vector<vector<int>> &m, int n, int x, int y){

if(m[x][y]==0)

return;

if(x==n-1 && y==n-1){

res.push\_back(str);

return;

}

m[x][y] = -1;

if(x<n-1 && m[x+1][y]==1){

str.push\_back('D');

fun(res, str, m, n, x+1, y);

str.pop\_back();

}

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

str.push\_back('L');

fun(res, str, m, n, x, y-1);

str.pop\_back();

}

if(y<n-1 && m[x][y+1]==1){

str.push\_back('R');

fun(res, str, m, n, x, y+1);

str.pop\_back();

}

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

str.push\_back('U');

fun(res, str, m, n, x-1, y);

str.pop\_back();

}

m[x][y] = 1;

}

vector<string> findPath(vector<vector<int>> &m, int n) {

vector<string> res;

string str = "";

fun(res, str, m, n, 0, 0);

return res;

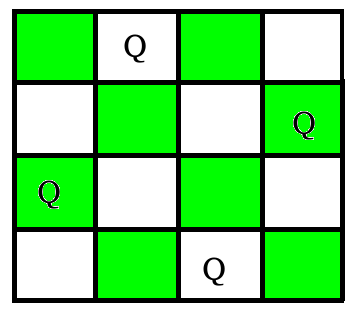
}

};

**Time Complexity:** O(3^(n^2))

**Space Complexity:** O(n^2)

# 255. Printing all solutions in N-Queen Problem

The n-queens puzzle is the problem of placing **n** queens on a (**n×n)** chessboard such that no two queens can attack each other.  
Given an integer n, find all distinct solutions to the n-queens puzzle. Each solution contains distinct board configurations of the n-queens’ placement, where the solutions are a permutation of [1,2,3..n] in increasing order, here the number in the *ith* place denotes that the *ith*-column queen is placed in the row with that number. For eg below figure represents a chessboard [3 1 4 2].  
  


**Example 1:**

**Input:**

1

**Output:**

[1]

**Explaination:**

Only one queen can be placed

in the single cell available.

**Example 2:**

**Input:**

4

**Output:**

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

**Explaination:**

These are the 2 possible solutions.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **nQueen()** which takes n as input parameter and returns a list containing all the possible chessboard configurations in sorted order. Return an empty list if no solution exists.

**Expected Time Complexity:** O(n!)  
**Expected Auxiliary Space:** O(n2)

**Constraints:**  
1 ≤ n ≤ 10

## Solution

**My Implementation**

class Solution{

public:

void fun(vector<vector<int>> &res, vector<int> &sol, int n, int i){

if(i==n){

res.push\_back(sol);

return;

}

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

int j=0;

for(; j<sol.size();j++){

int i1=sol[j], j1=j, i2=row, j2=i;

if(i1==i2 || j1==j2 || (i1-j1)==(i2-j2) || (i1+j1)==(i2+j2))

break;

}

if(j==sol.size()){

sol.push\_back(row);

fun(res, sol, n, i+1);

sol.pop\_back();

}

}

}

vector<vector<int>> nQueen(int n) {

vector<vector<int>> res;

vector<int> sol;

fun(res, sol, n, 0);

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

for(int j=0;j<res[0].size();j++)

res[i][j]++;

}

return res;

}

};

**Backtracking Algorithm**

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

1) Start in the leftmost column

2) If all queens are placed

return true

3) Try all rows in the current column. Do following

for every tried row.

a) If the queen can be placed safely in this row

then mark this [row, column] as part of the

solution and recursively check if placing

queen here leads to a solution.

b) If placing queen in [row, column] leads to a

solution then return true.

c) If placing queen doesn't lead to a solution

then unmark this [row, column] (Backtrack)

and go to step (a) to try other rows.

3) If all rows have been tried and nothing worked,

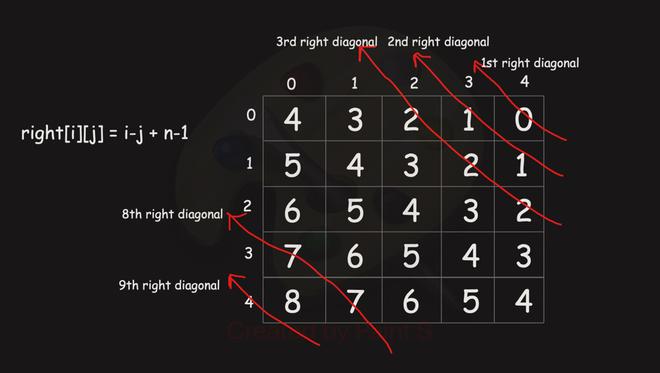
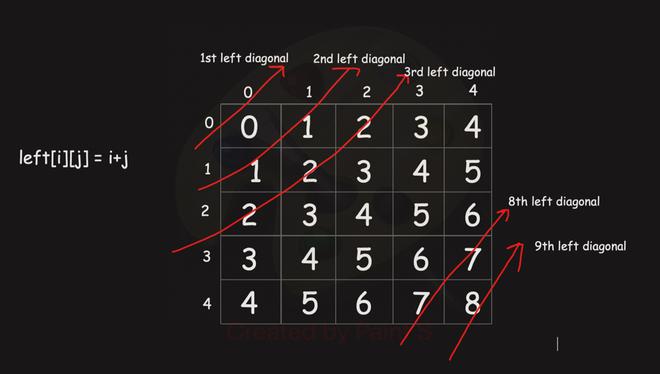
return false to trigger backtracking.

***A modification is that we can find whether we have a previously placed queen in a column or in left diagonal or in right diagonal in O(1) time. We can observe that***

***1)For all cells in a particular left diagonal , their row + col  = constant.***

***2)For all cells in a particular right diagonal, their row – col + n – 1 = constant.***

**Let say n = 5, then we have a total of 2n-1 left and right diagonals**



*Let say we placed a queen at (2,0)*

*(2,0) have leftDiagonal value = 2. Now we can not place another queen at (1,1) and (0,2) because both of these have same leftDiagonal value as for (2,0). Similar thing can be noticed for right diagonal as well.*

/\* C/C++ program to solve N Queen Problem using

backtracking \*/

#include <bits/stdc++.h>

using namespace std;

vector<vector<int> > result;

/\* A utility function to check if a queen can

be placed on board[row][col]. Note that this

function is called when "col" queens are

already placed in columns from 0 to col -1.

So we need to check only left side for

attacking queens \*/

bool isSafe(vector<vector<int> > board,

int row, int col)

{

int i, j;

int N = board.size();

/\* Check this row on left side \*/

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

if (board[row][i])

return false;

/\* Check upper diagonal on left side \*/

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

if (board[i][j])

return false;

/\* Check lower diagonal on left side \*/

for (i = row, j = col; j >= 0 && i < N; i++, j--)

if (board[i][j])

return false;

return true;

}

/\* A recursive utility function to solve N

Queen problem \*/

bool solveNQUtil(vector<vector<int> >& board, int col)

{

/\* base case: If all queens are placed

then return true \*/

int N = board.size();

if (col == N) {

vector<int> v;

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

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

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

v.push\_back(j + 1);

}

}

result.push\_back(v);

return true;

}

/\* Consider this column and try placing

this queen in all rows one by one \*/

bool res = false;

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

/\* Check if queen can be placed on

board[i][col] \*/

if (isSafe(board, i, col)) {

/\* Place this queen in board[i][col] \*/

board[i][col] = 1;

// Make result true if any placement

// is possible

res = solveNQUtil(board, col + 1) || res;

/\* If placing queen in board[i][col]

doesn't lead to a solution, then

remove queen from board[i][col] \*/

board[i][col] = 0; // BACKTRACK

}

}

/\* If queen can not be place in any row in

this column col then return false \*/

return res;

}

/\* This function solves the N Queen problem using

Backtracking. It mainly uses solveNQUtil() to

solve the problem. It returns false if queens

cannot be placed, otherwise return true and

prints placement of queens in the form of 1s.

Please note that there may be more than one

solutions, this function prints one of the

feasible solutions.\*/

vector<vector<int> > nQueen(int n)

{

result.clear();

vector<vector<int> > board(n, vector<int>(n, 0));

if (solveNQUtil(board, 0) == false) {

return {};

}

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

return result;

}

// Driver Code

int main()

{

int n = 4;

vector<vector<int> > v = nQueen(n);

for (auto ar : v) {

cout << "[";

for (auto it : ar)

cout << it << " ";

cout << "]";

}

return 0;

}

**Output**

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

**Efficient Backtracking Approach Using Bit-Masking**

**Algorithm:**   
There is always only one queen in each row and each column, so idea of backtracking is to start placing queen from the leftmost column of each row and find a column where the queen could be placed without collision with previously placed queens. It is repeated from the first row till the last row. While placing a queen, it is tracked as if it is not making a collision (row-wise, column-wise and diagonally) with queens placed in previous rows. Once it is found that the queen can’t be placed at a particular column index in a row, the algorithm backtracks and change the position of the queen placed in the previous row then moves forward to place the queen in the next row.

1. Start with three-bit vector which is used to track safe place for queen placement row-wise, column-wise and diagonally in each iteration.
2. Three-bit vector will contain information as bellow:
   * **rowmask:** set bit index (i) of this bit vector will indicate, the queen can’t be placed at ith column of next row.
   * **ldmask:** set bit index (i) of this bit vector will indicate, the queen can’t e placed at ith column of next row. It represents the unsafe column index for next row falls under left diagonal of queens placed in previous rows.
   * **rdmask:** set bit index (i) of this bit vector will indicate, the queen can’t be placed at ith column of next row. It represents the unsafe column index for next row falls right diagonal of queens placed in previous rows.
3. There is a 2-D (NxN) matrix (board), which will have ‘ ‘ character at all indexes in beginning and it gets filled by ‘Q’ row-by-row. Once all rows are filled by ‘Q’, the current solution is pushed into the result list.

Below is the implementation of the above approach:

// CPP program for above approach

#include <bits/stdc++.h>

using namespace std;

vector<vector<int> > result;

// Program to solve N Queens problem

void solveBoard(vector<vector<char> >& board, int row,

int rowmask, int ldmask, int rdmask,

int& ways)

{

int n = board.size();

// All\_rows\_filled is a bit mask having all N bits set

int all\_rows\_filled = (1 << n) - 1;

// If rowmask will have all bits set, means queen has

// been placed successfully in all rows and board is

// displayed

if (rowmask == all\_rows\_filled) {

vector<int> v;

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

for (int j = 0; j < board.size(); j++) {

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

v.push\_back(j + 1);

}

}

result.push\_back(v);

return;

}

// We extract a bit mask(safe) by rowmask,

// ldmask and rdmask. all set bits of 'safe'

// indicates the safe column index for queen

// placement of this iteration for row index(row).

int safe

= all\_rows\_filled & (~(rowmask | ldmask | rdmask));

while (safe) {

// Extracts the right-most set bit

// (safe column index) where queen

// can be placed for this row

int p = safe & (-safe);

int col = (int)log2(p);

board[row][col] = 'Q';

// This is very important:

// we need to update rowmask, ldmask and rdmask

// for next row as safe index for queen placement

// will be decided by these three bit masks.

// We have all three rowmask, ldmask and

// rdmask as 0 in beginning. Suppose, we are placing

// queen at 1st column index at 0th row. rowmask,

// ldmask and rdmask will change for next row as

// below:

// rowmask's 1st bit will be set by OR operation

// rowmask = 00000000000000000000000000000010

// ldmask will change by setting 1st

// bit by OR operation and left shifting

// by 1 as it has to block the next column

// of next row because that will fall on left

// diagonal. ldmask =

// 00000000000000000000000000000100

// rdmask will change by setting 1st bit

// by OR operation and right shifting by 1

// as it has to block the previous column

// of next row because that will fall on right

// diagonal. rdmask =

// 00000000000000000000000000000001

// these bit masks will keep updated in each

// iteration for next row

solveBoard(board, row + 1, rowmask | p,

(ldmask | p) << 1, (rdmask | p) >> 1,

ways);

// Reset right-most set bit to 0 so,

// next iteration will continue by placing the queen

// at another safe column index of this row

safe = safe & (safe - 1);

// Backtracking, replace 'Q' by ' '

board[row][col] = ' ';

}

return;

}

// Driver Code

int main()

{

// Board size

int n = 4;

int ways = 0;

vector<vector<char> > board;

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

vector<char> tmp;

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

tmp.push\_back(' ');

}

board.push\_back(tmp);

}

int rowmask = 0, ldmask = 0, rdmask = 0;

int row = 0;

// Function Call

result.clear();

solveBoard(board, row, rowmask, ldmask, rdmask, ways);

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

for (auto ar : result) {

cout << "[";

for (auto it : ar)

cout << it << " ";

cout << "]";

}

return 0;

}

// This code is contributed by Nikhil Vinay

**Output**

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

# 256. Word Break Problem using Backtracking

Given a string **s** and a dictionary of words **dict** of length **n,** add spaces in **s** to construct a sentence where each word is a valid dictionary word. Each dictionary word can be used more than once. Return all such possible sentences.

Follow examples for better understanding.

**Example 1:**

**Input:** s = "catsanddog", n = 5

dict = {"cats", "cat", "and", "sand", "dog"}

**Output:** (cats and dog)(cat sand dog)

**Explanation:** All the words in the given

sentences are present in the dictionary.

**Example 2:**

**Input:** s = "catsandog", n = 5

dict = {"cats", "cat", "and", "sand", "dog"}

**Output:** Empty

**Explanation:** There is no possible breaking

of the string s where all the words are present

in dict.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **wordBreak()** which takes **n**, **dict** and **s** as input parameters and returns a list of possible sentences. If no sentence is possible it returns an empty list.

**Expected Time Complexity:** O(N2\*n) where N = |s|  
**Expected Auxiliary Space:** O(N2)

**Constraints:**  
1 ≤ n ≤ 20  
1 ≤ dict[i] ≤ 15  
1 ≤ |s| ≤ 500

## Solution:

**My Implementation:**

class Solution{

public:

void fun(vector<string> &dict, string &s, string str, int len, vector<string> &res, int ind, int word, int word\_i){

if(len==ind){

if(word!=-1 && dict[word].size()==word\_i){

str = str + dict[word];

res.push\_back(str);

}

return;

}

if(word!=-1 && dict[word].size()==word\_i){

str = str + dict[word] + " ";

word = -1;

}

if(word==-1){

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

if(dict[i][0]==s[ind]){

fun(dict, s, str, len, res, ind+1, i, 1);

}

}

}

else{

if(dict[word][word\_i]==s[ind]){

fun(dict, s, str, len, res, ind+1, word, word\_i+1);

}

}

}

vector<string> wordBreak(int n, vector<string>& dict, string s)

{

vector<string> res;

string str = "";

int len = s.size();

fun(dict, s, str, len, res, 0, -1, 0);

return res;

}

};

We start scanning the sentence from the left. As we find a valid word, we need to check whether the rest of the sentence can make valid words or not. Because in some situations the first found word from the left side can leave a remaining portion that is not further separable. So, in that case, we should come back and leave the currently found word and keep on searching for the next word. And this process is recursive because to find out whether the right portion is separable or not, we need the same logic. So we will use recursion and backtracking to solve this problem. To keep track of the found words we will use a stack. Whenever the right portion of the string does not make valid words, we pop the top string from the stack and continue finding.

Below is the implementation of the above idea:

// A recursive program to print all possible

// partitions of a given string into dictionary

// words

#include <iostream>

using namespace std;

/\* A utility function to check whether a word

is present in dictionary or not. An array of

strings is used for dictionary. Using array

of strings for dictionary is definitely not

a good idea. We have used for simplicity of

the program\*/

int dictionaryContains(string &word)

{

string dictionary[] = {"mobile","samsung","sam","sung",

"man","mango", "icecream","and",

"go","i","love","ice","cream"};

int n = sizeof(dictionary)/sizeof(dictionary[0]);

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

if (dictionary[i].compare(word) == 0)

return true;

return false;

}

// Prototype of wordBreakUtil

void wordBreakUtil(string str, int size, string result);

// Prints all possible word breaks of given string

void wordBreak(string str)

{

// Last argument is prefix

wordBreakUtil(str, str.size(), "");

}

// Result store the current prefix with spaces

// between words

void wordBreakUtil(string str, int n, string result)

{

//Process all prefixes one by one

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

{

// Extract substring from 0 to i in prefix

string prefix = str.substr(0, i);

// If dictionary contains this prefix, then

// we check for remaining string. Otherwise

// we ignore this prefix (there is no else for

// this if) and try next

if (dictionaryContains(prefix))

{

// If no more elements are there, print it

if (i == n)

{

// Add this element to previous prefix

result += prefix;

cout << result << endl;

return;

}

wordBreakUtil(str.substr(i, n-i), n-i,

result + prefix + " ");

}

}

}

//Driver Code

int main()

{

// Function call

cout << "First Test:\n";

wordBreak("iloveicecreamandmango");

cout << "\nSecond Test:\n";

wordBreak("ilovesamsungmobile");

return 0;

}

**Output**

First Test:

i love ice cream and man go

i love ice cream and mango

i love icecream and man go

i love icecream and mango

Second Test:

i love sam sung mobile

i love samsung mobile

**Complexities:**

* **Time Complexity**: O(2n). Because there are 2n combinations in The Worst Case.
* **Auxiliary Space**: O(n2). Because of the Recursive Stack of wordBreakUtil(…) function in The Worst Case.

Where n is the length of the input string.

# 257. Remove Invalid Parentheses

Given a string s that contains parentheses and letters, remove the minimum number of invalid parentheses to make the input string valid.

Return *all the possible results*. You may return the answer in **any order**.

**Example 1:**

**Input:** s = "()())()"

**Output:** ["(())()","()()()"]

**Example 2:**

**Input:** s = "(a)())()"

**Output:** ["(a())()","(a)()()"]

**Example 3:**

**Input:** s = ")("

**Output:** [""]

**Constraints:**

* 1 <= s.length <= 25
* s consists of lowercase English letters and parentheses '(' and ')'.
* There will be at most 20 parentheses in s.

## Solution:

**My Implementation:**

class Solution {

public:

int minRemovals(string s, int n){

int temp=0, res=0;

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

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

temp++;

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

if(temp==0)

res++;

else

temp--;

}

}

return res+temp;

}

void solve(string s, int n, vector<string> &ans, int mr, unordered\_set<string> &st){

if(st.find(s)!=st.end())

return;

st.insert(s);

if(mr==0){

if(minRemovals(s, n)==0)

ans.push\_back(s);

return;

}

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

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

solve(s.substr(0,i)+s.substr(i+1), n-1, ans, mr-1, st);

}

}

vector<string> removeInvalidParentheses(string s) {

int n = s.size(), mr = minRemovals(s, n);

vector<string> ans;

unordered\_set<string> st;

solve(s, n, ans, mr, st);

return ans;

}

};

**Another Implementation:**

In this, we have computed the subsequence that needs to be deleted to make string valid and then, checked by deleting that subsequence in diff. ways if it generates valid string or not.

class Solution

{

List<String> list = new ArrayList<>();

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

int count1=0,count2=0;

boolean check(String s)

{

int temp=0;

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

{

if(s.charAt(i)=='(')

temp++;

else if(s.charAt(i)==')')

temp--;

if(temp<0)

return false;

}

if(temp==0)

return true;

return false;

}

void fun(String s,String str,int starti,int startj,String res)

{

if(starti==s.length() && startj==str.length()){

if(!hs.contains(res))

{

if(check(res))

hs.add(res);

}

}

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

{

if(startj==str.length())

{

res = res + s.substring(i,s.length());

if(!hs.contains(res))

{

if(check(res))

hs.add(res);

}

break;

}

else if(s.charAt(i)==str.charAt(startj))

{

fun(s,str,i+1,startj+1,res);

res = res + s.charAt(i);

}

else

res = res + s.charAt(i);

}

}

String removal(String s)

{

Stack<Character> stk = new Stack<>();

String str = "";

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

{

if(s.charAt(i)=='(')

{

count1++;

stk.push('(');

}

else if(s.charAt(i)==')')

{

count2++;

if(stk.isEmpty())

stk.push(')');

else

{

if(stk.peek()=='(')

stk.pop();

else

stk.push(')');

}

}

else

str = str + s.charAt(i);

}

if(count1==0 || count2==0)

return str;

StringBuilder sb = new StringBuilder();

while(!stk.isEmpty())

sb.append(stk.pop());

return sb.reverse().toString();

}

public List<String> removeInvalidParentheses(String s)

{

String str = removal(s);

if(count1==0 || count2==0)

{

list.add(str);

return list;

}

fun(s,str,0,0,"");

list = new ArrayList<>(hs);

return list;

}

}

**Approach 1: Backtracking**

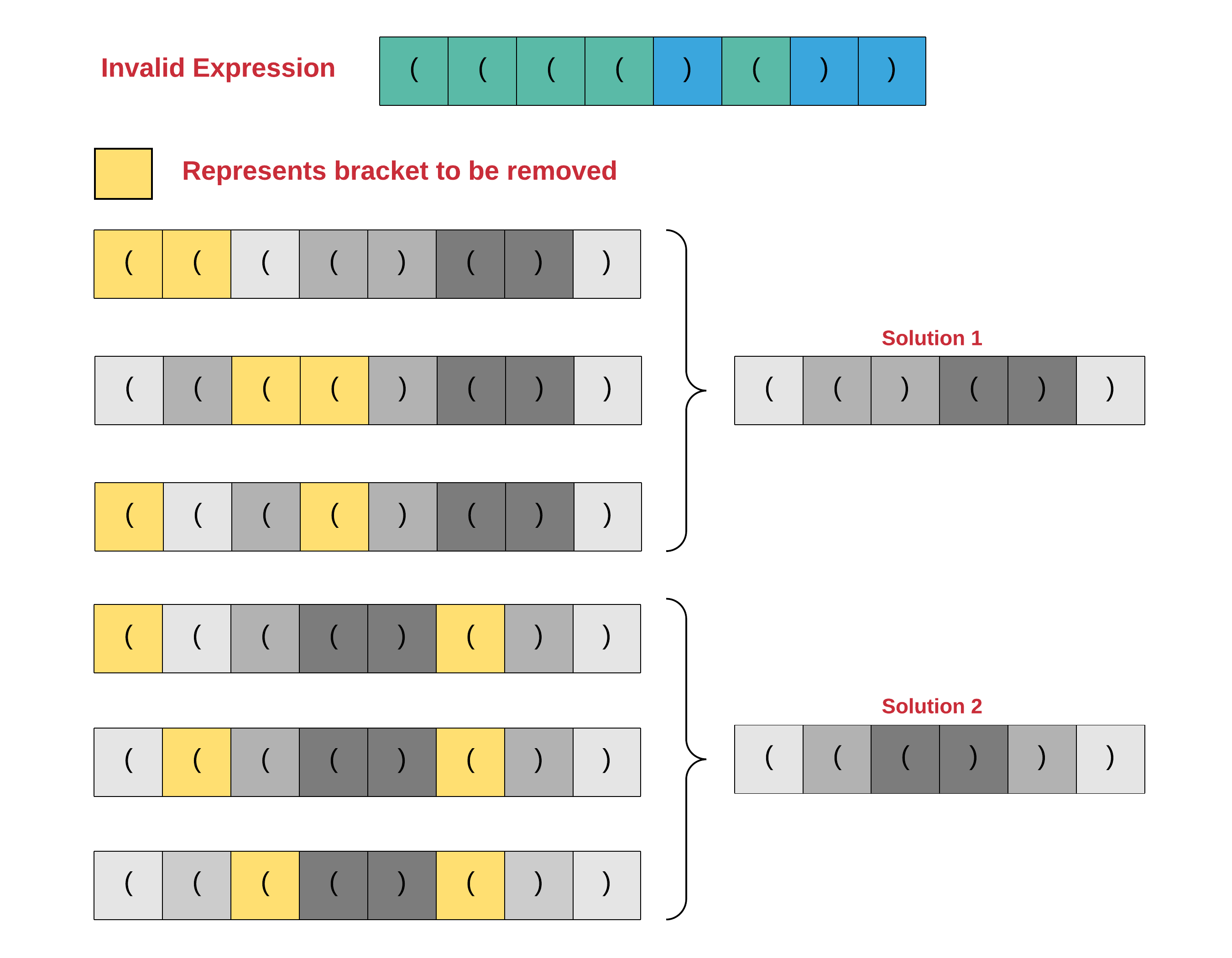
**Intuition**

For this question, we are given an expression consisting of parentheses and there can be some misplaced or extra brackets in the expression that cause it to be invalid. An expression consisting of parentheses is considered valid only when every closing bracket has a corresponding opening bracket and vice versa.

This means if we start looking at each of the bracket from left to right, as soon as we encounter a closing bracket, there should be an unmatched opening bracket available to match it. Otherwise the expression would become invalid. The expression can also become invalid if the number of opening parentheses i.e. ( are more than the number of closing parentheses i.e. ).

Let us look at an invalid expression and all the possible valid expressions that can be formed from it by removing some of the brackets. There is no restriction on which parentheses we can remove. We simply have to make the expression valid.

The only condition is that we should be removing the minimum number of brackets to make an invalid expression, valid. If this condition was not present, we could potentially remove most of the brackets and come down to say 2 brackets in the end which form () and that would be a valid expression.



An important thing to observe in the above diagram is that there are multiple ways of reaching the same solution i.e. say the optimal number of parentheses to be removed to make the original expression valid is K. We can remove multiple different sets of K brackets that will eventually give us the same final expression. But, each valid expression should be recorded only once. We have to take care of this in our solution. Note that there are other possible ways of reaching one of the two valid expressions shown above. We have simply shown 3 ways each for the two valid expressions.

Coming back to our problem, the question that now arises is, how to decide which of the parentheses to remove?

Since we don't know which of the brackets can possibly be removed, we try out all the options!

For every bracket we have two choices:

* Either it can be considered a part of the final expression OR
* It can be ignored i.e. we can delete it from our final expression.

Such kind of problems where we have multiple options and we have no strategy or metric of deciding greedily which option to take, we try out all of the options and see which ones lead to an answer. These type of problems are perfect candidates for the programming paradigm, Recursion.

**Algorithm**

1. Initialize an array that will store all of our valid expressions finally.
2. Start with the leftmost bracket in the given sequence and proceed right in the recursion.
3. The state of recursion is defined by the index which we are currently processing in the original expression. Let this index be represented by the character i. Also, we have two different variables left\_count and right\_count that represent the number of left and right parentheses we have added to our expression till now. These are the parentheses that were considered.
4. If the current character i.e. S[i] (considering S is the expression string) is neither a closing or an opening parenthesis, then we simply add this character to our final solution string for the current recursion.
5. However, if the current character is either of the two brackets i.e. S[i] == '(' or S[i] == ')', then we have two options. We can either discard this character by marking it an invalid character or we can consider this bracket to be a part of the final expression.
6. When all of the parentheses in the original expression have been processed, we simply check if the expression represented by expr i.e. the expression formed till now is valid one or not. The way we check if the final expression is valid or not is by looking at the values in left\_count and right\_count. For an expression to be valid left\_count == right\_count. If it is indeed valid, then it could be one of our possible solutions.
   * Even though we have a valid expression, we also need to keep track of the number of removals we did to get this expression. This is done by another variable passed in recursion called rem\_count.
   * Once recursion finishes we check if the current value of rem\_count is < the least number of steps we took to form a valid expression till now i.e. the global minima. If this is not the case, we don't record the new expression, else we record it.

One small optimization that we can do from an implementation perspective is introducing some sort of pruning in our algorithm. Right now we simply go till the very end i.e. process all of the parentheses and when we are done processing all of them, we check if the expression we have can be considered or not.

We have to wait till the very end to decide if the expression formed in recursion is a valid expression or not. Is there a way for us to cutoff from some of the recursion paths early on because they wouldn't lead to a solution? The answer to this is Yes! The optimization is based on the following idea.

For a left bracket encountered during recursion, if we decide to consider it, then it may or may not lead to an invalid final expression. It may lead to an invalid expression eventually if there are no matching closing bracket available afterwards. But, we don't know for sure if this will happen or not.

However, for a closing bracket, if we decide to keep it as a part of our final expression (remember for every bracket we have two options, either to keep it or to remove it and recurse further) and there is no corresponding opening bracket to match it in the expression till now, then it will definitely lead to an invalid expression no matter what we do afterwards.

e.g.

( ( ) ) )

In this case the third closing bracket will make the expression invalid. No matter what comes afterwards, this will give us an invalid expression and if such a thing happens, we shouldn't recurse further and simply prune the recursion tree.

That is why, in addition to having the index in the original string/expression which we are currently processing and the expression string formed till now, we also keep track of the number of left and right parentheses. Whenever we keep a left parenthesis in the expression, we increment its counter. For a right parenthesis, we check if right\_count < left\_count. If this is the case then only we consider that right parenthesis and recurse further. Otherwise we don't as we know it will make the expression invalid. This simple optimization saves a lot of runtime.

Now, let us look at the implementation for this algorithm.

class Solution {

private Set<String> validExpressions = new HashSet<String>();

private int minimumRemoved;

private void reset() {

this.validExpressions.clear();

this.minimumRemoved = Integer.MAX\_VALUE;

}

private void recurse(

String s,

int index,

int leftCount,

int rightCount,

StringBuilder expression,

int removedCount) {

// If we have reached the end of string.

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

// If the current expression is valid.

if (leftCount == rightCount) {

// If the current count of removed parentheses is <= the current minimum count

if (removedCount <= this.minimumRemoved) {

// Convert StringBuilder to a String. This is an expensive operation.

// So we only perform this when needed.

String possibleAnswer = expression.toString();

// If the current count beats the overall minimum we have till now

if (removedCount < this.minimumRemoved) {

this.validExpressions.clear();

this.minimumRemoved = removedCount;

}

this.validExpressions.add(possibleAnswer);

}

}

} else {

char currentCharacter = s.charAt(index);

int length = expression.length();

// If the current character is neither an opening bracket nor a closing one,

// simply recurse further by adding it to the expression StringBuilder

if (currentCharacter != '(' && currentCharacter != ')') {

expression.append(currentCharacter);

this.recurse(s, index + 1, leftCount, rightCount, expression, removedCount);

expression.deleteCharAt(length);

} else {

// Recursion where we delete the current character and move forward

this.recurse(s, index + 1, leftCount, rightCount, expression, removedCount + 1);

expression.append(currentCharacter);

// If it's an opening parenthesis, consider it and recurse

if (currentCharacter == '(') {

this.recurse(s, index + 1, leftCount + 1, rightCount, expression, removedCount);

} else if (rightCount < leftCount) {

// For a closing parenthesis, only recurse if right < left

this.recurse(s, index + 1, leftCount, rightCount + 1, expression, removedCount);

}

// Undoing the append operation for other recursions.

expression.deleteCharAt(length);

}

}

}

public List<String> removeInvalidParentheses(String s) {

this.reset();

this.recurse(s, 0, 0, 0, new StringBuilder(), 0);

return new ArrayList(this.validExpressions);

}

}

**Complexity analysis**

* Time Complexity : O(2^N)*O*(2*N*) since in the worst case we will have only left parentheses in the expression and for every bracket we will have two options i.e. whether to remove it or consider it. Considering that the expression has N*N* parentheses, the time complexity will be O(2^N)*O*(2*N*).
* Space Complexity : O(N)*O*(*N*) because we are resorting to a recursive solution and for a recursive solution there is always stack space used as internal function states are saved onto a stack during recursion. The maximum depth of recursion decides the stack space used. Since we process one character at a time and the base case for the recursion is when we have processed all of the characters of the expression string, the size of the stack would be O(N)*O*(*N*). Note that we are not considering the space required to store the valid expressions. We only count the intermediate space here.

**Approach 2: Limited Backtracking!**

Although the previous solution does get accepted on the platform, it is a very inefficient solution because we try removing each and every possible parentheses from the expression and in the end we check two things:

1. if the expression is valid or not
2. if the total number of removed parentheses removed in the current recursion is less than the global minimum till now or not.

We cannot determine which of the parentheses are misplaced because, as the problem statement puts across, we can remove multiple combinations of parentheses and end up with a valid expression. This means there can be multiple valid expressions from a single invalid expression and we have to find all of them.

The one thing all these valid expressions have in common is that they will all be of the same length i.e. as compared to the original expression, all of these expressions will have the same number of characters removed.

What if we could determine this count?

What if in addition to determining this count of characters to be removed, we could also determine the number of left parentheses and number of right parentheses to be removed from the original expression to get **any** valid expression?

This would cut down the computations immensely and the runtime would plummet as a result. The reason for this is, if we knew how many left and right parentheses are to be removed from the original expression to get a valid expression, we would cut down on so many unwanted recursive calls.

Imagine the original expression to be 1000 characters with only 3 misplaced ( parentheses and 2 misplaced ) parentheses. In our previous solution we would end up trying to remove each one of left and right parentheses and try to reach a valid expression in the end whereas we should only be trying out removing 3 ( brackets and 2 ) brackets.

This is the exact number of ( and ) that have to be removed to get a valid expression. No more, no less.

Let us look at how we can find out the number of misplaced left and right parentheses in a given expression first and then we will slightly modify our original algorithm to incorporate these counts as well.

1. We process the expression one bracket at a time starting from the left.
2. Suppose we encounter an opening bracket i.e. (, it may or may not lead to an invalid expression because there can be a matching ending bracket somewhere in the remaining part of the expression. Here, we simply increment the counter keeping track of left parentheses till now. left += 1
3. If we encounter a closing bracket, this has two meanings:
   * Either there was no matching opening bracket for this closing bracket and in that case we have an invalid expression. This is the case when left == 0 i.e. when there are no unmatched left brackets available. In such a case we increment another counter say right += 1 to represent misplaced right parentheses.
   * Or, we had some unmatched opening bracket available to match this closing bracket. This is the case when left > 0. In this case we simply decrement the left counter we had i.e. left -= 1
4. Continue processing the string until all parentheses have been processed.
5. In the end the values of left and right would tell us the number of unmatched ( and ) parentheses respectively.

Now that we have these two values available that tell us the total number of left i.e. ( and right i.e. ) parentheses that have to be removed to make the invalid expression valid, we will modify our original algorithm discussed in the previous session to avoid unwanted recursions.

**Algorithm**

The overall algorithm remains exactly the same as before. The changes that we will incorporate are listed below:

* The state of the recursion is now defined by five different variables:
  1. index which represents the current character that we have to process in the original string.
  2. left\_count which represents the number of left parentheses that have been added to the expression we are building.
  3. right\_count which represents the number of right parentheses that have been added to the expression we are building.
  4. left\_rem is the number of left parentheses that remain to be removed.
  5. right\_rem represents the number of right parentheses that remain to be removed. Overall, for the final expression to be valid, left\_rem == 0 and right\_rem == 0.
* When we decide to not consider a parenthesis i.e. delete a parenthesis, be it a left or a right parentheses, we have to consider their corresponding remaining counts as well. This means that we can only discard a left parentheses if left\_rem > 0 and similarly for the right one we will check for right\_rem > 0.
* There are no changes to checks for **considering** a parenthesis. Only the conditions change for **discarding** a parenthesis.
* Condition for an expression being valid in the base case would now become left\_rem == 0 and right\_rem == 0. Note that we don't have to check if left\_count == right\_count anymore because in the case of a valid expression, we would have removed all the misplaced or invalid parenthesis by the time the recursion ends. So, the only check we need if left\_rem == 0 and right\_rem == 0.

The most important thing here is that we have completely gotten rid of checking if the number of parentheses removed is lesser than the current minimum or not. The reason for this is we always remove the same number of parentheses as defined by left\_rem + right\_rem at the start of recursion.

Now let us look at the implementation for this modified version of algorithm.

class Solution {

private Set<String> validExpressions = new HashSet<String>();

private void recurse(

String s,

int index,

int leftCount,

int rightCount,

int leftRem,

int rightRem,

StringBuilder expression) {

// If we reached the end of the string, just check if the resulting expression is

// valid or not and also if we have removed the total number of left and right

// parentheses that we should have removed.

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

if (leftRem == 0 && rightRem == 0) {

this.validExpressions.add(expression.toString());

}

} else {

char character = s.charAt(index);

int length = expression.length();

// The discard case. Note that here we have our pruning condition.

// We don't recurse if the remaining count for that parenthesis is == 0.

if ((character == '(' && leftRem > 0) || (character == ')' && rightRem > 0)) {

this.recurse(

s,

index + 1,

leftCount,

rightCount,

leftRem - (character == '(' ? 1 : 0),

rightRem - (character == ')' ? 1 : 0),

expression);

}

expression.append(character);

// Simply recurse one step further if the current character is not a parenthesis.

if (character != '(' && character != ')') {

this.recurse(s, index + 1, leftCount, rightCount, leftRem, rightRem, expression);

} else if (character == '(') {

// Consider an opening bracket.

this.recurse(s, index + 1, leftCount + 1, rightCount, leftRem, rightRem, expression);

} else if (rightCount < leftCount) {

// Consider a closing bracket.

this.recurse(s, index + 1, leftCount, rightCount + 1, leftRem, rightRem, expression);

}

// Delete for backtracking.

expression.deleteCharAt(length);

}

}

public List<String> removeInvalidParentheses(String s) {

int left = 0, right = 0;

// First, we find out the number of misplaced left and right parentheses.

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

// Simply record the left one.

if (s.charAt(i) == '(') {

left++;

} else if (s.charAt(i) == ')') {

// If we don't have a matching left, then this is a misplaced right, record it.

right = left == 0 ? right + 1 : right;

// Decrement count of left parentheses because we have found a right

// which CAN be a matching one for a left.

left = left > 0 ? left - 1 : left;

}

}

this.recurse(s, 0, 0, 0, left, right, new StringBuilder());

return new ArrayList<String>(this.validExpressions);

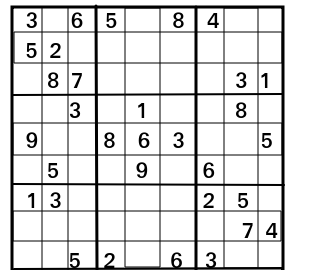
}

}

**Complexity analysis**

* Time Complexity : The optimization that we have performed is simply a better form of pruning. Pruning here is something that will vary from one test case to another. In the worst case, we can have something like ((((((((( and the left\_rem = len(S) and in such a case we can discard all of the characters because all are misplaced. So, in the worst case we **still** have 2 options per parenthesis and that gives us a complexity of O(2^N)*O*(2*N*).
* Space Complexity : The space complexity remains the same i.e. O(N)*O*(*N*) as previous solution. We have to go to a maximum recursion depth of N*N* before hitting the base case. Note that we are not considering the space required to store the valid expressions. We only count the intermediate space here.

# 258. Sudoku Solver

Given an incomplete Sudoku configuration in terms of a 9 x 9  2-D square matrix (grid[][]), the task to find a solved Sudoku. For simplicity, you may assume that there will be only one unique solution.  
  
**Sample Sudoku for you to get the logic for its solution:**  
  


**Example 1:**

**Input:**

grid[][] =

[[3 0 6 5 0 8 4 0 0],

[5 2 0 0 0 0 0 0 0],

[0 8 7 0 0 0 0 3 1 ],

[0 0 3 0 1 0 0 8 0],

[9 0 0 8 6 3 0 0 5],

[0 5 0 0 9 0 6 0 0],

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

[0 0 0 0 0 0 0 7 4],

[0 0 5 2 0 6 3 0 0]]

**Output:**

3 1 6 5 7 8 4 9 2

5 2 9 1 3 4 7 6 8

4 8 7 6 2 9 5 3 1

2 6 3 4 1 5 9 8 7

9 7 4 8 6 3 1 2 5

8 5 1 7 9 2 6 4 3

1 3 8 9 4 7 2 5 6

6 9 2 3 5 1 8 7 4

7 4 5 2 8 6 3 1 9

**Your Task:**  
You need to complete the two functions:  
**SolveSudoku()**: Takes a grid as its argument and returns true if a solution is possible and false if it is not.  
**printGrid()**: Takes a grid as its argument and prints the 81 numbers of the solved Sudoku in a single line with space separation.

**Expected Time Complexity:** O(9N\*N).  
**Expected Auxiliary Space:** O(N\*N).

**Constraints:**  
0 ≤ grid[i][j] ≤ 9

## Solution:

**My Implementation:**

class Solution

{

public:

bool flag = false;

bool check(int grid[N][N], int x, int y, int num){

int block\_x = (x/3)\*3, block\_y = (y/3)\*3;

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

if(grid[i][y]==num)

return false;

if(grid[x][i]==num)

return false;

if(grid[block\_x+(i/3)][block\_y+(i%3)]==num)

return false;

}

return true;

}

void solve(int grid[N][N], int x, int y){

if(x==N){

//printGrid(grid);

flag = true;

return;

}

if(grid[x][y]==0){

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

if(check(grid, x, y, i)==true){

grid[x][y] = i;

if(y==N-1)

solve(grid, x+1, 0);

else

solve(grid, x, y+1);

if(!flag)

grid[x][y]=0;

}

if(flag)

break;

}

}

else{

if(y==N-1)

solve(grid, x+1, 0);

else

solve(grid, x, y+1);

}

}

//Function to find a solved Sudoku.

bool SolveSudoku(int grid[N][N])

{

flag = false;

solve(grid, 0, 0);

return flag;

}

//Function to print grids of the Sudoku.

void printGrid (int grid[N][N])

{

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

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

cout<<grid[i][j]<<" ";

}

}

};

**Another Implementation:**

class Solution

{

public:

bool flag = false;

bool check(int grid[N][N], int x, int y, int num){

int block\_x = (x/3)\*3, block\_y = (y/3)\*3;

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

if(grid[i][y]==num)

return false;

if(grid[x][i]==num)

return false;

if(grid[block\_x+(i/3)][block\_y+(i%3)]==num)

return false;

}

return true;

}

//Function to find a solved Sudoku.

bool SolveSudoku(int grid[N][N])

{

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

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

if(grid[i][j]==0){

for(int num=1;num<=N;num++){

if(check(grid,i,j,num)==true){

grid[i][j] = num;

if(SolveSudoku(grid)==true)

return true;

else

grid[i][j] = 0;

}

}

return false;

}

}

}

return true;

}

//Function to print grids of the Sudoku.

void printGrid (int grid[N][N])

{

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

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

cout<<grid[i][j]<<" ";

}

}

};

**Method 1:** Simple.  
**Approach:** The naive approach is to generate all possible configurations of numbers from 1 to 9 to fill the empty cells. Try every configuration one by one until the correct configuration is found, i.e. for every unassigned position fill the position with a number from 1 to 9. After filling all the unassigned position check if the matrix is safe or not. If safe print else recurs for other cases.

**Algorithm:**

1. Create a function that checks if the given matrix is valid sudoku or not. Keep Hashmap for the row, column and boxes. If any number has a frequency greater than 1 in the hashMap return false else return true;
2. Create a recursive function that takes a grid and the current row and column index.
3. Check some base cases. If the index is at the end of the matrix, i.e. i=N-1 and j=N then check if the grid is safe or not, if safe print the grid and return true else return false. The other base case is when the value of column is N, i.e j = N, then move to next row, i.e. i++ and j = 0.
4. if the current index is not assigned then fill the element from 1 to 9 and recur for all 9 cases with the index of next element, i.e. i, j+1. if the recursive call returns true then break the loop and return true.
5. if the current index is assigned then call the recursive function with index of next element, i.e. i, j+1

#include <iostream>

using namespace std;

// N is the size of the 2D matrix N\*N

#define N 9

/\* A utility function to print grid \*/

void print(int arr[N][N])

{

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

{

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

cout << arr[i][j] << " ";

cout << endl;

}

}

// Checks whether it will be

// legal to assign num to the

// given row, col

bool isSafe(int grid[N][N], int row,

int col, int num)

{

// Check if we find the same num

// in the similar row , we

// return false

for (int x = 0; x <= 8; x++)

if (grid[row][x] == num)

return false;

// Check if we find the same num in

// the similar column , we

// return false

for (int x = 0; x <= 8; x++)

if (grid[x][col] == num)

return false;

// Check if we find the same num in

// the particular 3\*3 matrix,

// we return false

int startRow = row - row % 3,

startCol = col - col % 3;

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

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

if (grid[i + startRow][j +

startCol] == num)

return false;

return true;

}

/\* Takes a partially filled-in grid and attempts

to assign values to all unassigned locations in

such a way to meet the requirements for

Sudoku solution (non-duplication across rows,

columns, and boxes) \*/

bool solveSudoku(int grid[N][N], int row, int col)

{

// Check if we have reached the 8th

// row and 9th column (0

// indexed matrix) , we are

// returning true to avoid

// further backtracking

if (row == N - 1 && col == N)

return true;

// Check if column value becomes 9 ,

// we move to next row and

// column start from 0

if (col == N) {

row++;

col = 0;

}

// Check if the current position of

// the grid already contains

// value >0, we iterate for next column

if (grid[row][col] > 0)

return solveSudoku(grid, row, col + 1);

for (int num = 1; num <= N; num++)

{

// Check if it is safe to place

// the num (1-9) in the

// given row ,col ->we

// move to next column

if (isSafe(grid, row, col, num))

{

/\* Assigning the num in

the current (row,col)

position of the grid

and assuming our assigned

num in the position

is correct \*/

grid[row][col] = num;

// Checking for next possibility with next

// column

if (solveSudoku(grid, row, col + 1))

return true;

}

// Removing the assigned num ,

// since our assumption

// was wrong , and we go for

// next assumption with

// diff num value

grid[row][col] = 0;

}

return false;

}

// Driver Code

int main()

{

// 0 means unassigned cells

int grid[N][N] = { { 3, 0, 6, 5, 0, 8, 4, 0, 0 },

{ 5, 2, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 8, 7, 0, 0, 0, 0, 3, 1 },

{ 0, 0, 3, 0, 1, 0, 0, 8, 0 },

{ 9, 0, 0, 8, 6, 3, 0, 0, 5 },

{ 0, 5, 0, 0, 9, 0, 6, 0, 0 },

{ 1, 3, 0, 0, 0, 0, 2, 5, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 7, 4 },

{ 0, 0, 5, 2, 0, 6, 3, 0, 0 } };

if (solveSudoku(grid, 0, 0))

print(grid);

else

cout << "no solution exists " << endl;

return 0;

// This is code is contributed by Pradeep Mondal P

}

**Output**

3 1 6 5 7 8 4 9 2

5 2 9 1 3 4 7 6 8

4 8 7 6 2 9 5 3 1

2 6 3 4 1 5 9 8 7

9 7 4 8 6 3 1 2 5

8 5 1 7 9 2 6 4 3

1 3 8 9 4 7 2 5 6

6 9 2 3 5 1 8 7 4

7 4 5 2 8 6 3 1 9

**Complexity Analysis:**

* **Time complexity:** O(9^(n\*n)).   
  For every unassigned index, there are 9 possible options so the time complexity is O(9^(n\*n)).
* **Space Complexity:** O(n\*n).   
  To store the output array a matrix is needed.

**Method 2:** Backtracking.   
**Approach:**   
Like all other [Backtracking problems](https://www.geeksforgeeks.org/archives/tag/backtracking), Sudoku can be solved by one by one assigning numbers to empty cells. Before assigning a number, check whether it is safe to assign. Check that the same number is not present in the current row, current column and current 3X3 subgrid. After checking for safety, assign the number, and recursively check whether this assignment leads to a solution or not. If the assignment doesn’t lead to a solution, then try the next number for the current empty cell. And if none of the number (1 to 9) leads to a solution, return false and print no solution exists.

**Algorithm:**

1. Create a function that checks after assigning the current index the grid becomes unsafe or not. Keep Hashmap for a row, column and boxes. If any number has a frequency greater than 1 in the hashMap return false else return true; hashMap can be avoided by using loops.
2. Create a recursive function that takes a grid.
3. Check for any unassigned location. If present then assign a number from 1 to 9, check if assigning the number to current index makes the grid unsafe or not, if safe then recursively call the function for all safe cases from 0 to 9. if any recursive call returns true, end the loop and return true. If no recursive call returns true then return false.
4. If there is no unassigned location then return true.

// A Backtracking program in

// C++ to solve Sudoku problem

#include <bits/stdc++.h>

using namespace std;

// UNASSIGNED is used for empty

// cells in sudoku grid

#define UNASSIGNED 0

// N is used for the size of Sudoku grid.

// Size will be NxN

#define N 9

// This function finds an entry in grid

// that is still unassigned

bool FindUnassignedLocation(int grid[N][N],

int& row, int& col);

// Checks whether it will be legal

// to assign num to the given row, col

bool isSafe(int grid[N][N], int row,

int col, int num);

/\* Takes a partially filled-in grid and attempts

to assign values to all unassigned locations in

such a way to meet the requirements for

Sudoku solution (non-duplication across rows,

columns, and boxes) \*/

bool SolveSudoku(int grid[N][N])

{

int row, col;

// If there is no unassigned location,

// we are done

if (!FindUnassignedLocation(grid, row, col))

// success!

return true;

// Consider digits 1 to 9

for (int num = 1; num <= 9; num++)

{

// Check if looks promising

if (isSafe(grid, row, col, num))

{

// Make tentative assignment

grid[row][col] = num;

// Return, if success

if (SolveSudoku(grid))

return true;

// Failure, unmake & try again

grid[row][col] = UNASSIGNED;

}

}

// This triggers backtracking

return false;

}

/\* Searches the grid to find an entry that is

still unassigned. If found, the reference

parameters row, col will be set the location

that is unassigned, and true is returned.

If no unassigned entries remain, false is returned. \*/

bool FindUnassignedLocation(int grid[N][N],

int& row, int& col)

{

for (row = 0; row < N; row++)

for (col = 0; col < N; col++)

if (grid[row][col] == UNASSIGNED)

return true;

return false;

}

/\* Returns a boolean which indicates whether

an assigned entry in the specified row matches

the given number. \*/

bool UsedInRow(int grid[N][N], int row, int num)

{

for (int col = 0; col < N; col++)

if (grid[row][col] == num)

return true;

return false;

}

/\* Returns a boolean which indicates whether

an assigned entry in the specified column

matches the given number. \*/

bool UsedInCol(int grid[N][N], int col, int num)

{

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

if (grid[row][col] == num)

return true;

return false;

}

/\* Returns a boolean which indicates whether

an assigned entry within the specified 3x3 box

matches the given number. \*/

bool UsedInBox(int grid[N][N], int boxStartRow,

int boxStartCol, int num)

{

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

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

if (grid[row + boxStartRow]

[col + boxStartCol] ==

num)

return true;

return false;

}

/\* Returns a boolean which indicates whether

it will be legal to assign num to the given

row, col location. \*/

bool isSafe(int grid[N][N], int row,

int col, int num)

{

/\* Check if 'num' is not already placed in

current row, current column

and current 3x3 box \*/

return !UsedInRow(grid, row, num)

&& !UsedInCol(grid, col, num)

&& !UsedInBox(grid, row - row % 3,

col - col % 3, num)

&& grid[row][col] == UNASSIGNED;

}

/\* A utility function to print grid \*/

void printGrid(int grid[N][N])

{

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

{

for (int col = 0; col < N; col++)

cout << grid[row][col] << " ";

cout << endl;

}

}

// Driver Code

int main()

{

// 0 means unassigned cells

int grid[N][N] = { { 3, 0, 6, 5, 0, 8, 4, 0, 0 },

{ 5, 2, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 8, 7, 0, 0, 0, 0, 3, 1 },

{ 0, 0, 3, 0, 1, 0, 0, 8, 0 },

{ 9, 0, 0, 8, 6, 3, 0, 0, 5 },

{ 0, 5, 0, 0, 9, 0, 6, 0, 0 },

{ 1, 3, 0, 0, 0, 0, 2, 5, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 7, 4 },

{ 0, 0, 5, 2, 0, 6, 3, 0, 0 } };

if (SolveSudoku(grid) == true)

printGrid(grid);

else

cout << "No solution exists";

return 0;

}

**Output**

3 1 6 5 7 8 4 9 2

5 2 9 1 3 4 7 6 8

4 8 7 6 2 9 5 3 1

2 6 3 4 1 5 9 8 7

9 7 4 8 6 3 1 2 5

8 5 1 7 9 2 6 4 3

1 3 8 9 4 7 2 5 6

6 9 2 3 5 1 8 7 4

7 4 5 2 8 6 3 1 9

**Complexity Analysis:**

* **Time complexity:** O(9^(n\*n)).   
  For every unassigned index, there are 9 possible options so the time complexity is O(9^(n\*n)). The time complexity remains the same but there will be some early pruning so the time taken will be much less than the naive algorithm but the upper bound time complexity remains the same.
* **Space Complexity:**O(n\*n).   
  To store the output array a matrix is needed.

**Method 3**: Using Bit Masks.

**Approach:**This method is a slight optimization to the above 2 methods.  For each row/column/box create a bitmask and for each element in the grid set the bit at position ‘value’ to 1 in the corresponding bitmasks, for O(1) checks.

**Algorithm:**

1. Create 3 arrays of size N (one for rows, columns, and boxes).
2. The boxes are indexed from 0 to 8. (in order to find the box-index of an element we use the following formula: **row / 3 \* 3 + column / 3**).
3. Map the initial values of the grid first.
4. Each time we **add/remove** an element to/from the grid **set the bit to 1/0** to the corresponding bitmasks.

#include <bits/stdc++.h>

using namespace std;

#define N 9

// Bitmasks for each row/column/box

int row[N], col[N], box[N];

bool seted = false;

// Utility function to find the box index

// of an element at position [i][j] in the grid

int getBox(int i,int j)

{

return i / 3 \* 3 + j / 3;

}

// Utility function to check if a number

// is present in the coresponding row/column/box

bool isSafe(int i,int j,int number)

{

return !((row[i] >> number) & 1)

&& !((col[j] >> number) & 1)

&& !((box[getBox(i,j)] >> number) & 1);

}

// Utility function to set the initial values of a Sudoku board

// (map the values in the bitmasks)

void setInitialValues(int grid[N][N])

{

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

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

row[i] |= 1 << grid[i][j],

col[j] |= 1 << grid[i][j],

box[getBox(i, j)] |= 1 << grid[i][j];

}

/\* Takes a partially filled-in grid and attempts

to assign values to all unassigned locations in

such a way to meet the requirements for

Sudoku solution (non-duplication across rows,

columns, and boxes) \*/

bool SolveSudoku(int grid[N][N] ,int i, int j)

{

// Set the initial values

if(!seted)

seted = true,

setInitialValues(grid);

if(i == N - 1 && j == N)

return true;

if(j == N)

j = 0,

i++;

if(grid[i][j])

return SolveSudoku(grid, i, j + 1);

for (int nr = 1; nr <= N;nr++)

{

if(isSafe(i, j, nr))

{

/\* Assign nr in the

current (i, j)

position and

add nr to each bitmask

\*/

grid[i][j] = nr;

row[i] |= 1 << nr;

col[j] |= 1 << nr;

box[getBox(i, j)] |= 1 << nr;

if(SolveSudoku(grid, i,j + 1))

return true;

// Remove nr from each bitmask

// and search for another possibility

row[i] &= ~(1 << nr);

col[j] &= ~(1 << nr);

box[getBox(i, j)] &= ~(1 << nr);

}

grid[i][j] = 0;

}

return false;

}

// Utility function to print the solved grid

void print(int grid[N][N])

{

for (int i = 0; i < N; i++, cout << '\n')

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

cout << grid[i][j] << ' ';

}

// Driver Code

int main()

{

// 0 means unassigned cells

int grid[N][N] = { { 3, 0, 6, 5, 0, 8, 4, 0, 0 },

{ 5, 2, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 8, 7, 0, 0, 0, 0, 3, 1 },

{ 0, 0, 3, 0, 1, 0, 0, 8, 0 },

{ 9, 0, 0, 8, 6, 3, 0, 0, 5 },

{ 0, 5, 0, 0, 9, 0, 6, 0, 0 },

{ 1, 3, 0, 0, 0, 0, 2, 5, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 7, 4 },

{ 0, 0, 5, 2, 0, 6, 3, 0, 0 }};

if (SolveSudoku(grid,0 ,0))

print(grid);

else

cout << "No solution exists\n";

return 0;

}

**Output**

3 1 6 5 7 8 4 9 2

5 2 9 1 3 4 7 6 8

4 8 7 6 2 9 5 3 1

2 6 3 4 1 5 9 8 7

9 7 4 8 6 3 1 2 5

8 5 1 7 9 2 6 4 3

1 3 8 9 4 7 2 5 6

6 9 2 3 5 1 8 7 4

7 4 5 2 8 6 3 1 9

**Complexity Analysis:**

* **Time complexity:** O(9^(n\*n)). For every unassigned index, there are 9 possible options so the time complexity is O(9^(n\*n)). The time complexity remains the same but checking if a number is safe to use is much faster, O(1).
* **Space Complexity:** O(n\*n). To store the output array a matrix is needed, and 3 extra arrays of size n are needed for the bitmasks.

# 259. m Coloring Problem

Given an undirected graph and an integer **M**. The task is to determine if the graph can be colored with at most M colors 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. Print 1 if it is possible to colour vertices and 0 otherwise.

**Example 1:**

**Input:**

N = 4

M = 3

E = 5

Edges[] = {(0,1),(1,2),(2,3),(3,0),(0,2)}

**Output:** 1

**Explanation:** It is possible to colour the

given graph using 3 colours.

**Example 2:**

**Input:**

N = 3

M = 2

E = 3

Edges[] = {(0,1),(1,2),(0,2)}

**Output:** 0

**Your Task:**  
Your task is to complete the function **graphColoring()** which takes the 2d-array graph[], the number of colours and the number of nodes as inputs and returns **true** if answer exists otherwise **false**. 1 is printed if the returned value is **true,**0 otherwise. The printing is done by the driver's code.  
**Note**: In Example there are Edges not the graph.Graph will be like, if there is an edge between vertex X and vertex Y graph[] will contain 1 at graph[X-1][Y-1], else 0. In 2d-array graph[ ], nodes are 0-based indexed, i.e. from 0 to N-1.Function will be contain 2-D graph not the edges.  
  
**Expected Time Complexity:** O(MN).  
**Expected Auxiliary Space:** O(N).

**Constraints:**  
1 ≤ N ≤ 20  
1 ≤ E ≤ (N\*(N-1))/2  
1 ≤ M ≤ N

## Solution:

**Method 1:** Naive.

**Naive Approach:** Generate all possible configurations of colors. Since each node can be coloured using any of the m available colours, the total number of colour configurations possible are m^V.   
After generating a configuration of colour, check if the adjacent vertices have the same colour or not. If the conditions are met, print the combination and break the loop.

**Algorithm:**

1. Create a recursive function that takes current index, number of vertices and output color array.
2. If the current index is equal to number of vertices. Check if the output color configuration is safe, i.e check if the adjacent vertices do not have same color. If the conditions are met, print the configuration and break.
3. Assign a color to a vertex (1 to m).
4. For every assigned color recursively call the function with next index and number of vertices
5. If any recursive function returns true break the loop and returns true.

Below is the implementation of the above idea:

#include<bits/stdc++.h>

using namespace std;

// Number of vertices in the graph

#define V 4

void printSolution(int color[]);

// check if the colored

// graph is safe or not

bool isSafe(bool graph[V][V], int color[])

{

// check for every edge

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

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

if (graph[i][j] && color[j] == color[i])

return false;

return true;

}

/\* This function solves the m Coloring

problem using recursion. It returns

false if the m colours cannot be assigned,

otherwise, return true and prints

assignments of colours to all vertices.

Please note that there may be more than

one solutions, this function prints one

of the feasible solutions.\*/

bool graphColoring(bool graph[V][V], int m, int i,

int color[V])

{

// if current index reached end

if (i == V) {

// if coloring is safe

if (isSafe(graph, color)) {

// Print the solution

printSolution(color);

return true;

}

return false;

}

// Assign each color from 1 to m

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

color[i] = j;

// Recur of the rest vertices

if (graphColoring(graph, m, i + 1, color))

return true;

color[i] = 0;

}

return false;

}

/\* A utility function to print solution \*/

void printSolution(int color[])

{

cout << "Solution Exists:" " Following are the assigned colors \n";

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

cout << " " << color[i];

cout << "\n";

}

// Driver code

int main()

{

/\* Create following graph and

test whether it is 3 colorable

(3)---(2)

| / |

| / |

| / |

(0)---(1)

\*/

bool graph[V][V] = {

{ 0, 1, 1, 1 },

{ 1, 0, 1, 0 },

{ 1, 1, 0, 1 },

{ 1, 0, 1, 0 },

};

int m = 3; // Number of colors

// Initialize all color values as 0.

// This initialization is needed

// correct functioning of isSafe()

int color[V];

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

color[i] = 0;

if (!graphColoring(graph, m, 0, color))

cout << "Solution does not exist";

return 0;

}

**Output**

Solution Exists: Following are the assigned colors

1 2 3 2

**Complexity Analysis:**

* **Time Complexity:** O(m^V).   
  There is a total O(m^V) combination of colors. So the time complexity is O(m^V).
* **Space Complexity:** O(V).   
  Recursive Stack of graphColoring(…) function will require O(V) space.

**Method 2:** [Backtracking](https://www.geeksforgeeks.org/backtracking-algorithms/).

**Approach:** The idea is to assign colors one by one to different vertices, starting from the vertex 0. Before assigning a color, check for safety by considering already assigned colors to the adjacent vertices i.e check if the adjacent vertices have the same color or not. If there is any color assignment that does not violate the conditions, mark the color assignment as part of the solution. If no assignment of color is possible then backtrack and return false.

**Algorithm:**

1. Create a recursive function that takes the graph, current index, number of vertices, and output color array.
2. If the current index is equal to the number of vertices. Print the color configuration in output array.
3. Assign a color to a vertex (1 to m).
4. For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) recursively call the function with next index and number of vertices
5. If any recursive function returns true break the loop and return true.
6. If no recursive function returns true then return false.

Below is the implementation of the above idea:

// C++ program for solution of M

// Coloring problem using backtracking

#include <iostream>

using namespace std;

// Number of vertices in the graph

#define V 4

void printSolution(int color[]);

/\* A utility function to check if

the current color assignment

is safe for vertex v i.e. checks

whether the edge exists or not

(i.e, graph[v][i]==1). If exist

then checks whether the color to

be filled in the new vertex(c is

sent in the parameter) is already

used by its adjacent

vertices(i-->adj vertices) or

not (i.e, color[i]==c) \*/

bool isSafe(int v, bool graph[V][V],

int color[], int c)

{

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

if (graph[v][i] && c == color[i])

return false;

return true;

}

/\* A recursive utility function

to solve m coloring problem \*/

bool graphColoringUtil(bool graph[V][V], int m,

int color[], int v)

{

/\* base case: If all vertices are

assigned a color then return true \*/

if (v == V)

return true;

/\* Consider this vertex v and

try different colors \*/

for(int c = 1; c <= m; c++)

{

/\* Check if assignment of color

c to v is fine\*/

if (isSafe(v, graph, color, c))

{

color[v] = c;

/\* recur to assign colors to

rest of the vertices \*/

if (graphColoringUtil(

graph, m, color, v + 1) == true)

return true;

/\* If assigning color c doesn't

lead to a solution then remove it \*/

color[v] = 0;

}

}

/\* If no color can be assigned to

this vertex then return false \*/

return false;

}

/\* This function solves the m Coloring

problem using Backtracking. It mainly

uses graphColoringUtil() to solve the

problem. It returns false if the m

colors cannot be assigned, otherwise

return true and prints assignments of

colors to all vertices. Please note

that there may be more than one solutions,

this function prints one of the

feasible solutions.\*/

bool graphColoring(bool graph[V][V], int m)

{

// Initialize all color values as 0.

// This initialization is needed

// correct functioning of isSafe()

int color[V];

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

color[i] = 0;

// Call graphColoringUtil() for vertex 0

if (graphColoringUtil(graph, m, color, 0) == false)

{

cout << "Solution does not exist";

return false;

}

// Print the solution

printSolution(color);

return true;

}

/\* A utility function to print solution \*/

void printSolution(int color[])

{

cout << "Solution Exists:"

<< " Following are the assigned colors"

<< "\n";

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

cout << " " << color[i] << " ";

cout << "\n";

}

// Driver code

int main()

{

/\* Create following graph and test

whether it is 3 colorable

(3)---(2)

| / |

| / |

| / |

(0)---(1)

\*/

bool graph[V][V] = { { 0, 1, 1, 1 },

{ 1, 0, 1, 0 },

{ 1, 1, 0, 1 },

{ 1, 0, 1, 0 }, };

// Number of colors

int m = 3;

graphColoring(graph, m);

return 0;

}

**Output**

Solution Exists: Following are the assigned colors

1 2 3 2

**Complexity Analysis:**

* **Time Complexity:** O(m^V).   
  There are total O(m^V) combination of colors. So time complexity is O(m^V). The upperbound time complexity remains the same but the average time taken will be less.
* **Space Complexity:** O(V).   
  Recursive Stack of graphColoring(…) function will require O(V) space.

**Method 3:**Using BFS

The approach here is to color each node from 1 to n initially by color 1. And start travelling BFS from an unvisited starting node to cover all connected components in one go. On reaching each node during BFS traversal, do the following:

* Check all edges of the given node.
* For each vertex connected to our node via an edge:
  + check if the color of the nodes is the same. If same, increase the color of the other node (not the current) by one.
  + check if it visited or unvisited. If not visited, mark it as visited and push it in a queue.
* Check condition for maxColors till now. If it exceeds M, return false

After visiting all nodes, return true (As no violating condition could be found while travelling).

// CPP program for the above approach

#include <bits/stdc++.h>

#include <iostream>

using namespace std;

class node

{

// A node class which stores the color and the edges

// connected to the node

public:

int color = 1;

set<int> edges;

};

int canPaint(vector<node>& nodes, int n, int m)

{

// Create a visited array of n

// nodes, initialized to zero

vector<int> visited(n + 1, 0);

// maxColors used till now are 1 as

// all nodes are painted color 1

int maxColors = 1;

// Do a full BFS traversal from

// all unvisited starting points

for (int sv = 1; sv <= n; sv++)

{

if (visited[sv])

continue;

// If the starting point is unvisited,

// mark it visited and push it in queue

visited[sv] = 1;

queue<int> q;

q.push(sv);

// BFS Travel starts here

while (!q.empty())

{

int top = q.front();

q.pop();

// Checking all adjacent nodes

// to "top" edge in our queue

for (auto it = nodes[top].edges.begin();

it != nodes[top].edges.end(); it++)

{

// IMPORTANT: If the color of the

// adjacent node is same, increase it by 1

if (nodes[top].color == nodes[\*it].color)

nodes[\*it].color += 1;

// If number of colors used shoots m, return

// 0

maxColors

= max(maxColors, max(nodes[top].color,

nodes[\*it].color));

if (maxColors > m)

return 0;

// If the adjacent node is not visited,

// mark it visited and push it in queue

if (!visited[\*it]) {

visited[\*it] = 1;

q.push(\*it);

}

}

}

}

return 1;

}

// Driver code

int main()

{

int n = 4;

bool graph[n][n] = {

{ 0, 1, 1, 1 },

{ 1, 0, 1, 0 },

{ 1, 1, 0, 1 },

{ 1, 0, 1, 0 }};

int m = 3; // Number of colors

// Create a vector of n+1

// nodes of type "node"

// The zeroth position is just

// dummy (1 to n to be used)

vector<node> nodes(n + 1);

// Add edges to each node as per given input

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

{

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

{

if(graph[i][j])

{

// Connect the undirected graph

nodes[i].edges.insert(i);

nodes[j].edges.insert(j);

}

}

}

// Display final answer

cout << canPaint(nodes, n, m);

cout << "\n";

return 0;

}

**Output**

1

**Complexity Analysis:**

* **Time Complexity:** O(V + E).
* **Space Complexity:** O(V). For Storing Visited List.

# 260. Print all palindromic partitions of a string

Given a String **S,**Find all possible Palindromic partitions of the given String.

**Example 1:**

**Input:**

**S =** "geeks"

**Output:**

g e e k s

g ee k s

**Explanation:**

All possible palindromic partitions

are printed.

**Example 2:**

**Input:**

**S =** "madam"

**Output:**

m a d a m

m ada m

madam

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **allPalindromicPerms()** which takes a String S as input parameter and returns a list of lists denoting all the possible palindromic partitions in the order of their appearance in the original string.

**Expected Time Complexity:** O(N\*2N)  
**Expected Auxiliary Space:** O(N2), where N is the length of the String

**Constraints:**  
1 <= |S| <= 20

## Solution:

Note that this problem is different from[Palindrome Partitioning Problem](https://www.geeksforgeeks.org/dynamic-programming-set-17-palindrome-partitioning/), there the task was to find the partitioning with minimum cuts in input string. Here we need to print all possible partitions.  
The idea is to go through every substring starting from first character, check if it is palindrome. If yes, then add the substring to solution and recur for remaining part. Below is complete algorithm.  
Below is the implementation of above idea.

// C++ program to print all palindromic partitions of a given string

#include<bits/stdc++.h>

using namespace std;

// A utility function to check if str is palindrome

bool isPalindrome(string str, int low, int high)

{

while (low < high)

{

if (str[low] != str[high])

return false;

low++;

high--;

}

return true;

}

// Recursive function to find all palindromic partitions of str[start..n-1]

// allPart --> A vector of vector of strings. Every vector inside it stores

// a partition

// currPart --> A vector of strings to store current partition

void allPalPartUtil(vector<vector<string> >&allPart, vector<string> &currPart,

int start, int n, string str)

{

// If 'start' has reached len

if (start >= n)

{

allPart.push\_back(currPart);

return;

}

// Pick all possible ending points for substrings

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

{

// If substring str[start..i] is palindrome

if (isPalindrome(str, start, i))

{

// Add the substring to result

currPart.push\_back(str.substr(start, i-start+1));

// Recur for remaining remaining substring

allPalPartUtil(allPart, currPart, i+1, n, str);

// Remove substring str[start..i] from current

// partition

currPart.pop\_back();

}

}

}

// Function to print all possible palindromic partitions of

// str. It mainly creates vectors and calls allPalPartUtil()

void allPalPartitions(string str)

{

int n = str.length();

// To Store all palindromic partitions

vector<vector<string> > allPart;

// To store current palindromic partition

vector<string> currPart;

// Call recursive function to generate all partitions

// and store in allPart

allPalPartUtil(allPart, currPart, 0, n, str);

// Print all partitions generated by above call

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

{

for (int j=0; j<allPart[i].size(); j++)

cout << allPart[i][j] << " ";

cout << "\n";

}

}

// Driver program

int main()

{

string str = "nitin";

allPalPartitions(str);

return 0;

}

**Output**

n i t i n

n iti n

nitin

**Output:**

n i t i n

n iti n

nitin

**Approach 2: Expand around every palindrome**

The idea is to split the string into all palindromes of length 1 that is convert the string to a list of its characters (but as string data type) and then expand the smaller palindromes to bigger palindromes by checking if its left and right (reversed) are equal or not if they are equal then merge them and solve for new list recursively. Also if two adjacent strings of this list are equal (when one of them is reversed), merging them would also give a palindrome so merge them and solve recursively.

# 261. Subset Sum Problem

Given an array **arr[]** of size **N**, check if it can be partitioned into two parts such that the sum of elements in both parts is the same.

**Example 1:**

**Input:** N = 4

arr = {1, 5, 11, 5}

**Output:** YES

**Explaination:**

The two parts are {1, 5, 5} and {11}.

**Example 2:**

**Input:** N = 3

arr = {1, 3, 5}

**Output:** NO

**Explaination:** This array can never be

partitioned into two such parts.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **equalPartition()** which takes the value N and the array as input parameters and returns 1 if the partition is possible. Otherwise, returns 0.

**Expected Time Complexity:** O(N\*sum of elements)  
**Expected Auxiliary Space:** O(N\*sum of elements)

**Constraints:**  
1 ≤ N ≤ 100  
1 ≤ arr[i] ≤ 1000

## Solution:

We have already discussed a solution in [Partition Problem](https://www.geeksforgeeks.org/dynamic-programming-set-18-partition-problem/) to find if array can be partitioned or not. In this post, we print two sets that are also printed. We post pass two vectors set1 and set2 and two sum variables sum1 and sum2. Traverse the array recursively. At every array position there are two choices: either add the current element to set 1 or to set 2. Recursively call for both the conditions and update the vectors set1 and set2 accordingly. If the current element is added to set 1 then add the current element to sum1 and insert it in vector set 1. Repeat the same if the current element is included in set 2. At the end of array traversal compare both the sums. If both the sums are equal then print both the vectors otherwise backtrack to check other possibilities.

**Implementation:**

// CPP program to print equal sum two subsets of

// an array if it can be partitioned into subsets.

#include <bits/stdc++.h>

using namespace std;

/// Function to print the equal sum sets of the array.

void printSets(vector<int> set1, vector<int> set2)

{

int i;

/// Print set 1.

for (i = 0; i < set1.size(); i++) {

cout << set1[i] << " ";

}

cout << "\n";

/// Print set 2.

for (i = 0; i < set2.size(); i++) {

cout << set2[i] << " ";

}

}

/// Utility function to find the sets of the array which

/// have equal sum.

bool findSets(int arr[], int n, vector<int>& set1,

vector<int>& set2, int sum1, int sum2,

int pos)

{

/// If entire array is traversed, compare both the sums.

if (pos == n) {

/// If sums are equal print both sets and return

/// true to show sets are found.

if (sum1 == sum2) {

printSets(set1, set2);

return true;

}

/// If sums are not equal then return sets are not

/// found.

else

return false;

}

/// Add current element to set 1.

set1.push\_back(arr[pos]);

/// Recursive call after adding current element to

/// set 1.

bool res = findSets(arr, n, set1, set2, sum1 + arr[pos],

sum2, pos + 1);

/// If this inclusion results in equal sum sets

/// partition then return true to show desired sets are

/// found.

if (res)

return res;

/// If not then backtrack by removing current element

/// from set1 and include it in set 2.

set1.pop\_back();

set2.push\_back(arr[pos]);

/// Recursive call after including current element to

/// set 2.

res = findSets(arr, n, set1, set2, sum1,

sum2 + arr[pos], pos + 1);

if (res == false)

if (!set2.empty())

set2.pop\_back();

return res;

}

/// Return true if array arr can be partitioned

/// into two equal sum sets or not.

bool isPartitionPoss(int arr[], int n)

{

/// Calculate sum of elements in array.

int sum = 0;

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

sum += arr[i];

/// If sum is odd then array cannot be

/// partitioned.

if (sum % 2 != 0)

return false;

/// Declare vectors to store both the sets.

vector<int> set1, set2;

/// Find both the sets.

return findSets(arr, n, set1, set2, 0, 0, 0);

}

// Driver code

int main()

{

int arr[] = { 5, 5, 1, 11 };

int n = sizeof(arr) / sizeof(arr[0]);

if (!isPartitionPoss(arr, n)) {

cout << "-1";

}

return 0;

}

**Output:**

5 5 1

11

**Time Complexity:** Exponential O(2^n)   
**Auxiliary Space:** O(n) (Without considering size of function call stack)

**Approach:**In the [previous](https://www.geeksforgeeks.org/print-equal-sum-sets-array-partition-problem/) post, a solution using [recursion](https://www.geeksforgeeks.org/recursion/) is discussed. In this post, a solution using [Dynamic Programming](https://www.geeksforgeeks.org/dynamic-programming/) is explained.   
The idea is to declare two sets set 1 and set 2. To recover the solution, traverse the boolean dp table backward starting from the final result dp[n][k], where n = number of elements and k = sum/2. Set 1 will consist of elements that contribute to sum k and other elements that do not contribute are added to set 2. Follow these steps at each position to recover the solution. 

1. Check if dp[i-1][sum] is true or not. If it is true, then the current element does not contribute to sum k. Add this element to set 2. Update index i by i-1 and sum remains unchanged.
2. If dp[i-1][sum] is false, then current element contribute to sum k. Add current element to set 1. Update index i by i-1 and sum by sum-arr[i-1].

Repeat the above steps until each index position is traversed.  
**Implementation:**

// CPP program to print equal sum sets of array.

#include <bits/stdc++.h>

using namespace std;

// Function to print equal sum

// sets of array.

void printEqualSumSets(int arr[], int n)

{

int i, currSum;

// Finding sum of array elements

int sum = accumulate(arr, arr+n, 0);

// Check sum is even or odd. If odd

// then array cannot be partitioned.

// Print -1 and return.

if (sum & 1) {

cout << "-1";

return;

}

// Divide sum by 2 to find

// sum of two possible subsets.

int k = sum >> 1;

// Boolean DP table to store result

// of states.

// dp[i][j] = true if there is a

// subset of elements in first i elements

// of array that has sum equal to j.

bool dp[n + 1][k + 1];

// If number of elements are zero, then

// no sum can be obtained.

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

dp[0][i] = false;

// Sum 0 can be obtained by not selecting

// any element.

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

dp[i][0] = true;

// Fill the DP table in bottom up manner.

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

for (currSum = 1; currSum <= k; currSum++) {

// Excluding current element.

dp[i][currSum] = dp[i - 1][currSum];

// Including current element

if (arr[i - 1] <= currSum)

dp[i][currSum] = dp[i][currSum] |

dp[i - 1][currSum - arr[i - 1]];

}

}

// Required sets set1 and set2.

vector<int> set1, set2;

// If partition is not possible print

// -1 and return.

if (!dp[n][k]) {

cout << "-1\n";

return;

}

// Start from last element in dp table.

i = n;

currSum = k;

while (i > 0 && currSum >= 0) {

// If current element does not

// contribute to k, then it belongs

// to set 2.

if (dp[i - 1][currSum]) {

i--;

set2.push\_back(arr[i]);

}

// If current element contribute

// to k then it belongs to set 1.

else if (dp[i - 1][currSum - arr[i - 1]]) {

i--;

currSum -= arr[i];

set1.push\_back(arr[i]);

}

}

// Print elements of both the sets.

cout << "Set 1 elements: ";

for (i = 0; i < set1.size(); i++)

cout << set1[i] << " ";

cout << "\nSet 2 elements: ";

for (i = 0; i < set2.size(); i++)

cout << set2[i] << " ";

}

// Driver program.

int main()

{

int arr[] = { 5, 5, 1, 11 };

int n = sizeof(arr) / sizeof(arr[0]);

printEqualSumSets(arr, n);

return 0;

}

**Output :**

Set 1 elements: 1 5 5

Set 2 elements: 11

**Time Complexity:**O(n\*k), where k = sum(arr) / 2   
**Auxiliary Space:**O(n\*k)

The solution discussed above requires O(n \* sum) space and O(n \* sum) time. We can optimize space. We create a boolean 2D array subset[2][sum+1]. Using bottom up manner we can fill up this table. The idea behind using **2 in “subset[2][sum+1]”** is that for filling a row only the values from previous row is required. So alternate rows are used either making the first one as current and second as previous or the first as previous and second as current.

**Another Approach:** To further reduce space complexity, we create a boolean 1D array subset[sum+1]. Using bottom up manner we can fill up this table. The idea is that we can check if the sum till position “i” is possible then if the current element in the array at position j is x, then sum i+x is also possible. We traverse the sum array from back to front so that we don’t count any element twice.

Here’s the code for the given approach:

**Implementation:**

class Solution {

public:

bool canPartition(vector<int>& nums) {

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

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

sum += nums[i];

if(sum%2!=0)

return false;

sum /= 2;

vector<bool> dp(sum, false);

dp[0] = true;

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

for(int j=sum;j>=nums[i];j--){

if(dp[j - nums[i]] == true)

dp[j] = true;

}

}

return dp[sum];

}

};

**Time Complexity:** O(N\*K) where N is the number of elements in the array and K is total sum.  
**Space Complexity:** O(K)

# 262. The Knight’s tour problem

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that “works”. Problems that are typically solved using the backtracking technique have the following property in common. These problems can only be solved by trying every possible configuration and each configuration is tried only once. A Naive solution for these problems is to try all configurations and output a configuration that follows given problem constraints. Backtracking works incrementally and is an optimization over the Naive solution where all possible configurations are generated and tried.  
For example, consider the following [Knight’s Tour](http://en.wikipedia.org/wiki/Knight%27s_tour) problem.

**Problem Statement:**  
Given a N\*N board with the Knight placed on the first block of an empty board. Moving according to the rules of chess knight must visit each square exactly once. Print the order of each cell in which they are visited.

**Example:**

Input :

N = 8

Output:

0 59 38 33 30 17 8 63

37 34 31 60 9 62 29 16

58 1 36 39 32 27 18 7

35 48 41 26 61 10 15 28

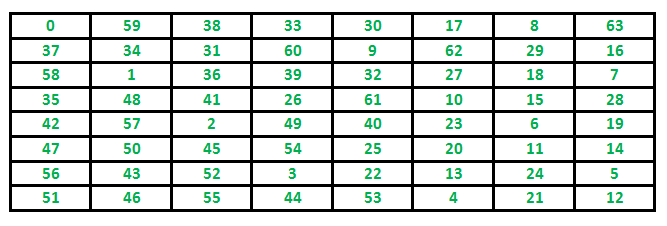
42 57 2 49 40 23 6 19

47 50 45 54 25 20 11 14

56 43 52 3 22 13 24 5

51 46 55 44 53 4 21 12

**The path followed by Knight to cover all the cells**  
Following is a chessboard with 8 x 8 cells. Numbers in cells indicate the move number of Knight.



## Solution:

Let us first discuss the Naive algorithm for this problem and then the Backtracking algorithm.

**Naive Algorithm for Knight’s tour**   
The Naive Algorithm is to generate all tours one by one and check if the generated tour satisfies the constraints.

while there are untried tours

{

generate the next tour

if this tour covers all squares

{

print this path;

}

}

**Backtracking**works in an incremental way to attack problems. Typically, we start from an empty solution vector and one by one add items (Meaning of item varies from problem to problem. In the context of Knight’s tour problem, an item is a Knight’s move). When we add an item, we check if adding the current item violates the problem constraint, if it does then we remove the item and try other alternatives. If none of the alternatives works out then we go to the previous stage and remove the item added in the previous stage. If we reach the initial stage back then we say that no solution exists. If adding an item doesn’t violate constraints then we recursively add items one by one. If the solution vector becomes complete then we print the solution.

**Backtracking Algorithm for Knight’s tour**

Following is the Backtracking algorithm for Knight’s tour problem.

If all squares are visited

print the solution

Else

a) Add one of the next moves to solution vector and recursively

check if this move leads to a solution. (A Knight can make maximum

eight moves. We choose one of the 8 moves in this step).

b) If the move chosen in the above step doesn't lead to a solution

then remove this move from the solution vector and try other

alternative moves.

c) If none of the alternatives work then return false (Returning false

will remove the previously added item in recursion and if false is

returned by the initial call of recursion then "no solution exists" )

Following are implementations for Knight’s tour problem. It prints one of the possible solutions in 2D matrix form. Basically, the output is a 2D 8\*8 matrix with numbers from 0 to 63 and these numbers show steps made by Knight.

// C++ program for Knight Tour problem

#include <bits/stdc++.h>

using namespace std;

#define N 8

int solveKTUtil(int x, int y, int movei, int sol[N][N],

int xMove[], int yMove[]);

/\* A utility function to check if i,j are

valid indexes for N\*N chessboard \*/

int isSafe(int x, int y, int sol[N][N])

{

return (x >= 0 && x < N && y >= 0 && y < N

&& sol[x][y] == -1);

}

/\* A utility function to print

solution matrix sol[N][N] \*/

void printSolution(int sol[N][N])

{

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

for (int y = 0; y < N; y++)

cout << " " << setw(2) << sol[x][y] << " ";

cout << endl;

}

}

/\* This function solves the Knight Tour problem using

Backtracking. This function mainly uses solveKTUtil()

to solve the problem. It returns false if no complete

tour is possible, otherwise return true and prints the

tour.

Please note that there may be more than one solutions,

this function prints one of the feasible solutions. \*/

int solveKT()

{

int sol[N][N];

/\* Initialization of solution matrix \*/

for (int x = 0; x < N; x++)

for (int y = 0; y < N; y++)

sol[x][y] = -1;

/\* xMove[] and yMove[] define next move of Knight.

xMove[] is for next value of x coordinate

yMove[] is for next value of y coordinate \*/

int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };

int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };

// Since the Knight is initially at the first block

sol[0][0] = 0;

/\* Start from 0,0 and explore all tours using

solveKTUtil() \*/

if (solveKTUtil(0, 0, 1, sol, xMove, yMove) == 0) {

cout << "Solution does not exist";

return 0;

}

else

printSolution(sol);

return 1;

}

/\* A recursive utility function to solve Knight Tour

problem \*/

int solveKTUtil(int x, int y, int movei, int sol[N][N],

int xMove[8], int yMove[8])

{

int k, next\_x, next\_y;

if (movei == N \* N)

return 1;

/\* Try all next moves from

the current coordinate x, y \*/

for (k = 0; k < 8; k++) {

next\_x = x + xMove[k];

next\_y = y + yMove[k];

if (isSafe(next\_x, next\_y, sol)) {

sol[next\_x][next\_y] = movei;

if (solveKTUtil(next\_x, next\_y, movei + 1, sol,

xMove, yMove)

== 1)

return 1;

else

// backtracking

sol[next\_x][next\_y] = -1;

}

}

return 0;

}

// Driver Code

int main()

{

// Function Call

solveKT();

return 0;

}

**Output**

0 59 38 33 30 17 8 63

37 34 31 60 9 62 29 16

58 1 36 39 32 27 18 7

35 48 41 26 61 10 15 28

42 57 2 49 40 23 6 19

47 50 45 54 25 20 11 14

56 43 52 3 22 13 24 5

51 46 55 44 53 4 21 12

**Time Complexity :**  
There are N2 Cells and for each, we have a maximum of 8 possible moves to choose from, so the worst running time is O(8N^2).

**Auxiliary Space:**O(N2)

**Important Note:**  
No order of the xMove, yMove is wrong, but they will affect the running time of the algorithm drastically. For example, think of the case where the 8th choice of the move is the correct one, and before that our code ran 7 different wrong paths. It’s always a good idea a have a heuristic than to try backtracking randomly. Like, in this case, we know the next step would probably be in the south or east direction, then checking the paths which lead their first is a better strategy.

Note that Backtracking is not the best solution for the Knight’s tour problem.

# 263. Tug of War

Given a set of n integers, divide the set in two subsets of n/2 sizes each such that the difference of the sum of two subsets is as minimum as possible. If n is even, then sizes of two subsets must be strictly n/2 and if n is odd, then size of one subset must be (n-1)/2 and size of other subset must be (n+1)/2.  
For example, let given set be {3, 4, 5, -3, 100, 1, 89, 54, 23, 20}, the size of set is 10. Output for this set should be {4, 100, 1, 23, 20} and {3, 5, -3, 89, 54}. Both output subsets are of size 5 and sum of elements in both subsets is same (148 and 148).   
Let us consider another example where n is odd. Let given set be {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4}. The output subsets should be {45, -34, 12, 98, -1} and {23, 0, -99, 4, 189, 4}. The sums of elements in two subsets are 120 and 121 respectively.  
The following solution tries every possible subset of half size. If one subset of half size is formed, the remaining elements form the other subset. We initialize current set as empty and one by one build it. There are two possibilities for every element, either it is part of current set, or it is part of the remaining elements (other subset). We consider both possibilities for every element. When the size of current set becomes n/2, we check whether this solutions is better than the best solution available so far. If it is, then we update the best solution.

## Solution:

Following is the implementation for Tug of War problem. It prints the required arrays.   
 #include <bits/stdc++.h>

using namespace std;

// function that tries every possible solution by calling itself recursively

void TOWUtil(int\* arr, int n, bool\* curr\_elements, int no\_of\_selected\_elements,

bool\* soln, int\* min\_diff, int sum, int curr\_sum, int curr\_position)

{

// checks whether the it is going out of bound

if (curr\_position == n)

return;

// checks that the numbers of elements left are not less than the

// number of elements required to form the solution

if ((n/2 - no\_of\_selected\_elements) > (n - curr\_position))

return;

// consider the cases when current element is not included in the solution

TOWUtil(arr, n, curr\_elements, no\_of\_selected\_elements,

soln, min\_diff, sum, curr\_sum, curr\_position+1);

// add the current element to the solution

no\_of\_selected\_elements++;

curr\_sum = curr\_sum + arr[curr\_position];

curr\_elements[curr\_position] = true;

// checks if a solution is formed

if (no\_of\_selected\_elements == n/2)

{

// checks if the solution formed is better than the best solution so far

if (abs(sum/2 - curr\_sum) < \*min\_diff)

{

\*min\_diff = abs(sum/2 - curr\_sum);

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

soln[i] = curr\_elements[i];

}

}

else

{

// consider the cases where current element is included in the solution

TOWUtil(arr, n, curr\_elements, no\_of\_selected\_elements, soln,

min\_diff, sum, curr\_sum, curr\_position+1);

}

// removes current element before returning to the caller of this function

curr\_elements[curr\_position] = false;

}

// main function that generate an arr

void tugOfWar(int \*arr, int n)

{

// the boolean array that contains the inclusion and exclusion of an element

// in current set. The number excluded automatically form the other set

bool\* curr\_elements = new bool[n];

// The inclusion/exclusion array for final solution

bool\* soln = new bool[n];

int min\_diff = INT\_MAX;

int sum = 0;

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

{

sum += arr[i];

curr\_elements[i] = soln[i] = false;

}

// Find the solution using recursive function TOWUtil()

TOWUtil(arr, n, curr\_elements, 0, soln, &min\_diff, sum, 0, 0);

// Print the solution

cout << "The first subset is: ";

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

{

if (soln[i] == true)

cout << arr[i] << " ";

}

cout << "\nThe second subset is: ";

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

{

if (soln[i] == false)

cout << arr[i] << " ";

}

}

// Driver program to test above functions

int main()

{

int arr[] = {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4};

int n = sizeof(arr)/sizeof(arr[0]);

tugOfWar(arr, n);

return 0;

}

**Output:**

The first subset is: 45 -34 12 98 -1

The second subset is: 23 0 -99 4 189 4

**Time Complexity:** O(2^n)

# 264. Find shortest safe route in a path with landmines

Given a path in the form of a rectangular matrix having few landmines arbitrarily placed (marked as 0), calculate length of the shortest safe route possible from any cell in the first column to any cell in the last column of the matrix. We have to avoid landmines and their four adjacent cells (left, right, above and below) as they are also unsafe. We are allowed to move to only adjacent cells which are not landmines. i.e. the route cannot contains any diagonal moves.

Examples:

**Input:**

A 12 x 10 matrix with landmines marked as 0

[ 1 1 1 1 1 1 1 1 1 1 ]

[ 1 0 1 1 1 1 1 1 1 1 ]

[ 1 1 1 0 1 1 1 1 1 1 ]

[ 1 1 1 1 0 1 1 1 1 1 ]

[ **1 1 1 1** 1 1 1 1 1 1 ]

[ 1 1 1 **1** 1 0 1 1 1 1 ]

[ 1 0 1 **1** 1 1 1 1 0 1 ]

[ 1 1 1 **1 1 1 1 1** 1 1 ]

[ 1 1 1 1 1 1 1 **1 1 1** ]

[ 0 1 1 1 1 0 1 1 1 1 ]

[ 1 1 1 1 1 1 1 1 1 1 ]

[ 1 1 1 0 1 1 1 1 1 1 ]

**Output:**

Length of shortest safe route is 13 (Highlighted in **Bold**)

## Solution:

The idea is to use Backtracking. We first mark all adjacent cells of the landmines as unsafe. Then for each safe cell of first column of the matrix, we move forward in all allowed directions and recursively checks if they leads to the destination or not. If destination is found, we update the value of shortest path else if none of the above solutions work we return false from our function.

Below is the implementation of above idea –

// C++ program to find shortest safe Route in

// the matrix with landmines

#include <bits/stdc++.h>

using namespace std;

#define R 12

#define C 10

// These arrays are used to get row and column

// numbers of 4 neighbours of a given cell

int rowNum[] = { -1, 0, 0, 1 };

int colNum[] = { 0, -1, 1, 0 };

// A function to check if a given cell (x, y)

// can be visited or not

bool isSafe(int mat[R][C], int visited[R][C],

int x, int y)

{

if (mat[x][y] == 0 || visited[x][y])

return false;

return true;

}

// A function to check if a given cell (x, y) is

// a valid cell or not

bool isValid(int x, int y)

{

if (x < R && y < C && x >= 0 && y >= 0)

return true;

return false;

}

// A function to mark all adjacent cells of

// landmines as unsafe. Landmines are shown with

// number 0

void markUnsafeCells(int mat[R][C])

{

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

{

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

{

// if a landmines is found

if (mat[i][j] == 0)

{

// mark all adjacent cells

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

if (isValid(i + rowNum[k], j + colNum[k]))

mat[i + rowNum[k]][j + colNum[k]] = -1;

}

}

}

// mark all found adjacent cells as unsafe

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

{

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

{

if (mat[i][j] == -1)

mat[i][j] = 0;

}

}

// Uncomment below lines to print the path

/\*for (int i = 0; i < R; i++)

{

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

{

cout << std::setw(3) << mat[i][j];

}

cout << endl;

}\*/

}

// Function to find shortest safe Route in the

// matrix with landmines

// mat[][] - binary input matrix with safe cells marked as 1

// visited[][] - store info about cells already visited in

// current route

// (i, j) are coordinates of the current cell

// min\_dist --> stores minimum cost of shortest path so far

// dist --> stores current path cost

void findShortestPathUtil(int mat[R][C], int visited[R][C],

int i, int j, int &min\_dist, int dist)

{

// if destination is reached

if (j == C-1)

{

// update shortest path found so far

min\_dist = min(dist, min\_dist);

return;

}

// if current path cost exceeds minimum so far

if (dist > min\_dist)

return;

// include (i, j) in current path

visited[i][j] = 1;

// Recurse for all safe adjacent neighbours

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

{

if (isValid(i + rowNum[k], j + colNum[k]) &&

isSafe(mat, visited, i + rowNum[k], j + colNum[k]))

{

findShortestPathUtil(mat, visited, i + rowNum[k],

j + colNum[k], min\_dist, dist + 1);

}

}

// Backtrack

visited[i][j] = 0;

}

// A wrapper function over findshortestPathUtil()

void findShortestPath(int mat[R][C])

{

// stores minimum cost of shortest path so far

int min\_dist = INT\_MAX;

// create a boolean matrix to store info about

// cells already visited in current route

int visited[R][C];

// mark adjacent cells of landmines as unsafe

markUnsafeCells(mat);

// start from first column and take minimum

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

{

// if path is safe from current cell

if (mat[i][0] == 1)

{

// initialize visited to false

memset(visited, 0, sizeof visited);

// find shortest route from (i, 0) to any

// cell of last column (x, C - 1) where

// 0 <= x < R

findShortestPathUtil(mat, visited, i, 0,

min\_dist, 0);

// if min distance is already found

if(min\_dist == C - 1)

break;

}

}

// if destination can be reached

if (min\_dist != INT\_MAX)

cout << "Length of shortest safe route is "

<< min\_dist;

else // if the destination is not reachable

cout << "Destination not reachable from "

<< "given source";

}

// Driver code

int main()

{

// input matrix with landmines shown with number 0

int mat[R][C] =

{

{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 1, 0, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 1, 1, 1, 0, 1, 1, 1, 1, 1, 1 },

{ 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 },

{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 1, 1, 1, 1, 1, 0, 1, 1, 1, 1 },

{ 1, 0, 1, 1, 1, 1, 1, 1, 0, 1 },

{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 0, 1, 1, 1, 1, 0, 1, 1, 1, 1 },

{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 1, 1, 1, 0, 1, 1, 1, 1, 1, 1 }

};

// find shortest path

findShortestPath(mat);

return 0;

}

**Output:**

Length of shortest safe route is 13

**Another method:** It can be solved in polynomial time with the help of Breadth First Search. Enqueue the cells with 1 value in the queue with the distance as 0. As the BFS proceeds, shortest path to each cell from the first column is computed. Finally for the reachable cells in the last column, output the minimum distance.

The implementation in C++ is as follows:

// C++ program to find shortest safe Route in

// the matrix with landmines

#include <bits/stdc++.h>

using namespace std;

#define R 12

#define C 10

struct Key{

int x,y;

Key(int i,int j){ x=i;y=j;};

};

// These arrays are used to get row and column

// numbers of 4 neighbours of a given cell

int rowNum[] = { -1, 0, 0, 1 };

int colNum[] = { 0, -1, 1, 0 };

// A function to check if a given cell (x, y) is

// a valid cell or not

bool isValid(int x, int y)

{

if (x < R && y < C && x >= 0 && y >= 0)

return true;

return false;

}

// A function to mark all adjacent cells of

// landmines as unsafe. Landmines are shown with

// number 0

void findShortestPath(int mat[R][C]){

int i,j;

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

{

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

{

// if a landmines is found

if (mat[i][j] == 0)

{

// mark all adjacent cells

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

if (isValid(i + rowNum[k], j + colNum[k]))

mat[i + rowNum[k]][j + colNum[k]] = -1;

}

}

}

// mark all found adjacent cells as unsafe

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

{

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

{

if (mat[i][j] == -1)

mat[i][j] = 0;

}

}

int dist[R][C];

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

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

dist[i][j] = -1;

}

queue<Key> q;

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

if(mat[i][0] == 1){

q.push(Key(i,0));

dist[i][0] = 0;

}

}

while(!q.empty()){

Key k = q.front();

q.pop();

int d = dist[k.x][k.y];

int x = k.x;

int y = k.y;

for (int k = 0; k < 4; k++) {

int xp = x + rowNum[k];

int yp = y + colNum[k];

if(isValid(xp,yp) && dist[xp][yp] == -1 && mat[xp][yp] == 1){

dist[xp][yp] = d+1;

q.push(Key(xp,yp));

}

}

}

// stores minimum cost of shortest path so far

int ans = INT\_MAX;

// start from first column and take minimum

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

if(mat[i][C-1] == 1 && dist[i][C-1] != -1){

ans = min(ans,dist[i][C-1]);

}

}

// if destination can be reached

if(ans == INT\_MAX)

cout << "NOT POSSIBLE\n";

else// if the destination is not reachable

cout << "Length of shortest safe route is " << ans << endl;

}

// Driver code

int main(){

// input matrix with landmines shown with number 0

int mat[R][C] =

{

{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 1, 0, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 1, 1, 1, 0, 1, 1, 1, 1, 1, 1 },

{ 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 },

{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 1, 1, 1, 1, 1, 0, 1, 1, 1, 1 },

{ 1, 0, 1, 1, 1, 1, 1, 1, 0, 1 },

{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 0, 1, 1, 1, 1, 0, 1, 1, 1, 1 },

{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 1, 1, 1, 0, 1, 1, 1, 1, 1, 1 }

};

// find shortest path

findShortestPath(mat);

}

**Output:**

Length of shortest safe route is 13

# 265. Combinational Sum

Given an array of integers and a sum B, find all unique combinations in the array where the sum is equal to B. The same number may be chosen from the array any number of times to make B.

**Note:**  
        **1.** All numbers will be positive integers.  
       **2.** Elements in a combination (a1, a2, …, ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).  
        **3.** The combinations themselves must be sorted in ascending order.

**Example 1:**

**Input:**

N = 4

arr[] = {7,2,6,5}

B = 16

**Output:**

(2 2 2 2 2 2 2 2)

(2 2 2 2 2 6)

(2 2 2 5 5)

(2 2 5 7)

(2 2 6 6)

(2 7 7)

(5 5 6)

**Example 2:**

**Input:**

N = 11

arr[] = {6,5,7,1,8,2,9,9,7,7,9}

B = 6

**Output:**

(1 1 1 1 1 1)

(1 1 1 1 2)

(1 1 2 2)

(1 5)

(2 2 2)

(6)

**Your Task:**  
Your task is to complete the function **combinationSum()**which takes the array A and a sum B as inputs and returns a list of list denoting the required combinations in the order specified in the problem description. The printing is done by the driver's code. If no set can be formed with the given set, then  "Empty" (without quotes) is printed.

**Expected Time Complexity:** O(X2 \* 2N), where X is average of summation B/arrifor every number in the array.  
**Expected Auxiliary Space:** O(X \* 2N)

**Constraints:**  
1 <= N <= 30  
1 <= A[i] <= 20  
1 <= B <= 100

## Solution:

Since the problem is to get all the possible results, not the best or the number of result, thus we don't need to consider DP(dynamic programming), recursion is needed to handle it.

We should use the following algorithm.

1. Sort the array(non-decreasing).

2. First remove all the duplicates from array.

3. Then use recursion and backtracking to solve

the problem.

(A) If at any time sub-problem sum == 0 then

add that array to the result (vector of

vectors).

(B) Else if sum is negative then ignore that

sub-problem.

(C) Else insert the present index in that

array to the current vector and call

the function with sum = sum-ar[index] and

index = index, then pop that element from

current index (backtrack) and call the

function with sum = sum and index = index+1

Below is the C++ implementation of the above steps.

// C++ program to find all combinations that

// sum to a given value

#include <bits/stdc++.h>

using namespace std;

// Print all members of ar[] that have given

void findNumbers(vector<int>& ar, int sum,

vector<vector<int> >& res, vector<int>& r,

int i)

{

// if we get exact answer

if (sum == 0) {

res.push\_back(r);

return;

}

// Recur for all remaining elements that

// have value smaller than sum.

while (i < ar.size() && sum - ar[i] >= 0) {

// Till every element in the array starting

// from i which can contribute to the sum

r.push\_back(ar[i]); // add them to list

// recur for next numbers

findNumbers(ar, sum - ar[i], res, r, i);

i++;

// Remove number from list (backtracking)

r.pop\_back();

}

}

// Returns all combinations

// of ar[] that have given

// sum.

vector<vector<int> > combinationSum(vector<int>& ar,

int sum)

{

// sort input array

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

// remove duplicates

ar.erase(unique(ar.begin(), ar.end()), ar.end());

vector<int> r;

vector<vector<int> > res;

findNumbers(ar, sum, res, r, 0);

return res;

}

// Driver code

int main()

{

vector<int> ar;

ar.push\_back(2);

ar.push\_back(4);

ar.push\_back(6);

ar.push\_back(8);

int n = ar.size();

int sum = 8; // set value of sum

vector<vector<int> > res = combinationSum(ar, sum);

// If result is empty, then

if (res.size() == 0) {

cout << "Emptyn";

return 0;

}

// Print all combinations stored in res.

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

if (res[i].size() > 0) {

cout << " ( ";

for (int j = 0; j < res[i].size(); j++)

cout << res[i][j] << " ";

cout << ")";

}

}

}

**Output**

( 2 2 2 2 ) ( 2 2 4 ) ( 2 6 ) ( 4 4 ) ( 8 )

**My Implementation:**

class Solution {

public:

void fun(vector<int> &A, int B, vector<int> &temp, vector<vector<int>> &res, int sum, int ind, int n, set<vector<int>> &mp){

if(sum==B){

if(mp.find(temp)==mp.end()){

res.push\_back(temp);

mp.insert(temp);

}

return;

}

else if(sum>B||ind>=n)

return;

else{

temp.push\_back(A[ind]);

fun(A, B, temp, res, sum+A[ind], ind, n, mp);

temp.pop\_back();

fun(A, B, temp, res, sum, ind+1, n, mp);

}

}

//Function to return a list of indexes denoting the required

//combinations whose sum is equal to given number.

vector<vector<int> > combinationSum(vector<int> &A, int B) {

set<vector<int>> mp;

vector<vector<int>> res;

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

auto e = unique(A.begin(), A.end());

A.resize((e-A.begin()));

vector<int> temp;

fun(A, B, temp, res, 0, 0, A.size(), mp);

return res;

}

};

# 266. Find Maximum number possible by doing at-most K swaps

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

**Example 1:**

**Input:**

K = 4

str = "1234567"

**Output:**

7654321

**Explanation:**

Three swaps can make the

input 1234567 to 7654321, swapping 1

with 7, 2 with 6 and finally 3 with 5

**Example 2:**

**Input:**

K = 3

str = "3435335"

**Output:**

5543333

**Explanation:**

Three swaps can make the input

3435335 to 5543333, swapping 3

with 5, 4 with 5 and finally 3 with 4

**Your task:**  
You don't have to read input or print anything. Your task is to complete the function **findMaximumNum()**which takes the string and an integer as input and returns a string containing the largest number formed by perfoming the swap operation at most k times.

**Expected Time Complexity:** O(n!/(n-k)!) , where n = length of input string  
**Expected Auxiliary Space:** O(n)

**Constraints**:  
1 ≤ |str|≤ 30  
1 ≤ K≤ 10

## Solution:

**Naive Solution:**  
**Approach:** The idea is to consider every digit and swap it with digits following it one at a time and see if it leads to the maximum number. The process is repeated K times. The code can be further optimized, if the current digit is swapped with a digit less than the following digit.  
**Algorithm:**

1. Create a global variable which will store the maximum string or number.
2. Define a recursive function that takes the string as number and value of k
3. Run a nested loop, the outer loop from 0 to length of string -1 and inner loop from i+1 to end of the string.
4. Swap the ith and jth character and check if the string is now maximum and update the maximum string.
5. Call the function recursively with parameters: string and k-1.
6. Now again swap back the ith and jth character.

// C++ program to find maximum

// integer possible by doing

// at-most K swap operations

// on its digits.

#include <bits/stdc++.h>

using namespace std;

// Function to find maximum

// integer possible by

// doing at-most K swap

// operations on its digits

void findMaximumNum(

string str, int k, string& max)

{

// Return if no swaps left

if (k == 0)

return;

int n = str.length();

// Consider every digit

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

// Compare it with all digits after it

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

// if digit at position i

// is less than digit

// at position j, swap it

// and check for maximum

// number so far and recurse

// for remaining swaps

if (str[i] < str[j]) {

// swap str[i] with str[j]

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

// If current num is more

// than maximum so far

if (str.compare(max) > 0)

max = str;

// recurse of the other k - 1 swaps

findMaximumNum(str, k - 1, max);

// Backtrack

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

}

}

}

}

// Driver code

int main()

{

string str = "129814999";

int k = 4;

string max = str;

findMaximumNum(str, k, max);

cout << max << endl;

return 0;

}

**Output:**

999984211

**Complexity Analysis:**

* **Time Complexity:** O((n^2)^k).   
  For every recursive call n^2 recursive calls is generated until the value of k is 0. So total recursive calls are O((n^2)^k).
* **Space Complexity:**O(n).   
  This is the space required to store the output string.

**Efficient Solution:**  
**Approach:** The above approach traverses the whole string at each recursive call which is highly inefficient and unnecessary. Also, pre-computing the maximum digit after the current at a recursive call avoids unnecessary exchanges with each digit. It can be observed that to make the maximum string, the maximum digit is shifted to the front. So, instead of trying all pairs, try only those pairs where one of the elements is the maximum digit which is not yet swapped to the front.   
*There is an improvement by 27580 microseconds for each test case*.  
**Algorithm:**

1. Create a global variable which will store the maximum string or number.
2. Define a recursive function that takes the string as a number, the value of k, and the current index.
3. Find the index of the maximum element in the range current index to end.
4. if the index of the maximum element is not equal to the current index then decrement the value of k.
5. Run a loop from the current index to the end of the array
6. If the ith digit is equal to the maximum element
7. Swap the ith and element at the current index and check if the string is now maximum and update the maximum string.
8. Call the function recursively with parameters: string and k.
9. Now again swap back the ith and element at the current index.

// C++ program to find maximum

// integer possible by doing

// at-most K swap operations on

// its digits.

#include <bits/stdc++.h>

using namespace std;

// Function to find maximum

// integer possible by

// doing at-most K swap operations

// on its digits

void findMaximumNum(

string str, int k,

string& max, int ctr)

{

// return if no swaps left

if (k == 0)

return;

int n = str.length();

// Consider every digit after

// the cur position

char maxm = str[ctr];

for (int j = ctr + 1; j < n; j++) {

// Find maximum digit greater

// than at ctr among rest

if (maxm < str[j])

maxm = str[j];

}

// If maxm is not equal to str[ctr],

// decrement k

if (maxm != str[ctr])

--k;

// search this maximum among the rest from behind

//first swap the last maximum digit if it occurs more then 1 time

//example str= 1293498 and k=1 then max string is 9293418 instead of 9213498

for (int j = n-1; j >=ctr; j--) {

// If digit equals maxm swap

// the digit with current

// digit and recurse for the rest

if (str[j] == maxm) {

// swap str[ctr] with str[j]

swap(str[ctr], str[j]);

// If current num is more than

// maximum so far

if (str.compare(max) > 0)

max = str;

// recurse other swaps after cur

findMaximumNum(str, k, max, ctr + 1);

// Backtrack

swap(str[ctr], str[j]);

}

}

}

// Driver code

int main()

{

string str = "129814999";

int k = 4;

string max = str;

findMaximumNum(str, k, max, 0);

cout << max << endl;

return 0;

}

**Output:**

999984211

**Complexity Analysis:**

* **Time Complexity:** O(n^k).   
  For every recursive call n recursive calls is generated until the value of k is 0. So total recursive calls are O((n)^k).
* **Space Complexity:** O(n).   
  The space required to store the output string.

**My Implementation:**

class Solution

{

public:

string mx;

void fun(string str, int k, int ind, int n){

if(k<=0 || ind==n){

if(str.compare(mx)>0){

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

mx[i] = str[i];

}

return;

}

fun(str, k, ind+1, n);

for(int j=ind+1; j<n; j++){

if(str[ind]<str[j]){

swap(str[ind], str[j]);

fun(str, k-1, ind+1, n);

swap(str[ind], str[j]);

}

}

}

//Function to find the largest number after k swaps.

string findMaximumNum(string str, int k)

{

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

mx += str[i];

fun(str, k, 0, str.size());

//cout<<mx;

return mx;

}

};

# 267. Print all permutations of a string

## Same as ques 57 of string

# 268. Find if there is a path of more than k length from a source

Given a graph, a source vertex in the graph and a number k, find if there is a simple path (without any cycle) starting from given source and ending at any other vertex such that the distance from source to that vertex is atleast ‘k’ length.

## Solution:

One important thing to note is, simply doing BFS or DFS and picking the longest edge at every step would not work. The reason is, a shorter edge can produce longer path due to higher weight edges connected through it.  
The idea is to use Backtracking. We start from given source, explore all paths from current vertex. We keep track of current distance from source. If distance becomes more than k, we return true. If a path doesn’t produces more than k distance, we backtrack.  
How do we make sure that the path is simple and we don’t loop in a cycle? The idea is to keep track of current path vertices in an array. Whenever we add a vertex to path, we check if it already exists or not in current path. If it exists, we ignore the edge.  
Below is implementation of above idea.

// Program to find if there is a simple path with

// weight more than k

#include<bits/stdc++.h>

using namespace std;

// iPair ==> Integer Pair

typedef pair<int, int> iPair;

// This class represents a dipathted graph using

// adjacency list representation

class Graph

{

int V; // No. of vertices

// In a weighted graph, we need to store vertex

// and weight pair for every edge

list< pair<int, int> > \*adj;

bool pathMoreThanKUtil(int src, int k, vector<bool> &path);

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int u, int v, int w);

bool pathMoreThanK(int src, int k);

};

// Returns true if graph has path more than k length

bool Graph::pathMoreThanK(int src, int k)

{

// Create a path array with nothing included

// in path

vector<bool> path(V, false);

// Add source vertex to path

path[src] = 1;

return pathMoreThanKUtil(src, k, path);

}

// Prints shortest paths from src to all other vertices

bool Graph::pathMoreThanKUtil(int src, int k, vector<bool> &path)

{

// If k is 0 or negative, return true;

if (k <= 0)

return true;

// Get all adjacent vertices of source vertex src and

// recursively explore all paths from src.

list<iPair>::iterator i;

for (i = adj[src].begin(); i != adj[src].end(); ++i)

{

// Get adjacent vertex and weight of edge

int v = (\*i).first;

int w = (\*i).second;

// If vertex v is already there in path, then

// there is a cycle (we ignore this edge)

if (path[v] == true)

continue;

// If weight of is more than k, return true

if (w >= k)

return true;

// Else add this vertex to path

path[v] = true;

// If this adjacent can provide a path longer

// than k, return true.

if (pathMoreThanKUtil(v, k-w, path))

return true;

// Backtrack

path[v] = false;

}

// If no adjacent could produce longer path, return

// false

return false;

}

// Allocates memory for adjacency list

Graph::Graph(int V)

{

this->V = V;

adj = new list<iPair> [V];

}

// Utility function to an edge (u, v) of weight w

void Graph::addEdge(int u, int v, int w)

{

adj[u].push\_back(make\_pair(v, w));

adj[v].push\_back(make\_pair(u, w));

}

// Driver program to test methods of graph class

int main()

{

// create the graph given in above fugure

int V = 9;

Graph g(V);

// making above shown graph

g.addEdge(0, 1, 4);

g.addEdge(0, 7, 8);

g.addEdge(1, 2, 8);

g.addEdge(1, 7, 11);

g.addEdge(2, 3, 7);

g.addEdge(2, 8, 2);

g.addEdge(2, 5, 4);

g.addEdge(3, 4, 9);

g.addEdge(3, 5, 14);

g.addEdge(4, 5, 10);

g.addEdge(5, 6, 2);

g.addEdge(6, 7, 1);

g.addEdge(6, 8, 6);

g.addEdge(7, 8, 7);

int src = 0;

int k = 62;

g.pathMoreThanK(src, k)? cout << "Yes\n" :

cout << "No\n";

k = 60;

g.pathMoreThanK(src, k)? cout << "Yes\n" :

cout << "No\n";

return 0;

}

**Output:** 

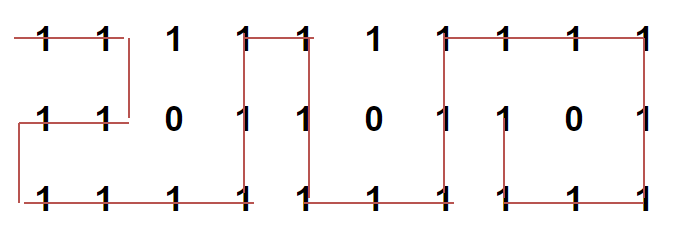
No

Yes

**Time Complexity:** O(n!)   
**Explanation:**   
From the source node, we one-by-one visit all the paths and check if the total weight is greater than k for each path. So, the worst case will be when the number of possible paths is maximum. This is the case when every node is connected to every other node.   
Beginning from the source node we have n-1 adjacent nodes. The time needed for a path to connect any two nodes is 2. One for joining the source and the next adjacent vertex. One for breaking the connection between the source and the old adjacent vertex.   
After selecting a node out of n-1 adjacent nodes, we are left with n-2 adjacent nodes (as the source node is already included in the path) and so on at every step of selecting a node our problem reduces by 1 node.  
We can write this in the form of a recurrence relation as: F(n) = n\*(2+F(n-1))   
This expands to: 2n + 2n\*(n-1) + 2n\*(n-1)\*(n-2) + ……. + 2n(n-1)(n-2)(n-3)…..1   
As n times 2n(n-1)(n-2)(n-3)….1 is greater than the given expression so we can safely say time complexity is: n\*2\*n!   
Here in the question the first node is defined so time complexity becomes   
F(n-1) = 2(n-1)\*(n-1)! = 2\*n\*(n-1)! – 2\*1\*(n-1)! = 2\*n!-2\*(n-1)! = O(n!)

# 269. Longest Possible Route in a Matrix with Hurdles

Given an M x N matrix, with a few hurdles arbitrarily placed, calculate the length of the longest possible route possible from source to a destination within the matrix. We are allowed to move to only adjacent cells which are not hurdles. The route cannot contain any diagonal moves and a location once visited in a particular path cannot be visited again.  
For example, the longest path with no hurdles from source to destination is highlighted below. The length of the path is 24.



## Solution:

The idea is to use Backtracking. We start from the source cell of the matrix, move forward in all four allowed directions, and recursively checks if they lead to the solution or not. If the destination is found, we update the value of the longest path else if none of the above solutions work we return false from our function.  
Below is the C++ implementation of the above idea –

// C++ program to find Longest Possible Route in a

// matrix with hurdles

#include <bits/stdc++.h>

using namespace std;

#define R 3

#define C 10

// A Pair to store status of a cell. found is set to

// true of destination is reachable and value stores

// distance of longest path

struct Pair {

// true if destination is found

bool found;

// stores cost of longest path from current cell to

// destination cell

int value;

};

// Function to find Longest Possible Route in the

// matrix with hurdles. If the destination is not reachable

// the function returns false with cost INT\_MAX.

// (i, j) is source cell and (x, y) is destination cell.

Pair findLongestPathUtil(int mat[R][C], int i, int j, int x,

int y, bool visited[R][C])

{

// if (i, j) itself is destination, return true

if (i == x && j == y) {

Pair p = { true, 0 };

return p;

}

// if not a valid cell, return false

if (i < 0 || i >= R || j < 0 || j >= C || mat[i][j] == 0

|| visited[i][j]) {

Pair p = { false, INT\_MAX };

return p;

}

// include (i, j) in current path i.e.

// set visited(i, j) to true

visited[i][j] = true;

// res stores longest path from current cell (i, j) to

// destination cell (x, y)

int res = INT\_MIN;

// go left from current cell

Pair sol

= findLongestPathUtil(mat, i, j - 1, x, y, visited);

// if destination can be reached on going left from

// current cell, update res

if (sol.found)

res = max(res, sol.value);

// go right from current cell

sol = findLongestPathUtil(mat, i, j + 1, x, y, visited);

// if destination can be reached on going right from

// current cell, update res

if (sol.found)

res = max(res, sol.value);

// go up from current cell

sol = findLongestPathUtil(mat, i - 1, j, x, y, visited);

// if destination can be reached on going up from

// current cell, update res

if (sol.found)

res = max(res, sol.value);

// go down from current cell

sol = findLongestPathUtil(mat, i + 1, j, x, y, visited);

// if destination can be reached on going down from

// current cell, update res

if (sol.found)

res = max(res, sol.value);

// Backtrack

visited[i][j] = false;

// if destination can be reached from current cell,

// return true

if (res != INT\_MIN) {

Pair p = { true, 1 + res };

return p;

}

// if destination can't be reached from current cell,

// return false

else {

Pair p = { false, INT\_MAX };

return p;

}

}

// A wrapper function over findLongestPathUtil()

void findLongestPath(int mat[R][C], int i, int j, int x,

int y)

{

// create a boolean matrix to store info about

// cells already visited in current route

bool visited[R][C];

// initialize visited to false

memset(visited, false, sizeof visited);

// find longest route from (i, j) to (x, y) and

// print its maximum cost

Pair p = findLongestPathUtil(mat, i, j, x, y, visited);

if (p.found)

cout << "Length of longest possible route is "

<< p.value;

// If the destination is not reachable

else

cout << "Destination not reachable from given "

"source";

}

// Driver code

int main()

{

// input matrix with hurdles shown with number 0

int mat[R][C] = { { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 1, 1, 0, 1, 1, 0, 1, 1, 0, 1 },

{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 } };

// find longest path with source (0, 0) and

// destination (1, 7)

findLongestPath(mat, 0, 0, 1, 7);

return 0;

}

Output: 

Length of longest possible route is 24

# 270. Print all possible paths from top left to bottom right of a mXn matrix

The problem is to print all the possible paths from top left to bottom right of a mXn matrix with the constraints that ***from each cell you can either move only to right or down***.

**Examples :**

Input : 1 2 3

4 5 6

Output : 1 4 5 6

1 2 5 6

1 2 3 6

Input : 1 2

3 4

Output : 1 2 4

1 3 4

## Solution:

The algorithm is a simple recursive algorithm, from each cell first print all paths by going down and then print all paths by going right. Do this recursively for each cell encountered.

Following are implementation of the above algorithm.

// C++ program to Print all possible paths from

// top left to bottom right of a mXn matrix

#include<iostream>

using namespace std;

/\* mat: Pointer to the starting of mXn matrix

i, j: Current position of the robot (For the first call use 0,0)

m, n: Dimensions of given the matrix

pi: Next index to be filed in path array

\*path[0..pi-1]: The path traversed by robot till now (Array to hold the

path need to have space for at least m+n elements) \*/

void printAllPathsUtil(int \*mat, int i, int j, int m, int n, int \*path, int pi)

{

// Reached the bottom of the matrix so we are left with

// only option to move right

if (i == m - 1)

{

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

path[pi + k - j] = \*((mat + i\*n) + k);

for (int l = 0; l < pi + n - j; l++)

cout << path[l] << " ";

cout << endl;

return;

}

// Reached the right corner of the matrix we are left with

// only the downward movement.

if (j == n - 1)

{

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

path[pi + k - i] = \*((mat + k\*n) + j);

for (int l = 0; l < pi + m - i; l++)

cout << path[l] << " ";

cout << endl;

return;

}

// Add the current cell to the path being generated

path[pi] = \*((mat + i\*n) + j);

// Print all the paths that are possible after moving down

printAllPathsUtil(mat, i+1, j, m, n, path, pi + 1);

// Print all the paths that are possible after moving right

printAllPathsUtil(mat, i, j+1, m, n, path, pi + 1);

// Print all the paths that are possible after moving diagonal

// printAllPathsUtil(mat, i+1, j+1, m, n, path, pi + 1);

}

// The main function that prints all paths from top left to bottom right

// in a matrix 'mat' of size mXn

void printAllPaths(int \*mat, int m, int n)

{

int \*path = new int[m+n];

printAllPathsUtil(mat, 0, 0, m, n, path, 0);

}

// Driver program to test above functions

int main()

{

int mat[2][3] = { {1, 2, 3}, {4, 5, 6} };

printAllPaths(\*mat, 2, 3);

return 0;

}

**Output**

1 4 5 6

1 2 5 6

1 2 3 6

Note that in the above code, the last line of printAllPathsUtil() is commented, If we uncomment this line, we get all the paths from the top left to bottom right of a nXm matrix if the diagonal movements are also allowed. And also if moving to some of the cells are not permitted then the same code can be improved by passing the restriction array to the above function.

Note all the above approach take some extra time and space for solving the problem ,we can simply use backtracking algorithm to solve problem in optimized manner

#include<bits/stdc++.h>

using namespace std;

// function to display the path

void display(vector<int> &ans) {

for(auto i :ans ) {

cout<<i <<" ";

}

cout<<endl;

}

// a function which check whether our step is safe or not

bool issafe(int r,int c,vector<vector<int>>& visited,int n,int m) {

return (r < n and c <m and visited[r] !=-1 ); // return true if all values satisfied else false

}

void FindPaths(vector<vector<int>> &grid,int r,int c, int n,int m,vector<int> &ans) {

// when we hit the last cell we reach to destination then directly push the path

if(r == n-1 and c == m-1) {

ans.push\_back(grid[r]);

display(ans); // function to display the path stored in ans vector

ans.pop\_back(); // pop back because we need to backtrack to explore more path

return ;

}

// we will store the current value in ch and mark the visited place as -1

int ch = grid[r];

ans.push\_back(ch); // push the path in ans array

grid[r] = -1; // mark the visited place with -1

// if is it safe to take next downward step then take it

if(issafe(r+1,c,grid,n,m)) {

FindPaths(grid,r+1,c,n,m,ans);

}

// if is it safe to take next rightward step then take it

if(issafe(r,c+1,grid,n,m)) {

FindPaths(grid,r,c+1,n,m,ans);

}

// backtracking step we need to make values original so to we can visit it by some another path

grid[r] = ch;

// remove the current path element we explore

ans.pop\_back();

return ;

}

int main() {

int n = 3 ,m =3;

vector<vector<int> >grid{ {1,2,3},{4,5,6},{7,8,9}};

vector<int>ans ; // it will store the path which we have covered

FindPaths(grid,0,0,n,m,ans); // here 0,0 are initial position to start with

return 0;

}

**Output**

1 4 7 8 9

1 4 5 8 9

1 4 5 6 9

1 2 5 8 9

1 2 5 6 9

1 2 3 6 9

So by these method you can optimized your code.

**TC-**O(2^n\*m)   **, SC –**O(n)

**Another Approach (Iterative) :**

1. In this approach we will use **BFS (breadth first search)** to find all possible paths.

2. We will make a queue which contains the following information :

    a)  Vector that stores the path up to a certain cell.

    b)  coordinates of the cell.

3. We will start from the top-left cell and push cell value and coordinates in the queue.

4. We will keep on exploring right and down cell (if possible) until queue is not empty

   and push them in the queue by updating the current cell vector.

5. If we reach the last cell then we have got one answer and we will print the answer vector.

// c++ implementation for the above approach

#include <bits/stdc++.h>

using namespace std;

// this structure stores information

// about a particular cell i.e

// path upto that cell and cell's

// coordinates

struct info {

vector<int> path;

int i;

int j;

};

void printAllPaths(vector<vector<int> >& maze)

{

int n = maze.size();

int m = maze[0].size();

queue<info> q;

// pushing top-left cell into the queue

q.push({ { maze[0][0] }, 0, 0 });

while (!q.empty()) {

info p = q.front();

q.pop();

// if we reached the bottom-right cell

// i.e the destination then print the path

if (p.i == n - 1 && p.j == m - 1) {

for (auto x : p.path)

cout << x << " ";

cout << "\n";

}

// if we are in the last row

// then only right movement is possible

else if (p.i == n - 1) {

vector<int> temp = p.path;

// updating the current path

temp.push\_back(maze[p.i][p.j + 1]);

q.push({ temp, p.i, p.j + 1 });

}

// if we are in the last column

// then only down movement is possible

else if (p.j == m - 1) {

vector<int> temp = p.path;

// updating the current path

temp.push\_back(maze[p.i + 1][p.j]);

q.push({ temp, p.i + 1, p.j });

}

// else both right and down movement

// are possible

else { // right movement

vector<int> temp = p.path;

// updating the current path

temp.push\_back(maze[p.i][p.j + 1]);

q.push({ temp, p.i, p.j + 1 });

// down movement

temp.pop\_back();

// updating the current path

temp.push\_back(maze[p.i + 1][p.j]);

q.push({ temp, p.i + 1, p.j });

}

}

}

// Driver Code

int main()

{

vector<vector<int> > maze{ { 1, 2, 3 },

{ 4, 5, 6 },

{ 7, 8, 9 } };

printAllPaths(maze);

return 0;

}

**Output**

1 2 3 6 9

1 2 5 6 9

1 2 5 8 9

1 4 5 6 9

1 4 5 8 9

1 4 7 8 9

# 271. Partition of a set into K subsets with equal sum

Given an integer array **a[ ]** of **N** elements and an integer **K**, the task is to check if the array **a[ ]** could be divided into **K** non-empty subsets with equal sum of elements.  
**Note:** All elements of this array should be part of exactly one partition.

**Example 1:**

**Input:**

N = 5

a[] = {2,1,4,5,6}

K = 3

**Output:**

1

**Explanation:** we can divide above array

into 3 parts with equal sum as (2, 4),

(1, 5), (6)

**Example 2:**

**Input**:

N = 5

a[] = {2,1,5,5,6}

K = 3

**Output:**

0

**Explanation**: It is not possible to divide

above array into 3 parts with equal sum.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **isKPartitionPossible()** which takes the array a[], the size of the array N, and the value of K as inputs and returns true(same as 1) if possible, otherwise false(same as 0).

**Expected Time Complexity:** O(N\*2N).  
**Expected Auxiliary Space:** O(2N).

**Constraints:**  
1 ≤ K ≤ N ≤ 10  
1 ≤ a[i] ≤ 100

## Solution:

We can solve this problem recursively, we keep an array for sum of each partition and a boolean array to check whether an element is already taken into some partition or not.  
First we need to check some base cases,  
If K is 1, then we already have our answer, complete array is only subset with same sum.  
If N < K, then it is not possible to divide array into subsets with equal sum, because we can’t divide the array into more than N parts.  
If sum of array is not divisible by K, then it is not possible to divide the array. We will proceed only if k divides sum. Our goal reduces to divide array into K parts where sum of each part should be array\_sum/K  
In below code a recursive method is written which tries to add array element into some subset. If sum of this subset reaches required sum, we iterate for next part recursively, otherwise we backtrack for different set of elements. If number of subsets whose sum reaches the required sum is (K-1), we flag that it is possible to partition array into K parts with equal sum, because remaining elements already have a sum equal to required sum.

// C++ program to check whether an array can be

// partitioned into K subsets of equal sum

#include <bits/stdc++.h>

using namespace std;

// Recursive Utility method to check K equal sum

// subsetition of array

/\*\*

array - given input array

subsetSum array - sum to store each subset of the array

taken - boolean array to check whether element

is taken into sum partition or not

K - number of partitions needed

N - total number of element in array

curIdx - current subsetSum index

limitIdx - lastIdx from where array element should

be taken \*/

bool isKPartitionPossibleRec(int arr[], int subsetSum[], bool taken[],

int subset, int K, int N, int curIdx, int limitIdx)

{

if (subsetSum[curIdx] == subset)

{

/\* current index (K - 2) represents (K - 1) subsets of equal

sum last partition will already remain with sum 'subset'\*/

if (curIdx == K - 2)

return true;

// recursive call for next subsetition

return isKPartitionPossibleRec(arr, subsetSum, taken, subset,

K, N, curIdx + 1, N - 1);

}

// start from limitIdx and include elements into current partition

for (int i = limitIdx; i >= 0; i--)

{

// if already taken, continue

if (taken[i])

continue;

int tmp = subsetSum[curIdx] + arr[i];

// if temp is less than subset then only include the element

// and call recursively

if (tmp <= subset)

{

// mark the element and include into current partition sum

taken[i] = true;

subsetSum[curIdx] += arr[i];

bool nxt = isKPartitionPossibleRec(arr, subsetSum, taken,

subset, K, N, curIdx, i - 1);

// after recursive call unmark the element and remove from

// subsetition sum

taken[i] = false;

subsetSum[curIdx] -= arr[i];

if (nxt)

return true;

}

}

return false;

}

// Method returns true if arr can be partitioned into K subsets

// with equal sum

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

{

// If K is 1, then complete array will be our answer

if (K == 1)

return true;

// If total number of partitions are more than N, then

// division is not possible

if (N < K)

return false;

// if array sum is not divisible by K then we can't divide

// array into K partitions

int sum = 0;

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

sum += arr[i];

if (sum % K != 0)

return false;

// the sum of each subset should be subset (= sum / K)

int subset = sum / K;

int subsetSum[K];

bool taken[N];

// Initialize sum of each subset from 0

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

subsetSum[i] = 0;

// mark all elements as not taken

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

taken[i] = false;

// initialize first subsubset sum as last element of

// array and mark that as taken

subsetSum[0] = arr[N - 1];

taken[N - 1] = true;

// call recursive method to check K-substitution condition

return isKPartitionPossibleRec(arr, subsetSum, taken,

subset, K, N, 0, N - 1);

}

// Driver code to test above methods

int main()

{

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

int N = sizeof(arr) / sizeof(arr[0]);

int K = 3;

if (isKPartitionPossible(arr, N, K))

cout << "Partitions into equal sum is possible.\n";

else

cout << "Partitions into equal sum is not possible.\n";

}

**Output:**

Partitions into equal sum is possible.

**My Implementation:**

class Solution{

public:

bool flag = false;

void fun(int a[], int n, int k, int sum, int curr, int ind){

if(ind==n){

flag = true;

return;

}

if(curr==sum)

curr = 0;

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

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

swap(a[ind], a[i]);

fun(a, n, k, sum, curr+a[ind], ind+1);

swap(a[ind], a[i]);

}

}

}

bool isKPartitionPossible(int a[], int n, int k)

{

int sum = 0;

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

sum += a[i];

if(sum%k!=0)

return false;

fun(a, n, k, sum/k, 0, 0);

return flag;

}

};

# 272. Find the K-th Permutation Sequence of first N natural numbers

Given two **integers N and K**, find the Kth permutation sequence of numbers from 1 to N without using STL function.  
*Note: Assume that the inputs are such that Kth permutation of N number is always possible.*

**Examples:**

***Input:****N = 3, K = 4****Output:****231****Explanation:****The ordered list of permutation sequence from integer 1 to 3 is : 123, 132, 213, 231, 312, 321. So, the 4th permutation sequence is “231”.*

***Input:****N = 2, K = 1****Output:****12   
Explanation:   
For n = 2, only 2 permutations are possible 12 21. So, the 1st permutation sequence is “12”.*

## Solution:

**Naive Approach:**  
To solve the problem mentioned above the simple approach is to find all permutation sequences and output the kth out of them. But this method is not so efficient and takes more time, hence it can be optimized.

**Efficient Approach:**  
To optimize the above method mentioned above, observe that the value of *k* can be directly used to find the number at each index of the sequence.

* The first position of an *n* length sequence is occupied by each of the numbers from 1 to n **exactly n! / n that is (n-1)! number of times** and in ascending order. So the **first position of the kth sequence will be occupied by the number present at index = k / (n-1)!** (according to 1-based indexing).
* The currently found number can not occur again so it is removed from the original n numbers and now the problem reduces to finding the ( k % (n-1)! )th permutation sequence of the remaining n-1 numbers.
* This process can be repeated until we have only one number left which will be placed in the first position of the last 1-length sequence.
* The factorial values involved here can be very large as compared to k. So, the trick used to avoid the full computation of such large factorials is that as soon as the **product n \* (n-1) \* … becomes greater than k**, we no longer need to find the actual factorial value because:

*k / n\_actual\_factorial\_value = 0   
and k / n\_partial\_factorial\_value = 0   
when partial\_factorial\_value > k*

Below is the implementation of the above approach:

// C++ program to Find the kth Permutation

// Sequence of first n natural numbers

#include <bits/stdc++.h>

using namespace std;

// Function to find the index of number

// at first position of

// kth sequence of set of size n

int findFirstNumIndex(int& k, int n)

{

if (n == 1)

return 0;

n--;

int first\_num\_index;

// n\_actual\_fact = n!

int n\_partial\_fact = n;

while (k >= n\_partial\_fact

&& n > 1) {

n\_partial\_fact

= n\_partial\_fact

\* (n - 1);

n--;

}

// First position of the

// kth sequence will be

// occupied by the number present

// at index = k / (n-1)!

first\_num\_index = k / n\_partial\_fact;

k = k % n\_partial\_fact;

return first\_num\_index;

}

// Function to find the

// kth permutation of n numbers

string findKthPermutation(int n, int k)

{

// Store final answer

string ans = "";

set<int> s;

// Insert all natural number

// upto n in set

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

s.insert(i);

set<int>::iterator itr;

// Mark the first position

itr = s.begin();

// subtract 1 to get 0 based indexing

k = k - 1;

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

int index

= findFirstNumIndex(k, n - i);

advance(itr, index);

// itr now points to the

// number at index in set s

ans += (to\_string(\*itr));

// remove current number from the set

s.erase(itr);

itr = s.begin();

}

return ans;

}

// Driver code

int main()

{

int n = 3, k = 4;

string kth\_perm\_seq

= findKthPermutation(n, k);

cout << kth\_perm\_seq << endl;

return 0;

}

**Output:**

231