JABALPUR ENGINEERING COLLEGE

GOKALPUR ,JABALPUR(M.P.) 482011

ASSIGNMENT

ANALYSIS AND DESIGN OF ALGORITHM

COMPUTER SCIENCE AND ENGINEERING

IV - SEMESTER

SUBMITTED TO-

Prof. J.S. THAKUR SIR

SUBMITTED BY-

KRISHNA KUMAR SURYAVANSHI

0201CS171038

Assignment : 1

1) Enlist and describe at least 5 problems (Other than those given in your syllabus) that can be solved using Divide & Conquer Approach. Give the solution of at-least one.

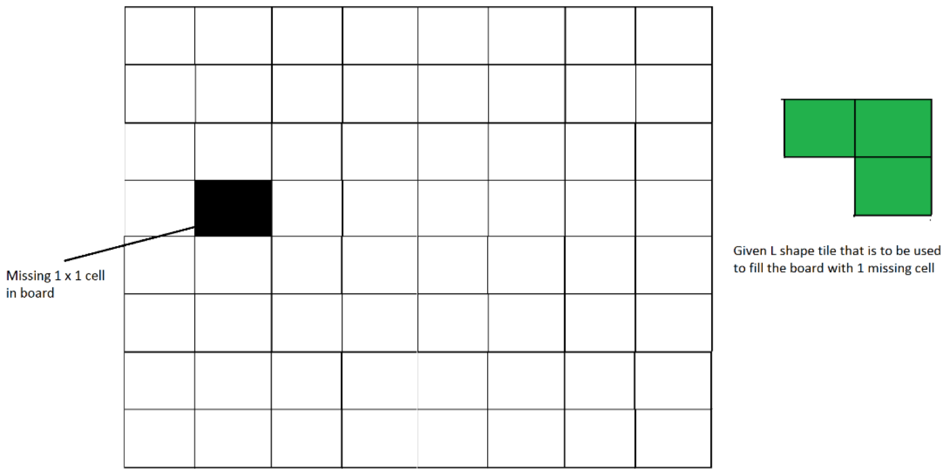
Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.

1. **Divide**: Break the given problem into subproblems of same type.
2. **Conquer**: Recursively solve these subproblems
3. **Combine**: Appropriately combine the answers

Following are some standard algorithms that are Divide and Conquer algorithms.

(a) Tiling Problem

Given a n by n board where n is of form 2^k where k >= 1 (Basically n is a power of 2 with minimum value as 2). The board has one missing cell (of size 1 x 1). Fill the board using L shaped tiles. A L shaped tile is a 2 x 2 square with one cell of size 1×1 missing.


Missing 1 x 1 cell in board
Given L shape tile that is to be used to fill the board with 1 missing cell

(b) Calculate pow(x,n)

Given two integers x and n, write a function to compute x^n. We may assume that x and n are small and overflow doesn't happen. We need to find $x^{n.}$

(c) Multiply two polynomials

Given,two polynomials represented by two arrays, write a function that multiplies given two polynomials. A simple solution is to one by one consider every term of first polynomial and multiply it with every term of second polynomial.

(d) Longest Common Prefix

Given a set of strings, find the longest common prefix.

Example :

```
Input  : {"apple", "ape", "april"}

Output : "ap"
```

(e) Maximum Subarray Sum using Divide and Conquer algorithm

You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.

For example, if the given array is {-2, -5, **6, -2, -3, 1, 5**, -6}, then the maximum subarray sum is 7 (see highlighted elements).

Detail Explanation of: Longest Common Prefix

Given two integers x and n, write a function to compute x^n. We may assume that x and n are small and overflow doesn’t happen.

Examples :

```
Input : x = 2, n = 3

Output : 8


Input : x = 7, n = 2

Output : 49
```

Below solution divides the problem into subproblems of size y/2 and call the subproblems recursively.

```
#include<iostream>

using namespace std;

class gfg

{


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

public:

int power(int x, unsigned int y)

{

    if (y == 0)

        return 1;

    else if (y % 2 == 0)

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

    else

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

}

};


/* Driver code */

int main()

{

    gfg g;
    int x = 2;

    unsigned int y = 3;

    cout << g.power(x, y);

    return 0;  }
```

2) Enlist and describe at least 5 problems (Other than those given in your syllabus) that can be solved using Greedy Approach. Give the solution of at least one.

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: *At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem*.

(a) Dijkstra's shortest path algorithm

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

(b) Greedy Algorithm to find Minimum number of Coins:

Given a value V, if we want to make change for V Rs, and we have infinite supply of each of the denominations in Indian currency, i.e., we have infinite supply of { 1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, what is the minimum number of coins and/or notes needed to make the change?

(c) Policemen catch thieves:

Given an array of size n that has the following specifications:

1. Each element in the array contains either a policeman or a thief.
2. Each policeman can catch only one thief.
3. A policeman cannot catch a thief who is more than K units away from the policeman.
   We need to find the maximum number of thieves that can be caught.

(d) Fitting Shelves Problem:

Given length of wall w and shelves of two lengths m and n, find the number of each type of shelf to be used and the remaining empty space in the optimal solution so that the empty space is minimum. The larger of the two shelves is cheaper so it is preferred. However cost is secondary and first priority is to minimize empty space on wall.

(e) Assign Mice to Holes:

There are N Mice and N holes are placed in a straight line. Each hole can accommodate only 1 mouse. A mouse can stay at his position, move one step right from x to x + 1, or move one step left from x to x -1. Any of these moves consumes 1 minute. Assign mice to holes so that the time when the last mouse gets inside a hole is minimized.

➢ Detail Explanation of:Dijkstra's shortest path algorithm

```
#include<iostream>

#include<climits>     /*Used for INT_MAX*/

using namespace std;

#define vertex 7

int minimumDist(int dist[], bool Dset[])

{

        int min=INT_MAX,index;                  =

        for(int v=0;v<vertex;v++)

        {

                if(Dset[v]==false && dist[v]<=min)

                {

                        min=dist[v];

                        index=v;
```

```
		}

	}

	return index;

}

void dijkstra(int graph[vertex][vertex],int src) /*Method to implement shortest path algorithm*/

{

	int dist[vertex];

	bool Dset[vertex];

	for(int i=0;i<vertex;i++)			/*Initialize distance of all the vertex to INFINITY and 
Dset as false*/

	{

		dist[i]=INT_MAX;

		Dset[i]=false;

	}

	dist[src]=0;					/*Initialize the distance of the source vertec to zero*/

	for(int c=0;c<vertex;c++)

	{

		int u=minimumDist(dist,Dset);

		Dset[u]=true;

		for(int v=0;v<vertex;v++)

		{

			if(!Dset[v] && graph[u][v] && dist[u]!=INT_MAX && 
dist[u]+graph[u][v]<dist[v])

			dist[v]=dist[u]+graph[u][v];

		}
```

```
        }

        cout<<"Vertex\t\tDistance from source"<<endl;

        for(int i=0;i<vertex;i++)                    /*will print the vertex with their distance from the
source to the console */

        {

                char c=65+i;

                cout<<c<<"\t\t"<<dist[i]<<endl;

        }

}

int main()

{

        int graph[vertex][vertex]={{0,5,3,0,0,0,0},{0,0,2,0,3,0,1},{0,0,0,7,7,0,0},

                                        {2,0,0,0,0,6,0},{0,0,0,2,0,1,0},{0,0,0,0,0,0,0},{0,0,0,0,1,0,0}};

        dijkstra(graph,0);

        return 0;
```

3). Enlist and describe at least 5 problems (Other than those given in your syllabus) that can be solved using Dynamic Programming Approach. Give the solution of at-least one.

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of sub problems, so that we do not have to re-compute them when needed later.

(a) Gold Mine Problem:

Given a gold mine of n*m dimensions. Each field in this mine contains a positive integer which is the amount of gold in tons. Initially the miner is at first column but can be at any row. He can move only (right->, right up /, right down\) that is from a given cell, the miner can move to the cell diagonally up towards the right or right or diagonally down towards the right. Find out maximum amount of gold he can collect.

(b) Friends Pairing Problem:

Given n friends, each one can remain single or can be paired up with some other friend. Each friend can be paired only once. Find out the total number of ways in which friends can remain single or can be paired up.

(c) Golomb sequence:

In mathematics, the Golomb sequence is a non-decreasing integer sequence where n-th term is equal to number of times n appears in the sequence.
The first few values are
1, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, ……

Explanation of few terms:
Third term is 2, note that three appears 2 times.
Second term is 2, note that two appears 2 times.
Fourth term is 3, note that four appears 3 times.

(d) Temple Offerings:

Consider a devotee wishing to give offerings to temples along a mountain range. The temples are located in a row at different heights. Each temple should receive at least one offering. If two adjacent temples are at different altitudes, then the temple that is higher up should receive more offerings than the one that is lower down. If two adjacent temples are at the same height, then their offerings relative to each other does not matter. Given the number of temples and the heights of the temples in order, find the minimum number of offerings to bring.

(e) Box Stacking Problem:

You are given a set of n types of rectangular 3-D boxes, where the i^th box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

➢ Detail Explanation of: Gold Mine Problem

Create a 2-D matrix goldTable[][]) of the same as given matrix mat[][]. If we observe the question closely, we can notice following.

1. Amount of gold is positive, so we would like to cover maximum cells of maximum values under given constraints.
2. In every move, we move one step toward right side. So we always end up in last column. If we are at the last column, then we are unable to move right

If we are at the first row or last column, then we are unable to move right-up so just assign 0 otherwise assign the value of goldTable[row-1][col+1] to right_up. If we are at the last row or last column, then we are unable to move right down so just assign 0 otherwise assign the value of goldTable[row+1][col+1] to right up.
Now find the maximum of right, right_up, and right_down and then add it with that mat[row][col].
At last, find the maximum of all rows and first column and return it.

Code:-

```
        int getMaxGold(int gold[][MAX], int m, int n)
{
   int goldTable[m][n];
   memset(goldTable, 0, sizeof(goldTable));

   for (int col=n-1; col>=0; col--)
   {
      for (int row=0; row<m; row++)
      {
         int right = (col==n-1)? 0: goldTable[row][col+1];
         int right_up = (row==0 || col==n-1)? 0:
                     goldTable[row-1][col+1];
                int right_down = (row==m-1 || col==n-1)? 0:
                     goldTable[row+1][col+1];
           goldTable[row][col] = gold[row][col] +
                     max(right, max(right_up, right_down));

      }
   }
        int res = goldTable[0][0];
```

```
    for (int i=1; i<m; i++)
        res = max(res, goldTable[i][0]);
    return res;
}
int main()
{
    int gold[MAX][MAX]= { {1, 3, 1, 5},
        {2, 2, 4, 1},
        {5, 0, 2, 3},
        {0, 6, 1, 2}
    };
    int m = 4, n = 4;
    cout << getMaxGold(gold, m, n);
    return 0;
}
```

4). Enlist and describe at-least 5 problems (Other than those given in your syllabus) that can be solved using Backtracking Approach. Give the solution of at-least one.

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).

OR

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

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.

Some of the problems which can be solved using Divide and Conquer technique

(a) Rat in a 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 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.

(b) Sudoku:

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 sub grid of size 3×3 contains exactly one instance of the digits from 1 to 9.

(c) Tug of War:

Given a set of n integers, divide the set in two subsets of n/2 sizes each such that the difference of the sum of two subsets is as minimum as possible. If n is even, then sizes of two subsets must be strictly n/2 and if n is odd, then size of one subset must be (n-1)/2 and size of other subset must be (n+1)/2.

For example, let given set be {3, 4, 5, -3, 100, 1, 89, 54, 23, 20}, the size of set is 10. Output for this set should be {4, 100, 1, 23, 20} and {3, 5, -3, 89, 54}. Both output subsets are of size 5 and sum of elements in both subsets is same (148 and 148).
Let us consider another example where n is odd. Let given set be {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4}. The output subsets should be {45, -34, 12, 98, -1} and {23, 0, -99, 4, 189, 4}. The sums of elements in two subsets are 120 and 121 respectively.

(d) Remove Invalid Parentheses:

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.

(e) Combinational Sum:

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).

➢ Detail Explanation of: Solution of Sudoku Problem:

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 sub grid. 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.

Code:-

```
        bool SolveSudoku(int grid[N][N])
{
   int row, col;



   if (!FindUnassignedLocation(grid, row, col))
      return true;



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

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

         grid[row][col] = num;
```

```
            if (SolveSudoku(grid))
                return true;



            grid[row][col] = UNASSIGNED;
        }
    }
    return false;
}
          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;
}
          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;
}
          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;
}
          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;
}
          bool isSafe(int grid[N][N], int row, int col, int num)
{
       current column and current 3x3 box */
    return !UsedInRow(grid, row, num) &&
           !UsedInCol(grid, col, num) &&
           !UsedInBox(grid, row - row%3 , col - col%3, num)&&
            grid[row][col]==UNASSIGNED;
}
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");
    }
}
          int main()
```

```
{

    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;
}
```

5). Enlist and describe at-east 5 problems (Other than those given in your syllabus) that can be solved using Branch & Bound Approach. Give the solution of at-least one.

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly. Some of the problems which can be solved using Divide and Conquer technique

(a). 8 puzzle Problem using Branch and Bound:

In this puzzle solution of 8 puzzle problem is discussed. *Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match final configuration using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space.*

(b). Job Assignment Problem:

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

(c). Construct a distinct elements array with given size, sum and element upper bound:

Given N size of the original array, SUM total sum of all elements present in the array and K such that no element in array is greater than K, construct the original array where all elements in the array are unique. If there is no solution, print “Not Possible”.

(d). Minimum positive integer required to split the array equally:

Given an array of N positive integers, the task is to find the smallest positive integer that can be placed between any two elements of the array such that, the sum of elements in the sub array occurring before it, is equal to the sum of elements occurring in the sub array after it, with the newly placed integer included in either of the two sub arrays.

(e). Minimum operations required to remove an array:

Given an array of **N** integers where N is even. There are two kinds of operations allowed on the array.

1. Increase the value of any element A[i] by 1.
2. If two adjacent elements in the array are consecutive prime number, delete both the element. That is, A[i] is a prime number and A[i+1] is the next prime number.

The task is to find the minimum number of operation required to remove all the element of the array.

➢ **Explanation of : 8 Puzzle Problem:**

The search for an answer node can often be speeded by using an “intelligent” ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an answer node. It is similar to backtracking technique but uses BFS-like search.

There are basically three types of nodes involved in Branch and Bound
1. Live node is a node that has been generated but whose children have not yet been generated.
2. E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

3. Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Code:

```
        /* Algorithm LCSearch uses c(x) to find an answer node
 * LCSearch uses Least() and Add() to maintain the list
   of live nodes
 * Least() finds a live node with least c(x), deletes
   it from the list and returns it
 * Add(x) adds x to the list of live nodes
 * Implement list of live nodes as a min heap */


struct list_node
{
   list_node *next;


   // Helps in tracing path when answer is found
   list_node *parent;
   float cost;
}


algorithm LCSearch(list_node *t)
{
   // Search t for an answer node
   // Input: Root node of tree t
   // Output: Path from answer node to root
   if (*t is an answer node)
   {
      print(*t);
      return;
```

```
}

E = t; // E-node

Initialize the list of live nodes to be empty;
while (true)
{
   for each child x of E
   {
       if x is an answer node
       {
          print the path from x to t;
          return;
       }
       Add (x); // Add x to list of live nodes;
       x->parent = E; // Pointer for path to root
   }

   if there are no more live nodes
   {
      print ("No answer node");
      return;
   }

   // Find a live node with least estimated cost
   E = Least();

   // The found node is deleted from the list of
   // live nodes
```

```
    }
}
```