Contents

1 Introduction **5**
Source 8

2 The Knight's tour problem **9**
Source 18

3 Warnsdorff's algorithm for Knight's tour problem **19**
Source 24

4 Rat in a Maze **25**
Source 33

5 N Queen Problem **34**
Source 43

6 N Queen in O(n) space **44**
Source 47

7 Printing all solutions in N-Queen Problem **48**
Source 52

8 8 queen problem **53**
Source 53

9 Subset Sum **54**
Source 60

10 m Coloring Problem **61**
Source 68

11 Hamiltonian Cycle **69**
Source 79

12 Sudoku **80**
Source 87

13 Solving Cryptarithmetic Puzzles **88**
Source 90

14 Magnet Puzzle **91**
Source 92

15 Boggle | Set 1 (Using DFS) **93**
Source 96

16 Boggle | Set 2 (Using Trie) **97**
Source 106

17 Tug of War **107**
Source 112

18 Backtracking to find all subsets **113**
Source 115

19 Print the DFS traversal step-wise (Backtracking also) **116**
Source 119

20 C++ program for Solving Cryptarithmetic Puzzles **120**
Source 125

21 A backtracking approach to generate n bit Gray Codes **126**
Source 128

22 Minimum queens required to cover all the squares of a chess board **129**
Source 133

23 Print all the combinations of a string in lexicographical order **134**
Source 137

24 Recursive program to generate power set **138**
Source 141

25 Smallest number with given sum of digits and sum of square of digits **142**
Source 145

26 Minimize number of unique characters in string **146**
Source 149

27 Rat in a Maze with multiple steps or jump allowed **150**
Source 154

28 Fill 8 numbers in grid with given conditions **155**
Source 160

29 Power Set in Lexicographic order **161**
Source 163

30 Prime numbers after prime P with sum S **164**
Source 178

31 Smallest expression to represent a number using single digit **179**
Source 185

32 Count all possible paths between two vertices **186**
Source 193

33 Check if a given string is sum-string **194**
Source 197

34 Print all possible strings that can be made by placing spaces **198**
Source 203

35 Combinational Sum **204**
Source 207

36 Combinations where every element appears twice and distance between appearances is equal to the value **208**
Source 213

37 Print all palindromic partitions of a string **214**
Source 219

38 Word Break Problem using Backtracking **220**
Source 222

39 Partition of a set into K subsets with equal sum **223**
Source 226

40 Print all longest common sub-sequences in lexicographical order **227**
Source 230

41 Remove Invalid Parentheses **231**
Source 233

42 Find all distinct subsets of a given set **234**
Source 236

43 Find shortest safe route in a path with landmines **237**
Source 242

44 Longest Possible Route in a Matrix with Hurdles **243**
Source 247

45 Match a pattern and String without using regular expressions **248**
Source 251

46 Find Maximum number possible by doing at-most K swaps **252**
Source 254

47 Find paths from corner cell to middle cell in maze **255**
Source 259

48 Find if there is a path of more than k length from a source 260

Source . . . 264

49 Fill two instances of all numbers from 1 to n in a specific way 265

Source . . . 267

50 Print all paths from a given source to a destination 268

Source . . . 275

51 Print all possible paths from top left to bottom right of a mXn matrix 276

Source . . . 279

52 Write a program to print all permutations of a given string 280

Source . . . 286

53 Given an array A[] and a number x, check for pair in A[] with sum as x 287

Source . . . 301

Chapter 1

Introduction

Backtracking | Introduction - GeeksforGeeks

Prerequisites :

- Recursion
- Complexity Analysis

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

According to the wiki definition,

> **Backtracking** can be defined as a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem.

How to determine if a problem can be solved using Backtracking?

Generally, every constraint satisfaction problem which has clear and well-defined constraints on any objective solution, that incrementally builds candidate to the solution and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution, can be solved by Backtracking. However, most of the problems that are discussed, can be solved using other known algorithms like *Dynamic Programming* or *Greedy Algorithms* in logarithmic, linear, linear-logarithmic time complexity in order of input size, and therefore, outshine the backtracking algorithm in every respect (since backtracking algorithms are generally exponential in both time and space). However, a few problems still remain, that only have backtracking algorithms to solve them until now.

Consider a situation that you have three boxes in front of you and only one of them has a gold coin in it but you do not know which one. So, in order to get the coin, you will have to open all of the boxes one by one. You will first check the first box, if it does not contain the coin, you will have to close it and check the second box and so on until you find the coin. This is what backtracking is, that is solving all sub-problems one by one in order to reach the best possible solution.

Consider the below example to understand the Backtracking approach more formally,

Given an instance of any computational problem P and data D corresponding to the instance, all the constraints that need to be satisfied in order to solve the problem are represented by C. A backtracking algorithm will then work as follows:

The Algorithm begins to build up a solution, starting with an empty solution set S. **S = {}**

1. Add to S the first move that is still left (All possible moves are added to S one by one). This now creates a new sub-tree S in the search tree of the algorithm.
2. Check if $S + s$ satisfies each of the constraints in C.
    - If Yes, then the sub-tree S is "eligible" to add more "children".
    - Else, the entire sub-tree S is useless, so recurs back to step 1 using argument S.
3. In the event of "eligibility" of the newly formed sub-tree S, recurs back to step 1, using argument $S + s$.
4. If the check for $S + s$ returns that it is a solution for the entire data D. Output and terminate the program.
   If not, then return that no solution is possible with the current S and hence discard it.

Pseudo Code for Backtracking :

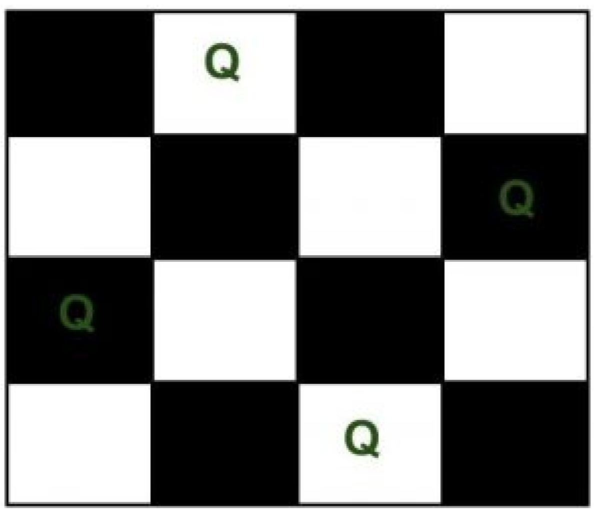
1. Recursive backtracking solution.

```
void findSolutions(n, other params) :
    if (found a solution) :
        solutionsFound = solutionsFound + 1;
        displaySolution();
        if (solutionsFound >= solutionTarget) :
            System.exit(0);
        return

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            findSolutions(n+1, other params);
            removeValue(val, n);
```

2. Finding whether a solution exists or not

```
boolean findSolutions(n, other params) :
    if (found a solution) :
        displaySolution();
        return true;

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            if (findSolutions(n+1, other params))
                return true;
            removeValue(val, n);
        return false;
```

Let us try to solve a standard Backtracking problem, **N-Queen Problem**.
The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for the above 4 queen solution.

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

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 the 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.
```

You may refer to the article on Backtracking | Set 3 (N Queen Problem) for complete implementation of the above approach.

More Backtracking Problems:

- Backtracking | Set 1 (The Knight's tour problem)
- Backtracking | Set 2 (Rat in a Maze)
- Backtracking | Set 4 (Subset Sum)
- Backtracking | Set 5 (m Coloring Problem)
- –> Click Here for More

Source

https://www.geeksforgeeks.org/backtracking-introduction/

Chapter 2

The Knight's tour problem

The Knight's tour problem | Backtracking-1 - GeeksforGeeks

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that "works". Problems which are typically solved using backtracking technique have 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 in incremental way and is an optimization over the Naive solution where all possible configurations are generated and tried.

For example, consider the following Knight's Tour problem.
The knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once.

Path followed by Knight to cover all the cells

Following is chessboard with 8 x 8 cells. Numbers in cells indicate move number of Knight.

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

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 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 work out then we go to 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

```
 // C program for Knight Tour problem
#include<stdio.h>
#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 */
bool 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++)
            printf(" %2d ", sol[x][y]);
        printf("\n");
    }
}

/* 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.  */
bool 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) == false)
    {
        printf("Solution does not exist");
        return false;
    }
    else
        printSolution(sol);

    return true;
}

/* A recursive utility function to solve Knight Tour
   problem */
int solveKTUtil(int x, int y, int movei, int sol[N][N],
                int xMove[N], int yMove[N])
{
   int k, next_x, next_y;
   if (movei == N*N)
       return true;

   /* 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) == true)
             return true;
         else
             sol[next_x][next_y] = -1;// backtracking
       }
   }

   return false;
}

/* Driver program to test above functions */
int main()
{
    solveKT();
```

```
    return 0;
}
```

Java

```
 // Java program for Knight Tour problem
class KnightTour {
    static int N = 8;

    /* A utility function to check if i,j are
       valid indexes for N*N chessboard */
    static boolean isSafe(int x, int y, int sol[][]) {
        return (x >= 0 && x < N && y >= 0 &&
                y < N && sol[x][y] == -1);
    }

    /* A utility function to print solution
       matrix sol[N][N] */
    static void printSolution(int sol[][]) {
        for (int x = 0; x < N; x++) {
            for (int y = 0; y < N; y++)
                System.out.print(sol[x][y] + " ");
            System.out.println();
        }
    }

    /* 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.  */
    static boolean solveKT() {
        int sol[][] = new int[8][8];

        /* 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[] = {2, 1, -1, -2, -2, -1, 1, 2};
        int yMove[] = {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)) {
            System.out.println("Solution does not exist");
            return false;
        } else
            printSolution(sol);

        return true;
    }

    /* A recursive utility function to solve Knight
       Tour problem */
    static boolean solveKTUtil(int x, int y, int movei,
                               int sol[][], int xMove[],
                               int yMove[]) {
        int k, next_x, next_y;
        if (movei == N * N)
            return true;

        /* 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))
                    return true;
                else
                    sol[next_x][next_y] = -1;// backtracking
            }
        }

        return false;
    }

    /* Driver program to test above functions */
    public static void main(String args[]) {
        solveKT();
    }
}
// This code is contributed by Abhishek Shankhadhar
```

C#

```
 // C# program for
// Knight Tour problem
using System;

class GFG
{
    static int N = 8;

    /* A utility function to
    check if i,j are valid
    indexes for N*N chessboard */
    static bool isSafe(int x, int y,
                       int[,] sol)
    {
        return (x >= 0 && x < N &&
                y >= 0 && y < N &&
                sol[x, y] == -1);
    }

    /* A utility function to
    print solution matrix sol[N][N] */
    static void printSolution(int[,] sol)
    {
        for (int x = 0; x < N; x++)
        {
            for (int y = 0; y < N; y++)
                Console.Write(sol[x, y] + " ");
            Console.WriteLine();
        }
    }

    /* 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. */
    static bool solveKT()
     {
        int[,] sol = new int[8, 8];
```

```
    /* 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 = {2, 1, -1, -2,
                  -2, -1, 1, 2};
    int[] yMove = {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))
    {
        Console.WriteLine("Solution does "+
                          "not exist");
        return false;
    }
    else
        printSolution(sol);

    return true;
}

/* A recursive utility function
to solve Knight Tour problem */
static bool solveKTUtil(int x, int y, int movei,
                        int[,] sol, int[] xMove,
                        int[] yMove)
{
    int k, next_x, next_y;
    if (movei == N * N)
        return true;

    /* 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))
                    return true;
                else
                    // backtracking
                    sol[next_x,next_y] = -1;
            }
        }

        return false;
    }

    // Driver Code
    public static void Main()
    {
        solveKT();
    }
}

// This code is contributed by mits.
```

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
```

Note that Backtracking is not the best solution for the Knight's tour problem. See below article for other better solutions. The purpose of this post is to explain Backtracking with an example.
Warnsdorff's algorithm for Knight's tour problem

References:
http://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf
http://www.cis.upenn.edu/~matuszek/cit594-2009/Lectures/35-backtracking.ppt

http://mathworld.wolfram.com/KnightsTour.html
http://en.wikipedia.org/wiki/Knight%27s_tour

Improved By : Mithun Kumar

Source

https://www.geeksforgeeks.org/the-knights-tour-problem-backtracking-1/

Chapter 3

Warnsdorff's algorithm for Knight's tour problem

Warnsdorff's algorithm for Knight's tour problem - GeeksforGeeks

Problem : A knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once.

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

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

We have discussed Backtracking Algorithm for solution of Knight's tour. In this post Warnsdorff's heuristic is discussed.

Warnsdorff's Rule:

1. We can start from any initial position of the knight on the board.
2. We always move to an adjacent, unvisited square with minimal degree (minimum number of unvisited adjacent).

This algorithm may also more generally be applied to any graph.

Some definitions:

- A position Q is accessible from a position P if P can move to Q by a single Knight's move, and Q has not yet been visited.
- The accessibility of a position P is the number of positions accessible from P.

Algorithm:

1. Set P to be a random initial position on the board
2. Mark the board at P with the move number "1"
3. Do following for each move number from 2 to the number of squares on the board:
   - let S be the set of positions accessible from P.
   - Set P to be the position in S with minimum accessibility
   - Mark the board at P with the current move number
4. Return the marked board — each square will be marked with the move number on which it is visited.

Below is implementation of above algorithm.

```
 // C++ program to for Kinight's tour problem usin
// Warnsdorff's algorithm
#include <bits/stdc++.h>
#define N 8

// Move pattern on basis of the change of
// x coordinates and y coordinates respectively
static int cx[N] = {1,1,2,2,-1,-1,-2,-2};
static int cy[N] = {2,-2,1,-1,2,-2,1,-1};

// function restricts the knight to remain within
// the 8x8 chessboard
bool limits(int x, int y)
{
    return ((x >= 0 && y >= 0) && (x < N && y < N));
}

/* Checks whether a square is valid and empty or not */
bool isempty(int a[], int x, int y)
{
    return (limits(x, y)) && (a[y*N+x] < 0);
}

/* Returns the number of empty squares adjacent
   to (x, y) */
```

```
int getDegree(int a[], int x, int y)
{
    int count = 0;
    for (int i = 0; i < N; ++i)
        if (isempty(a, (x + cx[i]), (y + cy[i])))
            count++;

    return count;
}

// Picks next point using Warnsdorff's heuristic.
// Returns false if it is not possible to pick
// next point.
bool nextMove(int a[], int *x, int *y)
{
    int min_deg_idx = -1, c, min_deg = (N+1), nx, ny;

    // Try all N adjacent of (*x, *y) starting
    // from a random adjacent. Find the adjacent
    // with minimum degree.
    int start = rand()%N;
    for (int count = 0; count < N; ++count)
    {
        int i = (start + count)%N;
        nx = *x + cx[i];
        ny = *y + cy[i];
        if ((isempty(a, nx, ny)) &&
           (c = getDegree(a, nx, ny)) < min_deg)
        {
            min_deg_idx = i;
            min_deg = c;
        }
    }

    // IF we could not find a next cell
    if (min_deg_idx == -1)
        return false;

    // Store coordinates of next point
    nx = *x + cx[min_deg_idx];
    ny = *y + cy[min_deg_idx];

    // Mark next move
    a[ny*N + nx] = a[(*y)*N + (*x)]+1;

    // Update next point
    *x = nx;
    *y = ny;
```

```
    return true;
}

/* displays the chessboard with all the
  legal knight's moves */
void print(int a[])
{
    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
            printf("%d\t",a[j*N+i]);
        printf("\n");
    }
}

/* checks its neighbouring sqaures */
/* If the knight ends on a square that is one
   knight's move from the beginning square,
   then tour is closed */
bool neighbour(int x, int y, int xx, int yy)
{
    for (int i = 0; i < N; ++i)
        if (((x+cx[i]) == xx)&&((y + cy[i]) == yy))
            return true;

    return false;
}

/* Generates the legal moves using warnsdorff's
  heuristics. Returns false if not possible */
bool findClosedTour()
{
    // Filling up the chessboard matrix with -1's
    int a[N*N];
    for (int i = 0; i< N*N; ++i)
        a[i] = -1;

    // Randome initial position
    int sx = rand()%N;
    int sy = rand()%N;

    // Current points are same as initial points
    int x = sx, y = sy;
    a[y*N+x] = 1; // Mark first move.

    // Keep picking next points using
    // Warnsdorff's heuristic
```

```
    for (int i = 0; i < N*N-1; ++i)
        if (nextMove(a, &x, &y) == 0)
            return false;

    // Check if tour is closed (Can end
    // at starting point)
    if (!neighbour(x, y, sx, sy))
        return false;

    print(a);
    return true;
}

// Driver code
int main()
{
    // To make sure that different random
    // initial positions are picked.
    srand(time(NULL));

    // While we don't get a solution
    while (!findClosedTour())
    {
    ;
    }

    return 0;
}
```

Output:

```
59   14   63   32   1    16   19   34
62   31   60   15   56    33    2    17
13   58   55   64   49    18    35    20
30   61   42   57   54    51    40    3
43   12   53   50   41    48    21    36
26   29   44   47   52    39    4    7
11   46   27   24   9    6    37    22
28   25   10   45   38    23    8    5
```

The Hamiltonian path problem is NP-hard in general. In practice, Warnsdorf's heuristic successfully finds a solution in linear time.

Do you know?

"On an 8 × 8 board, there are exactly 26,534,728,821,064 directed closed tours (i.e. two tours along the same path that travel in opposite directions are counted separately, as are rotations and reflections). The number of undirected closed tours is half this number, since every tour can be traced in reverse!"

Source

https://www.geeksforgeeks.org/warnsdorffs-algorithm-knights-tour-problem/

Chapter 4

Rat in a Maze

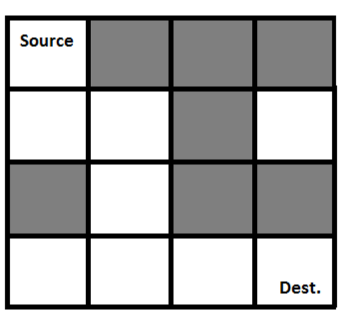
Rat in a Maze | Backtracking-2 - GeeksforGeeks

We have discussed Backtracking and Knight's tour problem in Set 1. Let us discuss Rat in a Mazeas another example problem that can be solved using Backtracking.

A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.
In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

Following is an example maze.

```
Gray blocks are dead ends (value = 0).
```

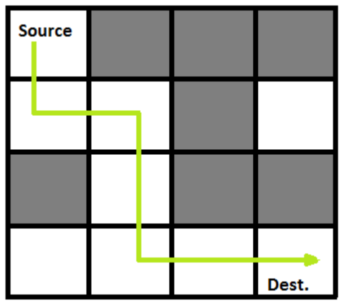


Following is binary matrix representation of the above maze.

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

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

Following is a maze with highlighted solution path.



Following is the solution matrix (output of program) for the above input matrx.

```
                {1, 0, 0, 0}
                {1, 1, 0, 0}
                {0, 1, 0, 0}
                {0, 1, 1, 1}
 All enteries in solution path are marked as 1.
```

Naive Algorithm

The Naive Algorithm is to generate all paths from source to destination and one by one check if the generated path satisfies the constraints.

```
while there are untried paths
{
   generate the next path
   if this path has all blocks as 1
   {
      print this path;
   }
}
```

Backtracking Algorithm

```
If destination is reached
    print the solution matrix
Else
   a) Mark current cell in solution matrix as 1.
   b) Move forward in the horizontal direction and recursively check if this
       move leads to a solution.
```

```
c) If the move chosen in the above step doesn't lead to a solution
    then move down and check if this move leads to a solution.
d) If none of the above solutions works then unmark this cell as 0
    (BACKTRACK) and return false.
```

Implementation of Backtracking solution

C/C++

```
 /* C/C++ program to solve Rat in a Maze problem using
   backtracking */
#include<stdio.h>

// Maze size
#define N 4

bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N]);

/* A utility function to print solution matrix sol[N][N] */
void printSolution(int sol[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf(" %d ", sol[i][j]);
        printf("\n");
    }
}

/* A utility function to check if x,y is valid index for N*N maze */
bool isSafe(int maze[N][N], int x, int y)
{
    // if (x,y outside maze) return false
    if(x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1)
        return true;

    return false;
}

/* This function solves the Maze problem using Backtracking.  It mainly
   uses solveMazeUtil() to solve the problem. It returns false if no
   path is possible, otherwise return true and prints the path in the
   form of 1s. Please note that there may be more than one solutions,
   this function prints one of the feasible solutions.*/
bool solveMaze(int maze[N][N])
{
    int sol[N][N] = { {0, 0, 0, 0},
```

```
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0}
    };

    if(solveMazeUtil(maze, 0, 0, sol) == false)
    {
        printf("Solution doesn't exist");
        return false;
    }

    printSolution(sol);
    return true;
}

/* A recursive utility function to solve Maze problem */
bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N])
{
    // if (x,y is goal) return true
    if(x == N-1 && y == N-1)
    {
        sol[x][y] = 1;
        return true;
    }

    // Check if maze[x][y] is valid
    if(isSafe(maze, x, y) == true)
    {
        // mark x,y as part of solution path
        sol[x][y] = 1;

        /* Move forward in x direction */
        if (solveMazeUtil(maze, x+1, y, sol) == true)
            return true;

        /* If moving in x direction doesn't give solution then
           Move down in y direction  */
        if (solveMazeUtil(maze, x, y+1, sol) == true)
            return true;

        /* If none of the above movements work then BACKTRACK:
            unmark x,y as part of solution path */
        sol[x][y] = 0;
        return false;
    }

    return false;
}
```

```
// driver program to test above function
int main()
{
    int maze[N][N]  =  { {1, 0, 0, 0},
        {1, 1, 0, 1},
        {0, 1, 0, 0},
        {1, 1, 1, 1}
    };

    solveMaze(maze);
    return 0;
}
```

Java

```
 /* Java program to solve Rat in a Maze problem using
   backtracking */

public class RatMaze
{
    final int N = 4;

    /* A utility function to print solution matrix
       sol[N][N] */
    void printSolution(int sol[][])
    {
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
                System.out.print(" " + sol[i][j] +
                                 " ");
            System.out.println();
        }
    }

    /* A utility function to check if x,y is valid
        index for N*N maze */
    boolean isSafe(int maze[][], int x, int y)
    {
        // if (x,y outside maze) return false
        return (x >= 0 && x < N && y >= 0 &&
                y < N && maze[x][y] == 1);
    }

    /* This function solves the Maze problem using
       Backtracking. It mainly uses solveMazeUtil()
       to solve the problem. It returns false if no
```

```
   path is possible, otherwise return true and
   prints the path in the form of 1s. Please note
   that there may be more than one solutions, this
   function prints one of the feasible solutions.*/
boolean solveMaze(int maze[][])
{
    int sol[][] = {{0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0}
    };

    if (solveMazeUtil(maze, 0, 0, sol) == false)
    {
        System.out.print("Solution doesn't exist");
        return false;
    }

    printSolution(sol);
    return true;
}

/* A recursive utility function to solve Maze
   problem */
boolean solveMazeUtil(int maze[][], int x, int y,
                      int sol[][])
{
    // if (x,y is goal) return true
    if (x == N - 1 && y == N - 1)
    {
        sol[x][y] = 1;
        return true;
    }

    // Check if maze[x][y] is valid
    if (isSafe(maze, x, y) == true)
    {
        // mark x,y as part of solution path
        sol[x][y] = 1;

        /* Move forward in x direction */
        if (solveMazeUtil(maze, x + 1, y, sol))
            return true;

        /* If moving in x direction doesn't give
           solution then  Move down in y direction */
        if (solveMazeUtil(maze, x, y + 1, sol))
            return true;
```

```
            /* If none of the above movements works then
               BACKTRACK: unmark x,y as part of solution
               path */
            sol[x][y] = 0;
            return false;
        }

        return false;
    }

    public static void main(String args[])
    {
        RatMaze rat = new RatMaze();
        int maze[][] = {{1, 0, 0, 0},
            {1, 1, 0, 1},
            {0, 1, 0, 0},
            {1, 1, 1, 1}
        };
        rat.solveMaze(maze);
    }
}
// This code is contributed by Abhishek Shankhadhar
```

Python3

```
 # Python3 program to solve Rat in a Maze
# problem using backracking

# Maze size
N = 4

# A utility function to print solution matrix sol
def printSolution( sol ):

    for i in sol:
        for j in i:
            print(str(j) + " ", end="")
        print("")

# A utility function to check if x,y is valid
# index for N*N Maze
def isSafe( maze, x, y ):

    if x >= 0 and x < N and y >= 0 and y < N and maze[x][y] == 1:
        return True

    return False
```

```
""" This function solves the Maze problem using Backtracking.
    It mainly uses solveMazeUtil() to solve the problem. It
    returns false if no path is possible, otherwise return
    true and prints the path in the form of 1s. Please note
    that there may be more than one solutions, this function
    prints one of the feasable solutions. """
def solveMaze( maze ):

    # Creating a 4 * 4 2-D list
    sol = [ [ 0 for j in range(4) ] for i in range(4) ]

    if solveMazeUtil(maze, 0, 0, sol) == False:
        print("Solution doesn't exist");
        return False

    printSolution(sol)
    return True

# A recursive utility function to solve Maze problem
def solveMazeUtil(maze, x, y, sol):

    #if (x,y is goal) return True
    if x == N - 1 and y == N - 1:
        sol[x][y] = 1
        return True

    # Check if maze[x][y] is valid
    if isSafe(maze, x, y) == True:
        # mark x, y as part of solution path
        sol[x][y] = 1

        # Move forward in x direction
        if solveMazeUtil(maze, x + 1, y, sol) == True:
            return True

        # If moving in x direction doesn't give solution
        # then Move down in y direction
        if solveMazeUtil(maze, x, y + 1, sol) == True:
            return True

        # If none of the above movements work then
        # BACKTRACK: unmark x,y as part of solution path
        sol[x][y] = 0
        return False

# Driver program to test above function
if __name__ == "__main__":
```

```
    # Initialising the maze
    maze = [ [1, 0, 0, 0],
             [1, 1, 0, 1],
             [0, 1, 0, 0],
             [1, 1, 1, 1] ]

    solveMaze(maze)

# This code is contributed by Shiv Shankar
```

Output: The 1 values show the path for rat

```
 1  0  0  0
 1  1  0  0
 0  1  0  0
 0  1  1  1
```

Below is an extended version of this problem.Count number of ways to reach destination in a Maze

Source

https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-2/

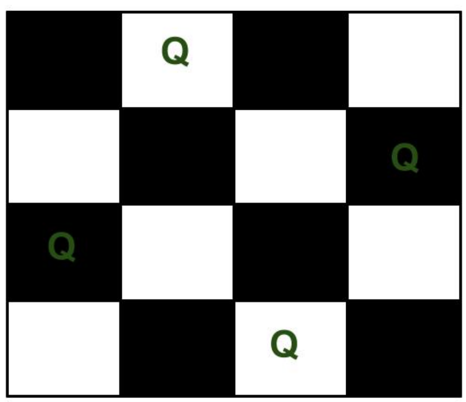
Chapter 5

N Queen Problem

N Queen Problem | Backtracking-3 - GeeksforGeeks

We have discussed Knight's tour and Rat in a Maze problems in Set 1 and Set 2 respectively. Let us discuss N Queen as another example problem that can be solved using Backtracking.

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

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

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

Naive Algorithm
Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

```
while there are untried configurations
{
   generate the next configuration
   if queens don't attack in this configuration then
   {
      print this configuration;
   }
}
```

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 so
    b) If placing the queen in [row, column] leads to a solution then return
        true.
    c) If placing queen doesn't lead to a solution then umark 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.
```

Implementation of Backtracking solution

C/C++

```
 /* C/C++ program to solve N Queen Problem using
   backtracking */
#define N 4
#include<stdio.h>
#include<stdbool.h>
```

```
/* A utility function to print solution */
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
}

/* 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(int board[N][N], int row, int col)
{
    int i, j;

    /* 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(int board[N][N], int col)
{
    /* base case: If all queens are placed
      then return true */
    if (col >= N)
        return true;
```

```
    /* Consider this column and try placing
       this queen in all rows one by one */
    for (int i = 0; i < N; i++)
    {
        /* Check if the queen can be placed on
          board[i][col] */
        if ( isSafe(board, i, col) )
        {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

            /* recur to place rest of the queens */
            if ( solveNQUtil(board, col + 1) )
                return true;

            /* 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 the queen cannot be placed in any row in
        this colum col  then return false */
    return false;
}

/* 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.*/
bool solveNQ()
{
    int board[N][N] = { {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0}
    };

    if ( solveNQUtil(board, 0) == false )
    {
      printf("Solution does not exist");
      return false;
```

```
    }

    printSolution(board);
    return true;
}

// driver program to test above function
int main()
{
    solveNQ();
    return 0;
}
```

Java

```
 /* Java program to solve N Queen Problem using
   backtracking */
public class NQueenProblem
{
    final int N = 4;

    /* A utility function to print solution */
    void printSolution(int board[][])
    {
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
                System.out.print(" " + board[i][j]
                                 + " ");
            System.out.println();
        }
    }

    /* 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
       placeed in columns from 0 to col -1. So we need
       to check only left side for attacking queens */
    boolean isSafe(int board[][], int row, int col)
    {
        int i, j;

        /* Check this row on left side */
        for (i = 0; i < col; i++)
            if (board[row][i] == 1)
                return false;

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

```
    for (i=row, j=col; i>=0 && j>=0; i--, j--)
        if (board[i][j] == 1)
            return false;

    /* Check lower diagonal on left side */
    for (i=row, j=col; j>=0 && i<N; i++, j--)
        if (board[i][j] == 1)
            return false;

    return true;
}

/* A recursive utility function to solve N
   Queen problem */
boolean solveNQUtil(int board[][], int col)
{
    /* base case: If all queens are placed
       then return true */
    if (col >= N)
        return true;

    /* Consider this column and try placing
       this queen in all rows one by one */
    for (int i = 0; i < N; i++)
    {
        /* Check if the queen can be placed on
           board[i][col] */
        if (isSafe(board, i, col))
        {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

            /* recur to place rest of the queens */
            if (solveNQUtil(board, col + 1) == true)
                return true;

            /* 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 the queen can not be placed in any row in
       this colum col, then return false */
    return false;
}
```

```
    /* 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.*/
    boolean solveNQ()
    {
        int board[][] = {{0, 0, 0, 0},
            {0, 0, 0, 0},
            {0, 0, 0, 0},
            {0, 0, 0, 0}
        };

        if (solveNQUtil(board, 0) == false)
        {
            System.out.print("Solution does not exist");
            return false;
        }

        printSolution(board);
        return true;
    }

    // driver program to test above function
    public static void main(String args[])
    {
        NQueenProblem Queen = new NQueenProblem();
        Queen.solveNQ();
    }
}
// This code is contributed by Abhishek Shankhadhar
```

Python

```
 # Python program to solve N Queen
# Problem using backtracking

global N
N = 4

def printSolution(board):
    for i in range(N):
        for j in range(N):
            print board[i][j],
        print
```

```
# 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
def isSafe(board, row, col):

    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i,j in zip(range(row,-1,-1), range(col,-1,-1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i,j in zip(range(row,N,1), range(col,-1,-1)):
        if board[i][j] == 1:
            return False

    return True

def solveNQUtil(board, col):
    # base case: If all queens are placed
    # then return true
    if col >= N:
        return True

    # Consider this column and try placing
    # this queen in all rows one by one
    for i in range(N):

        if isSafe(board, i, col):
            # Place this queen in board[i][col]
            board[i][col] = 1

            # recur to place rest of the queens
            if solveNQUtil(board, col+1) == True:
                return True

            # If placing queen in board[i][col
            # doesn't lead to a solution, then
            # queen from board[i][col]
```

```
            board[i][col] = 0

    # if the queen can not be placed in any row in
    # this colum col  then return false
    return False

# 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
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one  of the
# feasible solutions.
def solveNQ():
    board = [ [0, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0]
             ]

    if solveNQUtil(board, 0) == False:
        print "Solution does not exist"
        return False

    printSolution(board)
    return True

# driver program to test above function
solveNQ()

# This code is contributed by Divyanshu Mehta
```

Output: The 1 values indicate placements of queens

```
 0  0  1  0
 1  0  0  0
 0  0  0  1
 0  1  0  0
```

Printing all solutions in N-Queen Problem

Sources:
http://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf
http://en.literateprograms.org/Eight_queens_puzzle_%28C%29
http://en.wikipedia.org/wiki/Eight_queens_puzzle

Improved By : AniruddhaPandey, Parimal7

Source

https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/

Chapter 6

N Queen in O(n) space

N Queen in O(n) space - GeeksforGeeks

Given n, of a n x n chessboard, find the proper placement of queens on chessboard.

Previous Approach : N Queen

Algorithm :

```
Place(k, i)
// Returns true if a queen can be placed
// in kth row and ith column. Otherwise it
// returns false. X[] is a global array
// whose first (k-1) values have been set.
// Abs( ) returns absolute value of r
{
   for j := 1 to k-1 do

        // Two in the same column
        // or in the same diagonal
        if ((x[j] == i)  or
            (abs(x[j] - i) = Abs(j - k)))
           then return false;

   return true;
}
```

Algorithm nQueens(k, n) :

```
// Using backtracking, this procedure prints all
// possible placements of n queens on an n×n
```

```
// chessboard so that they are nonattacking.
{
      for i:= 1 to n do
      {
         if Place(k, i) then
         {
             x[k] = i;
             if (k == n)
                write (x[1:n]);
             else
               NQueens(k+1, n);
         }
      }
}
```

C++

```
 // CPP code to for n Queen placement
#include <bits/stdc++.h>

#define breakLine cout << "\n---------------------------------\n";
#define MAX 10

using namespace std;

int arr[MAX], no;

void nQueens(int k, int n);
bool canPlace(int k, int i);
void display(int n);

// Function to check queens placement
void nQueens(int k, int n){

    for (int i = 1;i <= n;i++){
        if (canPlace(k, i)){
            arr[k] = i;
            if (k == n)
                display(n);
            else
                nQueens(k + 1, n);
        }
    }
}

// Helper Function to check if queen can be placed
bool canPlace(int k, int i){
    for (int j = 1;j <= k - 1;j++){
```

```
        if (arr[j] == i ||
            (abs(arr[j] - i) == abs(j - k)))
           return false;
    }
    return true;
}

// Function to display placed queen
void display(int n){
    breakLine
    cout << "Arrangement No. " << ++no;
    breakLine

    for (int i = 1; i <= n; i++){
        for (int j = 1; j <= n; j++){
            if (arr[i] != j)
                cout << "\t_";
            else
                cout << "\tQ";
        }
        cout << endl;
    }

    breakLine
}

// Driver Code
int main(){
    int n = 4;
    nQueens(1, n);
    return 0;
}
```

Output:

```

---------------------------------
Arrangement No. 1
---------------------------------
    _    Q    _    _
    _    _    _    Q
    Q    _    _    _
    _    _    Q    _

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

---------------------------------
Arrangement No. 2
```

```
---------------------------------
   _    _    Q    _
   Q    _    _    _
   _    _    _    Q
   _    Q    _    _

---------------------------------
```

Source

https://www.geeksforgeeks.org/n-queen-in-on-space/

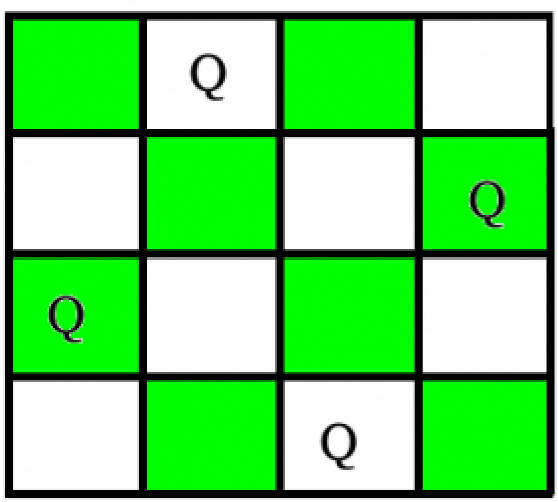
Chapter 7

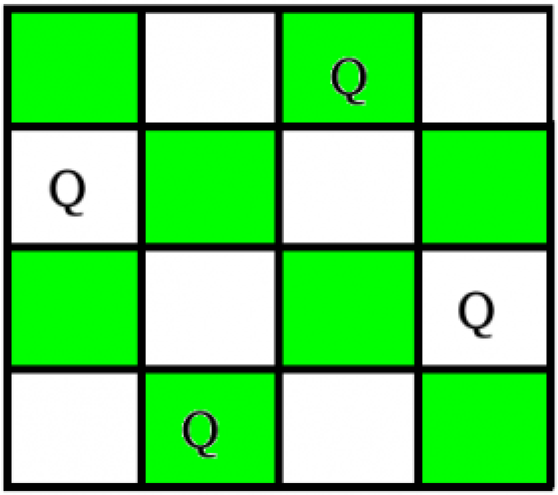
Printing all solutions in N-Queen Problem

Printing all solutions in N-Queen Problem - GeeksforGeeks

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following are two solutions for 4 Queen problem.





In previous post, we have discussed an approach that prints only one possible solution, so now in this post the task is to print all solutions in N-Queen Problem. The solution discussed here is an extension of same approach.

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.
```

There is only a slight modification in above algorithm that is highlighted in the code.

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

```
backtracking */
#include<bits/stdc++.h>
#define N 4

/* A utility function to print solution */
void printSolution(int board[N][N])
{
    static int k = 1;
    printf("%d-\n",k++);
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
    printf("\n");
}

/* 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(int board[N][N], int row, int col)
{
    int i, j;

    /* 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(int board[N][N], int col)
{
    /* base case: If all queens are placed
    then return true */
    if (col == N)
    {
        printSolution(board);
        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.*/
void solveNQ()
{
```

```
    int board[N][N];
    memset(board, 0, sizeof(board));

    if (solveNQUtil(board, 0) == false)
    {
        printf("Solution does not exist");
        return ;
    }

    return ;
}

// driver program to test above function
int main()
{
    solveNQ();
    return 0;
}
```

Output:

```
1-
 0  0  1  0
 1  0  0  0
 0  0  0  1
 0  1  0  0

2-
 0  1  0  0
 0  0  0  1
 1  0  0  0
 0  0  1  0
```

Source

https://www.geeksforgeeks.org/printing-solutions-n-queen-problem/

Chapter 8

8 queen problem

8 queen problem - GeeksforGeeks

The eight queens problem is the problem of placing eight queens on an 8×8 chessboard such that none of them attack one another (no two are in the same row, column, or diagonal). More generally, the n queens problem places n queens on an n×n chessboard.

There are different solutions for the problem.

Backtracking | Set 3 (N Queen Problem)

Branch and Bound | Set 5 (N Queen Problem)

You can find detailed solutions at http://en.literateprograms.org/Eight_queens_puzzle_(C)

Source

https://www.geeksforgeeks.org/8-queen-problem/

Chapter 9

Subset Sum

Subset Sum | Backtracking-4 - GeeksforGeeks

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

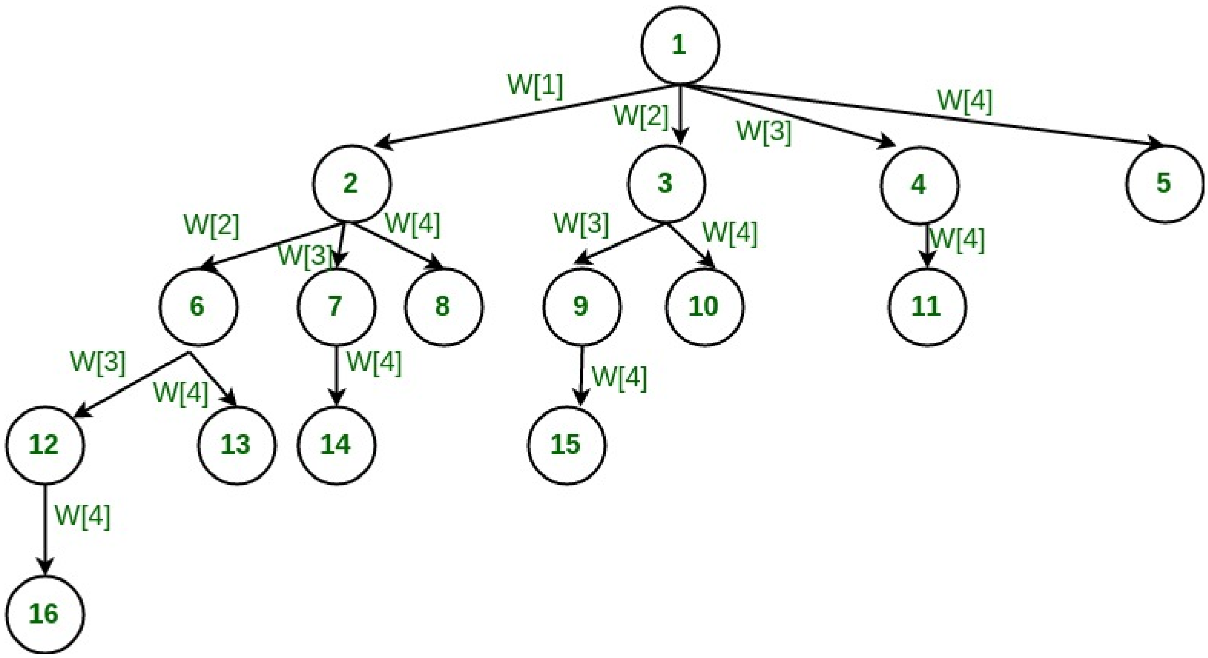
Exhaustive Search Algorithm for Subset Sum

One way to find subsets that sum to K is to consider all possible subsets. A power set contains all those subsets generated from a given set. The size of such a power set is 2^N.

Backtracking Algorithm for Subset Sum

Using exhaustive search we consider all subsets irrespective of whether they satisfy given constraints or not. Backtracking can be used to make a systematic consideration of the elements to be selected.

Assume given set of 4 elements, say **w[1] … w[4]**. Tree diagrams can be used to design backtracking algorithms. The following tree diagram depicts approach of generating variable sized tuple.



In the above tree, a node represents function call and a branch represents candidate element. The root node contains 4 children. In other words, root considers every element of the set as different branch. The next level sub-trees correspond to the subsets that includes the parent node. The branches at each level represent tuple element to be considered. For example, if we are at level 1, tuple_vector[1] can take any value of four branches generated. If we are at level 2 of left most node, tuple_vector[2] can take any value of three branches generated, and so on…

For example the left most child of root generates all those subsets that include w[1]. Similarly the second child of root generates all those subsets that includes w[2] and excludes w[1].

As we go down along depth of tree we add elements so far, and if the added sum is satisfying explicit constraints, we will continue to generate child nodes further. Whenever the constraints are not met, we stop further generation of sub-trees of that node, and backtrack to previous node to explore the nodes not yet explored. In many scenarios, it saves considerable amount of processing time.

The tree should trigger a clue to implement the backtracking algorithm (try yourself). It prints all those subsets whose sum add up to given number. We need to explore the nodes along the breadth and depth of the tree. Generating nodes along breadth is controlled by loop and nodes along the depth are generated using recursion (post order traversal). Pseudo code given below,

```
if(subset is satisfying the constraint)
    print the subset
    exclude the current element and consider next element
else
    generate the nodes of present level along breadth of tree and
    recur for next levels
```

Following is C implementation of subset sum using variable size tuple vector. Note that the following program explores all possibilities similar to exhaustive search. It is to demonstrate how backtracking can be used. See next code to verify, how we can optimize the backtracking solution.

```
 #include <stdio.h>
#include <stdlib.h>

#define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))

static int total_nodes;
// prints subset found
void printSubset(int A[], int size)
{
    for(int i = 0; i < size; i++)
    {
        printf("%*d", 5, A[i]);
    }

    printf("n");
}

// inputs
// s            - set vector
// t            - tuplet vector
// s_size       - set size
// t_size       - tuplet size so far
// sum          - sum so far
// ite          - nodes count
// target_sum   - sum to be found
void subset_sum(int s[], int t[],
                int s_size, int t_size,
                int sum, int ite,
                int const target_sum)
{
    total_nodes++;
    if( target_sum == sum )
    {
        // We found subset
        printSubset(t, t_size);
        // Exclude previously added item and consider next candidate
        subset_sum(s, t, s_size, t_size-1, sum - s[ite], ite + 1, target_sum);
        return;
    }
    else
    {
        // generate nodes along the breadth
        for( int i = ite; i < s_size; i++ )
```

```
        {
            t[t_size] = s[i];
            // consider next level node (along depth)
            subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1, target_sum);
        }
    }
}

// Wrapper to print subsets that sum to target_sum
// input is weights vector and target_sum
void generateSubsets(int s[], int size, int target_sum)
{
    int *tuplet_vector = (int *)malloc(size * sizeof(int));

    subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);

    free(tuplet_vector);
}

int main()
{
    int weights[] = {10, 7, 5, 18, 12, 20, 15};
    int size = ARRAYSIZE(weights);

    generateSubsets(weights, size, 35);
    printf("Nodes generated %dn", total_nodes);
    return 0;
}
```

The power of backtracking appears when we combine explicit and implicit constraints, and we stop generating nodes when these checks fail. We can improve the above algorithm by strengthening the constraint checks and presorting the data. By sorting the initial array, we need not to consider rest of the array, once the sum so far is greater than target number. We can backtrack and check other possibilities.

Similarly, assume the array is presorted and we found one subset. We can generate next node excluding the present node only when inclusion of next node satisfies the constraints. Given below is optimized implementation (it prunes the subtree if it is not satisfying contraints).

```
 #include <stdio.h>
#include <stdlib.h>

#define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))

static int total_nodes;

// prints subset found
void printSubset(int A[], int size)
```

```
{
    for(int i = 0; i < size; i++)
    {
        printf("%*d", 5, A[i]);
    }

    printf("n");
}

// qsort compare function
int comparator(const void *pLhs, const void *pRhs)
{
    int *lhs = (int *)pLhs;
    int *rhs = (int *)pRhs;

    return *lhs > *rhs;
}

// inputs
// s            - set vector
// t            - tuplet vector
// s_size       - set size
// t_size       - tuplet size so far
// sum          - sum so far
// ite          - nodes count
// target_sum   - sum to be found
void subset_sum(int s[], int t[],
                int s_size, int t_size,
                int sum, int ite,
                int const target_sum)
{
    total_nodes++;

    if( target_sum == sum )
    {
        // We found sum
        printSubset(t, t_size);

        // constraint check
        if( ite + 1 < s_size && sum - s[ite] + s[ite+1] <= target_sum )
        {
            // Exclude previous added item and consider next candidate
            subset_sum(s, t, s_size, t_size-1, sum - s[ite], ite + 1, target_sum);
        }
        return;
    }
    else
    {
```

```
        // constraint check
        if( ite < s_size && sum + s[ite] <= target_sum )
        {
            // generate nodes along the breadth
            for( int i = ite; i < s_size; i++ )
            {
                t[t_size] = s[i];

                if( sum + s[i] <= target_sum )
                {
                    // consider next level node (along depth)
                    subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1, target_sum);
                }
            }
        }
    }
}

// Wrapper that prints subsets that sum to target_sum
void generateSubsets(int s[], int size, int target_sum)
{
    int *tuplet_vector = (int *)malloc(size * sizeof(int));

    int total = 0;

    // sort the set
    qsort(s, size, sizeof(int), &comparator);

    for( int i = 0; i < size; i++ )
    {
        total += s[i];
    }

    if( s[0] <= target_sum && total >= target_sum )
    {

        subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);

    }

    free(tuplet_vector);
}

int main()
{
    int weights[] = {15, 22, 14, 26, 32, 9, 16, 8};
    int target = 53;
```

```
    int size = ARRAYSIZE(weights);

    generateSubsets(weights, size, target);

    printf("Nodes generated %dn", total_nodes);

    return 0;
}
```

As another approach, we can generate the tree in fixed size tuple analogs to binary pattern. We will kill the sub-trees when the constraints are not satisfied.

– – – **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/subset-sum-backtracking-4/

Chapter 10

m Coloring Problem

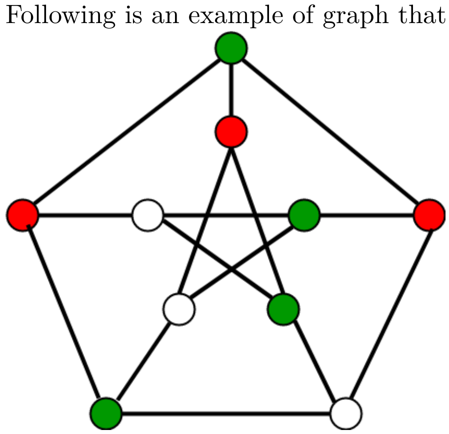
m Coloring Problem | Backtracking-5 - GeeksforGeeks

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

Input:
1) A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.
2) An integer m which is maximum number of colors that can be used.

Output:
An array color[V] that should have numbers from 1 to m. color[i] should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

Following is an example of graph that can be colored with 3 different colors.



Naive Algorithm
Generate all possible configurations of colors and print a configuration that satisfies the given constraints.

```
while there are untried conflagrations
{
   generate the next configuration
   if no adjacent vertices are colored with same color
   {
      print this configuration;
   }
}
```

There will be V^m configurations of colors.

Backtracking Algorithm
The idea is to assign colors one by one to different vertices, starting from the vertex 0. Before assigning a color, we check for safety by considering already assigned colors to the adjacent vertices. If we find a color assignment which is safe, we mark the color assignment as part of solution. If we do not a find color due to clashes then we backtrack and return false.

Implementation of Backtracking solution

C/C++

```
 #include<stdio.h>

// 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 = new int[V];
    for (int i = 0; i < V; i++)
       color[i] = 0;

    // Call graphColoringUtil() for vertex 0
    if (graphColoringUtil(graph, m, color, 0) == false)
    {
      printf("Solution does not exist");
      return false;
    }

    // Print the solution
    printSolution(color);
```

```
    return true;
}

/* A utility function to print solution */
void printSolution(int color[])
{
    printf("Solution Exists:"
            " Following are the assigned colors \n");
    for (int i = 0; i < V; i++)
      printf(" %d ", color[i]);
    printf("\n");
}

// driver program to test above function
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
    graphColoring (graph, m);
    return 0;
}
```

Java

```
 /* Java program for solution of M Coloring problem
   using backtracking */
public class mColoringProblem {
    final int V = 4;
    int color[];

    /* A utility function to check if the current
       color assignment is safe for vertex v */
    boolean isSafe(int v, int graph[][], int color[],
                   int c)
    {
        for (int i = 0; i < V; i++)
            if (graph[v][i] == 1 && c == color[i])
```

```
            return false;
    return true;
}

/* A recursive utility function to solve m
   coloring  problem */
boolean graphColoringUtil(int graph[][], 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))
                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.*/
boolean graphColoring(int graph[][], int m)
{
    // Initialize all color values as 0. This
    // initialization is needed correct functioning
    // of isSafe()
    color = new int[V];
    for (int i = 0; i < V; i++)
        color[i] = 0;

    // Call graphColoringUtil() for vertex 0
    if (!graphColoringUtil(graph, m, color, 0))
    {
        System.out.println("Solution does not exist");
        return false;
    }

    // Print the solution
    printSolution(color);
    return true;
}

/* A utility function to print solution */
void printSolution(int color[])
{
    System.out.println("Solution Exists: Following" +
                       " are the assigned colors");
    for (int i = 0; i < V; i++)
        System.out.print(" " + color[i] + " ");
    System.out.println();
}

// driver program to test above function
public static void main(String args[])
{
    mColoringProblem Coloring = new mColoringProblem();
    /* Create following graph and test whether it is
       3 colorable
      (3)---(2)
       |   / |
       |  /  |
       | /   |
      (0)---(1)
    */
    int graph[][] = {{0, 1, 1, 1},
        {1, 0, 1, 0},
        {1, 1, 0, 1},
        {1, 0, 1, 0},
```

```
        };
        int m = 3; // Number of colors
        Coloring.graphColoring(graph, m);
    }
}
// This code is contributed by Abhishek Shankhadhar
```

Python

```
 # Python program for solution of M Coloring
# problem using backtracking

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]\
                              for row in range(vertices)]

    # A utility function to check if the current color assignment
    # is safe for vertex v
    def isSafe(self, v, colour, c):
        for i in range(self.V):
            if self.graph[v][i] == 1 and colour[i] == c:
                return False
        return True

    # A recursive utility function to solve m
    # coloring  problem
    def graphColourUtil(self, m, colour, v):
        if v == self.V:
            return True

        for c in range(1, m+1):
            if self.isSafe(v, colour, c) == True:
                colour[v] = c
                if self.graphColourUtil(m, colour, v+1) == True:
                    return True
                colour[v] = 0

    def graphColouring(self, m):
        colour = [0] * self.V
        if self.graphColourUtil(m, colour, 0) == False:
            return False

        # Print the solution
        print "Solution exist and Following are the assigned colours:"
        for c in colour:
```

```
            print c,
        return True

# Driver Code
g  = Graph(4)
g.graph = [[0,1,1,1], [1,0,1,0], [1,1,0,1], [1,0,1,0]]
m=3
g.graphColouring(m)

# This code is contributed by Divyanshu Mehta
```

Output:

```
Solution Exists: Following are the assigned colors
 1  2  3  2
```

References:
http://en.wikipedia.org/wiki/Graph_coloring

Improved By : SarathChandra1

Source

https://www.geeksforgeeks.org/m-coloring-problem-backtracking-5/

Chapter 11

Hamiltonian Cycle

Hamiltonian Cycle | Backtracking-6 - GeeksforGeeks

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path. Following are the input and output of the required function.

Input:
A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

Output:
An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

For example, a Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}. There are more Hamiltonian Cycles in the graph like {0, 3, 4, 2, 1, 0}

```
(0)--(1)--(2)
 |   / \   |
 |  /   \  |
 | /     \ |
(3)-------(4)
```

And the following graph doesn't contain any Hamiltonian Cycle.

```
(0)--(1)--(2)
 |   / \   |
```

```
 |  /   \  |
 | /     \ |
(3)      (4)
```

Naive Algorithm
Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be n! (n factorial) configurations.

```
while there are untried conflagrations
{
   generate the next configuration
   if ( there are edges between two consecutive vertices of this
      configuration and there is an edge from the last vertex to
      the first ).
   {
      print this configuration;
      break;
   }
}
```

Backtracking Algorithm
Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

Implementation of Backtracking solution
Following are implementations of the Backtracking solution.

C/C++

```
 /* C/C++ program for solution of Hamiltonian Cycle problem
   using backtracking */
#include<stdio.h>

// Number of vertices in the graph
#define V 5

void printSolution(int path[]);

/* A utility function to check if the vertex v can be added at
   index 'pos' in the Hamiltonian Cycle constructed so far (stored
   in 'path[]') */
bool isSafe(int v, bool graph[V][V], int path[], int pos)
{
    /* Check if this vertex is an adjacent vertex of the previously
       added vertex. */
```

```
    if (graph [ path[pos-1] ][ v ] == 0)
        return false;

    /* Check if the vertex has already been included.
      This step can be optimized by creating an array of size V */
    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;

    return true;
}

/* A recursive utility function to solve hamiltonian cycle problem */
bool hamCycleUtil(bool graph[V][V], int path[], int pos)
{
    /* base case: If all vertices are included in Hamiltonian Cycle */
    if (pos == V)
    {
        // And if there is an edge from the last included vertex to the
        // first vertex
        if ( graph[ path[pos-1] ][ path[0] ] == 1 )
           return true;
        else
          return false;
    }

    // Try different vertices as a next candidate in Hamiltonian Cycle.
    // We don't try for 0 as we included 0 as starting point in in hamCycle()
    for (int v = 1; v < V; v++)
    {
        /* Check if this vertex can be added to Hamiltonian Cycle */
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;

            /* recur to construct rest of the path */
            if (hamCycleUtil (graph, path, pos+1) == true)
                return true;

            /* If adding vertex v doesn't lead to a solution,
               then remove it */
            path[pos] = -1;
        }
    }

    /* If no vertex can be added to Hamiltonian Cycle constructed so far,
       then return false */
    return false;
```

```
}

/* This function solves the Hamiltonian Cycle problem using Backtracking.
  It mainly uses hamCycleUtil() to solve the problem. It returns false
  if there is no Hamiltonian Cycle possible, otherwise return true and
  prints the path. Please note that there may be more than one solutions,
  this function prints one of the feasible solutions. */
bool hamCycle(bool graph[V][V])
{
    int *path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    /* Let us put vertex 0 as the first vertex in the path. If there is
       a Hamiltonian Cycle, then the path can be started from any point
       of the cycle as the graph is undirected */
    path[0] = 0;
    if ( hamCycleUtil(graph, path, 1) == false )
    {
        printf("\nSolution does not exist");
        return false;
    }

    printSolution(path);
    return true;
}

/* A utility function to print solution */
void printSolution(int path[])
{
    printf ("Solution Exists:"
            " Following is one Hamiltonian Cycle \n");
    for (int i = 0; i < V; i++)
        printf(" %d ", path[i]);

    // Let us print the first vertex again to show the complete cycle
    printf(" %d ", path[0]);
    printf("\n");
}

// driver program to test above function
int main()
{
   /* Let us create the following graph
      (0)--(1)--(2)
       |   / \   |
       |  /   \  |
       | /     \ |
```

```
      (3)-------(4)    */
   bool graph1[V][V] = {{0, 1, 0, 1, 0},
                       {1, 0, 1, 1, 1},
                       {0, 1, 0, 0, 1},
                       {1, 1, 0, 0, 1},
                       {0, 1, 1, 1, 0},
                      };

    // Print the solution
    hamCycle(graph1);

   /* Let us create the following graph
      (0)--(1)--(2)
       |   / \   |
       |  /   \  |
       | /     \ |
      (3)       (4)    */
    bool graph2[V][V] = {{0, 1, 0, 1, 0},
                       {1, 0, 1, 1, 1},
                       {0, 1, 0, 0, 1},
                       {1, 1, 0, 0, 0},
                       {0, 1, 1, 0, 0},
                      };

    // Print the solution
    hamCycle(graph2);

    return 0;
}
```

Java

```
 /* Java program for solution of Hamiltonian Cycle problem
   using backtracking */
class HamiltonianCycle
{
    final int V = 5;
    int path[];

    /* A utility function to check if the vertex v can be
       added at index 'pos'in the Hamiltonian Cycle
       constructed so far (stored in 'path[]') */
    boolean isSafe(int v, int graph[][], int path[], int pos)
    {
        /* Check if this vertex is an adjacent vertex of
           the previously added vertex. */
        if (graph[path[pos - 1]][v] == 0)
            return false;
```

```
    /* Check if the vertex has already been included.
       This step can be optimized by creating an array
       of size V */
    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;

    return true;
}

/* A recursive utility function to solve hamiltonian
   cycle problem */
boolean hamCycleUtil(int graph[][], int path[], int pos)
{
    /* base case: If all vertices are included in
       Hamiltonian Cycle */
    if (pos == V)
    {
        // And if there is an edge from the last included
        // vertex to the first vertex
        if (graph[path[pos - 1]][path[0]] == 1)
            return true;
        else
            return false;
    }

    // Try different vertices as a next candidate in
    // Hamiltonian Cycle. We don't try for 0 as we
    // included 0 as starting point in in hamCycle()
    for (int v = 1; v < V; v++)
    {
        /* Check if this vertex can be added to Hamiltonian
           Cycle */
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;

            /* recur to construct rest of the path */
            if (hamCycleUtil(graph, path, pos + 1) == true)
                return true;

            /* If adding vertex v doesn't lead to a solution,
               then remove it */
            path[pos] = -1;
        }
    }
```

```
    /* If no vertex can be added to Hamiltonian Cycle
       constructed so far, then return false */
    return false;
}

/* This function solves the Hamiltonian Cycle problem using
   Backtracking. It mainly uses hamCycleUtil() to solve the
   problem. It returns false if there is no Hamiltonian Cycle
   possible, otherwise return true and prints the path.
   Please note that there may be more than one solutions,
   this function prints one of the feasible solutions. */
int hamCycle(int graph[][])
{
    path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    /* Let us put vertex 0 as the first vertex in the path.
       If there is a Hamiltonian Cycle, then the path can be
       started from any point of the cycle as the graph is
       undirected */
    path[0] = 0;
    if (hamCycleUtil(graph, path, 1) == false)
    {
        System.out.println("\nSolution does not exist");
        return 0;
    }

    printSolution(path);
    return 1;
}

/* A utility function to print solution */
void printSolution(int path[])
{
    System.out.println("Solution Exists: Following" +
                       " is one Hamiltonian Cycle");
    for (int i = 0; i < V; i++)
        System.out.print(" " + path[i] + " ");

    // Let us print the first vertex again to show the
    // complete cycle
    System.out.println(" " + path[0] + " ");
}

// driver program to test above function
public static void main(String args[])
{
```

```
        HamiltonianCycle hamiltonian =
                                new HamiltonianCycle();
        /* Let us create the following graph
           (0)--(1)--(2)
            |   / \   |
            |  /   \  |
            | /     \ |
           (3)-------(4)    */
        int graph1[][] = {{0, 1, 0, 1, 0},
            {1, 0, 1, 1, 1},
            {0, 1, 0, 0, 1},
            {1, 1, 0, 0, 1},
            {0, 1, 1, 1, 0},
        };

        // Print the solution
        hamiltonian.hamCycle(graph1);

        /* Let us create the following graph
           (0)--(1)--(2)
            |   / \   |
            |  /   \  |
            | /     \ |
           (3)       (4)    */
        int graph2[][] = {{0, 1, 0, 1, 0},
            {1, 0, 1, 1, 1},
            {0, 1, 0, 0, 1},
            {1, 1, 0, 0, 0},
            {0, 1, 1, 0, 0},
        };

        // Print the solution
        hamiltonian.hamCycle(graph2);
    }
}
// This code is contributed by Abhishek Shankhadhar
```

Python

```
 # Python program for solution of
# hamiltonian cycle problem

class Graph():
    def __init__(self, vertices):
        self.graph = [[0 for column in range(vertices)]\
                            for row in range(vertices)]
        self.V = vertices
```

```
''' Check if this vertex is an adjacent vertex
    of the previously added vertex and is not
    included in the path earlier '''
def isSafe(self, v, pos, path):
    # Check if current vertex and last vertex
    # in path are adjacent
    if self.graph[ path[pos-1] ][v] == 0:
        return False

    # Check if current vertex not already in path
    for vertex in path:
        if vertex == v:
            return False

    return True

# A recursive utility function to solve
# hamiltonian cycle problem
def hamCycleUtil(self, path, pos):

    # base case: if all vertices are
    # included in the path
    if pos == self.V:
        # Last vertex must be adjacent to the
        # first vertex in path to make a cyle
        if self.graph[ path[pos-1] ][ path[0] ] == 1:
            return True
        else:
            return False

    # Try different vertices as a next candidate
    # in Hamiltonian Cycle. We don't try for 0 as
    # we included 0 as starting point in in hamCycle()
    for v in range(1,self.V):

        if self.isSafe(v, pos, path) == True:

            path[pos] = v

            if self.hamCycleUtil(path, pos+1) == True:
                return True

            # Remove current vertex if it doesn't
            # lead to a solution
            path[pos] = -1

    return False
```

```
    def hamCycle(self):
        path = [-1] * self.V

        ''' Let us put vertex 0 as the first vertex
            in the path. If there is a Hamiltonian Cycle,
            then the path can be started from any point
            of the cycle as the graph is undirected '''
        path[0] = 0

        if self.hamCycleUtil(path,1) == False:
            print "Solution does not exist\n"
            return False

        self.printSolution(path)
        return True

    def printSolution(self, path):
        print "Solution Exists: Following is one Hamiltonian Cycle"
        for vertex in path:
            print vertex,
        print path[0], "\n"

# Driver Code

''' Let us create the following graph
      (0)--(1)--(2)
       |   / \   |
       |  /   \  |
       | /     \ |
      (3)-------(4)    '''
g1 = Graph(5)
g1.graph = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],
             [0, 1, 0, 0, 1,],[1, 1, 0, 0, 1],
             [0, 1, 1, 1, 0], ]

# Print the solution
g1.hamCycle();

''' Let us create the following graph
      (0)--(1)--(2)
       |   / \   |
       |  /   \  |
       | /     \ |
      (3)       (4)    '''
g2 = Graph(5)
g2.graph = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],
           [0, 1, 0, 0, 1,], [1, 1, 0, 0, 0],
           [0, 1, 1, 0, 0], ]
```

```
# Print the solution
g2.hamCycle();

# This code is contributed by Divyanshu Mehta
```

Output:

```
Solution Exists: Following is one Hamiltonian Cycle
 0  1  2  4  3  0

Solution does not exist
```

Note that the above code always prints cycle starting from 0. Starting point should not matter as cycle can be started from any point. If you want to change the starting point, you should make two changes to above code.
Change "path[0] = 0;" to "path[0] = s;" where s is your new starting point. Also change loop "for (int v = 1; v < V; v++)" in hamCycleUtil() to "for (int v = 0; v < V; v++)".
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/hamiltonian-cycle-backtracking-6/

Chapter 12

Sudoku

Sudoku | Backtracking-7 - GeeksforGeeks

Given a partially filled 9×9 2D array 'grid[9][9]', the goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and subgrid of size 3×3 contains exactly one instance of the digits from 1 to 9.

3		6	5		8	4		
5	2							
	8	7					3	1
		3		1			8	
9			8	6	3			5
	5			9		6		
1	3					2	5	
							7	4
		5	2		6	3		

Naive Algorithm
The Naive Algorithm 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.

Backtracking Algorithm
Like all other Backtracking problems, we can solve Sudoku by one by one assigning numbers to empty cells. Before assigning a number, we check whether it is safe to assign. We basically check that the same number is not present in the current row, current column and current 3X3 subgrid. After checking for safety, we 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

we try next number for the current empty cell. And if none of the number (1 to 9) leads to a solution, we return false.

```
  Find row, col of an unassigned cell
  If there is none, return true
  For digits from 1 to 9
    a) If there is no conflict for digit at row, col
        assign digit to row, col and recursively try fill in rest of grid
    b) If recursion successful, return true
    c) Else, remove digit and try another
  If all digits have been tried and nothing worked, return false
```

Following are C++ and Python implementation for Sudoku problem. It prints the completely filled grid as output.

C/C++

```
 // A Backtracking program  in C++ to solve Sudoku problem
#include <stdio.h>

// 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))
       return true; // success!

    // consider digits 1 to 9
    for (int num = 1; num <= 9; num++)
    {
```

```
        // if looks promising
        if (isSafe(grid, row, col, num))
        {
            // make tentative assignment
            grid[row][col] = num;

            // return, if success, yay!
            if (SolveSudoku(grid))
                return true;

            // failure, unmake & try again
            grid[row][col] = UNASSIGNED;
        }
    }
    return false; // this triggers backtracking
}

/* 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);
}

/* 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++)
             printf("%2d", grid[row][col]);
        printf("\n");
    }
}

/* Driver Program to test above functions */
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
         printf("No solution exists");

    return 0;
}
```

Python

```
 # A Backtracking program  in Pyhton to solve Sudoku problem


# A Utility Function to print the Grid
def print_grid(arr):
    for i in range(9):
        for j in range(9):
            print arr[i][j],
        print ('n')


# Function to Find the entry in the Grid that is still  not used
# 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.
# 'l' is a list  variable that has been passed from the solve_sudoku function
# to keep track of incrementation of Rows and Columns
def find_empty_location(arr,l):
    for row in range(9):
        for col in range(9):
            if(arr[row][col]==0):
                l[0]=row
                l[1]=col
                return True
    return False

# Returns a boolean which indicates whether any assigned entry
# in the specified row matches the given number.
def used_in_row(arr,row,num):
    for i in range(9):
        if(arr[row][i] == num):
            return True
    return False

# Returns a boolean which indicates whether any assigned entry
# in the specified column matches the given number.
def used_in_col(arr,col,num):
```

```
    for i in range(9):
        if(arr[i][col] == num):
            return True
    return False

# Returns a boolean which indicates whether any assigned entry
# within the specified 3x3 box matches the given number
def used_in_box(arr,row,col,num):
    for i in range(3):
        for j in range(3):
            if(arr[i+row][j+col] == num):
                return True
    return False

# Checks whether it will be legal to assign num to the given row,col
#  Returns a boolean which indicates whether it will be legal to assign
#  num to the given row,col location.
def check_location_is_safe(arr,row,col,num):

    # Check if 'num' is not already placed in current row,
    # current column and current 3x3 box
    return not used_in_row(arr,row,num) and not used_in_col(arr,col,num) and not used_in_box(arr

# 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)
def solve_sudoku(arr):

    # 'l' is a list variable that keeps the record of row and col in find_empty_location Functio

    l=[0,0]

    # If there is no unassigned location, we are done
    if(not find_empty_location(arr,l)):
        return True

    # Assigning list values to row and col that we got from the above Function
    row=l[0]
    col=l[1]

    # consider digits 1 to 9
    for num in range(1,10):

        # if looks promising
        if(check_location_is_safe(arr,row,col,num)):

            # make tentative assignment
            arr[row][col]=num
```

```
            # return, if sucess, ya!
            if(solve_sudoku(arr)):
                return True

            # failure, unmake & try again
            arr[row][col] = 0

    # this triggers backtracking
    return False

# Driver main function to test above functions
if __name__=="__main__":

    # creating a 2D array for the grid
    grid=[[0 for x in range(9)]for y in range(9)]

    # assigning values to the grid
    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]]

    # if sucess print the grid
    if(solve_sudoku(grid)):
        print_grid(grid)
    else:
        print "No solution exists"

# The above code has been contributed by Harshit Sidhwa.
```

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
```

References:
http://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf

Source

https://www.geeksforgeeks.org/sudoku-backtracking-7/

Chapter 13

Solving Cryptarithmetic Puzzles

Solving Cryptarithmetic Puzzles | Backtracking-8 - GeeksforGeeks

Newspapers and magazines often have crypt-arithmetic puzzles of the form:

```
  SEND
+ MORE
--------
 MONEY
--------
```

The goal here is to assign each letter a digit from 0 to 9 so that the arithmetic works out correctly. The rules are that all occurrences of a letter must be assigned the same digit, and no digit can be assigned to more than one letter.

- First, create a list of all the characters that need assigning to pass to Solve
- If all characters are assigned, return true if puzzle is solved, false otherwise
- Otherwise, consider the first unassigned character
- for (every possible choice among the digits not in use)
- If all digits have been tried and nothing worked, return false to trigger backtracking

```
 /* ExhaustiveSolve
* ---------------
* This is the "not-very-smart" version of cryptarithmetic solver. It takes
* the puzzle itself (with the 3 strings for the two addends and sum) and a
* string of letters as yet unassigned. If no more letters to assign
* then we've hit a base-case, if the current letter-to-digit mapping solves
* the puzzle, we're done, otherwise we return false to trigger backtracking
* If we have letters to assign, we take the first letter from that list, and
* try assigning it the digits from 0 to 9 and then recursively working
* through solving puzzle from here. If we manage to make a good assignment
```

```
* that works, we've succeeded, else we need to unassign that choice and try
* another digit. This version is easy to write, since it uses a simple
* approach (quite similar to permutations if you think about it) but it is
* not so smart because it doesn't take into account the structure of the
* puzzle constraints (for example, once the two digits for the addends have
* been assigned, there is no reason to try anything other than the correct
* digit for the sum) yet it tries a lot of useless combos regardless
*/
bool ExhaustiveSolve(puzzleT puzzle, string lettersToAssign)
{
    if (lettersToAssign.empty()) // no more choices to make
        return PuzzleSolved(puzzle); // checks arithmetic to see if works
    for (int digit = 0; digit <= 9; digit++)   // try all digits
    {
        if (AssignLetterToDigit(lettersToAssign[0], digit))
        {
            if (ExhaustiveSolve(puzzle, lettersToAssign.substr(1)))
                return true;
            UnassignLetterFromDigit(lettersToAssign[0], digit);
        }
    }
    return false;  // nothing worked, need to backtrack
}
```

The algorithm above actually has a lot in common with the permutations algorithm, it pretty much just creates all arrangements of the mapping from characters to digits and tries each until one works or all have been successfully tried. For a large puzzle, this could take a while.
A smarter algorithm could take into account the structure of the puzzle and avoid going down dead-end paths. For example, if we assign the characters starting from the ones place and moving to the left, at each stage, we can verify the correctness of what we have so far before we continue onwards. This definitely complicates the code but leads to a tremendous improvement in efficiency, making it much more feasible to solve large puzzles.

Below pseudocode in this case has more special cases, but the same general design

- Start by examining the rightmost digit of the topmost row, with a carry of 0
- If we are beyond the leftmost digit of the puzzle, return true if no carry, false otherwise
- If we are currently trying to assign a char in one of the addends
  If char already assigned, just recur on row beneath this one, adding value into sum
  If not assigned, then
  - for (every possible choice among the digits not in use)
    make that choice and then on row beneath this one, if successful, return true
    if !successful, unmake assignment and try another digit
  - return false if no assignment worked to trigger backtracking
- Else if trying to assign a char in the sum
- If char assigned & matches correct,
  recur on next column to the left with carry, if success return true,

- If char assigned & doesn't match, return false
- If char unassigned & correct digit already used, return false
- If char unassigned & correct digit unused,
  assign it and recur on next column to left with carry, if success return true
- return false to trigger backtracking

Source:
http://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf

Source

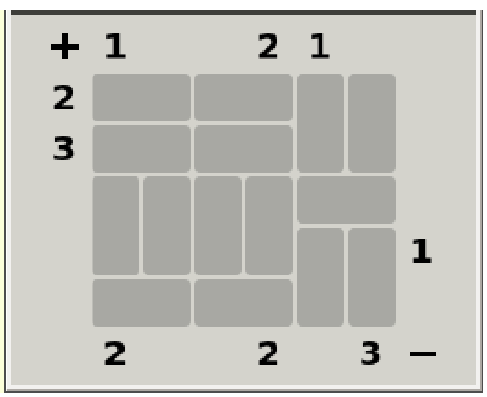
https://www.geeksforgeeks.org/solving-cryptarithmetic-puzzles-backtracking-8/

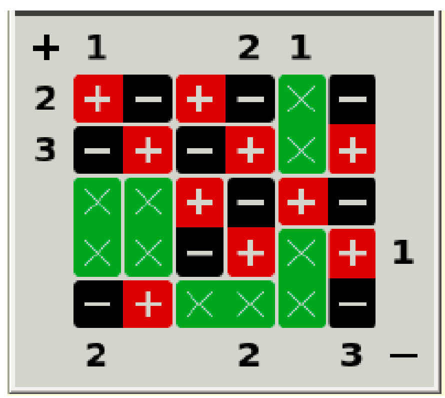
Chapter 14

Magnet Puzzle

Magnet Puzzle | Backtracking-9 - GeeksforGeeks

The puzzle game Magnets involves placing a set of domino-shaped magnets (or electrets or other polarized objects) in a subset of slots on a board so as to satisfy a set of constraints. For example, the puzzle on the left has the solution shown on the right:





Each slot contains either a blank entry (indicated by 'x's), or a "magnet" with a positive and negative end. The numbers along the left and top sides show the numbers of '+' squares in particular rows or columns. Those along the right and bottom show the number of '-' signs in particular rows or columns. Rows and columns without a number at one or both ends are unconstrained as to the number of '+' or '-' signs, depending on which number is not present. In addition to fulfilling these numerical constraints, a puzzle solution must also satisfy the constraint that no two orthogonally touching squares may have the same sign (diagonally joined squares are not constrained).

You are given top[], bottom[], left[], right[] arrays indicates the count of + or − along the top(+), bottom(-), left(+) and right(-) edges respectively. Values of -1 indicate any number of + and − signs. Also given matrix rules[][] contain any one T, B, L or R characters. For a vertical slot in the board, T indicates its top end and B indicates the bottom end. For a horizontal slot in the board, L indicates left end and R indicates the right end.

Examples:

```
Input : M = 5, N = 6
        top[] = { 1, -1, -1, 2, 1, -1 }
        bottom[] = { 2, -1, -1, 2, -1, 3 }
        left[] = { 2, 3, -1, -1, -1 }
        right[] = { -1, -1, -1, 1, -1 }
        rules[][] = { { L, R, L, R, T, T },
                      { L, R, L, R, B, B },
                      { T, T, T, T, L, R },
                      { B, B, B, B, T, T },
                      { L, R, L, R, B, B }};
Output : + - + - X -
         - + - + X +
         X X + - + -
         X X - + X +
         - + X X X -

Input : M = 4, N = 3
        top[] = { 2, -1, -1 }
        bottom[] = { -1, -1, 2 }
        left[] = { -1, -1, 2, -1 }
        right[] = { 0, -1, -1, -1 }
        rules[][] = { { T, T, T },
                      { B, B, B },
                      { T, L, R },
                      { B, L, R } };
Output : + X +
         - X -
        + - +
        - + -
```

We can solve this problem using Backtracking.

Source :https://people.eecs.berkeley.edu/~hilfingr/programming-contest/f2012-contest.pdf

Source

https://www.geeksforgeeks.org/magnet-puzzle-backtracking-9/

Chapter 15

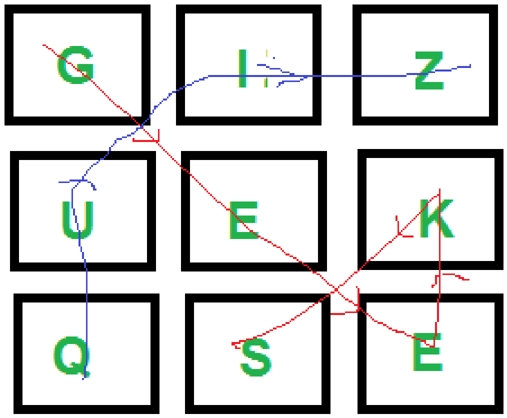
Boggle | Set 1 (Using DFS)

Boggle (Find all possible words in a board of characters) | Set 1 - GeeksforGeeks

Given a dictionary, a method to do lookup in dictionary and a M x N board where every cell has one character. Find all possible words that can be formed by a sequence of adjacent characters. Note that we can move to any of 8 adjacent characters, but a word should not have multiple instances of same cell.

Example:

```
Input: dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
       boggle[][]   = {{'G','I','Z'},
                       {'U','E','K'},
                       {'Q','S','E'}};
      isWord(str): returns true if str is present in dictionary
                   else false.

Output:  Following words of dictionary are present
         GEEKS
         QUIZ
```



The idea is to consider every character as a starting character and find all words starting with it. All words starting from a character can be found using Depth First Traversal. We do depth first traversal starting from every cell. We keep track of visited cells to make sure that a cell is considered only once in a word.

```
 // C++ program for Boggle game
#include<iostream>
#include<cstring>
using namespace std;

#define M 3
#define N 3

// Let the given dictionary be following
string dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
int n = sizeof(dictionary)/sizeof(dictionary[0]);

// A given function to check if a given string is present in
// dictionary. The implementation is naive for simplicity. As
// per the question dictionary is given to us.
bool isWord(string &str)
{
    // Linearly search all words
    for (int i=0; i<n; i++)
        if (str.compare(dictionary[i]) == 0)
          return true;
    return false;
}

// A recursive function to print all words present on boggle
void findWordsUtil(char boggle[M][N], bool visited[M][N], int i,
                   int j, string &str)
{
```

```
    // Mark current cell as visited and append current character
    // to str
    visited[i][j] = true;
    str = str + boggle[i][j];

    // If str is present in dictionary, then print it
    if (isWord(str))
        cout << str << endl;

    // Traverse 8 adjacent cells of boggle[i][j]
    for (int row=i-1; row<=i+1 && row<M; row++)
      for (int col=j-1; col<=j+1 && col<N; col++)
        if (row>=0 && col>=0 && !visited[row][col])
          findWordsUtil(boggle,visited, row, col, str);

    // Erase current character from string and mark visited
    // of current cell as false
    str.erase(str.length()-1);
    visited[i][j] = false;
}

// Prints all words present in dictionary.
void findWords(char boggle[M][N])
{
    // Mark all characters as not visited
    bool visited[M][N] = {{false}};

    // Initialize current string
    string str = "";

    // Consider every character and look for all words
    // starting with this character
    for (int i=0; i<M; i++)
       for (int j=0; j<N; j++)
             findWordsUtil(boggle, visited, i, j, str);
}

// Driver program to test above function
int main()
{
    char boggle[M][N] = {{'G','I','Z'},
                         {'U','E','K'},
                         {'Q','S','E'}};

    cout << "Following words of dictionary are present\n";
    findWords(boggle);
    return 0;
}
```

Output:

```
Following words of dictionary are present
GEEKS
QUIZ
```

Note that the above solution may print the same word multiple times. For example, if we add "SEEK" to the dictionary, it is printed multiple times. To avoid this, we can use hashing to keep track of all printed words.

In below set 2, we have discussed Trie based optimized solution:
Boggle | Set 2 (Using Trie)

This article is contributed by **Rishabh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/boggle-find-possible-words-board-characters/

Chapter 16

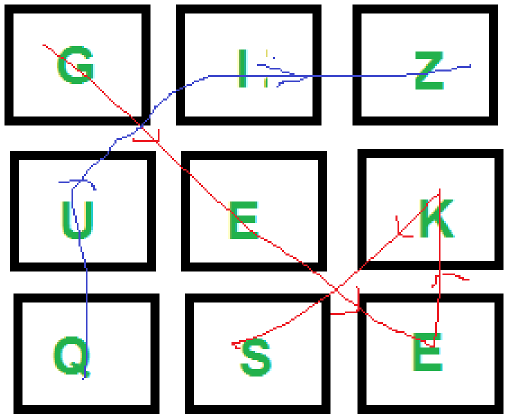
Boggle | Set 2 (Using Trie)

Boggle | Set 2 (Using Trie) - GeeksforGeeks

Given a dictionary, a method to do lookup in dictionary and a M x N board where every cell has one character. Find all possible words that can be formed by a sequence of adjacent characters. Note that we can move to any of 8 adjacent characters, but a word should not have multiple instances of same cell.

Example:

```
Input: dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
       boggle[][]   = {{'G','I','Z'},
                        {'U','E','K'},
                        {'Q','S','E'}};
      isWord(str): returns true if str is present in dictionary
                   else false.

Output:  Following words of the dictionary are present
         GEEKS
         QUIZ
```



We have discussed a Graph DFS based solution in below post.
Boggle (Find all possible words in a board of characters) | Set 1

Here we discuss a Trie based solution which is better then DFS based solution.
Given Dictionary dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"}
1. Create an Empty trie and insert all words of given dictionary into trie

```
      After insertion, Trie looks like (leaf nodes are in RED )
                           root
                       /
                       G   F     Q
                   /   |   |     |
                  O    E   O     U
                       |   |     |
                       E    R     I
                       |         |
                       K         Z
                       |
                       S
```

2. After that we have pick only those character in boggle[][] which are child of root of Trie
Let for above we pick 'G' boggle[0][0] , 'Q' boggle[2][0] (they both are present in boggle matrix)
3. search a word in a trie which start with character that we pick in step 2

```
1) Create bool visited boolean matrix (Visited[M][N] = false )
2) Call SearchWord() for every cell (i, j) which has one of the
   the first characters of dictionary words. In above example,
   we have 'G' and 'Q' as first characters.

SearchWord(Trie *root, i, j, visited[][N])
```

```
if root->leaf == true
   print word

if we have seen this element first time then make it visited.
   visited[i][j] = true
   do
      traverse all child of current root
      k goes (0 to 26 ) [there are only 26 Alphabet]
      add current char and search for next character

      find next character which is adjacent to boggle[i][j]
      they are 8 adjacent cells of boggle[i][j] (i+1, j+1),
      (i+1, j) (i-1, j) and so on.

   make it unvisited visited[i][j] = false
```

Below is the implementation of above idea

C++

```
 // C++ program for Boggle game
#include<bits/stdc++.h>
using namespace std;

// Converts key current character into index
// use only 'A' through 'Z'
#define char_int(c) ((int)c - (int)'A')

// Alphabet size
#define SIZE (26)

#define M 3
#define N 3

// trie Node
struct TrieNode
{
    TrieNode *Child[SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool leaf;
};

// Returns new trie node (initialized to NULLs)
TrieNode *getNode()
{
```

```
    TrieNode * newNode = new TrieNode;
    newNode->leaf = false;
    for (int i =0 ; i< SIZE ; i++)
        newNode->Child[i] = NULL;
    return newNode;
}

// If not present, inserts a key into the trie
// If the key is a prefix of trie node, just
// marks leaf node
void insert(TrieNode *root, char *Key)
{
    int n = strlen(Key);
    TrieNode * pChild = root;

    for (int i=0; i<n; i++)
    {
        int index = char_int(Key[i]);

        if (pChild->Child[index] == NULL)
            pChild->Child[index] = getNode();

        pChild = pChild->Child[index];
    }

    // make last node as leaf node
    pChild->leaf = true;
}

// function to check that current location
// (i and j) is in matrix range
bool isSafe(int i, int j, bool visited[M][N])
{
    return (i >=0 && i < M && j >=0 &&
            j < N && !visited[i][j]);
}

// A recursive function to print all words present on boggle
void searchWord(TrieNode *root, char boggle[M][N], int i,
                int j, bool visited[][N], string str)
{
    // if we found word in trie / dictionary
    if (root->leaf == true)
        cout << str << endl ;

    // If both I and j in  range and we visited
    // that element of matrix first time
    if (isSafe(i, j, visited))
```

```
    {
        // make it visited
        visited[i][j] = true;

        // traverse all childs of current root
        for (int K =0; K < SIZE; K++)
        {
            if (root->Child[K] != NULL)
            {
                // current character
                char ch = (char)K + (char)'A' ;

                // Recursively search reaming character of word
                // in trie for all 8 adjacent cells of boggle[i][j]
                if (isSafe(i+1,j+1,visited) && boggle[i+1][j+1] == ch)
                    searchWord(root->Child[K],boggle,i+1,j+1,visited,str+ch);
                if (isSafe(i, j+1,visited)  && boggle[i][j+1] == ch)
                    searchWord(root->Child[K],boggle,i, j+1,visited,str+ch);
                if (isSafe(i-1,j+1,visited) && boggle[i-1][j+1] == ch)
                    searchWord(root->Child[K],boggle,i-1, j+1,visited,str+ch);
                if (isSafe(i+1,j, visited)  && boggle[i+1][j] == ch)
                    searchWord(root->Child[K],boggle,i+1, j,visited,str+ch);
                if (isSafe(i+1,j-1,visited) && boggle[i+1][j-1] == ch)
                    searchWord(root->Child[K],boggle,i+1, j-1,visited,str+ch);
                if (isSafe(i, j-1,visited)&& boggle[i][j-1] == ch)
                    searchWord(root->Child[K],boggle,i,j-1,visited,str+ch);
                if (isSafe(i-1,j-1,visited) && boggle[i-1][j-1] == ch)
                    searchWord(root->Child[K],boggle,i-1, j-1,visited,str+ch);
                if (isSafe(i-1, j,visited) && boggle[i-1][j] == ch)
                    searchWord(root->Child[K],boggle,i-1, j, visited,str+ch);
            }
        }

        // make current element unvisited
        visited[i][j] = false;
    }
}

// Prints all words present in dictionary.
void findWords(char boggle[M][N], TrieNode *root)
{
    // Mark all characters as not visited
    bool visited[M][N];
    memset(visited,false,sizeof(visited));

    TrieNode *pChild = root ;

    string str = "";
```

```
    // traverse all matrix elements
    for (int i = 0 ; i < M; i++)
    {
        for (int j = 0 ; j < N ; j++)
        {
            // we start searching for word in dictionary
            // if we found a character which is child
            // of Trie root
            if (pChild->Child[char_int(boggle[i][j])] )
            {
                str = str+boggle[i][j];
                searchWord(pChild->Child[char_int(boggle[i][j])],
                           boggle, i, j, visited, str);
                str = "";
            }
        }
    }
}

//Driver program to test above function
int main()
{
    // Let the given dictionary be following
    char *dictionary[] = {"GEEKS", "FOR", "QUIZ", "GEE"};

    // root Node of trie
    TrieNode *root = getNode();

    // insert all words of dictionary into trie
    int n = sizeof(dictionary)/sizeof(dictionary[0]);
    for (int i=0; i<n; i++)
        insert(root, dictionary[i]);

    char boggle[M][N] = {{'G','I','Z'},
        {'U','E','K'},
        {'Q','S','E'}
    };

    findWords(boggle, root);

    return 0;
}
```

Java

```
 // Java program for Boggle game
public class Boggle {
```

```
// Alphabet size
static final int SIZE = 26;

static final int M = 3;
static final int N = 3;

// trie Node
static class TrieNode
{
    TrieNode[] Child = new TrieNode[SIZE];

    // isLeaf is true if the node represents
    // end of a word
    boolean leaf;

    //constructor
    public TrieNode() {
        leaf = false;
        for (int i =0 ; i< SIZE ; i++)
           Child[i] = null;
    }
}

// If not present, inserts a key into the trie
// If the key is a prefix of trie node, just
// marks leaf node
static void insert(TrieNode root, String Key)
{
    int n = Key.length();
    TrieNode pChild = root;

    for (int i=0; i<n; i++)
    {
        int index = Key.charAt(i) - 'A';

        if (pChild.Child[index] == null)
            pChild.Child[index] = new TrieNode();

        pChild = pChild.Child[index];
    }

    // make last node as leaf node
    pChild.leaf = true;
}

// function to check that current location
// (i and j) is in matrix range
```

```
static boolean isSafe(int i, int j, boolean visited[][])
{
    return (i >=0 && i < M && j >=0 &&
            j < N && !visited[i][j]);
}

// A recursive function to print all words present on boggle
static void searchWord(TrieNode root, char boggle[][], int i,
                int j, boolean visited[][], String str)
{
    // if we found word in trie / dictionary
    if (root.leaf == true)
       System.out.println(str);

    // If both I and j in  range and we visited
    // that element of matrix first time
    if (isSafe(i, j, visited))
    {
        // make it visited
        visited[i][j] = true;

        // traverse all child of current root
        for (int K =0; K < SIZE; K++)
        {
            if (root.Child[K] != null)
            {
                // current character
                char ch = (char) (K + 'A') ;

                // Recursively search reaming character of word
                // in trie for all 8 adjacent cells of
                // boggle[i][j]
                if (isSafe(i+1,j+1,visited) && boggle[i+1][j+1]
                                                == ch)
                    searchWord(root.Child[K],boggle,i+1,j+1,
                                         visited,str+ch);
                if (isSafe(i, j+1,visited)  && boggle[i][j+1]
                                                == ch)
                    searchWord(root.Child[K],boggle,i, j+1,
                                        visited,str+ch);
                if (isSafe(i-1,j+1,visited) && boggle[i-1][j+1]
                                                == ch)
                    searchWord(root.Child[K],boggle,i-1, j+1,
                                        visited,str+ch);
                if (isSafe(i+1,j, visited)  && boggle[i+1][j]
                                               == ch)
                    searchWord(root.Child[K],boggle,i+1, j,
                                       visited,str+ch);
```

```
                if (isSafe(i+1,j-1,visited) && boggle[i+1][j-1]
                                                    == ch)
                    searchWord(root.Child[K],boggle,i+1, j-1,
                                               visited,str+ch);
                if (isSafe(i, j-1,visited)&& boggle[i][j-1]
                                                   == ch)
                    searchWord(root.Child[K],boggle,i,j-1,
                                              visited,str+ch);
                if (isSafe(i-1,j-1,visited) && boggle[i-1][j-1]
                                                   == ch)
                    searchWord(root.Child[K],boggle,i-1, j-1,
                                             visited,str+ch);
                if (isSafe(i-1, j,visited) && boggle[i-1][j]
                                                  == ch)
                    searchWord(root.Child[K],boggle,i-1, j,
                                           visited,str+ch);
            }
        }

        // make current element unvisited
        visited[i][j] = false;
    }
}

// Prints all words present in dictionary.
static void findWords(char boggle[][], TrieNode root)
{
    // Mark all characters as not visited
    boolean[][] visited = new boolean[M][N];
    TrieNode pChild = root ;

    String str = "";

    // traverse all matrix elements
    for (int i = 0 ; i < M; i++)
    {
        for (int j = 0 ; j < N ; j++)
        {
            // we start searching for word in dictionary
            // if we found a character which is child
            // of Trie root
            if (pChild.Child[(boggle[i][j]) - 'A'] != null)
            {
                str = str+boggle[i][j];
                searchWord(pChild.Child[(boggle[i][j]) - 'A'],
                           boggle, i, j, visited, str);
                str = "";
            }
```

```
            }
        }
    }

    // Driver program to test above function
    public static void main(String args[])
    {
        // Let the given dictionary be following
        String dictionary[] = {"GEEKS", "FOR", "QUIZ", "GEE"};

        // root Node of trie
        TrieNode root = new TrieNode();

        // insert all words of dictionary into trie
        int n = dictionary.length;
        for (int i=0; i<n; i++)
            insert(root, dictionary[i]);

        char boggle[][] = {{'G','I','Z'},
                           {'U','E','K'},
                           {'Q','S','E'}
        };

        findWords(boggle, root);

    }
}
// This code is contributed by Sumit Ghosh
```

Output:

```
GEE, GEEKS, QUIZ
```

Source

https://www.geeksforgeeks.org/boggle-set-2-using-trie/

Chapter 17

Tug of War

Tug of War - GeeksforGeeks

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.

Following is the implementation for Tug of War problem. It prints the required arrays.

C++

```
 #include <iostream>
#include <stdlib.h>
#include <limits.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 boolen 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;
}
```

Java

```
 // Java program for Tug of war
import java.util.*;
import java.lang.*;
import java.io.*;

class TugOfWar
```

```
{
    public int min_diff;

    // function that tries every possible solution
    // by calling itself recursively
    void TOWUtil(int arr[], int n, boolean curr_elements[],
                int no_of_selected_elements, boolean soln[],
                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, 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 (Math.abs(sum / 2 - curr_sum) <
                                  min_diff)
            {
                min_diff = Math.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, 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 = arr.length;

    // the boolen array that contains the
    // inclusion and exclusion of an element
    // in current set. The number excluded
    // automatically form the other set
    boolean[] curr_elements = new boolean[n];

    // The inclusion/exclusion array for
    // final solution
    boolean[] soln = new boolean[n];

    min_diff = Integer.MAX_VALUE;

    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, sum, 0, 0);

    // Print the solution
    System.out.print("The first subset is: ");
    for (int i = 0; i < n; i++)
    {
```

```
            if (soln[i] == true)
                System.out.print(arr[i] + " ");
        }
        System.out.print("\nThe second subset is: ");
        for (int i = 0; i < n; i++)
        {
            if (soln[i] == false)
                System.out.print(arr[i] + " ");
        }
    }

    // Driver program to test above functions
    public static void main (String[] args)
    {
        int arr[] = {23, 45, -34, 12, 0, 98,
                     -99, 4, 189, -1, 4};
        TugOfWar a = new TugOfWar();
        a.tugOfWar(arr);
    }
}

// This code is contributed by Chhavi
```

Output:

```
The first subset is: 45 -34 12 98 -1
The second subset is: 23 0 -99 4 189 4
```

This article is compiled by Ashish Anand and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/tug-of-war/

Chapter 18

Backtracking to find all subsets

Backtracking to find all subsets - GeeksforGeeks

Given a set of positive integers, find all its subsets.
Examples:

```
Input : 1 2 3
Output :     // this space denotes null element.
         1
         1 2
         1 2 3
         1 3
         2
         2 3
         3

Input : 1 2
Output :
         1
         2
         1 2
```

We have already discussed iterative approach to find all subsets. This article aims to provide a backtracking approach.

Idea is that if we have n number of elements inside an array, we have exactly two choices for each of the elements. Either we include that element in our subset or we do not include it.

```
 // CPP program to find all subsets by backtracking.
#include <bits/stdc++.h>
using namespace std;
```

```
// In the array A at every step we have two
// choices for each element either  we can
// ignore the element or we can include the
// element in our subset
void subsetsUtil(vector<int>& A, vector<vector<int> >& res,
                 vector<int>& subset, int index)
{
    for (int i = index; i < A.size(); i++) {

        // include the A[i] in subset.
        subset.push_back(A[i]);
        res.push_back(subset);

        // move onto the next element.
        subsetsUtil(A, res, subset, i + 1);

        // exclude the A[i] from subset and triggers
        // backtracking.
        subset.pop_back();
    }

    return;
}

// below function returns the subsets of vector A.
vector<vector<int> > subsets(vector<int>& A)
{
    vector<int> subset;
    vector<vector<int> > res;

    // include the null element in the set.
    res.push_back(subset);

    // keeps track of current element in vector A;
    int index = 0;
    subsetsUtil(A, res, subset, index);

    return res;
}

// Driver Code.
int main()
{
    // find the subsets of below vector.
    vector<int> array = { 1, 2, 3 };

    // res will store all subsets.
```

```
    // O(2 ^ (number of elements inside array))
    // because at every step we have two choices
    // either include or ignore.
    vector<vector<int> > res = subsets(array);

    // Print result
    for (int i = 0; i < res.size(); i++) {
        for (int j = 0; j < res[i].size(); j++)
            cout << res[i][j] << " ";
        cout << endl;
    }

    return 0;
}
```

Output:

```


1
1 2
1 2 3
1 3
2
2 3
3
```

Time Complexity : O(2 ^ n)

Source

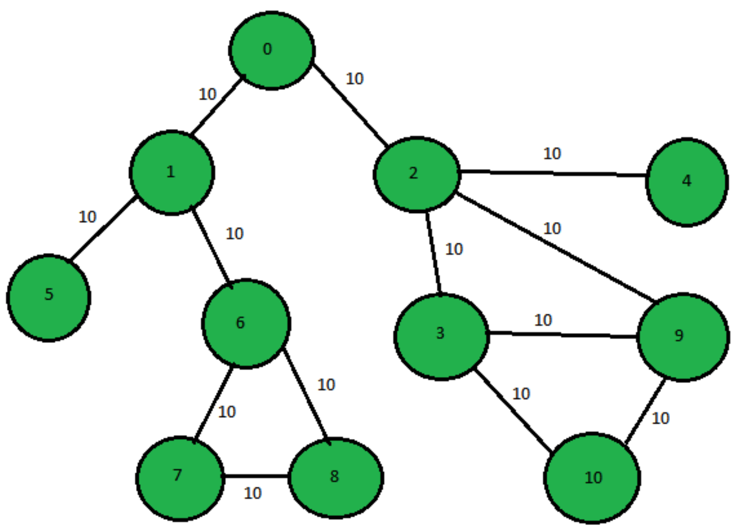
https://www.geeksforgeeks.org/backtracking-to-find-all-subsets/

Chapter 19

Print the DFS traversal step-wise (Backtracking also)

Print the DFS traversal step-wise (Backtracking also) - GeeksforGeeks

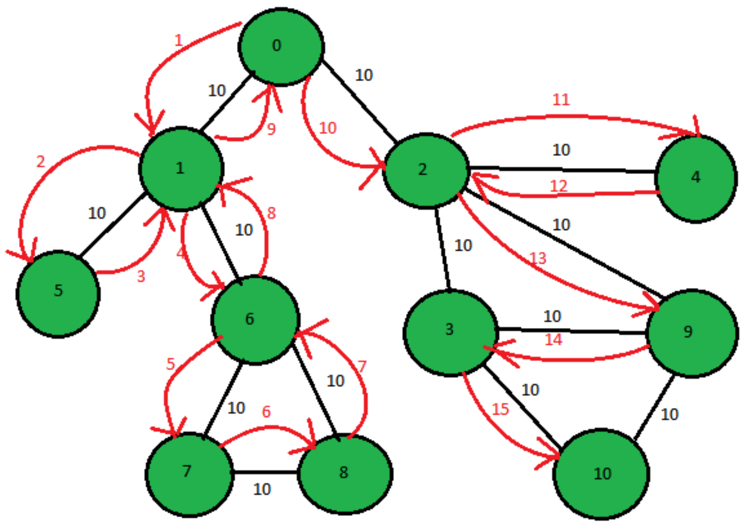
Given a graph, the task is to print the DFS traversal of a graph which includes the every step including the backtracking.




```
1st step:- 0 -> 1
2nd step:- 1 -> 5
3rd step:- 5 -> 1 (backtracking step)
4th step:- 1 -> 6...
and so on till all the nodes are visited.
```

```
Dfs step-wise(including backtracking) is:
0 1 5 1 6 7 8 7 6 1 0 2 4 2 9 3 10
```

Note: In this above diagram the weight between the edges has just been added, it does not have any role in DFS-traversal



Approach: DFS with Backtracking will be used here. First, visit every node using DFS simultaneously and keep track of the previously used edge and the parent node. If a node comes whose all the adjacent node has been visited, backtrack using the last used edge and print the nodes. Continue the steps and at every step, the parent node will become the present node. Continue the above steps to find the complete DFS traversal of the graph.

Below is the implementation of the above approach:

```
 // C++ program to print the complete
// DFS-traversal of graph
// using back-tracking
#include <bits/stdc++.h>
using namespace std;
const int N = 1000;
vector<int> adj[N];

// Function to print the complete DFS-traversal
void dfsUtil(int u, int node, bool visited[],
             vector<pair<int, int> > road_used, int parent, int it)
{
    int c = 0;
```

```
    // Check if all th node is visited or not
    // and count unvisited nodes
    for (int i = 0; i < node; i++)
        if (visited[i])
            c++;

    // If all the node is visited return;
    if (c == node)
        return;

    // Mark not visited node as visited
    visited[u] = true;

    // Track the current edge
    road_used.push_back({ parent, u });

    // Print the node
    cout << u << " ";

    // Check for not visited node and proceed with it.
    for (int x : adj[u]) {

        // call the DFs function if not visited
        if (!visited[x])
            dfsUtil(x, node, visited, road_used, u, it + 1);
    }

    // Backtrack through the last
    // visited nodes
    for (auto y : road_used)
        if (y.second == u)
            dfsUtil(y.first, node, visited,
                    road_used, u, it + 1);
}

// Function to call the DFS function
// which prints the DFS-travesal stepwise
void dfs(int node)
{

    // Create a array of visited ndoe
    bool visited[node];

    // Vector to track last visited road
    vector<pair<int, int> > road_used;

    // Initialize all the node with false
    for (int i = 0; i < node; i++)
```

```
        visited[i] = false;

    // call the function
    dfsUtil(0, node, visited, road_used, -1, 0);
}

// Function to insert edges in Graph
void insertEdge(int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// Driver Code
int main()
{
    // number of nodes and edges in the graph
    int node = 11, edge = 13;

    // Function call to create the graph
    insertEdge(0, 1);
    insertEdge(0, 2);
    insertEdge(1, 5);
    insertEdge(1, 6);
    insertEdge(2, 4);
    insertEdge(2, 9);
    insertEdge(6, 7);
    insertEdge(6, 8);
    insertEdge(7, 8);
    insertEdge(2, 3);
    insertEdge(3, 9);
    insertEdge(3, 10);
    insertEdge(9, 10);

    // Call the function to print
    dfs(node);

    return 0;
}
```

Output:

```
0 1 5 1 6 7 8 7 6 1 0 2 4 2 9 3 10
```

Source

https://www.geeksforgeeks.org/print-the-dfs-traversal-step-wise-backtracking-also/

Chapter 20

C++ program for Solving Cryptarithmetic Puzzles

C++ program for Solving Cryptarithmetic Puzzles - GeeksforGeeks

Newspapers and magazines often have crypt-arithmetic puzzles of the form:
Examples:

```
Input : s1 = SEND, s2 = "MORE", s3 = "MONEY"
Output : One of the possible solution is:
         D=1 E=5 M=0 N=3 O=8 R=2 S=7 Y=6
Explanation:
The above values satisfy below equation :
  SEND
+ MORE
--------
 MONEY
--------
```

It is strongly recommended to refer Backtracking | Set 8 (Solving Cryptarithmetic Puzzles) for approach of this problem.

The idea is to assign each letter a digit from 0 to 9 so that the arithmetic works out correctly. A permutation is a recursive function which calls a check function for every possible permutation of integers.
Check function checks whether the sum of first two numbers corresponding to first two string is equal to the third number corresponding to third string. If the solution is found then print the solution.

```
 // CPP program for solving cryptographic puzzles
#include <bits/stdc++.h>
```

```
using namespace std;

// vector stores 1 corresponding to index
// number which is already assigned
// to any char, otherwise stores 0
vector<int> use(10);

// structure to store char and its corresponding integer
struct node
{
    char c;
    int v;
};

// function check for correct solution
int check(node* nodeArr, const int count, string s1,
                               string s2, string s3)
{
    int val1 = 0, val2 = 0, val3 = 0, m = 1, j, i;

    // calculate number corresponding to first string
    for (i = s1.length() - 1; i >= 0; i--)
    {
        char ch = s1[i];
        for (j = 0; j < count; j++)
            if (nodeArr[j].c == ch)
                break;

        val1 += m * nodeArr[j].v;
        m *= 10;
    }
    m = 1;

    // calculate number corresponding to second string
    for (i = s2.length() - 1; i >= 0; i--)
    {
        char ch = s2[i];
        for (j = 0; j < count; j++)
            if (nodeArr[j].c == ch)
                break;

        val2 += m * nodeArr[j].v;
        m *= 10;
    }
    m = 1;

    // calculate number corresponding to third string
    for (i = s3.length() - 1; i >= 0; i--)
```

```
    {
        char ch = s3[i];
        for (j = 0; j < count; j++)
            if (nodeArr[j].c == ch)
                break;

        val3 += m * nodeArr[j].v;
        m *= 10;
    }

    // sum of first two number equal to third return true
    if (val3 == (val1 + val2))
        return 1;

    // else return false
    return 0;
}

// Recursive function to check solution for all permutations
bool permutation(const int count, node* nodeArr, int n,
                 string s1, string s2, string s3)
{
    // Base case
    if (n == count - 1)
    {

        // check for all numbers not used yet
        for (int i = 0; i < 10; i++)
        {

            // if not used
            if (use[i] == 0)
            {

                // assign char at index n integer i
                nodeArr[n].v = i;

                // if solution found
                if (check(nodeArr, count, s1, s2, s3) == 1)
                {
                    cout << "\nSolution found: ";
                    for (int j = 0; j < count; j++)
                        cout << " " << nodeArr[j].c << " = "
                             << nodeArr[j].v;
                    return true;
                }
            }
        }
```

```
        return false;
    }

    for (int i = 0; i < 10; i++)
    {

        // if ith integer not used yet
        if (use[i] == 0)
        {

            // assign char at index n integer i
            nodeArr[n].v = i;

            // mark it as not available for other char
            use[i] = 1;

            // call recursive function
            if (permutation(count, nodeArr, n + 1, s1, s2, s3))
                return true;

            // backtrack for all other possible solutions
            use[i] = 0;
        }
    }
    return false;
}

bool solveCryptographic(string s1, string s2,
                                   string s3)
{
    // count to store number of unique char
    int count = 0;

    // Length of all three strings
    int l1 = s1.length();
    int l2 = s2.length();
    int l3 = s3.length();

    // vector to store frequency of each char
    vector<int> freq(26);

    for (int i = 0; i < l1; i++)
        ++freq[s1[i] - 'A'];

    for (int i = 0; i < l2; i++)
        ++freq[s2[i] - 'A'];

    for (int i = 0; i < l3; i++)
```

```
        ++freq[s3[i] - 'A'];

    // count number of unique char
    for (int i = 0; i < 26; i++)
        if (freq[i] > 0)
            count++;

    // solution not possible for count greater than 10
    if (count > 10)
    {
        cout << "Invalid strings";
        return 0;
    }

    // array of nodes
    node nodeArr[count];

    // store all unique char in nodeArr
    for (int i = 0, j = 0; i < 26; i++)
    {
        if (freq[i] > 0)
        {
            nodeArr[j].c = char(i + 'A');
            j++;
        }
    }
    return permutation(count, nodeArr, 0, s1, s2, s3);
}

// Driver function
int main()
{
    string s1 = "SEND";
    string s2 = "MORE";
    string s3 = "MONEY";

    if (solveCryptographic(s1, s2, s3) == false)
        cout << "No solution";
    return 0;
}
```

Output:

```
Solution found:  D=1 E=5 M=0 N=3 O=8 R=2 S=7 Y=6
```

Source

https://www.geeksforgeeks.org/c-code-article-backtracking-set-8-solving-cryptarithmetic-puzzles/

Chapter 21

A backtracking approach to generate n bit Gray Codes

A backtracking approach to generate n bit Gray Codes - GeeksforGeeks

Given a number n, the task is to generate n bit Gray codes (generate bit patterns from 0 to 2^n-1 such that successive patterns differ by one bit)
Examples:

```
Input : 2
Output : 0 1 3 2
Explanation :
00 - 0
01 - 1
11 - 3
10 - 2

Input : 3
Output : 0 1 3 2 6 7 5 4
```

We have discussed an approach in Generate n-bit Gray Codes

This article provides a **backtracking approach** to the same problem. Idea is that for each bit out of n bit we have a choice either we can ignore it or we can invert the bit so this means our gray sequence goes upto 2 ^ n for n bits. So we make two recursive calls for either inverting the bit or leaving the bit as it is.

```
 // CPP program to find the gray sequence of n bits.
#include <iostream>
#include <vector>
using namespace std;
```

```
/* we have 2 choices for each of the n bits either we
   can include i.e invert the bit or we can exclude the
   bit i.e we can leave the number as it is. */
void grayCodeUtil(vector<int>& res, int n, int& num)
{
    // base case when we run out bits to process
    // we simply include it in gray code sequence.
    if (n == 0) {
        res.push_back(num);
        return;
    }

    // ignore the bit.
    grayCodeUtil(res, n - 1, num);

    // invert the bit.
    num = num ^ (1 << (n - 1));
    grayCodeUtil(res, n - 1, num);
}

// returns the vector containing the gray
// code sequence of n bits.
vector<int> grayCodes(int n)
{
    vector<int> res;

    // num is passed by reference to keep
    // track of current code.
    int num = 0;
    grayCodeUtil(res, n, num);

    return res;
}

// Driver function.
int main()
{
    int n = 3;
    vector<int> code = grayCodes(n);
    for (int i = 0; i < code.size(); i++)
        cout << code[i] << endl;
    return 0;
}
```

Output:

```
0
1
3
2
6
7
5
4
```

Source

https://www.geeksforgeeks.org/backtracking-approach-generate-n-bit-gray-codes/

Chapter 22

Minimum queens required to cover all the squares of a chess board

Minimum queens required to cover all the squares of a chess board - GeeksforGeeks

Given the dimension of a chess board (N x M), determine the minimum number of queens required to cover all the squares of the board. A queen can attack any square along its row, column or diagonals.

Examples:

```
Input : N = 8, M = 8
Output : 5
Layout : Q X X X X X X X
         X X Q X X X X X
         X X X X Q X X X
         X Q X X X X X X
         X X X Q X X X X
         X X X X X X X X
         X X X X X X X X
         X X X X X X X X

Input : N = 3, M = 5
Output : 2
Layout : Q X X X X
         X X X X X
         X X X Q X
```

This article attempts to solve the problem in a very simple way without much optimization.

Step 1: Starting from any corner square of the board, find an 'uncovered' square (Uncovered square is a square which isn't attacked by any of the queens already placed). If none found, goto Step 4.
Step 2: Place a Queen on this square and increment variable 'count' by 1.
Step 3: Repeat step 1.
Step 4: Now, you've got a layout where every square is covered. Therefore, the value of 'count' can be the answer. However, you might be able to do better, as there might exist a better layout with lesser number of queens. So, store this 'count' as the best value till now and proceed to find a better solution.
Step 5: Remove the last queen placed and place it in the next 'uncovered' cell.
Step 6: Proceed recursively and try out all the possible layouts. Finally, the one with the least number of queens is the answer.

Dry run the following code for better understanding.

```
 // Java program to find minimum number of queens needed
// to cover a given chess board.

public class Backtracking {

    // The chessboard is represented by a 2D array.
    static boolean[][] board;

    // N x M is the dimension of the chess board.
    static int N, M;

    // The minimum number of queens required.
    // Initially, set to MAX_VAL.
    static int minCount = Integer.MAX_VALUE;

    static String layout; // Stores the best layout.

    // Driver code
    public static void main(String[] args)
    {
        N = 8;
        M = 8;
        board = new boolean[N][M];
        placeQueen(0);

        System.out.println(minCount);
        System.out.println("\nLayout: \n" + layout);
    }

    // Finds minimum count of queens needed and places them.
    static void placeQueen(int countSoFar)
    {
        int i, j;
```

```
    if (countSoFar >= minCount)

        // We've already obtained a solution with lesser or
        // same number of queens. Hence, no need to proceed.
        return;

    // Checks if there exists any unattacked cells.
    findUnattackedCell : {
    for (i = 0; i < N; ++i)
        for (j = 0; j < M; ++j)
            if (!isAttacked(i, j))

                // Square (i, j) is unattacked.
                break findUnattackedCell;

    // All squares all covered. Hence, this
    // is the best solution till now.
    minCount = countSoFar;
    storeLayout();

    return;
    }

    for (i = 0; i < N; ++i)
        for (j = 0; j < M; ++j) {
            if (!isAttacked(i, j)) {

                // Square (i, j) is unattacked.
                // Therefore, place a queen here.
                board[i][j] = true;

                // Increment 'count' and proceed recursively.
                placeQueen(countSoFar + 1);

                // Remove this queen and attempt to
                // find a better solution.
                board[i][j] = false;
            }
        }
}

// Returns 'true' if the square (row, col) is
// being attacked by at least one queen.
static boolean isAttacked(int row, int col)
{
    int i, j;
```

```
        // Check the 'col'th column for any queen.
        for (i = 0; i < N; ++i)
            if (board[i][col])
                return true;

        // Check the 'row'th row for any queen.
        for (j = 0; j < M; ++j)
            if (board[row][j])
                return true;

        // Check the diagonals for any queen.
        for (i = 0; i < Math.min(N, M); ++i)
            if (row - i >= 0 && col - i >= 0 &&
                        board[row - i][col - i])
                return true;
            else if (row - i >= 0 && col + i < M &&
                           board[row - i][col + i])
                return true;
            else if (row + i < N && col - i >= 0 &&
                            board[row + i][col - i])
                return true;
            else if (row + i < N && col + i < M &&
                            board[row + i][col + i])
                return true;

        // This square is unattacked. Hence return 'false'.
        return false;
    }

    // Stores the current layout in 'layout'
    // variable as String.
    static void storeLayout()
    {
        StringBuilder sb = new StringBuilder();
        for (boolean[] row : board) {
            for (boolean cell : row)
                sb.append(cell ? "Q " : "X ");
            sb.append("\n");
        }
        layout = sb.toString();
    }
}
```

Output:

```
5
```

```
Layout:
Q X X X X X X X
X X Q X X X X X
X X X X Q X X X
X Q X X X X X X
X X X Q X X X X
X X X X X X X X
X X X X X X X X
X X X X X X X X
```

Improved By : sarthakmannaofficial

Source

https://www.geeksforgeeks.org/minimum-queens-required-to-cover-all-the-squares-of-a-chess-board/

Chapter 23

Print all the combinations of a string in lexicographical order

Print all the combinations of a string in lexicographical order - GeeksforGeeks

Given a string str, print of all the combinations of a string in lexicographical order.

Examples:

```
Input: str = "ABC"
Output:
A
AB
ABC
AC
ACB
B
BA
BAC
BC
BCA
C
CA
CAB
CB
CBA

Input: ED
Output:
D
DE
E
```

ED

Approach: Count the occurrences of all the characters in the string using a map, then using recursion all the possible combinations can be printed. Store the elements and their counts in two different arrays. Three arrays are used, input[] array which has the characters, count[] array has the count of characters and result[] is a temporary array which is used in recursion to generate all the combinations. Using recursion and backtracking all the combinations can be printed.

Below is the implementation of the above approach.

```
 // C++ program to find all combinations
// of a string in lexicographical order
#include <bits/stdc++.h>
using namespace std;

// function to print string
void printResult(char* result, int len)
{
    for (int i = 0; i <= len; i++)
        cout << result[i];
    cout << endl;
}

// Method to found all combination
// of string it is based in tree
void stringCombination(char result[], char str[], int count[],
                       int level, int size, int length)
{
    // return if level is equal size of string
    if (level == size)
        return;

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

        // if occurrence of char is 0 then
        // skip the iteration of loop
        if (count[i] == 0)
            continue;

        // decrease the char occurrence by 1
        count[i]--;

        // store the char in result
        result[level] = str[i];

        // print the string till level
        printResult(result, level);
```

```
        // call the function from level +1
        stringCombination(result, str, count,
                          level + 1, size, length);

        // backtracking
        count[i]++;
    }
}

void combination(string str)
{

    // declare the map for store
    // each char with occurrence
    map<char, int> mp;

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

        if (mp.find(str[i]) != mp.end())
            mp[str[i]] = mp[str[i]] + 1;
        else
            mp[str[i]] = 1;
    }

    // initialize the input array
    // with all unique char
    char* input = new char[mp.size()];

    // initialize the count array with
    // occurrence the unique char
    int* count = new int[mp.size()];

    // temporary char array for store the result
    char* result = new char[str.size()];

    map<char, int>::iterator it = mp.begin();
    int i = 0;

    for (it; it != mp.end(); it++) {
        // store the element of input array
        input[i] = it->first;

        // store the element of count array
        count[i] = it->second;
        i++;
    }
```

```
    // size of map(no of unique char)
    int length = mp.size();

    // size of original string
    int size = str.size();

    // call function for print string combination
    stringCombination(result, input, count,
                      0, size, length);
}

// Driver code
int main()
{
    string str = "ABC";
    cin >> str;

    combination(str);

    return 0;
}
```

Output:

```
A
AB
ABC
AC
ACB
B
BA
BAC
BC
BCA
C
CA
CAB
CB
CBA
```

Source

https://www.geeksforgeeks.org/print-all-the-combinations-of-a-string-in-lexicographical-order/

Chapter 24

Recursive program to generate power set

Recursive program to generate power set - GeeksforGeeks

Given a set represented as string, write a recursive code to print all subsets of it. The subsets can be printed in any order.

Examples:

```
Input :  set = "abc"
Output : "". "a", "b", "c", "ab", "ac", "bc", "abc"

Input : set = "abcd"
Output : "" "a" "ab" "abc" "abcd" "abd" "ac" "acd"
         "ad" "b" "bc" "bcd" "bd" "c" "cd" "d"
```

Method 1 : The idea is to fix a prefix, general all subsets beginning with current prefix. After all subsets with a prefix are generated, replace last character with one of the remaining characters.

```
 // CPP program to generate power set
#include <bits/stdc++.h>
using namespace std;

// str : Stores input string
// curr : Stores current subset
// index : Index in current subset, curr
void powerSet(string str, int index = -1,
              string curr = "")
{
    int n = str.length();
```

```
    // base case
    if (index == n)
        return;

    // First print current subset
    cout << curr << "\n";

    // Try appending remaining characters
    // to current subset
    for (int i = index + 1; i < n; i++) {

        curr += str[i];
        powerSet(str, i, curr);

        // Once all subsets beginning with
        // initial "curr" are printed, remove
        // last character to consider a different
        // prefix of subsets.
        curr.erase(curr.size() - 1);
    }
    return;
}

// Driver code
int main()
{
    string str = "abc";
    powerSet(str);
    return 0;
}
```

Output:

```

a
ab
abc
ac
b
bc
c
```

Method 2 : The idea is to consider two cases for every character. (i) Consider current character as part of current subset (ii) Do not consider current character as part of current subset.

```
 // CPP program to generate power set
#include <bits/stdc++.h>
using namespace std;

// str : Stores input string
// curr : Stores current subset
// index : Index in current subset, curr
void powerSet(string str, int index = 0,
            string curr = "")
{
    int n = str.length();

    // base case
    if (index == n)
    {
         cout << curr << endl;
         return;
    }

    // Two cases for every character
    // (i) We consider the character
    //     as part of current subset
    // (ii) We do not consider current
    //     character as part of current
    //     subset
    powerSet(str, index+1, curr+str[index]);
    powerSet(str, index+1, curr);
}

// Driver code
int main()
{
    string str = "abc";
    powerSet(str);
    return 0;
}
```

Output:

```
abc
ab
ac
a
bc
b
c
```

Iterative program for power set.

Source

https://www.geeksforgeeks.org/recursive-program-to-generate-power-set/

Chapter 25

Smallest number with given sum of digits and sum of square of digits

Smallest number with given sum of digits and sum of square of digits - GeeksforGeeks

Given sum of digits a and sum of square of digits b. Find the smallest number with given sum of digits and sum of the square of digits. The number should not contain more than 100 digits. Print -1 if no such number exists or if the number of digits is more than 100.

Examples:

> **Input :** a = 18, b = 162
> **Output :** 99
> **Explanation :** 99 is the smallest possible number whose sum of digits = 9 + 9 = 18 and sum of squares of digits is $9^2+9^2 = 162$.
>
> **Input :** a = 12, b = 9
> **Output :** -1

Approach:
Since the smallest number can be of 100 digits, it cannot be stored. Hence the first step to solve it will be to find the minimum number of digits which can give us the sum of digits as a and sum of the square of digits as b. To find the minimum number of digits, we can use Dynamic Programming. DP[a][b] signifies the minimum number of digits in a number whose sum of the digits will be a and sum of the square of digits will be b. If there does not exist any such number then DP[a][b] will be -1.

Since the number cannot exceed 100 digits, DP array will be of size **101*8101**. Iterate for every digit, and try all possible combination of digits which gives us the sum of digits as a and sum of the square of digits as b. Store the minimum number of digits in DP[a][b] using the below recurrence relation:

DP[a][b] = min(minimumNumberOfDigits(a – i, b – (i * i)) + 1)
where 1<=i<=9

After getting the minimum number of digits, find the digits. To find the digits, check for all combinations and print those digits which satisfies the condition below:

1 + dp[a – i][b – i * i] == dp[a][b]
where 1<=i<=9

If the condition above is met by any of i, reduce a by i and b by i*i and break. Keep on repeating the above process to find all the digits till a is 0 and b is 0.

Below is the C++ implementation of above approach:

```
 // CPP program to find the Smallest number
// with given sum of digits and
// sum of square of digits
#include <bits/stdc++.h>
using namespace std;

int dp[901][8101];

// Top down dp to find minimum number of digits with
// given sum of dits a and sum of square of digits as b
int minimumNumberOfDigits(int a, int b)
{
    // Invalid condition
    if (a > b || a < 0 || b < 0 || a > 900 || b > 8100)
        return -1;

    // Number of digits satisfied
    if (a == 0 && b == 0)
        return 0;

    // Memoization
    if (dp[a][b] != -1)
        return dp[a][b];

    // Intialize ans as maximum as we have to find the
    // minimum number of digits
    int ans = 101;

    // Check for all possible combinations of digits
    for (int i = 9; i >= 1; i--) {

        // recurrence call
        int k = minimumNumberOfDigits(a - i, b - (i * i));
```

```
        // If the combination of digits cannot give sum as a
        // and sum of square of digits as b
        if (k != -1)
            ans = min(ans, k + 1);
    }

    // Returns the minimum number of digits
    return dp[a][b] = ans;
}

// Function to print the digits that gives
// sum as a and sum of square of digits as b
void printSmallestNumber(int a,int b)
{

    // initialize the dp array as -1
    memset(dp, -1, sizeof(dp));

    // base condition
    dp[0][0] = 0;

    // function call to get the minimum number of digits
    int k = minimumNumberOfDigits(a, b);

    // When there does not exists any number
    if (k == -1 || k > 100)
        cout << "-1";
    else {
        // Printing the digits from the most significant digit
        while (a > 0 && b > 0) {

            // Trying all combinations
            for (int i = 1; i <= 9; i++) {
                // checking conditions for minimum digits
                if (a >= i && b >= i * i &&
                    1 + dp[a - i][b - i * i] == dp[a][b]) {
                    cout << i;
                    a -= i;
                    b -= i * i;
                    break;
                }
            }
        }
    }
}

// Driver Code
```

```
int main()
{
    int a = 18, b = 162;
    // Function call to print the smallest number
    printSmallestNumber(a,b);
}
```

Output:

```
99
```

Time Complexity : O(900*8100*9)
Auxiliary Space : O(900*8100)

Note: Time complexity is in terms of numbers as we are trying all possible combinations of digits.

Source

https://www.geeksforgeeks.org/smallest-number-with-given-sum-of-digits-and-sum-of-square-of-digits/

Chapter 26

Minimize number of unique characters in string

Minimize number of unique characters in string - GeeksforGeeks

Given two strings A and B. Minimize the number of unique characters in string A by either swapping A[i] with B[i] or keeping it unchanged. The number of swaps can be greater than or equal to 0. Note that A[i] can be swapped only with same index element in B. Print the minimum number of unique characters. Constraints: 0 < length of A   15.

Examples:

```
Input : A = ababa
        B = babab
Output : 1
Swapping all b's in string A, with
a's in string B results in string
A having all characters as a.

Input : A = abaaa
        B = bbabb
Output : 2
Initially string A has 2 unique
characters. Swapping at any index
does not change this count.
```

Approach: The problem can be solved using backtracking. Create a map in which key is A[i] and value is count of corresponding character. The size of the map tells the number of distinct characters as only those elements which are present in string A are present as key in map. At every index position, there are two choices: either swap A[i] with B[i] or keep A[i] unchanged. Start from index 0 and do following for each index:

1. Keep A[i] unchanged, increment count of A[i] by one in map and call recursively for next index.
2. Backtrack by decreasing count of A[i] by one, swap A[i] with B[i], increment count of A[i] by one in map and again recursively call for next index.

Keep a variable ans to store overall minimum value of distinct characters. In both the cases mentioned above, when entire string is traversed compare current number of distinct characters with overall minimum in ans and update ans accordingly.

Implementation:

```
 // CPP program to minimize number of
// unique characters in a string.

#include <bits/stdc++.h>
using namespace std;

// Utility function to find minimum
// number of unique characters in string.
void minCountUtil(string A, string B,
                  unordered_map<char, int>& ele,
                  int& ans, int ind)
{

    // If entire string is traversed, then
    // compare current number of distinct
    // characters in A with overall minimum.
    if (ind == A.length()) {
        ans = min(ans, (int)ele.size());
        return;
    }

    // swap A[i] with B[i], increase count of
    // corresponding character in map and call
    // recursively for next index.
    swap(A[ind], B[ind]);
    ele[A[ind]]++;
    minCountUtil(A, B, ele, ans, ind + 1);

    // Backtrack (Undo the changes done)
    ele[A[ind]]--;

    // If count of character is reduced to zero,
    // then that character is not present in A.
    // So remove that character from map.
    if (ele[A[ind]] == 0)
        ele.erase(A[ind]);
```

```
    // Restore A to original form.
    // (Backtracking step)
    swap(A[ind], B[ind]);

    // Increase count of A[i] in map and
    // call recursively for next index.
    ele[A[ind]]++;
    minCountUtil(A, B, ele, ans, ind + 1);

    // Restore the changes done
    // (Backtracking step)
    ele[A[ind]]--;
    if (ele[A[ind]] == 0)
        ele.erase(A[ind]);
}

// Function to find minimum number of
// distinct characters in string.
int minCount(string A, string B)
{
    // Variable to store minimum number
    // of distinct character.
    // Initialize it with length of A
    // as maximum possible value is
    // length of A.
    int ans = A.length();

    // Map to store count of distinct
    // characters in A. To keep
    // complexity of insert operation
    // constant unordered_map is used.
    unordered_map<char, int> ele;

    // Call utility function to find
    // minimum number of unique
    // characters.
    minCountUtil(A, B, ele, ans, 0);

    return ans;
}

int main()
{
    string A = "abaaa";
    string B = "bbabb";

    cout << minCount(A, B);
```

```
    return 0;
}
```

Output:

```
2
```

Time Complexity: $O(2^n)$
Auxiliary Space: $O(n)$

Improved By : SahilMalik1

Source

https://www.geeksforgeeks.org/minimize-number-unique-characters-string/

Chapter 27

Rat in a Maze with multiple steps or jump allowed

Rat in a Maze with multiple steps or jump allowed - GeeksforGeeks

This is the variation of Rat in Maze

A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach destination. The rat can move only in two directions: forward and down.
In the maze matrix, 0 means the block is dead end and non-zero number means the block can be used in the path from source to destination. The non-zero value of mat[i][j] indicates number of maximum jumps rat can make from cell mat[i][j].

In this variation, Rat is allowed to jump multiple steps at a time instead of 1.

Examples
Examples:

```
Input : { {2, 1, 0, 0},
         {3, 0, 0, 1},
         {0, 1, 0, 1},
          {0, 0, 0, 1}
        }
Output : { {1, 0, 0, 0},
           {1, 0, 0, 1},
           {0, 0, 0, 1},
           {0, 0, 0, 1}
         }

Explanation
Rat started with M[0][0] and can jump upto 2 steps right/down.
```

```
Let's try in horizontal direction -
M[0][1] won't lead to solution and M[0][2] is 0 which is dead end.
So, backtrack and try in down direction.
Rat jump down to M[1][0] which eventually leads to solution.

Input : {
      {2, 1, 0, 0},
      {2, 0, 0, 1},
      {0, 1, 0, 1},
      {0, 0, 0, 1}
    }
Output : Solution doesn't exist
```

Naive Algorithm

The Naive Algorithm is to generate all paths from source to destination and one by one check if the generated path satisfies the constraints.

```
while there are untried paths
{
   generate the next path
   if this path has all blocks as non-zero
   {
      print this path;
   }
}
```

Backtracking Algorithm

```
If destination is reached
    print the solution matrix
Else
   a) Mark current cell in solution matrix as 1.
   b) Move forward/jump (for each valid steps) in horizontal direction
      and recursively check if this move leads to a solution.
   c) If the move chosen in the above step doesn't lead to a solution
       then move down and check if this move leads to a solution.
   d) If none of the above solutions work then unmark this cell as 0
       (BACKTRACK) and return false.
```

Implementation of Backtracking solution

C/C++

```
 /* C/C++ program to solve Rat in a Maze problem
   using backtracking */
#include <stdio.h>
```

```
// Maze size
#define N 4

bool solveMazeUtil(int maze[N][N], int x, int y,
                                int sol[N][N]);

/* A utility function to print solution matrix
   sol[N][N] */
void printSolution(int sol[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf(" %d ", sol[i][j]);
        printf("\n");
    }
}

/* A utility function to check if x, y is valid
   index for N*N maze */
bool isSafe(int maze[N][N], int x, int y)
{
    // if (x, y outside maze) return false
    if (x >= 0 && x < N && y >= 0 &&
       y < N && maze[x][y] != 0)
        return true;

    return false;
}

/* This function solves the Maze problem using
Backtracking. It mainly uses solveMazeUtil() to
solve the problem. It returns false if no path
is possible, otherwise return true and prints
the path in the form of 1s. Please note that
there may be more than one solutions,
this function prints one of the feasible solutions.*/
bool solveMaze(int maze[N][N])
{
    int sol[N][N] = { { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 } };

    if (solveMazeUtil(maze, 0, 0, sol) == false) {
        printf("Solution doesn't exist");
        return false;
    }
```

```
    printSolution(sol);
    return true;
}

/* A recursive utility function to solve Maze problem */
bool solveMazeUtil(int maze[N][N], int x, int y,
                                 int sol[N][N])
{
    // if (x, y is goal) return true
    if (x == N - 1 && y == N - 1) {
        sol[x][y] = 1;
        return true;
    }

    // Check if maze[x][y] is valid
    if (isSafe(maze, x, y) == true) {

        // mark x, y as part of solution path
        sol[x][y] = 1;

        /* Move forward in x direction */
        for (int i = 1; i <= maze[x][y] && i < N; i++) {

            /* Move forward in x direction */
            if (solveMazeUtil(maze, x + i, y, sol) == true)
                return true;

            /* If moving in x direction doesn't give
               solution then Move down in y direction */
            if (solveMazeUtil(maze, x, y + i, sol) == true)
                return true;
        }

        /* If none of the above movements work then
           BACKTRACK: unmark x, y as part of solution
           path */
        sol[x][y] = 0;
        return false;
    }

    return false;
}

// driver program to test above function
int main()
{
    int maze[N][N] = { { 2, 1, 0, 0 },
                       { 3, 0, 0, 1 },
```

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

    solveMaze(maze);
    return 0;
}
```

Output:

```
1  0  0  0
 1  0  0  1
 0  0  0  1
 0  0  0  1
```

Source

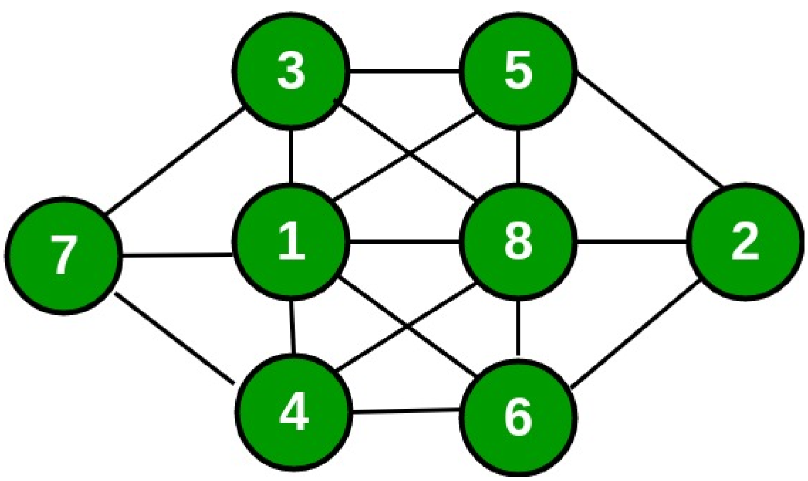
https://www.geeksforgeeks.org/rat-in-a-maze-with-multiple-steps-jump-allowed/

Chapter 28

Fill 8 numbers in grid with given conditions

Fill 8 numbers in grid with given conditions - GeeksforGeeks

Place the numbers 1, 2, 3, 4, 5, 6, 7, 8 into the eight circles in the figure given below, in such a way that no number is adjacent to a number that is next to it in the sequence. For example, 1 should not be adjacent to 2 but can be adjacent to 3, 4, 5, 6, 7, 8. Similarly for others.



Naive Algorithm
The Naive Algorithm is to generate all possible configurations of numbers from 1 to 8 to fill the empty cells. Try every configuration one by one until the correct configuration is found.

Backtracking Algorithm
Like all other Backtraking problems, we can solve this problem by one by one assigning numbers to empty cells. Before assigning a number, we check whether it is safe to assign. We basically check that the same number is not present to its adjacent cell (vertically, horizontally or diagonally). After checking for safety, we 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 we try next number for the current empty cell. And if none of the number (1 to 8) leads to solution, we return false.

```
  Find row, col of an unassigned cell
  If there is none, return true
  For digits from 1 to 8
    a) If there is no conflict for digit at row, col
        assign digit to row, col and recursively try fill in rest of grid
    b) If recursion successful, return true
    c) Else, remove digit and try another
  If all digits have been tried and nothing worked, return false
```

```
 // A Backtracking program in
// C++ to solve given problem
#include <cmath>
#include <iostream>

#define N 3 // row of grid
#define M 4 // column of grid
#define UNASSIGNED -1
using namespace std;

/* Returns a boolean which indicates
whether any assigned entry within the
specified grid matches the given number. */
bool UsedInGrid(int grid[N][M], int num)
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++)
            if (grid[i][j] == 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][M], int row, int col, int num)
{
    /* Check if 'num' is not already placed in Whole Grid*/
    if (row == 0 && col == 1) {

        if (UsedInGrid(grid, num)
            || (abs(num - grid[row][col + 1]) <= 1)
            || (abs(num - grid[row + 1][col]) <= 1)
```

```
            || (abs(num - grid[row + 1][col - 1]) <= 1)
            || (abs(num - grid[row + 1][col + 1]) <= 1))
            return false;
    }
    else if (row == 0 && col == 2) {
        if (UsedInGrid(grid, num)
            || (abs(num - grid[row][col - 1]) <= 1)
            || (abs(num - grid[row + 1][col]) <= 1)
            || (abs(num - grid[row + 1][col + 1]) <= 1)
            || (abs(num - grid[row + 1][col - 1]) <= 1))
            return false;
    }
    else if (row == 1 && col == 0) {
        if (UsedInGrid(grid, num)
            || (abs(num - grid[row - 1][col + 1]) <= 1)
            || (abs(num - grid[row][col + 1]) <= 1)
            || (abs(num - grid[row + 1][col + 1]) <= 1))
            return false;
    }
    else if (row == 1 && col == 3) {
        if (UsedInGrid(grid, num)
            || (abs(num - grid[row - 1][col - 1]) <= 1)
            || (abs(num - grid[row][col - 1]) <= 1)
            || (abs(num - grid[row + 1][col - 1]) <= 1))
            return false;
    }
    else if (row == 2 && col == 1) {
        if (UsedInGrid(grid, num)
        || (abs(num - grid[row - 1][col - 1]) <= 1)
        || (abs(num - grid[row - 1][col]) <= 1)
        || (abs(num - grid[row - 1][col + 1]) <= 1)
        || (abs(num - grid[row][col + 1]) <= 1))
            return false;
    }
    else if (row == 2 && col == 2) {
        if (UsedInGrid(grid, num)
        || (abs(num - grid[row][col - 1]) <= 1)
        || (abs(num - grid[row - 1][col]) <= 1)
        || (abs(num - grid[row - 1][col + 1]) <= 1)
        || (abs(num - grid[row - 1][col - 1]) <= 1))
            return false;
    }
    else if (row == 1 && col == 1) {
        if (UsedInGrid(grid, num)
        || (abs(num - grid[row][col - 1]) <= 1)
        || (abs(num - grid[row - 1][col]) <= 1)
        || (abs(num - grid[row - 1][col + 1]) <= 1)
        || (abs(num - grid[row][col + 1]) <= 1)
```

```
        || (abs(num - grid[row + 1][col + 1]) <= 1)
        || (abs(num - grid[row + 1][col]) <= 1))
            return false;
    }
    else if (row == 1 && col == 2) {
        if (UsedInGrid(grid, num)
        || (abs(num - grid[row][col - 1]) <= 1)
        || (abs(num - grid[row - 1][col]) <= 1)
        || (abs(num - grid[row + 1][col - 1]) <= 1)
        || (abs(num - grid[row][col + 1]) <= 1)
        || (abs(num - grid[row - 1][col - 1]) <= 1)
        || (abs(num - grid[row + 1][col]) <= 1))
            return false;
    }
    return true;
}

// This function finds an entry
// in grid that is still unassigned
bool FindUnassignedLocation(int grid[N][M],
                        int& row, int& col)
{
    for (row = 0; row < N; row++)
        for (col = 0; col < M; col++) {
            if (grid[row][col] == UNASSIGNED)
                return true;
        }
    return false;
}

/* A utility function to print grid */
void printGrid(int grid[N][M])
{
    for (int i = 0; i < N; i++) {
        if (i == 0 || i == N - 1)
            cout << " ";
        for (int j = 0; j < M; j++) {
            if (grid[i][j] == 0)
                cout << " ";
            else
                cout << grid[i][j] << " ";
        }
        cout << endl;
    }
}

/* Takes a grid and attempts to assign values to
all unassigned locations in such a way to meet
```

```
the requirements for this solution.*/
bool Solve(int grid[N][M])
{
    int row, col;

    // If there is no unassigned location, we are done
    if (!FindUnassignedLocation(grid, row, col))
        return true; // success!

    // consider digits 1 to 8
    for (int num = 1; num <= 8; num++) {

        // if looks promising
        if (isSafe(grid, row, col, num)) {

            // make tentative assignment
            grid[row][col] = num;

            // return, if success, yay!
            if (Solve(grid))
                return true;

            // failure, unmake & try again
            grid[row][col] = UNASSIGNED;
        }
    }
    return false; // this triggers backtracking
}

/* Driver Program to test above functions */
int main()
{
    // -1 means unassigned cells
    int grid[N][M] = { { 0, -1, -1, 0 },
                    { -1, -1, -1, -1 },
                    { 0, -1, -1, 0 } };

    if (Solve(grid) == true)
        printGrid(grid);
    else
        cout << "Not possible";

    return 0;
}
```

Output:

```
  3 5
7 1 8 2
  4 6
```

Source

https://www.geeksforgeeks.org/fill-grid-1-8-numbers/

Chapter 29

Power Set in Lexicographic order

Power Set in Lexicographic order - GeeksforGeeks

This article is about generating Power set in lexicographical order.

Examples :

```
Input : abc
Output : a ab abc ac b bc c
```

The idea is to sort array first. After sorting, one by one fix characters and recursively generates all subsets starting from them. After every recursive call, we remove last character so that next permutation can be generated.

C++

```
 // CPP program to generate power set in
// lexicographic order.
#include <bits/stdc++.h>
using namespace std;

// str : Stores input string
// n : Length of str.
// curr : Stores current permutation
// index : Index in current permutation, curr
void permuteRec(string str, int n,
           int index = -1, string curr = "")
{
    // base case
```

```
    if (index == n)
        return;

    cout << curr << "\n";
    for (int i = index + 1; i < n; i++) {

        curr += str[i];
        permuteRec(str, n, i, curr);

        // backtracking
        curr = curr.erase(curr.size() - 1);
    }
    return;
}

// Generates power set in lexicographic
// order.
void powerSet(string str)
{
    sort(str.begin(), str.end());
    permuteRec(str, str.size());
}

// Driver code
int main()
{
    string str = "cab";
    powerSet(str);
    return 0;
}
```

PHP

```
 <?php
// PHP program to generate power
// set in lexicographic order.

// str : Stores input string
// n : Length of str.
// curr : Stores current permutation
// index : Index in current permutation, curr
function permuteRec($str, $n, $index = -1,
                              $curr = "")
{
    // base case
    if ($index == $n)
        return;
```

```
    echo $curr."\n";
    for ($i = $index + 1; $i < $n; $i++)
    {

        $curr=$curr.$str[$i];
        permuteRec($str, $n, $i, $curr);

        // backtracking
        $curr ="";
    }
    return;
}

// Generates power set in lexicographic
// order.
function powerSet($str)
{

    $str = str_split($str);
    sort($str);
    permuteRec($str, sizeof($str));
}

// Driver code
$str = "cab";
powerSet($str);

// This code is contributed by Mithun Kumar
?>
```

Output :

```

a
ab
abc
ac
b
bc
c
```

Improved By : Mithun Kumar

Source

https://www.geeksforgeeks.org/powet-set-lexicographic-order/

Chapter 30

Prime numbers after prime P with sum S

Prime numbers after prime P with sum S - GeeksforGeeks

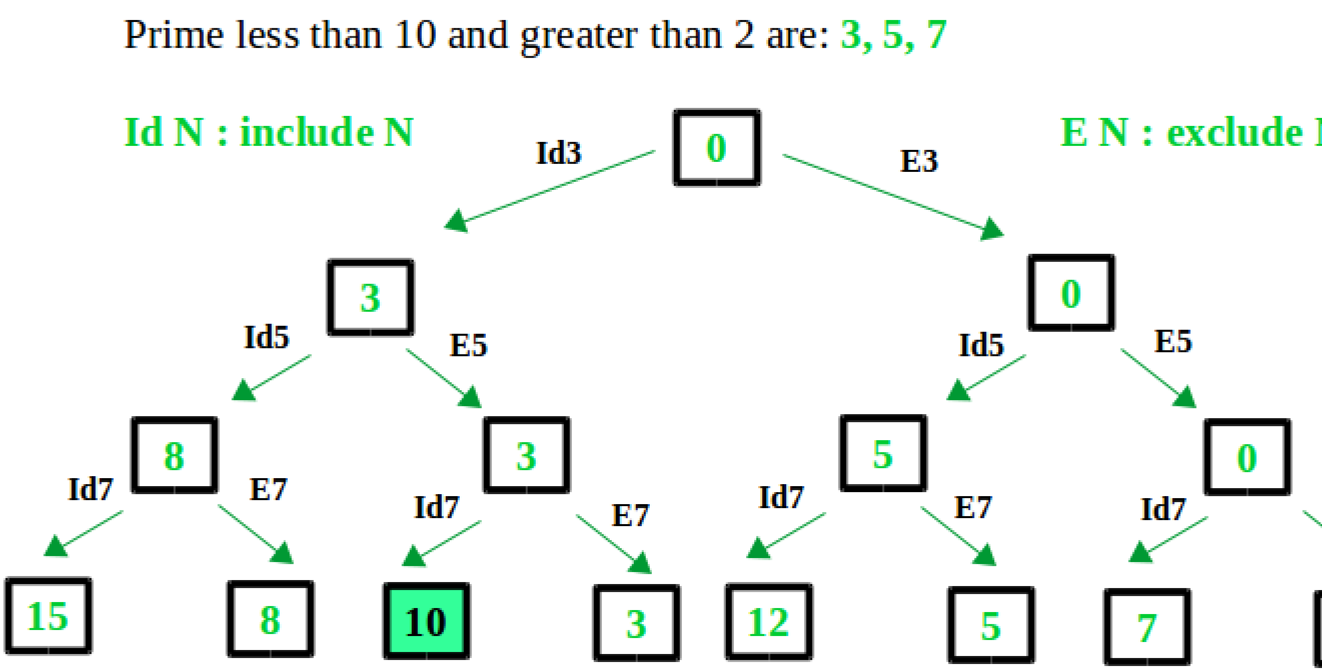
Given three numbers sum S, prime P and N, find all N prime numbers after prime P such that their sum is equal to S.

Examples :

```
Input :  N = 2, P = 7, S = 28
Output : 11 17
Explanation : 11 and 17 are primes after
prime 7 and (11 + 17 = 28)

Input :  N = 3, P = 2, S = 23
Output : 3 7 13
         5 7 11
Explanation : 3, 5, 7, 11 and 13 are primes
after prime 2. And (3 + 7 + 13 = 5 + 7 + 11
= 23)

Input :  N = 4, P = 3, S = 54
Output : 5 7 11 31
         5 7 13 29
         5 7 19 23
         5 13 17 19
         7 11 13 23
         7 11 17 19
Explanation : All are prime numbers and
their sum is 54
```

Approach : The approach used is to produce all the primes less than S and greater than P. And then backtracking to find if such N primes exist whose sum equals S.

For example, S = 10, N = 2, P = 2



C++

```
 // CPP Program to print all N primes after
// prime P whose sum equals S
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

// vector to store prime and N primes
// whose sum equals given S
vector<int> set;
vector<int> prime;

// function to check prime number
bool isPrime(int x)
{
    // square root of x
    int sqroot = sqrt(x);
    bool flag = true;
```

```
    // since 1 is not prime number
    if (x == 1)
        return false;

    // if any factor is found return false
    for (int i = 2; i <= sqroot; i++)
        if (x % i == 0)
            return false;

    // no factor found
    return true;
}

// function to display N primes whose sum equals S
void display()
{
    int length = set.size();
    for (int i = 0; i < length; i++)
        cout << set[i] << " ";
    cout << "\n";
}

// function to evaluate all possible N primes
// whose sum equals S
void primeSum(int total, int N, int S, int index)
{
    // if total equals S And
    // total is reached using N primes
    if (total == S && set.size() == N)
    {
        // display the N primes
        display();
        return;
    }

    // if total is greater than S
    // or if index has reached last element
    if (total > S || index == prime.size())
        return;

    // add prime[index] to set vector
    set.push_back(prime[index]);

    // include the (index)th prime to total
    primeSum(total+prime[index], N, S, index+1);

    // remove element from set vector
    set.pop_back();
```

```
    // exclude (index)th prime
    primeSum(total, N, S, index+1);
}

// function to generate all primes
void allPrime(int N, int S, int P)
{
    // all primes less than S itself
    for (int i = P+1; i <=S ; i++)
    {
        // if i is prime add it to prime vector
        if (isPrime(i))
            prime.push_back(i);
    }

    // if primes are less than N
    if (prime.size() < N)
        return;
    primeSum(0, N, S, 0);
}

// Driver Code
int main()
{
    int S = 54, N = 2, P = 3;
    allPrime(N, S, P);
    return 0;
}
```

Java

```
 // Java Program to print
// all N primes after prime
// P whose sum equals S
import java.io.*;
import java.util.*;

class GFG
{
    // vector to store prime
    // and N primes whose sum
    // equals given S
    static ArrayList<Integer> set =
                     new ArrayList<Integer>();
    static ArrayList<Integer> prime =
                     new ArrayList<Integer>();
```

```
// function to check
// prime number
static boolean isPrime(int x)
{
    // square root of x
    int sqroot = (int)Math.sqrt(x);

    // since 1 is not
    // prime number
    if (x == 1)
        return false;

    // if any factor is
    // found return false
    for (int i = 2;
             i <= sqroot; i++)
        if (x % i == 0)
            return false;

    // no factor found
    return true;
}

// function to display N
// primes whose sum equals S
static void display()
{
    int length = set.size();
    for (int i = 0;
             i < length; i++)
        System.out.print(
               set.get(i) + " ");
    System.out.println();
}

// function to evaluate
// all possible N primes
// whose sum equals S
static void primeSum(int total, int N,
                     int S, int index)
{
    // if total equals S
    // And total is reached
    // using N primes
    if (total == S &&
        set.size() == N)
    {
        // display the N primes
```

```
            display();
            return;
        }

        // if total is greater
        // than S or if index
        // has reached last
        // element
        if (total > S ||
            index == prime.size())
            return;

        // add prime.get(index)
        // to set vector
        set.add(prime.get(index));

        // include the (index)th
        // prime to total
        primeSum(total + prime.get(index),
                        N, S, index + 1);

        // remove element
        // from set vector
        set.remove(set.size() - 1);

        // exclude (index)th prime
        primeSum(total, N,
                S, index + 1);
    }

    // function to generate
    // all primes
    static void allPrime(int N,
                        int S, int P)
    {
        // all primes less
        // than S itself
        for (int i = P + 1;
                i <= S ; i++)
        {
            // if i is prime add
            // it to prime vector
            if (isPrime(i))
                prime.add(i);
        }

        // if primes are
        // less than N
```

```
        if (prime.size() < N)
            return;
        primeSum(0, N, S, 0);
    }

    // Driver Code
    public static void main(String args[])
    {
        int S = 54, N = 2, P = 3;
        allPrime(N, S, P);
    }
}

// This code is contributed by
// Manish Shaw(manishshaw1)
```

Python3

```
 # Python Program to print
# all N primes after prime
# P whose sum equals S
import math

# vector to store prime
# and N primes whose
# sum equals given S
set = []
prime = []

# function to
# check prime number
def isPrime(x) :

    # square root of x
    sqroot = int(math.sqrt(x))
    flag = True

    # since 1 is not
    # prime number
    if (x == 1) :
        return False

    # if any factor is
    # found return false
    for i in range(2, sqroot + 1) :
        if (x % i == 0) :
            return False
```

```
    # no factor found
    return True

# function to display N
# primes whose sum equals S
def display() :

    global set, prime
    length = len(set)
    for i in range(0, length) :
        print (set[i], end = " ")
    print ()

# function to evaluate
# all possible N primes
# whose sum equals S
def primeSum(total, N,
             S, index) :

    global set, prime

    # if total equals S
    # And total is reached
    # using N primes
    if (total == S and
         len(set) == N) :

        # display the N primes
        display()
        return

    # if total is greater
    # than S or if index
    # has reached last element
    if (total > S or
        index == len(prime)) :
        return

    # add prime[index]
    # to set vector
    set.append(prime[index])

    # include the (index)th
    # prime to total
    primeSum(total + prime[index],
                  N, S, index + 1)

    # remove element
```

```
    # from set vector
    set.pop()

    # exclude (index)th prime
    primeSum(total, N,
             S, index + 1)

# function to generate
# all primes
def allPrime(N, S, P) :

    global set, prime

    # all primes less
    # than S itself
    for i in range(P + 1,
                   S + 1) :

        # if i is prime add
        # it to prime vector
        if (isPrime(i)) :
            prime.append(i)

    # if primes are
    # less than N
    if (len(prime) < N) :
        return
    primeSum(0, N, S, 0)

# Driver Code
S = 54
N = 2
P = 3
allPrime(N, S, P)

# This code is contributed by
# Manish Shaw(manishshaw1)
```

C#

```
 // C# Program to print all
// N primes after prime P
// whose sum equals S
using System;
using System.Collections.Generic;

class GFG
{
```

```
// vector to store prime
// and N primes whose sum
// equals given S
static List<int> set = new List<int>();
static List<int> prime = new List<int>();

// function to check prime number
static bool isPrime(int x)
{
    // square root of x
    int sqroot = (int)Math.Sqrt(x);

    // since 1 is not prime number
    if (x == 1)
        return false;

    // if any factor is
    // found return false
    for (int i = 2; i <= sqroot; i++)
        if (x % i == 0)
            return false;

    // no factor found
    return true;
}

// function to display N
// primes whose sum equals S
static void display()
{
    int length = set.Count;
    for (int i = 0; i < length; i++)
        Console.Write(set[i] + " ");
    Console.WriteLine();
}

// function to evaluate
// all possible N primes
// whose sum equals S
static void primeSum(int total, int N,
                     int S, int index)
{
    // if total equals S And
    // total is reached using N primes
    if (total == S && set.Count == N)
    {
        // display the N primes
        display();
```

```
        return;
    }

    // if total is greater than
    // S or if index has reached
    // last element
    if (total > S || index == prime.Count)
        return;

    // add prime[index]
    // to set vector
    set.Add(prime[index]);

    // include the (index)th
    // prime to total
    primeSum(total + prime[index],
                     N, S, index + 1);

    // remove element
    // from set vector
    set.RemoveAt(set.Count - 1);

    // exclude (index)th prime
    primeSum(total, N, S, index + 1);
}

// function to generate
// all primes
static void allPrime(int N,
                     int S, int P)
{
    // all primes less than S itself
    for (int i = P + 1; i <=S ; i++)
    {
        // if i is prime add
        // it to prime vector
        if (isPrime(i))
            prime.Add(i);
    }

    // if primes are
    // less than N
    if (prime.Count < N)
        return;
    primeSum(0, N, S, 0);
}

// Driver Code
```

```
    static void Main()
    {
        int S = 54, N = 2, P = 3;
        allPrime(N, S, P);
    }
}

// This code is contributed by
// Manish Shaw(manishshaw1)
```

PHP

```
 <?php
// PHP Program to print all
// N primes after prime P
// whose sum equals S

// vector to store prime
// and N primes whose
// sum equals given S
$set = array();
$prime = array();

// function to
// check prime number
function isPrime($x)
{
    // square root of x
    $sqroot = sqrt($x);
    $flag = true;

    // since 1 is not
    // prime number
    if ($x == 1)
        return false;

    // if any factor is
    // found return false
    for ($i = 2; $i <= $sqroot; $i++)
        if ($x % $i == 0)
            return false;

    // no factor found
    return true;
}

// function to display N
// primes whose sum equals S
```

```
function display()
{
    global $set, $prime;
    $length = count($set);
    for ($i = 0; $i < $length; $i++)
        echo ($set[$i] . " ");
    echo ("\n");
}

// function to evaluate
// all possible N primes
// whose sum equals S
function primeSum($total, $N,
                  $S, $index)
{
    global $set, $prime;

    // if total equals S
    // And total is reached
    // using N primes
    if ($total == $S &&
        count($set) == $N)
    {
        // display the N primes
        display();
        return;
    }

    // if total is greater
    // than S or if index
    // has reached last element
    if ($total > $S ||
        $index == count($prime))
        return;

    // add prime[index]
    // to set vector
    array_push($set,
               $prime[$index]);

    // include the (index)th
    // prime to total
    primeSum($total + $prime[$index],
             $N, $S, $index + 1);

    // remove element
    // from set vector
    array_pop($set);
```

```
    // exclude (index)th prime
    primeSum($total, $N, $S,
             $index + 1);
}

// function to generate
// all primes
function allPrime($N, $S, $P)
{
    global $set, $prime;

    // all primes less
    // than S itself
    for ($i = $P + 1;
         $i <= $S ; $i++)
    {
        // if i is prime add
        // it to prime vector
        if (isPrime($i))
            array_push($prime, $i);
    }

    // if primes are
    // less than N
    if (count($prime) < $N)
        return;
    primeSum(0, $N, $S, 0);
}

// Driver Code
$S = 54; $N = 2; $P = 3;
allPrime($N, $S, $P);

// This code is contributed by
// Manish Shaw(manishshaw1)
?>
```

Output:

```
7 47
11 43
13 41
17 37
23 31
```

Optimizations :

The above solution can be optimized by pre-computing all required primes usingSieve of Eratosthenes

Improved By : manishshaw1

Source

https://www.geeksforgeeks.org/prime-numbers-after-prime-p-with-sum-s/

Chapter 31

Smallest expression to represent a number using single digit

Smallest expression to represent a number using single digit - GeeksforGeeks

Given a number N and a digit D, we have to form an expression or equation that contains only D and that expression evaluates to N. Allowed operators in expression are **+, -, *, and /** . Find the minimum length expression that satisfy the condition above and D can only appear in the expression at most 10(limit) times. Hence limit the values of N (Although the value of limit depends upon how far you want to go. But a large value of limit can take longer time for below approach).

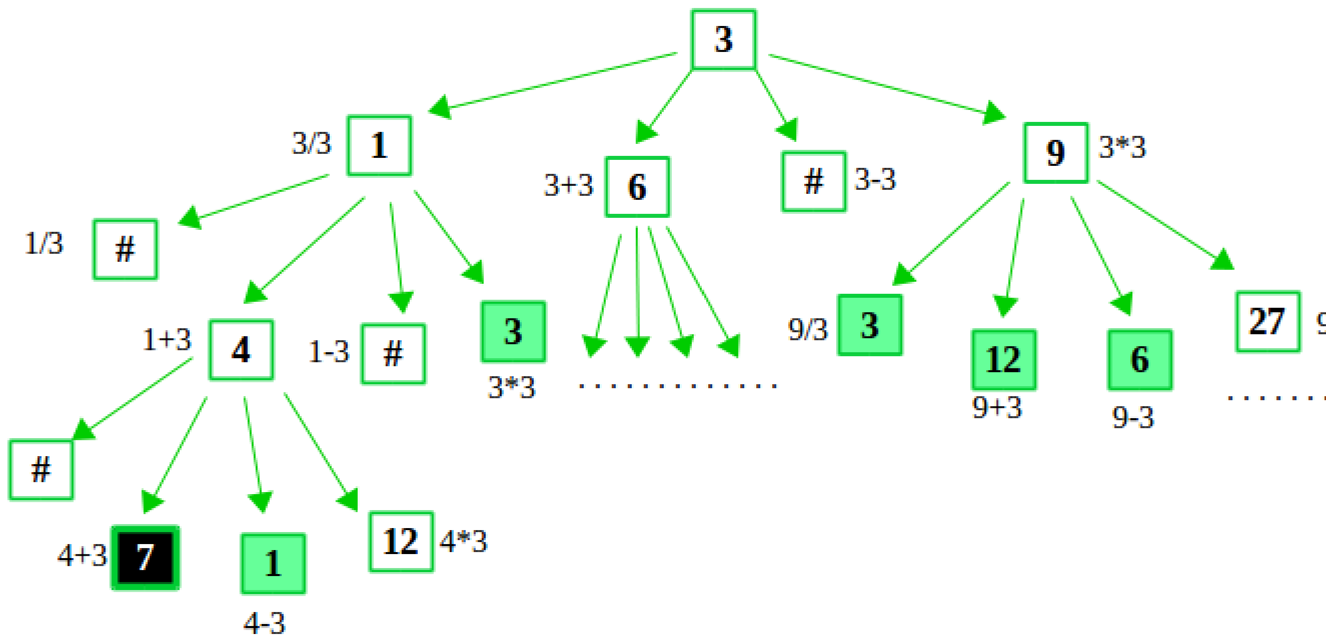
Remember, there can be more than one minimum expression of D that evaluates to N but the length of that expression will be minimum.

Examples:

```
Input :  N = 7, D = 3
Output : 3/3+ 3 + 3
Explanation : 3/3 = 1, and 1+3+3 = 7
This is the minimum expression.

Input :  N = 7, D = 4
Output : (4+4+4)/4 + 4
Explanation : (4+4+4) = 12, and 12/4 = 3 and 3+4 = 7
Also this is the minimum expression. Although
you may find another expression but that
expression can have only five 4's

Input :  N = 200, D = 9
Output : Expression not found!
Explanation : Not possible within 10 digits.
```

The approach we use is **Backtracking**. We start with the given Digit D and start multiplying, adding, subtracting, and dividing if possible. This process is done until we find the total as N or we reach end and we backtrack to start another path. To find the minimum expression, we find the minimum level of the recursive tree. And then apply our backtracking algorithm.

Let's say N = 7, D = 3



The above approach is exponential. At every level, we recurse 4 more ways (at-most). So, we can say the time complexity of the method is [illegible] where n is the number of levels in recursive tree (or we can say the number of times we want D to appear at-most in the expression which in our case is 10).

Note: We use the above approach two times. First to find minimum level and then to find the expression that is possible in that level. So, we have two passes in this approach. Although we can get the expression in one go, but you'll need to scratch your head for that.

```
 // CPP Program to generate minimum expression containing
// only given digit D that evaluates to number N.
#include <climits>
#include <iostream>
#include <map>
#include <sstream>
#include <stack>

// limit of Digits in the expression
```

```
#define LIMIT 10

using namespace std;

// map that store if a number is seen or not
map<int, int> seen;

// stack for storing operators
stack<char> operators;
int minimum = LIMIT;

// function to find minimum levels in the recursive tree
void minLevel(int total, int N, int D, int level) {

  // if total is equal to given N
  if (total == N) {

    // store if level is minimum
    minimum = min(minimum, level);
    return;
  }

  // if the last level is reached
  if (level == minimum)
    return;

  // if total can be divided by D.
  // recurse by dividing the total by D
  if (total % D == 0)
    minLevel(total / D, N, D, level + 1);

  // recurse for total + D
  minLevel(total + D, N, D, level + 1);

  // if total - D is greater than 0
  if (total - D > 0)

    // recurse for total - D
    minLevel(total - D, N, D, level + 1);

  // recurse for total multiply D
  minLevel(total * D, N, D, level + 1);
}

// function to generate the minimum expression
bool generate(int total, int N, int D, int level) {
  // if total is equal to N
  if (total == N)
```

```
    return true;

  // if the last level is reached
  if (level == minimum)
    return false;

  // if total is seen at level greater than current level
  // or if we haven't seen total before. Mark the total
  // as seen at current level
  if (seen.find(total) == seen.end() ||
      seen.find(total)->second >= level) {

    seen[total] = level;

    int divide = INT_MAX;

    // if total is divisible by D
    if (total % D == 0) {
      divide = total / D;

      // if divide isn't seen before
      // mark it as seen
      if (seen.find(divide) == seen.end())
        seen[divide] = level + 1;
    }

    int addition = total + D;

    // if addition isn't seen before
    // mark it as seen
    if (seen.find(addition) == seen.end())
      seen[addition] = level + 1;

    int subtraction = INT_MAX;
    // if D can be subtracted from total
    if (total - D > 0) {
      subtraction = total - D;

      // if subtraction isn't seen before
      // mark it as seen
      if (seen.find(subtraction) == seen.end())
        seen[subtraction] = level + 1;
    }

    int multiply = total * D;

    // if multiply isn't seen before
    // mark it as seen
```

```
    if (seen.find(multiply) == seen.end())
      seen[multiply] = level + 1;

    // recurse by dividing the total if possible
    if (divide != INT_MAX)
      if (generate(divide, N, D, level + 1)) {

        // store the operator.
        operators.push('/');
        return true;
      }

    // recurse by adding D to total
    if (generate(addition, N, D, level + 1)) {

      // store the operator.
      operators.push('+');
      return true;
    }

    // recurse by subtracting D from total
    if (subtraction != INT_MAX)
      if (generate(subtraction, N, D, level + 1)) {

        // store the operator.
        operators.push('-');
        return true;
      }

    // recurse by multiplying D by total
    if (generate(multiply, N, D, level + 1)) {

      // store the operator.
      operators.push('*');
      return true;
    }
  }

  // expression is not found yet
  return false;
}

// function to print the expression
void printExpression(int N, int D) {
  // find minimum level
  minLevel(D, N, D, 1);

  // generate expression if possible
```

```
  if (generate(D, N, D, 1)) {
    // stringstream for converting to D to string
    ostringstream num;
    num << D;

    string expression;

    // if stack is not empty
    if (!operators.empty()) {

      // concatenate D and operator at top of stack
      expression = num.str() + operators.top();
      operators.pop();
    }

    // until stack is empty
    // concatenate the operator with parenthesis for precedence
    while (!operators.empty()) {
      if (operators.top() == '/' || operators.top() == '*')
        expression = "(" + expression + num.str() + ")" + operators.top();
      else
        expression = expression + num.str() + operators.top();
      operators.pop();
    }

    expression = expression + num.str();

    // print the expression
    cout << "Expression: " << expression << endl;
  }

  // not possible within 10 digits.
  else
    cout << "Expression not found!" << endl;
}

// Driver's Code
int main() {
  int N = 7, D = 4;

  // print the Expression if possible
  printExpression(N, D);

  // print expression for N =100, D =7
  minimum = LIMIT;
  printExpression(100, 7);

  // print expression for N =200, D =9
```

```
  minimum = LIMIT;
  printExpression(200, 9);

  return 0;
}
```

Output:

```
Expression: (4+4+4)/4+4
Expression: (((7+7)*7)*7+7+7)/7
Expression not found!
```

Source

https://www.geeksforgeeks.org/smallest-expression-represent-number-using-single-digit/

Chapter 32

Count all possible paths between two vertices

Count all possible paths between two vertices - GeeksforGeeks

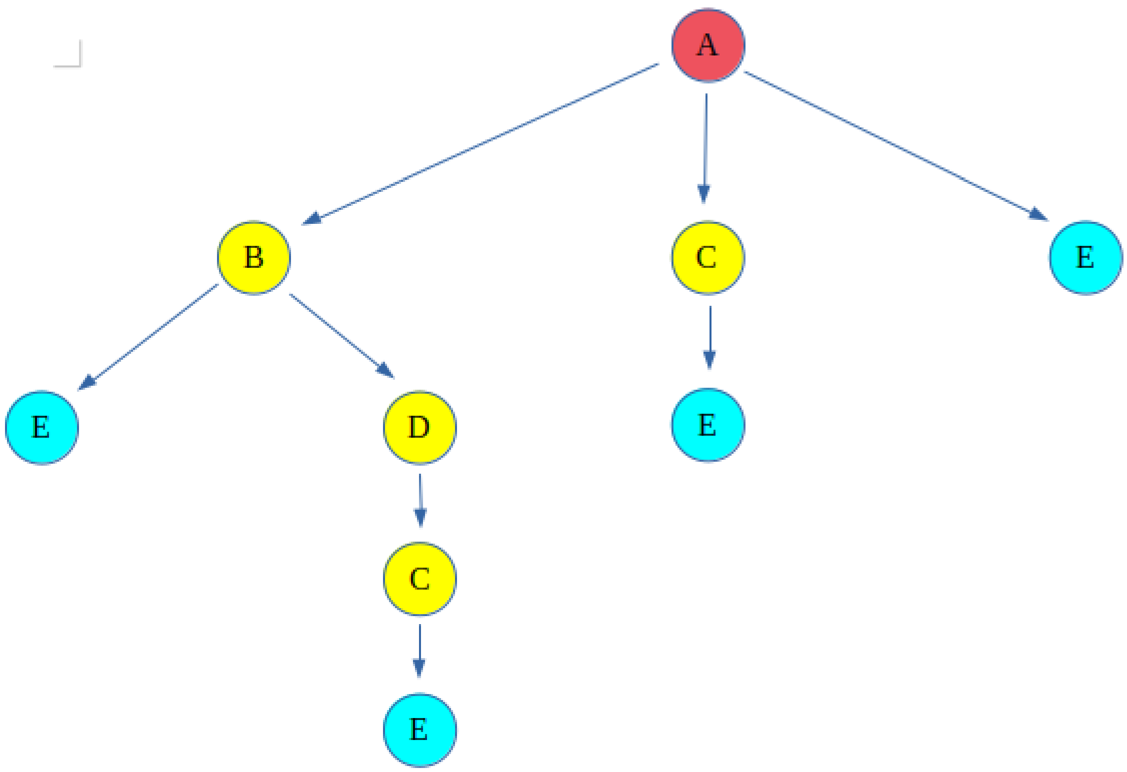
Count the total number of ways or paths that exist between two vertices in a directed graph. These paths doesn't contain a cycle, the simple enough reason is that a cylce contain infinite number of paths and hence they create problem.

Examples:

```
Input : Count paths between A and E


Output : Total paths between A and E are 4
         Explanation: The 4 paths between A and E are:
                      A -> E
                      A -> B -> E
                      A -> C -> E
                      A -> B -> D -> C -> E
```

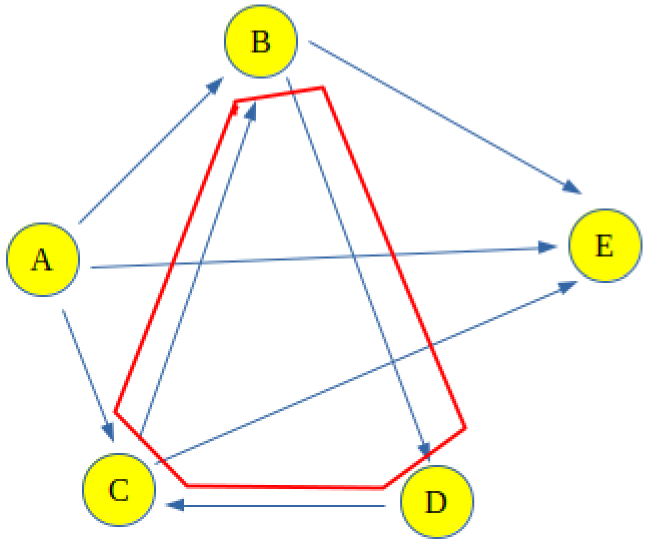
The problem can be solved using backtracking, that is we take a path and start walking it, if it leads us to the destination vertex then we count the path and backtrack to take another path. If the path doesn't leads us to the destination vertex, we discard the path.

Backtracking for above graph can be shown like this:
The red color vertex is the source vertex and the light-blue color vertex is destination, rest are either intermediate or discarded paths.



This gives us four paths between **source(A)** and **destination(E)** vertex.

Problem Associated with this: Now if we add just one more edge between C and B, it would make a cycle **(B -> D -> C -> B)**. And hence we could loop the cycles any number of times to get a new path, and there would be infinitely many paths because of the cycle.



C++

```
 // C++ program to count all paths from a
// source to a destination.
#include<bits/stdc++.h>

using namespace std;

// A directed graph using adjacency list
// representation
class Graph
{

    // No. of vertices in graph
    int V;
    list<int> *adj;

    // A recursive function
    // used by countPaths()
    void countPathsUtil(int, int, bool [],
                              int &);

public:

    // Constructor
    Graph(int V);
    void addEdge(int u, int v);
    int countPaths(int s, int d);
```

```
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int u, int v)
{

    // Add v to u’s list.
    adj[u].push_back(v);
}

// Returns count of paths from 's' to 'd'
int Graph::countPaths(int s, int d)
{

    // Mark all the vertices
    // as not visited
    bool *visited = new bool[V];
    memset(visited, false, sizeof(visited));

    // Call the recursive helper
    // function to print all paths
    int pathCount = 0;
    countPathsUtil(s, d, visited, pathCount);
    return pathCount;
}

// A recursive function to print all paths
// from 'u' to 'd'. visited[] keeps track of
// vertices in current path. path[] stores
// actual vertices and path_index is
// current index in path[]
void Graph::countPathsUtil(int u, int d, bool visited[],
                                        int &pathCount)
{
    visited[u] = true;

    // If current vertex is same as destination,
    // then increment count
    if (u == d)
        pathCount++;

    // If current vertex is not destination
    else
```

```
    {
        // Recur for all the vertices adjacent to
        // current vertex
        list<int>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (!visited[*i])
                countPathsUtil(*i, d, visited,
                                      pathCount);
    }

    visited[u] = false;
}

// Driver Code
int main()
{

    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(0, 3);
    g.addEdge(2, 0);
    g.addEdge(2, 1);
    g.addEdge(1, 3);

    int s = 2, d = 3;
    cout << g.countPaths(s, d);

    return 0;
}
```

Java

```
 // Java program to count all paths from a source
// to a destination.
import java.util.Arrays;
import java.util.Iterator;
import java.util.LinkedList;

// This class represents a directed graph using
// adjacency list representation

class Graph {

    // No. of vertices
    private int V;
```

```
// Array of lists for
// Adjacency List
// Representation
private LinkedList<Integer> adj[];

@SuppressWarnings("unchecked")
Graph(int v)
{
    V = v;
    adj = new LinkedList[v];
    for (int i = 0; i < v; ++i)
        adj[i] = new LinkedList<>();
}

// Method to add an edge into the graph
void addEdge(int v, int w)
{

    // Add w to v's list.
    adj[v].add(w);
}


// A recursive method to count
// all paths from 'u' to 'd'.
int countPathsUtil(int u, int d,
                   boolean visited[],
                   int pathCount)
{

    // Mark the current node as
    // visited and print it
    visited[u] = true;

    // If current vertex is same as
    // destination, then increment count
    if (u == d)
    {
        pathCount++;
    }

    // Recur for all the vertices
    // adjacent to this vertex
    else
    {
        Iterator<Integer> i = adj[u].listIterator();
        while (i.hasNext())
        {
```

```
                int n = i.next();
                if (!visited[n])
                {
                    pathCount = countPathsUtil(n, d,
                                                visited,
                                                pathCount);
                }
            }
        }

        visited[u] = false;
        return pathCount;
    }

    // Returns count of
    // paths from 's' to 'd'
    int countPaths(int s, int d)
    {

        // Mark all the vertices
        // as not visited
        boolean visited[] = new boolean[V];
        Arrays.fill(visited, false);

        // Call the recursive method
        // to count all paths
        int pathCount = 0;
        pathCount = countPathsUtil(s, d,
                                    visited,
                                    pathCount);
        return pathCount;
    }

    // Driver Code
    public static void main(String args[])
    {
        Graph g = new Graph(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(0, 3);
        g.addEdge(2, 0);
        g.addEdge(2, 1);
        g.addEdge(1, 3);

        int s = 2, d = 3;
        System.out.println(g.countPaths(s, d));
    }
}
```

```
// This code is contributed by shubhamjd.
```

Output:

```
3
```

Improved By : shubhamjd, IshaanKanwar

Source

https://www.geeksforgeeks.org/count-possible-paths-two-vertices/

Chapter 33

Check if a given string is sum-string

Check if a given string is sum-string - GeeksforGeeks

Given a string of digits, determine whether it is a 'sum-string'. A string S is called a sum-string if a rightmost substring can be written as sum of two substrings before it and same is recursively true for substrings before it.

Examples :

```
"12243660" is a sum string.
Explanation : 24 + 36 = 60, 12 + 24 = 36

"1111112223" is a sum string.
Explanation: 111+112 = 223, 1+111 = 112

"2368" is not a sum string
```

In general a string S is called sum-string if it satisfies the following properties:

```
sub-string(i, x) + sub-string(x+1, j)
 = sub-string(j+1, l)
and
sub-string(x+1, j)+sub-string(j+1, l)
 = sub-string(l+1, m)
and so on till end.
```

From the examples, we can see that our decision depends on first two chosen numbers.
So we choose all possible first two number for given string. Then for every chosen two

numbers we check whether it is sum-string or not? So the approach is very simple. We generate all possible first two numbers using two substrings s1 and s2 using two loops. then we check whether it is possible to make number s3 = (s1 + s2) or not. If we can make s3 then we recursively check for s2 + s3 so on.

```
 // C++ program to check if a given string
// is sum-string or not
#include <bits/stdc++.h>
using namespace std;

// this is function for finding sum of two
// numbers as string
string string_sum(string str1, string str2)
{
    if (str1.size() < str2.size())
       swap(str1, str2);

    int m = str1.size();
    int n = str2.size();
    string ans = "";

    // sum the str2 with str1
    int carry = 0;
    for (int i = 0; i < n; i++) {

        // Sum of current digits
        int ds = ((str1[m - 1 - i] - '0') +
                  (str2[n - 1 - i] - '0') +
                  carry) % 10;

        carry = ((str1[m - 1 - i] - '0') +
                 (str2[n - 1 - i] - '0') +
                 carry) / 10;

        ans = char(ds + '0') + ans;
    }

    for (int i = n; i < m; i++) {
        int ds = (str1[m - 1 - i] - '0' +
                  carry) % 10;
        carry = (str1[m - 1 - i] - '0' +
                 carry) / 10;
        ans = char(ds + '0') + ans;
    }

    if (carry)
        ans = char(carry + '0') + ans;
    return ans;
```

```
}

// Returns true of two substrings of ginven
// lengths of str[beg..] can cause a positive
// result.
bool checkSumStrUtil(string str, int beg,
                     int len1, int len2)
{

    // Finding two substrings of given lengths
    // and their sum
    string s1 = str.substr(beg, len1);
    string s2 = str.substr(beg + len1, len2);
    string s3 = string_sum(s1, s2);

    int s3_len = s3.size();

    // if number of digits s3 is greater then
    // the available string size
    if (s3_len > str.size() - len1 - len2 - beg)
        return false;

    // we got s3 as next number in main string
    if (s3 == str.substr(beg + len1 + len2, s3_len)) {

        // if we reach at the end of the string
        if (beg + len1 + len2 + s3_len == str.size())
            return true;

        // otherwise call recursively for n2, s3
        return checkSumStrUtil(str, beg + len1, len2,
                                          s3_len);
    }

    // we do not get s3 in main string
    return false;
}

// Returns true if str is sum string, else false.
bool isSumStr(string str)
{
    int n = str.size();

    // choosing first two numbers and checking
    // whether it is sum-string or not.
    for (int i = 1; i < n; i++)
        for (int j = 1; i + j < n; j++)
            if (checkSumStrUtil(str, 0, i, j))
```

```
                return true;

    return false;
}

// Driver code
int main()
{
    cout << isSumStr("1212243660") << endl;
    cout << isSumStr("123456787");
    return 0;
}
```

Output:

```
1
0
```

Source

https://www.geeksforgeeks.org/check-given-string-sum-string/

Chapter 34

Print all possible strings that can be made by placing spaces

Print all possible strings that can be made by placing spaces - GeeksforGeeks

Given a string you need to print all possible strings that can be made by placing spaces (zero or one) in between them.

```
Input:  str[] = "ABC"
Output: ABC
        AB C
        A BC
        A B C
```

Source: Amazon Interview Experience | Set 158, Round 1 ,Q 1.

The idea is to use recursion and create a buffer that one by one contains all output strings having spaces. We keep updating buffer in every recursive call. If the length of given string is 'n' our updated string can have maximum length of n + (n-1) i.e. 2n-1. So we create buffer size of 2n (one extra character for string termination).
We leave 1st character as it is, starting from the 2nd character, we can either fill a space or a character. Thus one can write a recursive function like below.

C/C++

```
 // C++ program to print permutations of a given string with spaces.
#include <iostream>
#include <cstring>
using namespace std;

/* Function recursively prints the strings having space pattern.
   i and j are indices in 'str[]' and 'buff[]' respectively */
```

```
void printPatternUtil(char str[], char buff[], int i, int j, int n)
{
    if (i==n)
    {
        buff[j] = '\0';
        cout << buff << endl;
        return;
    }

    // Either put the character
    buff[j] = str[i];
    printPatternUtil(str, buff, i+1, j+1, n);

    // Or put a space followed by next character
    buff[j] = ' ';
    buff[j+1] = str[i];

    printPatternUtil(str, buff, i+1, j+2, n);
}

// This function creates buf[] to store individual output string and uses
// printPatternUtil() to print all permutations.
void printPattern(char *str)
{
    int n = strlen(str);

    // Buffer to hold the string containing spaces
    char buf[2*n]; // 2n-1 characters and 1 string terminator

    // Copy the first character as it is, since it will be always
    // at first position
    buf[0] = str[0];

    printPatternUtil(str, buf, 1, 1, n);
}

// Driver program to test above functions
int main()
{
    char *str = "ABCD";
    printPattern(str);
    return 0;
}
```

Java

```
 // Java program to print permutations of a given string with spaces
import java.io.*;
```

```
class Permutation
{
    // Function recursively prints the strings having space pattern
    // i and j are indices in 'String str' and 'buf[]' respectively
    static void printPatternUtil(String str, char buf[], int i, int j, int n)
    {
        if(i == n)
        {
            buf[j] = '\0';
            System.out.println(buf);
            return;
        }

        // Either put the character
        buf[j] = str.charAt(i);
        printPatternUtil(str, buf, i+1, j+1, n);

        // Or put a space followed by next character
        buf[j] = ' ';
        buf[j+1] = str.charAt(i);

        printPatternUtil(str, buf, i+1, j+2, n);
    }

    // Function creates buf[] to store individual output string and uses
    // printPatternUtil() to print all permutations
    static void printPattern(String str)
    {
        int len = str.length();

        // Buffer to hold the string containing spaces
        // 2n-1 characters and 1 string terminator
        char[] buf = new char[2*len];

        // Copy the first character as it is, since it will be always
        // at first position
        buf[0] = str.charAt(0);
        printPatternUtil(str, buf, 1, 1, len);
    }

    // Driver program
    public static void main (String[] args)
    {
        String str = "ABCD";
        printPattern(str);
    }
}
```

Python

```
 # Python program to print permutations of a given string with
# spaces.

# Utility function
def toString(List):
    s = ""
    for x in List:
        if x == '\0':
            break
        s += x
    return s

# Function recursively prints the strings having space pattern.
# i and j are indices in 'str[]' and 'buff[]' respectively
def printPatternUtil(string, buff, i, j, n):
    if i == n:
        buff[j] = '\0'
        print toString(buff)
        return

    # Either put the character
    buff[j] = string[i]
    printPatternUtil(string, buff, i+1, j+1, n)

    # Or put a space followed by next character
    buff[j] = ' '
    buff[j+1] = string[i]

    printPatternUtil(string, buff, i+1, j+2, n)

# This function creates buf[] to store individual output string
# and uses printPatternUtil() to print all permutations.
def printPattern(string):
    n = len(string)

    # Buffer to hold the string containing spaces
    buff = [0] * (2*n) # 2n-1 characters and 1 string terminator

    # Copy the first character as it is, since it will be always
    # at first position
    buff[0] = string[0]

    printPatternUtil(string, buff, 1, 1, n)

# Driver program
string = "ABCD"
```

```
printPattern(string)

# This code is contributed by BHAVYA JAIN
```

C#

```
 // C# program to print permutations of a
// given string with spaces
using System;

class GFG {

    // Function recursively prints the
    // strings having space pattern
    // i and j are indices in 'String
    // str' and 'buf[]' respectively
    static void printPatternUtil(string str,
             char []buf, int i, int j, int n)
    {
        if(i == n)
        {
            buf[j] = '\0';
            Console.WriteLine(buf);
            return;
        }

        // Either put the character
        buf[j] = str[i];
        printPatternUtil(str, buf, i+1, j+1, n);

        // Or put a space followed by next
        // character
        buf[j] = ' ';
        buf[j+1] = str[i];

        printPatternUtil(str, buf, i+1, j+2, n);
    }

    // Function creates buf[] to store
    // individual output string and uses
    // printPatternUtil() to print all
    // permutations
    static void printPattern(string str)
    {
        int len = str.Length;

        // Buffer to hold the string containing
        // spaces 2n-1 characters and 1 string
```

```
        // terminator
        char []buf = new char[2*len];

        // Copy the first character as it is,
        // since it will be always at first
        // position
        buf[0] = str[0];
        printPatternUtil(str, buf, 1, 1, len);
    }

    // Driver program
    public static void Main ()
    {
        string str = "ABCD";
        printPattern(str);
    }
}

// This code is contributed by nitin mittal.
```

Output:

```
ABCD
ABC D
AB CD
AB C D
A BCD
A BC D
A B CD
A B C D
```

Time Complexity: Since number of Gaps are n-1, there are total 2^(n-1) patters each having length ranging from n to 2n-1. Thus overall complexity would be O(n*(2^n)).

This article is contributed by **Gaurav Sharma**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : nitin mittal

Source

https://www.geeksforgeeks.org/print-possible-strings-can-made-placing-spaces/

Chapter 35

Combinational Sum

Combinational Sum - GeeksforGeeks

Given an array of positive integers **arr[]** and a sum **x**, find all unique combinations in arr[] where the sum is equal to x. The same repeated number may be chosen from arr[] unlimited number of times. Elements in a combination (a1, a2, …, ak) must be printed in non-descending order. (ie, a1 <= a2 <= … <= ak).
The combinations themselves must be sorted in ascending order, i.e., the combination with smallest first element should be printed first. If there is no combination possible the print "Empty" (without quotes).

Examples:

```
Input : arr[] = 2, 4, 6, 8
            x = 8
Output : [2, 2, 2, 2]
         [2, 2, 4]
         [2, 6]
         [4, 4]
         [8]
```

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 if negative then ignore that
    sub-problem.
 (C) Else insert the present array in that
    index to the current vector and call
    the function with sum = sum-ar[index] and
    index = index, then pop that element from
    current index (backtrack) and call the
    function with sum = sum and index = index+1
```

Below is C++ implementation of 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  current sum becomes negative
    if (sum < 0)
        return;

    // 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 )
```

Source

https://www.geeksforgeeks.org/combinational-sum/

Chapter 36

Combinations where every element appears twice and distance between appearances is equal to the value

Combinations where every element appears twice and distance between appearances is equal to the value - GeeksforGeeks

Given a positive number n, we need to find all the combinations of 2*n elements such that every element from 1 to n appears exactly twice and distance between its appearances is exactly equal to value of the element.

Examples:

```
Input :  n = 3
Output : 3 1 2 1 3 2
         2 3 1 2 1 3
All elements from 1 to 3 appear
twice and distance between two
appearances is equal to value
of the element.

Input :  n = 4
Output : 4 1 3 1 2 4 3 2
         2 3 4 2 1 3 1 4
```

Explanation

We can use backtracking to solve this problem. The idea is to all possible combinations for the first element and recursively explore remaining element to check if they will lead to

the solution or not. If current configuration doesn't result in solution, we backtrack. Note that an element k can be placed at position i and (i+k+1) in the output array i >= 0 and (i+k+1) < 2*n.
Note that no combination of element is possible for some value of n like 2, 5, 6 etc.

C++

```
 // C++ program to find all combinations where every
// element appears twice and distance between
// appearances is equal to the value
#include <bits/stdc++.h>
using namespace std;

// Find all combinations that satisfies given constraints
void allCombinationsRec(vector<int> &arr, int elem, int n)
{
    // if all elements are filled, print the solution
    if (elem > n)
    {
        for (int i : arr)
            cout << i << " ";
        cout << endl;

        return;
    }

    // try all possible combinations for element elem
    for (int i = 0; i < 2*n; i++)
    {
        // if position i and (i+elem+1) are not occupied
        // in the vector
        if (arr[i] == -1 && (i + elem + 1) < 2*n &&
                arr[i + elem + 1] == -1)
        {
            // place elem at position i and (i+elem+1)
            arr[i] = elem;
            arr[i + elem + 1] = elem;

            // recurse for next element
            allCombinationsRec(arr, elem + 1, n);

            // backtrack (remove elem from position i and (i+elem+1) )
            arr[i] = -1;
            arr[i + elem + 1] = -1;
        }
    }
}
```

```
void allCombinations(int n)
{
    // create a vector of double the size of given number with
    vector<int> arr(2*n, -1);

    // all its elements initialized by 1
    int elem = 1;

    // start from element 1
    allCombinationsRec(arr, elem, n);
}

// Driver code
int main()
{
    // given number
    int n = 3;
    allCombinations(n);
    return 0;
}
```

Java

```
 // Java program to find all combinations where every
// element appears twice and distance between
// appearances is equal to the value

import java.util.Vector;

class Test
{
    // Find all combinations that satisfies given constraints
    static void allCombinationsRec(Vector<Integer> arr, int elem, int n)
    {
        // if all elements are filled, print the solution
        if (elem > n)
        {
            for (int i : arr)
                System.out.print(i + " ");
            System.out.println();

            return;
        }

        // try all possible combinations for element elem
        for (int i = 0; i < 2*n; i++)
        {
            // if position i and (i+elem+1) are not occupied
```

```
            // in the vector
            if (arr.get(i) == -1 && (i + elem + 1) < 2*n &&
                    arr.get(i + elem + 1) == -1)
            {
                // place elem at position i and (i+elem+1)
                arr.set(i, elem);
                arr.set(i + elem + 1, elem);

                // recurse for next element
                allCombinationsRec(arr, elem + 1, n);

                // backtrack (remove elem from position i and (i+elem+1) )
                arr.set(i, -1);
                arr.set(i + elem + 1, -1);
            }
        }
    }

    static void allCombinations(int n)
    {

        // create a vector of double the size of given number with
        Vector<Integer> arr = new Vector<>();

        for (int i = 0; i < 2*n; i++) {
            arr.add(-1);
        }

        // all its elements initialized by 1
        int elem = 1;

        // start from element 1
        allCombinationsRec(arr, elem, n);
    }

    // Driver method
    public static void main(String[] args)
    {
        // given number
        int n = 3;
        allCombinations(n);
    }
}
```

Python3

```
 # Python3 program to find all combinations
# where every element appears twice and distance
```

```
# between appearances is equal to the value

# Find all combinations that
# satisfies given constraints
def allCombinationsRec(arr, elem, n):

    # if all elements are filled,
    # print the solution
    if (elem > n):

        for i in (arr):
            print(i, end = " ")

        print("")
        return

    # Try all possible combinations
    # for element elem
    for i in range(0, 2 * n):

        # if position i and (i+elem+1) are
        # not occupied in the vector
        if (arr[i] == -1 and
           (i + elem + 1) < 2*n and
           arr[i + elem + 1] == -1):

            # place elem at position
            # i and (i+elem+1)
            arr[i] = elem
            arr[i + elem + 1] = elem

            # recurse for next element
            allCombinationsRec(arr, elem + 1, n)

            # backtrack (remove elem from
            # position i and (i+elem+1) )
            arr[i] = -1
            arr[i + elem + 1] = -1

def allCombinations(n):

    # create a vector of double
    # the size of given number with
    arr = [-1] * (2 * n)

    # all its elements initialized by 1
    elem = 1
```

```
    # start from element 1
    allCombinationsRec(arr, elem, n)

# Driver code
n = 3
allCombinations(n)

# This code is contributed by Smitha Dinesh Semwal.
```

Output:

```
 3 1 2 1 3 2
 2 3 1 2 1 3
```

Source

https://www.geeksforgeeks.org/combinations-every-element-appears-twice-distance-appearances-equal-value/

Chapter 37

Print all palindromic partitions of a string

Print all palindromic partitions of a string - GeeksforGeeks

Given a string s, partition s such that every string of the partition is a palindrome. Return all possible palindrome partitioning of s.

Example :

```
Input  : s = "bcc"
Output : [["b", "c", "c"], ["b", "cc"]]

Input  : s = "geeks"
Output : [["g", "e", "e", "k", "s"],
          ["g", "ee", "k", "s"]]
```

We have to list the all possible partitions so we will think in the direction of recursion. When we are on index i, we incrementally check all substrings starting from i for being palindromic. If found, we recursively solve the problem for the remaining string and add this in our solution.

Following is the solution-

1. We will maintain a 2-dimensional vector for storing all possible partitions and a temporary vector for storing the current partition, new starting index of string to check partitions as we have already checked partitions before this index.
2. Now keep on iterating further on string and check if it is palindrome or not.
3. If it is a palindrome than add this string in current partitions vector. Recurse on this new string if it is not the end of the string. After coming back again change the current partition vector to the old one as it might have changed in the recursive step.

4. If we reach the end of string while iterating than we have our partitions in our temporary vector so we will add it in results.

To check whether it's a palindrome or not, iterate on string by taking two pointers. Initialize the first to start and other to end of string. If both characters are same increase the first and decrease the last pointer and keep on iterating until first is less than last one.

C

```
 // C++ program to print all palindromic partitions
// of a given string.
#include <bits/stdc++.h>
using namespace std;

// Returns true if str is palindrome, else false
bool checkPalindrome(string str)
{
    int len = str.length();
    len--;
    for (int i=0; i<len; i++)
    {
        if (str[i] != str[len])
            return false;
        len--;
    }
    return true;
}

void printSolution(vector<vector<string> > partitions)
{
    for (int i = 0; i < partitions.size(); ++i)
    {
        for(int j = 0; j < partitions[i].size(); ++j)
            cout << partitions[i][j] << " ";
        cout << endl;
    }
    return;
}

// Goes through all indexes and recursively add remaining
// partitions if current string is palindrome.
void addStrings(vector<vector<string> > &v, string &s,
                vector<string> &temp, int index)
{
    int len = s.length();
    string str;
    vector<string> current = temp;
```

```
    if (index == 0)
        temp.clear();
    for (int i = index; i < len; ++i)
    {
        str = str + s[i];
        if (checkPalindrome(str))
        {
            temp.push_back(str);
            if (i+1 < len)
                addStrings(v,s,temp,i+1);
            else
                v.push_back(temp);
            temp = current;
        }
    }
    return;
}

// Generates all palindromic partitions of 's' and
// stores the result in 'v'.
void partition(string s, vector<vector<string> >&v)
{
    vector<string> temp;
    addStrings(v, s, temp, 0);
    printSolution(v);
    return;
}

// Driver code
int main()
{
    string s = "geeks";
    vector<vector<string> > partitions;
    partition(s, partitions);
    return 0;
}
```

Java

```
 // Java program to print all palindromic partitions
// of a given string.
import java.util.ArrayList;
public class GFG
{
    // Returns true if str is palindrome, else false
    static boolean checkPalindrome(String str)
    {
        int len = str.length();
```

```
    len--;
    for (int i=0; i<len; i++)
    {
        if (str.charAt(i) != str.charAt(len))
            return false;
        len--;
    }
    return true;
}

// Prints the partition list
static void printSolution(ArrayList<ArrayList<String>>
                                      partitions)
{
    for(ArrayList<String> i: partitions)
    {
        for(String j: i)
        {
            System.out.print(j+" ");
        }
        System.out.println();
    }
}

// Goes through all indexes and recursively add remaining
// partitions if current string is palindrome.
static ArrayList<ArrayList<String>> addStrings(ArrayList<
   ArrayList<String>> v, String s, ArrayList<String> temp,
                                         int index)
{
    int len = s.length();
    String str = "";
    ArrayList<String> current = new ArrayList<>(temp);

    if (index == 0)
        temp.clear();

    // Iterate over the string
    for (int i = index; i < len; ++i)
    {
        str = str + s.charAt(i);

        // check whether the substring is
        // palindromic or not
        if (checkPalindrome(str))
        {
            // if palindrome add it to temp list
            temp.add(str);
```

```
                if (i + 1 < len)
                {
                    // recurr to get all the palindromic
                    // partitions for the substrings
                    v = addStrings(v,s,temp,i+1);
                }
                else
                {
                    // if end of the string is reached
                    // add temp to v
                    v.add(temp);
                }

                // temp is reinitialize with the
                // current i.
                temp = new ArrayList<>(current);
            }
        }
        return v;
    }

    // Generates all palindromic partitions of 's' and
    // stores the result in 'v'.
    static void partition(String s, ArrayList<ArrayList<
                                      String>> v)
    {
        // temporary ArrayList to store each
        // palindromic string
        ArrayList<String> temp = new ArrayList<>();

        // calling addString method it adds all
        // the palindromic partitions to v
        v = addStrings(v, s, temp, 0);

        // printing the solution
        printSolution(v);
    }

    // Driver code
    public static void main(String args[])
    {
        String s = "geeks";
        ArrayList<ArrayList<String>> partitions = new
                                       ArrayList<>();
        partition(s, partitions);
    }
}
```

```
// This code is contributed by Sumit Ghosh
```

Output :

```
g e e k s
g ee k s
```

Related Article:
Dynamic Programming | Set 17 (Palindrome Partitioning)

Source

https://www.geeksforgeeks.org/print-palindromic-partitions-string/

Chapter 38

Word Break Problem using Backtracking

Word Break Problem using Backtracking - GeeksforGeeks

Given a valid sentence without any spaces between the words and a dictionary of valid English words, find all possible ways to break the sentence in individual dictionary words.

Example

```
Consider the following dictionary
{ i, like, sam, sung, samsung, mobile, ice,
  cream, icecream, man, go, mango}

Input: "ilikesamsungmobile"
Output: i like sam sung mobile
        i like samsung mobile

Input: "ilikeicecreamandmango"
Output: i like ice cream and man go
        i like ice cream and mango
        i like icecream and man go
        i like icecream and mango
```

We have discussed a Dynamic Programming solution in below post.
Dynamic Programming | Set 32 (Word Break Problem)

The Dynamic Programming solution only finds whether it is possible to break a word or not. Here we need to print all possible word breaks.

We start scanning the sentence from left. As we find a valid word, we need to check whether rest of the sentence can make valid words or not. Because in some situations the first found word from left side can leave a remaining portion which is not further separable. So in that case we should come back and leave the current 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 stack and continue finding.

```
 // 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 conatins 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; //print result
                return;
            }
            wordBreakUtil(str.substr(i, n-i), n-i,
                                result + prefix + " ");
        }
    }      //end for
}//end function

int main()
{
    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
```

Source

https://www.geeksforgeeks.org/word-break-problem-using-backtracking/

Chapter 39

Partition of a set into K subsets with equal sum

Partition of a set into K subsets with equal sum - GeeksforGeeks

Given an integer array of N elements, the task is to divide this array into K non-empty subsets such that the sum of elements in every subset is same. All elements of this array should be part of exactly one partition.
Examples:

```
Input : arr = [2, 1, 4, 5, 6], K = 3
Output : Yes
we can divide above array into 3 parts with equal
sum as [[2, 4], [1, 5], [6]]

Input  : arr = [2, 1, 5, 5, 6], K = 3
Output : No
It is not possible to divide above array into 3
parts with equal sum
```

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.
```

Improved By : ayush0824

Source

https://www.geeksforgeeks.org/partition-set-k-subsets-equal-sum/

Chapter 40

Print all longest common sub-sequences in lexicographical order

Print all longest common sub-sequences in lexicographical order - GeeksforGeeks

You are given two strings.Now you have to print all longest common sub-sequences in lexicographical order?

Examples:

```
Input : str1 = "abcabcaa", str2 = "acbacba"
Output: ababa
        abaca
        abcba
        acaba
        acaca
        acbaa
        acbca
```

This problem is an extension of longest common subsequence. We first find length of LCS and store all LCS in 2D table using Memoization (or Dynamic Programming). Then we search all characters from 'a' to 'z' (to output sorted order) in both strings. If a character is found in both strings and current positions of character lead to LCS, we recursively search all occurrences with current LCS length plus 1.

Below is the implementation of algorithm.

```
 // C++ program to find all LCS of two strings in
// sorted order.
#include<bits/stdc++.h>
```

```
#define MAX 100
using namespace std;

// length of lcs
int lcslen = 0;

// dp matrix to store result of sub calls for lcs
int dp[MAX][MAX];

// A memoization based function that returns LCS of
// str1[i..len1-1] and str2[j..len2-1]
int lcs(string str1, string str2, int len1, int len2,
                                      int i, int j)
{
    int &ret = dp[i][j];

    // base condition
    if (i==len1 || j==len2)
        return ret = 0;

    // if lcs has been computed
    if (ret != -1)
        return ret;

    ret = 0;

    // if characters are same return previous + 1 else
    // max of two sequences after removing i'th and j'th
    // char one by one
    if (str1[i] == str2[j])
        ret = 1 + lcs(str1, str2, len1, len2, i+1, j+1);
    else
        ret = max(lcs(str1, str2, len1, len2, i+1, j),
                  lcs(str1, str2, len1, len2, i, j+1));
    return ret;
}

// Function to print all routes common sub-sequences of
// length lcslen
void printAll(string str1, string str2, int len1, int len2,
              char data[], int indx1, int indx2, int currlcs)
{
    // if currlcs is equal to lcslen then print it
    if (currlcs == lcslen)
    {
        data[currlcs] = '\0';
        puts(data);
        return;
```

```
    }

    // if we are done with all the characters of both string
    if (indx1==len1 || indx2==len2)
        return;

    // here we have to print all sub-sequences lexicographically,
    // that's why we start from 'a'to'z' if this character is
    // present in both of them then append it in data[] and same
    // remaining part
    for (char ch='a'; ch<='z'; ch++)
    {
        // done is a flag to tell that we have printed all the
        // subsequences corresponding to current character
        bool done = false;

        for (int i=indx1; i<len1; i++)
        {
            // if character ch is present in str1 then check if
            // it is present in str2
            if (ch==str1[i])
            {
              for (int j=indx2; j<len2; j++)
              {
                // if ch is present in both of them and
                // remaining length is equal to remaining
                // lcs length then add ch in sub-sequenece
                if (ch==str2[j] &&
                  lcs(str1, str2, len1, len2, i, j) == lcslen-currlcs)
                {
                  data[currlcs] = ch;
                  printAll(str1, str2, len1, len2, data, i+1, j+1, currlcs+1);
                  done = true;
                  break;
                }
              }
            }

            // If we found LCS beginning with current character.
            if (done)
                break;
        }
    }
}

// This function prints all LCS of str1 and str2
// in lexicographic order.
void prinlAllLCSSorted(string str1, string str2)
```

```
{
    // Find lengths of both strings
    int len1 = str1.length(), len2 = str2.length();

    // Find length of LCS
    memset(dp, -1, sizeof(dp));
    lcslen = lcs(str1, str2, len1, len2, 0, 0);

    // Print all LCS using recursive backtracking
    // data[] is used to store individual LCS.
    char data[MAX];
    printAll(str1, str2, len1, len2, data, 0, 0, 0);
}

// Driver program to run the case
int main()
{
    string str1 = "abcabcaa", str2 = "acbacba";
    prinlAllLCSSorted(str1, str2);
    return 0;
}
```

Output:

```
ababa
abaca
abcba
acaba
acaca
acbaa
acbca
```

Source

https://www.geeksforgeeks.org/print-longest-common-sub-sequences-lexicographical-order/

Chapter 41

Remove Invalid Parentheses

Remove Invalid Parentheses - GeeksforGeeks

An expression will be given which can contain open and close parentheses and optionally some characters, No other operator will be there in string. We need to remove minimum number of parentheses to make the input string valid. If more than one valid output are possible removing same number of parentheses then print all such output.
Examples:

```
Input  : str = "()())()" -
Output : ()()() (())()
There are two possible solutions
"()()()" and "(())()"

Input  : str = (v)())()
Output : (v)()()  (v())()
```

As we need to generate all possible output we will backtrack among all states by removing one opening or closing bracket and check if they are valid if invalid then add the removed bracket back and go for next state. We will use BFS for moving through states, use of BFS will assure removal of minimal number of brackets because we traverse into states level by level and each level corresponds to one extra bracket removal. Other than this BFS involve no recursion so overhead of passing parameters is also saved.
Below code has a method isValidString to check validity of string, it counts open and closed parenthesis at each index ignoring non-parenthesis character. If at any instant count of close parenthesis becomes more than open then we return false else we keep update the count variable.

```
 /*  C/C++ program to remove invalid parenthesis */
#include <bits/stdc++.h>
using namespace std;
```

```
//  method checks if character is parenthesis(open
// or closed)
bool isParenthesis(char c)
{
    return ((c == '(') || (c == ')'));
}

//  method returns true if string contains valid
// parenthesis
bool isValidString(string str)
{
    int cnt = 0;
    for (int i = 0; i < str.length(); i++)
    {
        if (str[i] == '(')
            cnt++;
        else if (str[i] == ')')
            cnt--;
        if (cnt < 0)
            return false;
    }
    return (cnt == 0);
}

//  method to remove invalid parenthesis
void removeInvalidParenthesis(string str)
{
    if (str.empty())
        return ;

    //  visit set to ignore already visited string
    set<string> visit;

    //  queue to maintain BFS
    queue<string> q;
    string temp;
    bool level;

    //  pushing given string as starting node into queue
    q.push(str);
    visit.insert(str);
    while (!q.empty())
    {
        str = q.front();  q.pop();
        if (isValidString(str))
        {
            cout << str << endl;
```

```
            // If answer is found, make level true
            // so that valid string of only that level
            // are processed.
            level = true;
        }
        if (level)
            continue;
        for (int i = 0; i < str.length(); i++)
        {
            if (!isParenthesis(str[i]))
                continue;

            // Removing parenthesis from str and
            // pushing into queue,if not visited already
            temp = str.substr(0, i) + str.substr(i + 1);
            if (visit.find(temp) == visit.end())
            {
                q.push(temp);
                visit.insert(temp);
            }
        }
    }
}

//  Driver code to check above methods
int main()
{
    string expression = "()())()";
    removeInvalidParenthesis(expression);

    expression = "()v)";
    removeInvalidParenthesis(expression);

    return 0;
}
```

Output:

```
(())()
()()()

(v)
()v
```

Source

https://www.geeksforgeeks.org/remove-invalid-parentheses/

Chapter 42

Find all distinct subsets of a given set

Find all distinct subsets of a given set - GeeksforGeeks

Given a set of positive integers, find all its subsets. The set can contain duplicate elements, so any repeated subset should be considered only once in the output.

Examples:

```
Input:  S = {1, 2, 2}
Output:  {}, {1}, {2}, {1, 2}, {2, 2}, {1, 2, 2}

Explanation:
The total subsets of given set are -
{}, {1}, {2}, {2}, {1, 2}, {1, 2}, {2, 2}, {1, 2, 2}
Here {2} and {1, 2} are repeated twice so they are considered
only once in the output
```

Prerequisite: Power Set

The idea is to use a bit-mask pattern to generate all the combinations as discussed in previous post. But previous post will print duplicate subsets if the elements are repeated in the given set. To handle duplicate elements, we construct a string out of given subset such that subsets having similar elements will result in same string. We maintain a list of such unique strings and finally we decode all such string to print its individual elements.

Below is its C++ implementation –

```
 // C++ program to find all subsets of given set. Any
// repeated subset is considered only once in the output
#include <bits/stdc++.h>
using namespace std;
```

```
// Utility function to split the string using a delim. Refer -
// http://stackoverflow.com/questions/236129/split-a-string-in-c
vector<string> split(const string &s, char delim)
{
    vector<string> elems;
    stringstream ss(s);
    string item;
    while (getline(ss, item, delim))
        elems.push_back(item);

    return elems;
}

// Function to find all subsets of given set. Any repeated
// subset is considered only once in the output
int printPowerSet(int arr[], int n)
{
    vector<string> list;

    /* Run counter i from 000..0 to 111..1*/
    for (int i = 0; i < (int) pow(2, n); i++)
    {
        string subset = "";

        // consider each element in the set
        for (int j = 0; j < n; j++)
        {
            // Check if jth bit in the i is set. If the bit
            // is set, we consider jth element from set
            if ((i & (1 << j)) != 0)
                subset += to_string(arr[j]) + "|";
        }

        // if subset is encountered for the first time
        // If we use set<string>, we can directly insert
        if (find(list.begin(), list.end(), subset) == list.end())
            list.push_back(subset);
    }

    // consider every subset
    for (string subset : list)
    {
        // split the subset and print its elements
        vector<string> arr = split(subset, '|');
        for (string str: arr)
            cout << str << " ";
        cout << endl;
```

```
    }
}

// Driver code
int main()
{
    int arr[] = { 10, 12, 12 };
    int n = sizeof(arr)/sizeof(arr[0]);

    printPowerSet(arr, n);

    return 0;
}
```

Output:

```

10
12
10 12
12 12
10 12 12
```

This article is contributed by **Aditya Goel**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.GeeksforGeeks.org or mail your article to contribute@GeeksforGeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Source

https://www.geeksforgeeks.org/find-distinct-subsets-given-set/

Chapter 43

Find shortest safe route in a path with landmines

Find shortest safe route in a path with landmines - GeeksforGeeks

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)
```

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 C++ 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 cordinates 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)
        {
            // initailize 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
```

Source

https://www.geeksforgeeks.org/find-shortest-safe-route-in-a-path-with-landmines/

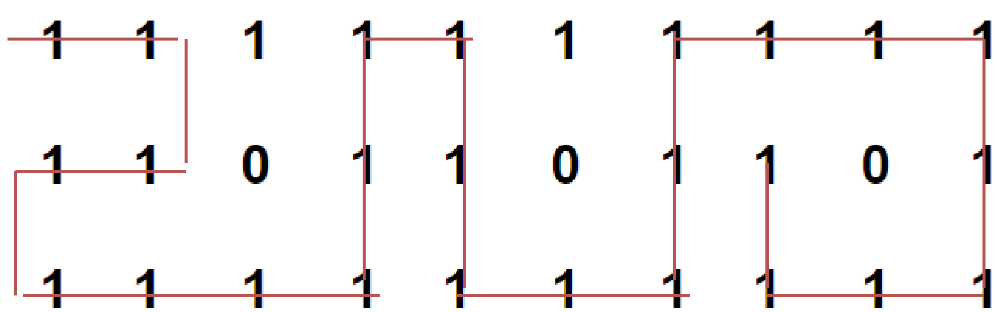
Chapter 44

Longest Possible Route in a Matrix with Hurdles

Longest Possible Route in a Matrix with Hurdles - GeeksforGeeks

Given an M x N matrix, with a few hurdles arbitrarily placed, calculate the length of longest possible route possible from source to destination within the matrix. We are allowed to move to only adjacent cells which are not hurdles. The route cannot contains any diagonal moves and a location once visited in a particular path cannot be visited again.

For example, longest path with no hurdles from source to destination is highlighted for below matrix. The length of the path is 24.



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 leads to the solution or not. If destination is found, we update the value of longest path else if none of the above solutions work we return false from our function.

Below is C++ implementation of 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];

    // initailize 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
```

Source

https://www.geeksforgeeks.org/longest-possible-route-in-a-matrix-with-hurdles/

Chapter 45

Match a pattern and String without using regular expressions

Match a pattern and String without using regular expressions - GeeksforGeeks

Given a string, find out if string follows a given pattern or not without using any regular expressions.

Examples:

```
Input:
string - GraphTreesGraph
pattern - aba
Output:
a->Graph
b->Trees

Input:
string - GraphGraphGraph
pattern - aaa
Output:
a->Graph

Input:
string - GeeksforGeeks
pattern - GfG
Output:
G->Geeks
f->for
```

```
Input:
string - GeeksforGeeks
pattern - GG
Output:
No solution exists
```

We can solve this problem with the help of Backtracking. For each character in the pattern, if the character is not seen before, we consider all possible sub-strings and recurse to see if it leads to the solution or not. We maintain a map that stores sub-string mapped to a pattern character. If pattern character is seen before, we use the same sub-string present in the map. If we found a solution, for each distinct character in the pattern, we print string mapped to it using our map.

Below is C++ implementation of above idea –

```
 // C++ program to find out if string follows
// a given pattern or not
#include <bits/stdc++.h>
using namespace std;

/*  Function to find out if string follows a given
    pattern or not

    str - given string
    n - length of given string
    i - current index in input string
    pat - given pattern
    m - length of given pattern
    j - current index in given pattern
    map - stores mapping between pattern and string */
bool patternMatchUtil(string str, int n, int i,
                    string pat, int m, int j,
                    unordered_map<char, string>& map)
{
    // If both string and pattern reach their end
    if (i == n && j == m)
        return true;

    // If either string or pattern reach their end
    if (i == n || j == m)
        return false;

    // read next character from the pattern
    char ch = pat[j];

    // if character is seen before
    if (map.find(ch)!= map.end())
    {
```

```
        string s = map[ch];
        int len = s.size();

        // consider next len characters of str
        string subStr = str.substr(i, len);

        // if next len characters of string str
        // don't match with s, return false
        if (subStr.compare(s))
            return false;

        // if it matches, recurse for remaining characters
        return patternMatchUtil(str, n, i + len, pat, m,
                                    j + 1, map);
    }

    // If character is seen for first time, try out all
    // remaining characters in the string
    for (int len = 1; len <= n - i; len++)
    {
        // consider substring that starts at position i
        // and spans len characters and add it to map
        map[ch] = str.substr(i, len);

        // see if it leads to the solution
        if (patternMatchUtil(str, n, i + len, pat, m,
                                    j + 1, map))
            return true;

        // if not, remove ch from the map
        map.erase(ch);
    }

    return false;
}

// A wrapper over patternMatchUtil()function
bool patternMatch(string str, string pat, int n, int m)
{
    if (n < m)
    return false;

    // create an empty hashmap
    unordered_map<char, string> map;

    // store result in a boolean variable res
    bool res = patternMatchUtil(str, n, 0, pat, m, 0, map);
```

```
    // if solution exists, print the mappings
    for (auto it = map.begin(); res && it != map.end(); it++)
        cout << it->first << "->" << it->second << endl;

    // return result
    return res;
}

// Driver code
int main()
{
    string str = "GeeksforGeeks", pat = "GfG";

    int n = str.size(), m = pat.size();

    if (!patternMatch(str, pat, n, m))
        cout << "No Solution exists";

    return 0;
}
```

Output:

```
f->for
G->Geeks
```

Source

https://www.geeksforgeeks.org/match-a-pattern-and-string-without-using-regular-expressions/

Chapter 46

Find Maximum number possible by doing at-most K swaps

Find Maximum number possible by doing at-most K swaps - GeeksforGeeks

Given a positive integer, find maximum integer possible by doing at-most K swap operations on its digits.

Examples:

```
Input: M = 254, K = 1
Output: 524

Input: M = 254, K = 2
Output: 542

Input: M = 68543, K = 1
Output: 86543

Input: M = 7599, K = 2
Output: 9975

Input: M = 76543, K = 1
Output: 76543

Input: M = 129814999, K = 4
Output: 999984211
```

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. We repeat the process K times. The code can be further optimized if we swap only if current digit is less than the following digit.

Below is C++ implementation of above idea –

```
 // 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++)
    {

        // and 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
```

The above code can further be optimized by stopping our search if all digits are already sorted in decreasing order. Please do share with us if you find more efficient ways to solve this problem.

Exercise :
1. Find minimum integer possible by doing at-least K swap operations on its digits.
2. Find maximum/minimum integer possible by doing exactly K swap operations on its digits.

Source

https://www.geeksforgeeks.org/find-maximum-number-possible-by-doing-at-most-k-swaps/

Chapter 47

Find paths from corner cell to middle cell in maze

Find paths from corner cell to middle cell in maze - GeeksforGeeks

Given a square maze containing positive numbers, find all paths from a corner cell (any of the extreme four corners) to the middle cell. We can move exactly n steps from a cell in 4 directions i.e. North, East, West and South where **n is value of the cell**,

We can move to mat[i+n][j], mat[i-n][j], mat[i][j+n], and mat[i][j-n] from a cell mat[i][j] where n is value of mat[i][j].

Example

```
Input:  9 x 9 maze
[ 3, 5, 4, 4, 7, 3, 4, 6, 3 ]
[ 6, 7, 5, 6, 6, 2, 6, 6, 2 ]
[ 3, 3, 4, 3, 2, 5, 4, 7, 2 ]
[ 6, 5, 5, 1, 2, 3, 6, 5, 6 ]
[ 3, 3, 4, 3, 0, 1, 4, 3, 4 ]
[ 3, 5, 4, 3, 2, 2, 3, 3, 5 ]
[ 3, 5, 4, 3, 2, 6, 4, 4, 3 ]
[ 3, 5, 1, 3, 7, 5, 3, 6, 4 ]
[ 6, 2, 4, 3, 4, 5, 4, 5, 1 ]

Output:
(0, 0) -> (0, 3) -> (0, 7) ->
(6, 7) -> (6, 3) -> (3, 3) ->
(3, 4) -> (5, 4) -> (5, 2) ->
(1, 2) -> (1, 7) -> (7, 7) ->
(7, 1) -> (2, 1) -> (2, 4) ->
(4, 4) -> MID
```

The idea is to use backtracking. We start with each corner cell of the maze and recursively checks if it leads to the solution or not. Following is the Backtracking algorithm –

If destination is reached

1. print the path

Else

1. Mark current cell as visited and add it to path array.
2. Move forward in all 4 allowed directions and recursively check if any of them leads to a solution.
3. If none of the above solutions work then mark this cell as not visitedand remove it from path array.

Below is its C++ implementation

```
 // C++ program to find a path from corner cell to
// middle cell in maze containing positive numbers
#include <bits/stdc++.h>
using namespace std;

// Rows and columns in given maze
#define N 9

// check whether given cell is a valid cell or not.
bool isValid(set<pair<int, int> > visited,
             pair<int, int> pt)
{
    // check if cell is not visited yet to
    // avoid cycles (infinite loop) and its
    // row and column number is in range
    return (pt.first >= 0) && (pt.first  < N) &&
           (pt.second >= 0) && (pt.second < N) &&
           (visited.find(pt) == visited.end());
}

// Function to print path from source to middle coordinate
void printPath(list<pair<int, int> > path)
{
    for (auto it = path.begin(); it != path.end(); it++)
        cout << "(" << it->first << ", "
             << it->second << ") -> ";

    cout << "MID" << endl << endl;
}
```

```
// For searching in all 4 direction
int row[] = {-1, 1, 0, 0};
int col[] = { 0, 0, -1, 1};

// Cordinates of 4 corners of matrix
int _row[] = { 0, 0, N-1, N-1};
int _col[] = { 0, N-1, 0, N-1};

void findPathInMazeUtil(int maze[N][N],
                list<pair<int, int> > &path,
                set<pair<int, int> > &visited,
                pair<int, int> &curr)
{
    // If we have reached the destination cell.
    // print the complete path
    if (curr.first == N / 2 && curr.second == N / 2)
    {
        printPath(path);
        return;
    }

    // consider each direction
    for (int i = 0; i < 4; ++i)
    {
        // get value of current cell
        int n = maze[curr.first][curr.second];

        // We can move N cells in either of 4 directions
        int x = curr.first + row[i]*n;
        int y = curr.second + col[i]*n;

        // Constructs a pair object with its first element
        // set to x and its second element set to y
        pair<int, int> next = make_pair(x, y);

        // if valid pair
        if (isValid(visited, next))
        {
            // mark cell as visited
            visited.insert(next);

            // add cell to current path
            path.push_back(next);

            // recuse for next cell
            findPathInMazeUtil(maze, path, visited, next);

            // backtrack
```

```
            path.pop_back();

            // remove cell from current path
            visited.erase(next);
        }
    }
}

// Function to find a path from corner cell to
// middle cell in maze contaning positive numbers
void findPathInMaze(int maze[N][N])
{
    // list to store complete path
    // from source to destination
    list<pair<int, int> > path;

    // to store cells already visisted in current path
    set<pair<int, int> > visited;

    // Consider each corners as the starting
    // point and search in maze
    for (int i = 0; i < 4; ++i)
    {
        int x = _row[i];
        int y = _col[i];

        // Constructs a pair object
        pair<int, int> pt = make_pair(x, y);

        // mark cell as visited
        visited.insert(pt);

        // add cell to current path
        path.push_back(pt);

        findPathInMazeUtil(maze, path, visited, pt);

        // backtrack
        path.pop_back();

        // remove cell from current path
        visited.erase(pt);
    }
}

int main()
{
    int maze[N][N] =
```

```
    {
        { 3, 5, 4, 4, 7, 3, 4, 6, 3 },
        { 6, 7, 5, 6, 6, 2, 6, 6, 2 },
        { 3, 3, 4, 3, 2, 5, 4, 7, 2 },
        { 6, 5, 5, 1, 2, 3, 6, 5, 6 },
        { 3, 3, 4, 3, 0, 1, 4, 3, 4 },
        { 3, 5, 4, 3, 2, 2, 3, 3, 5 },
        { 3, 5, 4, 3, 2, 6, 4, 4, 3 },
        { 3, 5, 1, 3, 7, 5, 3, 6, 4 },
        { 6, 2, 4, 3, 4, 5, 4, 5, 1 }
    };

    findPathInMaze(maze);

    return 0;
}
```

Output :

```
(0, 0) -> (0, 3) -> (0, 7) ->
(6, 7) -> (6, 3) -> (3, 3) ->
(3, 4) -> (5, 4) -> (5, 2) ->
(1, 2) -> (1, 7) -> (7, 7) ->
(7, 1) -> (2, 1) -> (2, 4) ->
(4, 4) -> MID
```

Source

https://www.geeksforgeeks.org/find-paths-from-corner-cell-to-middle-cell-in-maze/

Chapter 48

Find if there is a path of more than k length from a source

Find if there is a path of more than k length from a source - GeeksforGeeks

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.

```
Input  : Source s = 0, k = 58
Output : True
There exists a simple path 0 -> 7 -> 1
-> 2 -> 8 -> 6 -> 5 -> 3 -> 4
Which has a total distance of 60 km which
is more than 58.

Input  : Source s = 0, k = 62
Output : False

In the above graph, the longest simple
path has distance 61 (0 -> 7 -> 1-> 2
 -> 3 -> 4 -> 5-> 6 -> 8, so output
should be false for any input greater
than 61.
```

We strongly recommend you to minimize your browser and try this yourself first.

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 C++ 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
```

Exercise:
Modify the above solution to find weight of longest path from a given source.

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!)

This article is contributed by **Shivam Gupta**. The explanation for time complexity is contributed by **Pranav Nambiar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/find-if-there-is-a-path-of-more-than-k-length-from-a-source/

Chapter 49

Fill two instances of all numbers from 1 to n in a specific way

Fill two instances of all numbers from 1 to n in a specific way - GeeksforGeeks

Given a number n, create an array of size 2n such that the array contains 2 instances of every number from 1 to n, and the number of elements between two instances of a number i is equal to i. If such a configuration is not possible, then print the same.

Examples:

```
Input: n = 3
Output: res[] = {3, 1, 2, 1, 3, 2}

Input: n = 2
Output: Not Possible

Input: n = 4
Output: res[] = {4, 1, 3, 1, 2, 4, 3, 2}
```

We strongly recommend to minimize the browser and try this yourself first.

One solution is to Backtracking. The idea is simple, we place two instances of n at a place, then recur for n-1. If recurrence is successful, we return true, else we backtrack and try placing n at different location. Following is C implementation of the idea.

```
 // A backtracking based C Program to fill two instances of all numbers
// from 1 to n in a specific way
#include <stdio.h>
#include <stdbool.h>

// A recursive utility function to fill two instances of numbers from
```

```
// 1 to n in res[0..2n-1].  'curr' is current value of n.
bool fillUtil(int res[], int curr, int n)
{
     // If current number becomes 0, then all numbers are filled
     if (curr == 0) return true;

     // Try placing two instances of 'curr' at all possible locations
     // till solution is found
     int i;
     for (i=0; i<2*n-curr-1; i++)
     {
        // Two 'curr' should be placed at 'curr+1' distance
        if (res[i] == 0 && res[i + curr + 1] == 0)
        {
           // Plave two instances of 'curr'
           res[i] = res[i + curr + 1] = curr;

           // Recur to check if the above placement leads to a solution
           if (fillUtil(res, curr-1, n))
               return true;

           // If solution is not possible, then backtrack
           res[i] = res[i + curr + 1] = 0;
        }
     }
     return false;
}

// This function prints the result for input number 'n' using fillUtil()
void fill(int n)
{
    // Create an array of size 2n and initialize all elements in it as 0
    int res[2*n], i;
    for (i=0; i<2*n; i++)
       res[i] = 0;

    // If solution is possible, then print it.
    if (fillUtil(res, n, n))
    {
        for (i=0; i<2*n; i++)
           printf("%d ", res[i]);
    }
    else
        puts("Not Possible");
}

// Driver program
int main()
```

```
{
  fill(7);
  return 0;
}
```

Output:

```
7 3 6 2 5 3 2 4 7 6 5 1 4 1
```

The above solution may not be the best possible solution. There seems to be a pattern in the output. I an Looking for a better solution from other geeks.

This article is contributed by **Asif**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/fill-two-instances-numbers-1-n-specific-way/

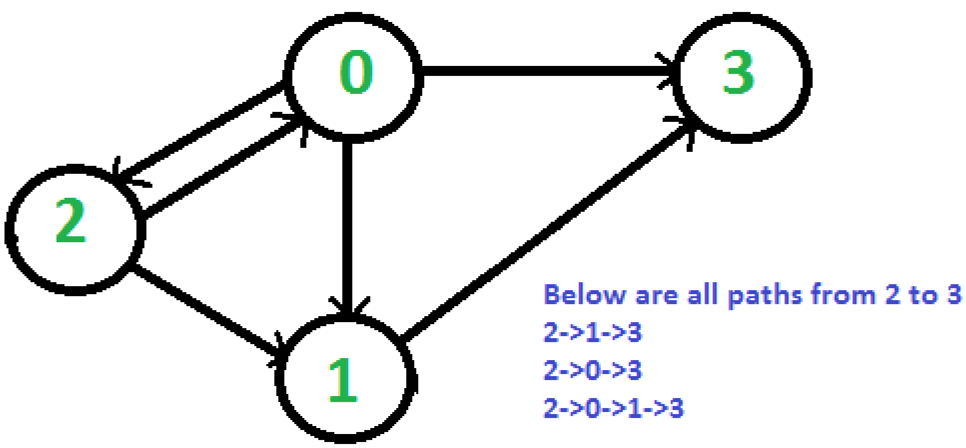
Chapter 50

Print all paths from a given source to a destination

Print all paths from a given source to a destination - GeeksforGeeks

Given a directed graph, a source vertex 's' and a destination vertex 'd', print all paths from given 's' to 'd'.

Consider the following directed graph. Let the s be 2 and d be 3. There are 4 different paths from 2 to 3.



The idea is to do Depth First Traversal of given directed graph. Start the traversal from source. Keep storing the visited vertices in an array say 'path[]'. If we reach the destination vertex, print contents of path[]. The important thing is to mark current vertices in path[] as visited also, so that the traversal doesn't go in a cycle.

Following is implementation of above idea.

C/C++

```
 // C++ program to print all paths from a source to destination.
#include<iostream>
#include <list>
using namespace std;

// A directed graph using adjacency list representation
class Graph
{
    int V;    // No. of vertices in graph
    list<int> *adj; // Pointer to an array containing adjacency lists

    // A recursive function used by printAllPaths()
    void printAllPathsUtil(int , int , bool [], int [], int &);

public:
    Graph(int V);   // Constructor
    void addEdge(int u, int v);
    void printAllPaths(int s, int d);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int u, int v)
{
    adj[u].push_back(v); // Add v to u’s list.
}

// Prints all paths from 's' to 'd'
void Graph::printAllPaths(int s, int d)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];

    // Create an array to store paths
    int *path = new int[V];
    int path_index = 0; // Initialize path[] as empty

    // Initialize all vertices as not visited
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to print all paths
    printAllPathsUtil(s, d, visited, path, path_index);
}
```

```
// A recursive function to print all paths from 'u' to 'd'.
// visited[] keeps track of vertices in current path.
// path[] stores actual vertices and path_index is current
// index in path[]
void Graph::printAllPathsUtil(int u, int d, bool visited[],
                              int path[], int &path_index)
{
    // Mark the current node and store it in path[]
    visited[u] = true;
    path[path_index] = u;
    path_index++;

    // If current vertex is same as destination, then print
    // current path[]
    if (u == d)
    {
        for (int i = 0; i<path_index; i++)
            cout << path[i] << " ";
        cout << endl;
    }
    else // If current vertex is not destination
    {
        // Recur for all the vertices adjacent to current vertex
        list<int>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (!visited[*i])
                printAllPathsUtil(*i, d, visited, path, path_index);
    }

    // Remove current vertex from path[] and mark it as unvisited
    path_index--;
    visited[u] = false;
}

// Driver program
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(0, 3);
    g.addEdge(2, 0);
    g.addEdge(2, 1);
    g.addEdge(1, 3);

    int s = 2, d = 3;
```

```
    cout << "Following are all different paths from " << s
         << " to " << d << endl;
    g.printAllPaths(s, d);

    return 0;
}
```

Java

```
 // JAVA program to print all
// paths from a source to
// destination.
import java.util.ArrayList;
import java.util.List;

// A directed graph using
// adjacency list representation
public class Graph {

    // No. of vertices in graph
    private int v;

    // adjacency list
    private ArrayList<Integer>[] adjList;

    //Constructor
    public Graph(int vertices){

        //initialise vertex count
        this.v = vertices;

        // initialise adjacency list
        initAdjList();
    }

    // utility method to initialise
    // adjacency list
    @SuppressWarnings("unchecked")
    private void initAdjList()
    {
        adjList = new ArrayList[v];

        for(int i = 0; i < v; i++)
        {
            adjList[i] = new ArrayList<>();
        }
    }
```

```
    // add edge from u to v
    public void addEdge(int u, int v)
    {
        // Add v to u's list.
        adjList[u].add(v);
    }

    // Prints all paths from
    // 's' to 'd'
    public void printAllPaths(int s, int d)
    {
        boolean[] isVisited = new boolean[v];
        ArrayList<Integer> pathList = new ArrayList<>();

        //add source to path[]
        pathList.add(s);

        //Call recursive utility
        printAllPathsUtil(s, d, isVisited, pathList);
    }

    // A recursive function to print
    // all paths from 'u' to 'd'.
    // isVisited[] keeps track of
    // vertices in current path.
    // localPathList<> stores actual
    // vertices in the current path
    private void printAllPathsUtil(Integer u, Integer d,
                                    boolean[] isVisited,
                            List<Integer> localPathList) {

        // Mark the current node
        isVisited[u] = true;

        if (u.equals(d))
        {
            System.out.println(localPathList);
        }

        // Recur for all the vertices
        // adjacent to current vertex
        for (Integer i : adjList[u])
        {
            if (!isVisited[i])
            {
                // store current node
                // in path[]
                localPathList.add(i);
```

```
                printAllPathsUtil(i, d, isVisited, localPathList);

                // remove current node
                // in path[]
                localPathList.remove(i);
            }
        }

        // Mark the current node
        isVisited[u] = false;
    }

    // Driver program
    public static void main(String[] args)
    {
        // Create a sample graph
        Graph g = new Graph(4);
        g.addEdge(0,1);
        g.addEdge(0,2);
        g.addEdge(0,3);
        g.addEdge(2,0);
        g.addEdge(2,1);
        g.addEdge(1,3);

        // arbitrary source
        int s = 2;

        // arbitrary destination
        int d = 3;

        System.out.println("Following are all different paths from "+s+" to "+d);
        g.printAllPaths(s, d);

    }
}

// This code is contributed by Himanshu Shekhar.
```

Python

```
 # Python program to print all paths from a source to destination.

from collections import defaultdict

#This class represents a directed graph
# using adjacency list representation
class Graph:
```

```
def __init__(self,vertices):
    #No. of vertices
    self.V= vertices

    # default dictionary to store graph
    self.graph = defaultdict(list)

# function to add an edge to graph
def addEdge(self,u,v):
    self.graph[u].append(v)

'''A recursive function to print all paths from 'u' to 'd'.
visited[] keeps track of vertices in current path.
path[] stores actual vertices and path_index is current
index in path[]'''
def printAllPathsUtil(self, u, d, visited, path):

    # Mark the current node as visited and store in path
    visited[u]= True
    path.append(u)

    # If current vertex is same as destination, then print
    # current path[]
    if u ==d:
        print path
    else:
        # If current vertex is not destination
        #Recur for all the vertices adjacent to this vertex
        for i in self.graph[u]:
            if visited[i]==False:
                self.printAllPathsUtil(i, d, visited, path)

    # Remove current vertex from path[] and mark it as unvisited
    path.pop()
    visited[u]= False


# Prints all paths from 's' to 'd'
def printAllPaths(self,s, d):

    # Mark all the vertices as not visited
    visited =[False]*(self.V)

    # Create an array to store paths
    path = []

    # Call the recursive helper function to print all paths
    self.printAllPathsUtil(s, d,visited, path)
```

```
# Create a graph given in the above diagram
g = Graph(4)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(0, 3)
g.addEdge(2, 0)
g.addEdge(2, 1)
g.addEdge(1, 3)

s = 2 ; d = 3
print ("Following are all different paths from %d to %d :" %(s, d))
g.printAllPaths(s, d)
#This code is contributed by Neelam Yadav
```

Output:

```
Following are all different paths from 2 to 3
2 0 1 3
2 0 3
2 1 3
```

This article is contributed by **Shivam Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

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

Chapter 51

Print all possible paths from top left to bottom right of a mXn matrix

Print all possible paths from top left to bottom right of a mXn matrix - GeeksforGeeks

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
```

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 is C++ implementation of the above algorithm.

```
 // CPP 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: Dimentions 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 abve 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 and that is left as an exercise.

This article is contributed by **Hariprasad NG**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Python implementation

```
 # Python3 program to Print all possible paths from
# top left to bottom right of a mXn matrix
allPaths = []
def findPaths(maze,m,n):
    path = [0 for d in range(m+n-1)]
    findPathsUtil(maze,m,n,0,0,path,0)

def findPathsUtil(maze,m,n,i,j,path,indx):
    global allPaths
    # if we reach the bottom of maze, we can only move right
    if i==m-1:
        for k in range(j,n):
            #path.append(maze[i][k])
            path[indx+k-j] = maze[i][k]
        # if we hit this block, it means one path is completed.
        # Add it to paths list and print
```

```
        print(path)
        allPaths.append(path)
        return
    # if we reach to the right most corner, we can only move down
    if j == n-1:
        for k in range(i,m):
            path[indx+k-i] = maze[k][j]
          #path.append(maze[j][k])
        # if we hit this block, it means one path is completed.
        # Add it to paths list and print
        print(path)
        allPaths.append(path)
        return

    # add current element to the path list
    #path.append(maze[i][j])
    path[indx] = maze[i][j]

    # move down in y direction and call findPathsUtil recursively
    findPathsUtil(maze, m, n, i+1, j, path, indx+1)

    # move down in y direction and call findPathsUtil recursively
    findPathsUtil(maze, m, n, i, j+1, path, indx+1)

if __name__ == '__main__':
    maze = [[1,2,3],
            [4,5,6],
            [7,8,9]]
    findPaths(maze,3,3)
    #print(allPaths)
```

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]
```

Improved By : ashritkumar

Source

https://www.geeksforgeeks.org/print-all-possible-paths-from-top-left-to-bottom-right-of-a-mxn-matrix/

Chapter 52

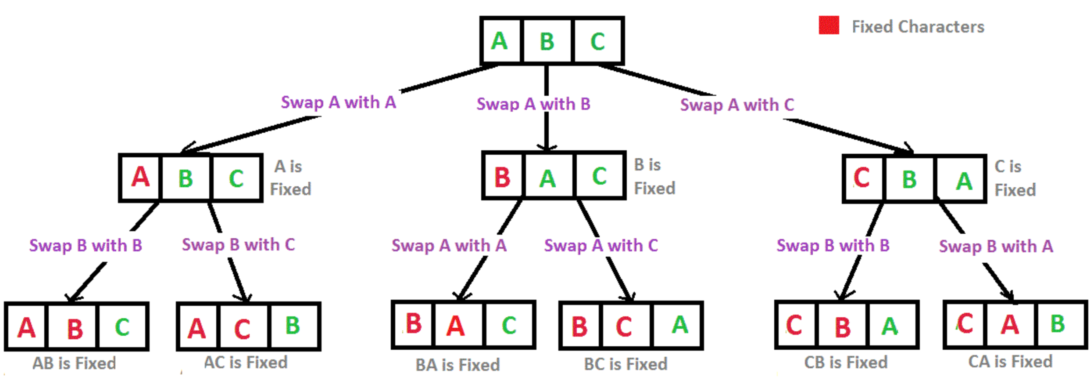
Write a program to print all permutations of a given string

Write a program to print all permutations of a given string - GeeksforGeeks

A permutation, also called an “arrangement number” or “order,” is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself. A string of length n has n! permutation.
Source: Mathword(http://mathworld.wolfram.com/Permutation.html)

Below are the permutations of string ABC.
ABC ACB BAC BCA CBA CAB

Here is a solution that is used as a basis in backtracking.



Recursion Tree for Permutations of String "ABC"

C/C++

```
 // C program to print all permutations with duplicates allowed
#include <stdio.h>
```

```
#include <string.h>

/* Function to swap values at two pointers */
void swap(char *x, char *y)
{
    char temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

/* Function to print permutations of string
   This function takes three parameters:
   1. String
   2. Starting index of the string
   3. Ending index of the string. */
void permute(char *a, int l, int r)
{
   int i;
   if (l == r)
     printf("%s\n", a);
   else
   {
       for (i = l; i <= r; i++)
       {
          swap((a+l), (a+i));
          permute(a, l+1, r);
          swap((a+l), (a+i)); //backtrack
       }
   }
}

/* Driver program to test above functions */
int main()
{
    char str[] = "ABC";
    int n = strlen(str);
    permute(str, 0, n-1);
    return 0;
}
```

Java

```
 // Java program to print all permutations of a
// given string.
public class Permutation
{
    public static void main(String[] args)
```

```
    {
        String str = "ABC";
        int n = str.length();
        Permutation permutation = new Permutation();
        permutation.permute(str, 0, n-1);
    }

    /**
     * permutation function
     * @param str string to calculate permutation for
     * @param l starting index
     * @param r end index
     */
    private void permute(String str, int l, int r)
    {
        if (l == r)
            System.out.println(str);
        else
        {
            for (int i = l; i <= r; i++)
            {
                str = swap(str,l,i);
                permute(str, l+1, r);
                str = swap(str,l,i);
            }
        }
    }

    /**
     * Swap Characters at position
     * @param a string value
     * @param i position 1
     * @param j position 2
     * @return swapped string
     */
    public String swap(String a, int i, int j)
    {
        char temp;
        char[] charArray = a.toCharArray();
        temp = charArray[i] ;
        charArray[i] = charArray[j];
        charArray[j] = temp;
        return String.valueOf(charArray);
    }

}

// This code is contributed by Mihir Joshi
```

Python

```
 # Python program to print all permutations with
# duplicates allowed

def toString(List):
    return ''.join(List)

# Function to print permutations of string
# This function takes three parameters:
# 1. String
# 2. Starting index of the string
# 3. Ending index of the string.
def permute(a, l, r):
    if l==r:
        print toString(a)
    else:
        for i in xrange(l,r+1):
            a[l], a[i] = a[i], a[l]
            permute(a, l+1, r)
            a[l], a[i] = a[i], a[l] # backtrack

# Driver program to test the above function
string = "ABC"
n = len(string)
a = list(string)
permute(a, 0, n-1)

# This code is contributed by Bhavya Jain
```

C#

```
 // C# program to print all
// permutations of a given string.
using System;

class GFG
{
    /**
    * permutation function
    * @param str string to
       calculate permutation for
    * @param l starting index
    * @param r end index
    */
    private static void permute(String str,
                                int l, int r)
```

```
    {
        if (l == r)
            Console.WriteLine(str);
        else
        {
            for (int i = l; i <= r; i++)
            {
                str = swap(str, l, i);
                permute(str, l + 1, r);
                str = swap(str, l, i);
            }
        }
    }

    /**
    * Swap Characters at position
    * @param a string value
    * @param i position 1
    * @param j position 2
    * @return swapped string
    */
    public static String swap(String a,
                              int i, int j)
    {
        char temp;
        char[] charArray = a.ToCharArray();
        temp = charArray[i] ;
        charArray[i] = charArray[j];
        charArray[j] = temp;
        string s = new string(charArray);
        return s;
    }

// Driver Code
public static void Main()
{
    String str = "ABC";
    int n = str.Length;
    permute(str, 0, n-1);
}
}

// This code is contributed by mits
```

PHP

```
 <?php
// PHP program to print all
```

```
// permutations of a given string.


/**
* permutation function
* @param str string to
*  calculate permutation for
* @param l starting index
* @param r end index
*/
function permute($str, $l, $r)
{
    if ($l == $r)
        echo $str. "\n";
    else
    {
        for ($i = $l; $i <= $r; $i++)
        {
            $str = swap($str, $l, $i);
            permute($str, $l + 1, $r);
            $str = swap($str, $l, $i);
        }
    }
}

/**
* Swap Characters at position
* @param a string value
* @param i position 1
* @param j position 2
* @return swapped string
*/
function swap($a, $i, $j)
{
    $temp;
    $charArray = str_split($a);
    $temp = $charArray[$i] ;
    $charArray[$i] = $charArray[$j];
    $charArray[$j] = $temp;
    return implode($charArray);
}

// Driver Code
$str = "ABC";
$n = strlen($str);
permute($str, 0, $n - 1);

// This code is contributed by mits.
```

```
?>
```

Output:

```
ABC
ACB
BAC
BCA
CBA
CAB
```

Algorithm Paradigm: Backtracking
Time Complexity: O(n*n!) Note that there are n! permutations and it requires O(n) time to print a a permutation.

Note : The above solution prints duplicate permutations if there are repeating characters in input string. Please see below link for a solution that prints only distinct permutations even if there are duplicates in input.

Print all distinct permutations of a given string with duplicates.
Permutations of a given string using STL

Improved By : Mithun Kumar

Source

https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/

Chapter 53

Given an array A[] and a number x, check for pair in A[] with sum as x

Given an array A[] and a number x, check for pair in A[] with sum as x - GeeksforGeeks

Write a program that, given an array A[] of n numbers and another number x, determines whether or not there exist two elements in S whose sum is exactly x.

METHOD 1 (Use Sorting)

Algorithm :

```
hasArrayTwoCandidates (A[], ar_size, sum)
1) Sort the array in non-decreasing order.
2) Initialize two index variables to find the candidate
   elements in the sorted array.
       (a) Initialize first to the leftmost index: l = 0
       (b) Initialize second  the rightmost index:  r = ar_size-1
3) Loop while l < r.
       (a) If (A[l] + A[r] == sum)  then return 1
       (b) Else if( A[l] + A[r] <  sum )  then l++
       (c) Else r--
4) No candidates in whole array - return 0
```

Time Complexity: Depends on what sorting algorithm we use. If we use Merge Sort or Heap Sort then (-)(nlogn) in worst case. If we use Quick Sort then O(n^2) in worst case.
Auxiliary Space : Again, depends on sorting algorithm. For example auxiliary space is O(n) for merge sort and O(1) for Heap Sort.

Example :
Let Array be {1, 4, 45, 6, 10, -8} and sum to find be 16

Sort the array
A = {-8, 1, 4, 6, 10, 45}

Initialize l = 0, r = 5
A[l] + A[r] (-8 + 45) > 16 => decrement r. Now r = 10
A[l] + A[r] (-8 + 10) increment l. Now l = 1
A[l] + A[r] (1 + 10) increment l. Now l = 2
A[l] + A[r] (4 + 10) increment l. Now l = 3
A[l] + A[r] (6 + 10) == 16 => Found candidates (return 1)

Note: If there are more than one pair having the given sum then this algorithm reports only one. Can be easily extended for this though.

Below is the implementation of the above approach.

C

```
 // C program to check if given array
// has 2 elements whose sum is equal
// to the given value

# include <stdio.h>
# define bool int

void quickSort(int *, int, int);

bool hasArrayTwoCandidates(int A[], int arr_size, int sum)
{
    int l, r;

    /* Sort the elements */
    quickSort(A, 0, arr_size-1);

    /* Now look for the two candidates in the sorted
       array*/
    l = 0;
    r = arr_size-1;
    while (l < r)
    {
         if(A[l] + A[r] == sum)
              return 1;
         else if(A[l] + A[r] < sum)
              l++;
         else // A[i] + A[j] > sum
              r--;
    }
    return 0;
```

```
}

/* FOLLOWING FUNCTIONS ARE ONLY FOR SORTING
    PURPOSE */
void exchange(int *a, int *b)
{
    int temp;
    temp = *a;
    *a   = *b;
    *b   = temp;
}

int partition(int A[], int si, int ei)
{
    int x = A[ei];
    int i = (si - 1);
    int j;

    for (j = si; j <= ei - 1; j++)
    {
        if(A[j] <= x)
        {
            i++;
            exchange(&A[i], &A[j]);
        }
    }
    exchange (&A[i + 1], &A[ei]);
    return (i + 1);
}

/* Implementation of Quick Sort
A[] --> Array to be sorted
si  --> Starting index
ei  --> Ending index
*/
void quickSort(int A[], int si, int ei)
{
    int pi;    /* Partitioning index */
    if(si < ei)
    {
        pi = partition(A, si, ei);
        quickSort(A, si, pi - 1);
        quickSort(A, pi + 1, ei);
    }
}

/* Driver program to test above function */
int main()
```

```
{
    int A[] = {1, 4, 45, 6, 10, -8};
    int n = 16;
    int arr_size = 6;

    if( hasArrayTwoCandidates(A, arr_size, n))
        printf("Array has two elements with given sum");
    else
        printf("Array doesn't have two elements with given sum");

    getchar();
    return 0;
}
```

C++

```
 // C++ program to check if given array
// has 2 elements whose sum is equal
// to the given value

#include <bits/stdc++.h>

using namespace std;

// Function to check if array has 2 elements
// whose sum is equal to the given value
bool hasArrayTwoCandidates(int A[], int arr_size,
                                        int sum)
{
    int l, r;

    /* Sort the elements */
    sort(A, A + arr_size);

    /* Now look for the two candidates in
       the sorted array*/
    l = 0;
    r = arr_size - 1;
    while (l < r)
    {
        if(A[l] + A[r] == sum)
            return 1;
        else if(A[l] + A[r] < sum)
            l++;
        else // A[i] + A[j] > sum
            r--;
    }
    return 0;
```

```
}

/* Driver program to test above function */
int main()
{
    int A[] = {1, 4, 45, 6, 10, -8};
    int n = 16;
    int arr_size = sizeof(A) / sizeof(A[0]);

    // Function calling
    if(hasArrayTwoCandidates(A, arr_size, n))
        cout << "Array has two elements with given sum";
    else
        cout << "Array doesn't have two elements with given sum";

    return 0;
}
```

Java

```
 // Java program to check if given array
// has 2 elements whose sum is equal
// to the given value
import java.util.*;

class GFG
{
    // Function to check if array has 2 elements
    // whose sum is equal to the given value
    static boolean hasArrayTwoCandidates(int A[],
                            int arr_size, int sum)
    {
        int l, r;

        /* Sort the elements */
        Arrays.sort(A);

        /* Now look for the two candidates
        in the sorted array*/
        l = 0;
        r = arr_size-1;
        while (l < r)
        {
            if(A[l] + A[r] == sum)
                return true;
            else if(A[l] + A[r] < sum)
                l++;
            else // A[i] + A[j] > sum
```

```
                r--;
        }
        return false;
    }

    // Driver code
    public static void main(String args[])
    {
        int A[] = {1, 4, 45, 6, 10, -8};
        int n = 16;
        int arr_size = A.length;

        // Function calling
        if(hasArrayTwoCandidates(A, arr_size, n))
            System.out.println("Array has two " +
                               "elements with given sum");
        else
            System.out.println("Array doesn't have " +
                               "two elements with given sum");

    }

}
```

Python

```
 # Python program to check for the sum condition to be satisified

def hasArrayTwoCandidates(A,arr_size,sum):

    # sort the array
    quickSort(A,0,arr_size-1)
    l = 0
    r = arr_size-1

    # traverse the array for the two elements
    while l<r:
        if (A[l] + A[r] == sum):
            return 1
        elif (A[l] + A[r] < sum):
            l += 1
        else:
            r -= 1
    return 0

# Implementation of Quick Sort
# A[] --> Array to be sorted
# si  --> Starting index
```

```
# ei  --> Ending index
def quickSort(A, si, ei):
    if si < ei:
        pi=partition(A,si,ei)
        quickSort(A,si,pi-1)
        quickSort(A,pi+1,ei)

# Utility function for partitioning the array(used in quick sort)
def partition(A, si, ei):
    x = A[ei]
    i = (si-1)
    for j in range(si,ei):
        if A[j] <= x:
            i += 1

            # This operation is used to swap two variables is python
            A[i], A[j] = A[j], A[i]

        A[i+1], A[ei] = A[ei], A[i+1]

    return i+1


# Driver program to test the functions
A = [1,4,45,6,10,-8]
n = 16
if (hasArrayTwoCandidates(A, len(A), n)):
    print("Array has two elements with the given sum")
else:
    print("Array doesn't have two elements with the given sum")

## This code is contributed by __Devesh Agrawal__
```

C#

```
 // C# program to check for pair
// in A[] with sum as x

using System;

class GFG
{
    static bool hasArrayTwoCandidates(int []A,
                            int arr_size, int sum)
    {
        int l, r;

        /* Sort the elements */
```

```
    sort(A, 0, arr_size-1);

    /* Now look for the two candidates
    in the sorted array*/
    l = 0;
    r = arr_size-1;
    while (l < r)
    {
        if(A[l] + A[r] == sum)
            return true;
        else if(A[l] + A[r] < sum)
            l++;
        else // A[i] + A[j] > sum
            r--;
    }
    return false;
}

/* Below functions are only to sort the
array using QuickSort */

/* This function takes last element as pivot,
places the pivot element at its correct
position in sorted array, and places all
smaller (smaller than pivot) to left of
pivot and all greater elements to right
of pivot */
static int partition(int []arr, int low, int high)
{
    int pivot = arr[high];

    // index of smaller element
    int i = (low-1);
    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller
        // than or equal to pivot
        if (arr[j] <= pivot)
        {
            i++;

            // swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
```

```
        // swap arr[i+1] and arr[high] (or pivot)
        int temp1 = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp1;

        return i+1;
    }

    /* The main function that
    implements QuickSort()
    arr[] --> Array to be sorted,
    low --> Starting index,
    high --> Ending index */
    static void sort(int []arr, int low, int high)
    {
        if (low < high)
        {
            /* pi is partitioning index, arr[pi]
            is now at right place */
            int pi = partition(arr, low, high);

            // Recursively sort elements before
            // partition and after partition
            sort(arr, low, pi-1);
            sort(arr, pi+1, high);
        }
    }

    // Driver code
    public static void Main()
    {
        int []A = {1, 4, 45, 6, 10, -8};
        int n = 16;
        int arr_size = 6;

        if( hasArrayTwoCandidates(A, arr_size, n))
            Console.Write("Array has two elements"+
                        " with given sum");
        else
            Console.Write("Array doesn't have "+
                        "two elements with given sum");

    }
}

// This code is contributed by Sam007
```

PHP

```
 <?php
// PHP program to check if given
// array has 2 elements whose sum
// is equal to the given value

// Function to check if array has
// 2 elements whose sum is equal
// to the given value
function hasArrayTwoCandidates($A, $arr_size,
                                        $sum)
{
    $l; $r;

    /* Sort the elements */
    //sort($A, A + arr_size);
    sort($A);

    /* Now look for the two candidates
    in the sorted array*/
    $l = 0;
    $r = $arr_size - 1;
    while ($l < $r)
    {
        if($A[$l] + $A[$r] == $sum)
            return 1;
        else if($A[$l] + $A[$r] < $sum)
            $l++;
        else // A[i] + A[j] > sum
            $r--;
    }
    return 0;
}

// Driver Code
$A = array (1, 4, 45, 6, 10, -8);
$n = 16;
$arr_size = sizeof($A);

// Function calling
if(hasArrayTwoCandidates($A, $arr_size, $n))
    echo "Array has two elements " .
                   "with given sum";
else
    echo "Array doesn't have two " .
          "elements with given sum";

// This code is contributed by m_kit
?>
```

Output :

```
Array has two elements with the given sum
```

METHOD 2 (Use Hashing)
This method works in O(n) time.

```
1) Initialize an empty hash table s.
2) Do following for each element A[i] in A[]
   (a)    If s[x - A[i]] is set then print the pair (A[i], x - A[i])
   (b)    Insert A[i] into s.
```

Below is the implementation of the above approach :

C

```
 // C++ program to check if given array
// has 2 elements whose sum is equal
// to the given value

// Works only if range elements is limited
#include <stdio.h>
#define MAX 100000

void printPairs(int arr[], int arr_size, int sum)
{
  int i, temp;
  bool s[MAX] = {0}; /*initialize hash set as 0*/

  for (i = 0; i < arr_size; i++)
  {
      temp = sum - arr[i];
      if (temp >= 0 && s[temp] == 1)
         printf("Pair with given sum %d is (%d, %d) n",
                 sum, arr[i], temp);
      s[arr[i]] = 1;
  }
}

/* Driver program to test above function */
int main()
{
    int A[] = {1, 4, 45, 6, 10, 8};
    int n = 16;
    int arr_size = sizeof(A)/sizeof(A[0]);
```

```
    printPairs(A, arr_size, n);

    getchar();
    return 0;
}
```

C++

```
 // C++ program to check if given array
// has 2 elements whose sum is equal
// to the given value
#include <bits/stdc++.h>

using namespace std;

void printPairs(int arr[], int arr_size, int sum)
{
    unordered_set<int> s;
    for (int i = 0; i < arr_size; i++)
    {
        int temp = sum - arr[i];

        if (temp >= 0 && s.find(temp) != s.end())
            cout << "Pair with given sum " << sum <<
                 " is (" << arr[i] << ", " << temp <<
                 ")" << endl;

        s.insert(arr[i]);
    }
}

/* Driver program to test above function */
int main()
{
    int A[] = {1, 4, 45, 6, 10, 8};
    int n = 16;
    int arr_size = sizeof(A)/sizeof(A[0]);

    // Function calling
    printPairs(A, arr_size, n);

    return 0;
}
```

Java

```
 // Java implementation using Hashing
```

```
import java.io.*;
import java.util.HashSet;

class PairSum
{
    static void printpairs(int arr[],int sum)
    {
        HashSet<Integer> s = new HashSet<Integer>();
        for (int i=0; i<arr.length; ++i)
        {
            int temp = sum-arr[i];

            // checking for condition
            if (temp>=0 && s.contains(temp))
            {
                System.out.println("Pair with given sum " +
                                    sum + " is (" + arr[i] +
                                    ", "+temp+")");
            }
            s.add(arr[i]);
        }
    }

    // Main to test the above function
    public static void main (String[] args)
    {
        int A[] = {1, 4, 45, 6, 10, 8};
        int n = 16;
        printpairs(A,  n);
    }
}

// This article is contributed by Aakash Hasija
```

Python

```
 # Python program to find if there are
# two elements wtih given sum

# function to check for the given sum
# in the array
def printPairs(arr, arr_size, sum):

    # Create an empty hash set
    s = set()

    for i in range(0,arr_size):
        temp = sum-arr[i]
```

```
        if (temp>=0 and temp in s):
            print ("Pair with the given sum is", arr[i], "and", temp)
        s.add(arr[i])

# driver program to check the above function
A = [1,4,45,6,10,8]
n = 16
printPairs(A, len(A), n)

# This code is contributed by __Devesh Agrawal__
```

C#

```
 // C# implementation using Hashing
using System;
using System.Collections.Generic;

class GFG
{
static void printpairs(int []arr,
                       int sum)
{
    HashSet<int> s = new HashSet<int>();
    for (int i = 0; i < arr.Length; ++i)
    {
        int temp = sum - arr[i];

        // checking for condition
        if (temp >= 0 && s.Contains(temp))
        {
            Console.Write("Pair with given sum " +
                          sum + " is (" + arr[i] +
                               ", " + temp + ")");
        }
        s.Add(arr[i]);
    }
}

// Driver Code
static void Main ()
{
    int []A = new int[]{1, 4, 45,
                        6, 10, 8};
    int n = 16;
    printpairs(A, n);
}
}
```

```
// This code is contributed by
// Manish Shaw(manishshaw1)
```

Output:

```
Pair with given sum 16 is (10, 6)
```

Time Complexity: O(n)
Auxiliary Space: O(n) where n is size of array.

If range of numbers include negative numbers then also it works. All we have to do for negative numbers is to make everything positive by adding the absolute value of smallest negative integer to all numbers.

Related Problems:
Given two unsorted arrays, find all pairs whose sum is x
Count pairs with given sumCount all distinct pairs with difference equal to k

Improved By : jit_t, manishshaw1

Source

https://www.geeksforgeeks.org/given-an-array-a-and-a-number-x-check-for-pair-in-a-with-sum-as-x/