Algorithm Analysis And Design

Contents

Chapter1: Introduction

1.1 What is an algorithm

An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

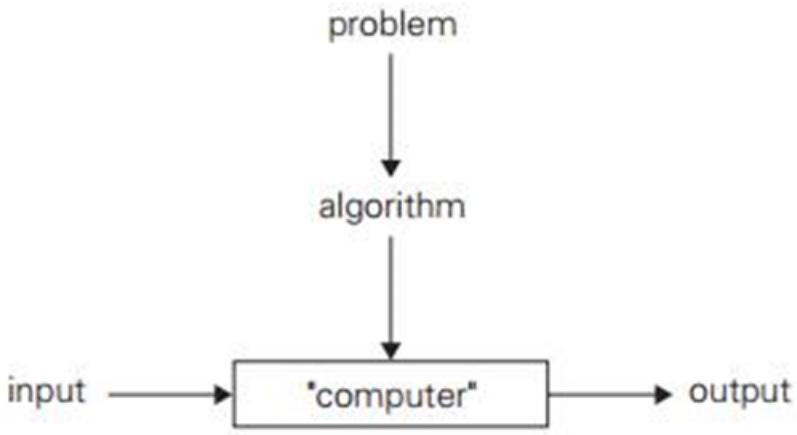


FIGURE 1.1 The notion of the algorithm.

1.2 Fundamentals of Algorithmic Problem Solving

We now list and briefly discuss a sequence of steps one typically goes through in designing and analyzing an algorithm (Figure 1.2).

Understanding the Problem

From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

Choosing between Exact and Approximate Problem Solving

The next principal decision is to choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an ***exact algo-rithm***; in the latter case, an algorithm is called an ***approximation algorithm.***

First, there are important prob-lems that simply cannot be solved exactly for most of their instances; examples include extracting square roots, solving nonlinear equations, and evaluating def-inite integrals. Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity.

Algorithm Design Techniques

Now, with all the components of the algorithmic problem solving in place, how do you design an algorithm to solve a given problem?

What is an algorithm design technique?

An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing like brute force, dynamic programming etc.

Of course, one should pay close attention to choosing data structures appro-priate for the operations performed by the algorithm.

Proving an Algorithm's Correctness

Once an algorithm has been specified, you have to prove its ***correctness***. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.

Analyzing an Algorithm

We usually want our algorithms to possess several qualities. After correctness, by far the most important is efficiency. In fact, there are two kinds of algorithm efficiency: time efficiency, indicating how fast the algorithm runs, and space efficiency, indicating how much extra memory it uses.

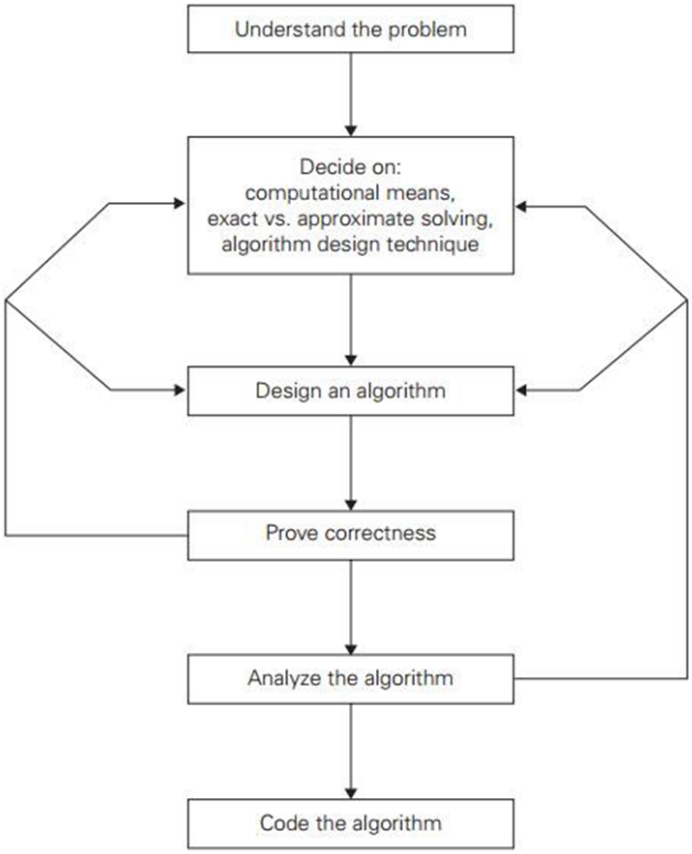


FIGURE 1.2 Algorithm design and analysis process.

1.3 Time Complexity

Time complexity is the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm.

When an algorithm uses statements that get executed only once, will always require the same amount of time, and when the statement is in loop condition, the time required increases depending on the number of times the loop is set to run. And, when an algorithm has a combination of both single executed statements and loop statements or with nested loop statements, the time increases proportionately, based on the number of times each statement gets executed.

Now, we will discuss algorithm design strategies.

CHAPTER-2: BRUTE-FORCE

Brute force is the straight-forward approach of solving algorithms, based on the problem statement and concepts involved.

Example1:

In the exponentiation problem, compute a to the power n (a^n) for a number ***a*** and a nonnegative integer ***n.*** This problem might seem trivial, it is very useful for illustrating several algorithm design strategies, such as the brute force. By the definition of exponentiation,

$$a^n = \underbrace{a * \cdots * a}_{n \text{ times}}.$$

This suggests simply computing a^n by multiplying 1 by a n times. Code goes as follows

```
int power( int a, int n)
{
        for(int i=0;i<n;i++)
  {
        x = x*a;
  }
 return x;
}
```

Problem1: Selection Sort

Selection sort algorithm sorts an array by finding the minimum element(if it is ascending order) and putting at the first and sorting the remaining unsorted array.

Pseudo-code for the algorithm goes as follows:

```
Selection_sort algorithm (int array, int n):  //n is length of array
        for i=0 to n:
        minimum_element  = array[0]
        for each unsorted element:
                if element < minimum_element
        element = new_minimum
        swap (minimum_element, first unsorted position)
```

Main-code:

```
// C program for implementation of selection sort
#include <stdio.h>

void selectionSort(int arr[], int n)
{
        int i, j, min_index;

        // One by one move boundary of unsorted subarray
        for (i = 0; i < n-1; i++)
```

```
    {
        // Find the minimum element in unsorted array
        min_index = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_index])
            min_index = j;

        // Swap the found minimum element with the first element
            int temp = arr[min_index];
            arr[min_index] = arr[i];
            arr[i]= temp;
    }
}
```

Time complexity: O(n^2) as there are two nested for loops.

Problem2: Bubble Sort:

It is one of the simplest sorting algorithms which functions by repeatedly swapping the adjacent numbers in the array if they are in wrong order.

code for bubble sort:

```
// C program for implementation of Bubble sort


// A function in c-language to implement bubble sort
void BubbleSort(int array[], int n)
{
int i, j;
for (i = 0; i < n-1; i++)
        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++)
                if (array[j] > array[j+1])
                        int temp = array[j];
                        array[j] = array[j+1];
                        array[j+1] = temp;
}
```

Time-complexity of the algorithm is O(n^2) due to two for-loops.

Problem3: Travelling Sales Man Problem:

Problem-Statement: The traveler should visit all the cities from the list, knowing the distance between all the cities and each city should visit only once. What is the nearest route to each city once and for all?

```
#include <bits/stdc++.h>

using namespace std;

int shortest_path_sum(int** edges_list, int num_nodes)

{

int source = 0;

vector<int> nodes;

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

{

        if(i != source)

        {

        nodes.push_back(i);

        }

}

int n = nodes.size();

int shortest_path = INT_MAX;

while(next_permutation(nodes.begin(),nodes.end()))
```

```
        {
            int path_weight = 0;
            int j = source;
            for (int i = 0; i < n; i++)
            {
            path_weight += edges_list[j][nodes[i]];
            j = nodes[i];
            }
            path_weight += edges_list[j][source];

            shortest_path = min(shortest_path, path_weight);
}
        return shortest_path;
        }
        };
```

Time-complexity of the algoririthm is O(n!).

CHAPTER-3: DIVIDE-CONQUER

In Divide and Conquer algorithms, the idea is to solve the problem in two sections.
1. The first section divides the problem into subproblems of the same type.

2. The second section is to solve the smaller problem independently and then add the combined result to produce the final answer to the problem.

Master-Theorem:

Master theorem is a theorem which we use to solve the recurrence relations of the form

$T(n) = a * T(n/b) + O(n^d)$ and the solution is shown as below -->(1)

Here the meaning is: cost of the work done for the problem of size n is equal cost of workdone of (a) number of subproblems of size n/b(assuming n/b as integer otherwise take a ceil of it) plus the additional workdone outside the recursive call which includes the cost of dividing the problem and merging.

$$T(n) = \begin{cases} O(n^d) & if d > log_b a \\ O(n^d log n) & if d = log_b a \\ O(n^{log_b a}) & if d < log_b a \end{cases}$$

If we draw the tree generated by the recurrence relation whose depth is log_b^a and branching factor a. If we check the tree at the k^{th} level there will be a^{k} nodes and each is a subproblem of size $\frac{n}{b^k}$ so

workdone at the k^{th} level is $O(\frac{a}{b^k})^d * a^k$

(Explanation: There are a^{k} nodes and each costs a constant time O(n^{d}) but here n is $\frac{a}{b^k}$ at kth level,see equation 1)

Therefore workdone at the k^{th} level can also be written as $O(n^d).(\frac{a}{b^d})^k$ It is summation of geometric series with ratio $\frac{a}{b^d}$ and first term n^{d}

So we have three cases when the ratio $\frac{a}{b^d}$ is

a) greater than 1, then the series is increasing and its sum is given by its last term $O(n^{log_b^a})$

$$= n^d(\frac{a}{b^d})^{log_b^n} = n^d(\frac{a^{log_b^n}}{b^{d*log_b^n}}) = a^{log_b^n} = n^{log_b^a}$$

b) less than 1, then the series is decreasing and its sum is given by the first term $O(n^d)$

c) equal to 1, in this case all O(logn) terms of the series are equal to n^{d}.Hence cost of work done = $O(n^d logn)$

There are several applications of the master theorem in different algorithms which uses divide and conquer method.It helps us to calculate the time complexity in simple and quick way.

Problem2: Strassens-Matrix Multiplication

Strassen's Matrix Mutiplication:

To multiply two n cross n matrices we can write divide the matrix into 4, n/2 cross n/2 size sub-matrices and can write the original matrix as $X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$

Here A, B, C, D are of n/2 cross n/2 size sub-matrices

Multiplication of two matrices X,Y are given by

$$X.Y = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

There are eight n/2-sized products: AE,BG,AF,BH,CE,DG,CF and DH

Time complexity of this is given by T(n)=8T(n/2)+O(n^2) where O(n^2) comes from the addition of numbers.

Using master theorem third case we get $T(n) = O(n^{log_2 8})$

We can do better than this by the following approach

$$X.Y = \begin{bmatrix} P_5 + P_4 - P2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where P1 = A(F-H) , P2=(A+B)H , P3= (C+D)E, P4=D(G-E), P5=(A+D)(E+H) P6=(B-D)(G+H) P7=(A-C)(E+F)

Problem3: Closest pair of points

Statement: Given an array of n points in the plane, and the problem is to find out the closest pair of points in the array.

Solution: Distance between two points p and q is given by

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

The Brute force solution is O(n^2), compute the distance between each pair of points and return the smallest.

Algorithm:

Let P[] be the given input array of points.

Firstly, the input array is sorted in increasing order according to x coordinates.

1) Find the middle point in the array sorted before, we can take *P[n/2]* as middle point.

2) Then we divide the given array in two halves. The first subarray have points from P[0] to P[n/2] and The second array have points from P[n/2+1] to P[n-1].

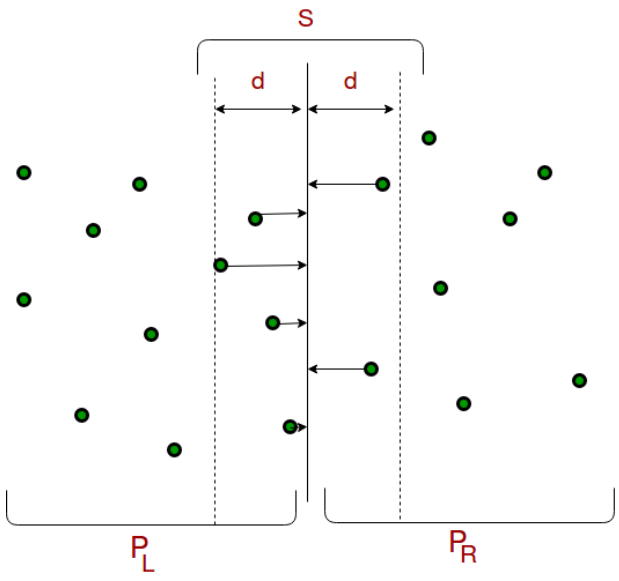
3) Do recursion to find the smallest distances in both subarrays P[0] to P[n/2] and P[n/2+1] to P[n-1]. Let the distances obtained be dl and dr and then Find the minimum of the two and let the minimum be given by d.

4)Now, we have an upper bound of the value d,(minimum distance). Now we need to consider the pairs such that one point in pair is from the left half and the other is from the right half. Consider the vertical line passing through P[n/2] and find all points whose x coordinate is closer than d to the middle vertical line. Build an array strip[] of all such points.

5) Sort the array strip[] according to y coordinates. This step is O(nLogn). It can be optimized to O(n) by recursively sorting and merging.

6) Find the smallest distance in strip[]. This is tricky. From the first look, it seems to be a O(n^2) step, but it is actually O(n). It can be proved geometrically that for every point in the strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate).

7) We will atlast return the minimum of value d and distance calculated in the above step



We need not consider all the points in the strip. You will consider only those points which are less than d distance from that point. For that, consider a 2d by d rectangle divided into 8 squares each of side d/2 such that either of the squares belongs completely to left or right of the drawn vertical line P[n/2] . No, two points can belong to the same d/2 square because the maximum distance between two points in the square is less than $d/\sqrt{2}$ which itself is less than d(minimum distance

between two points in the same square). So, 1 point can belong to one square at max and hence the 2d by d rectangle can have utmost 8 points and any point on this strip have to be compared to only other 7 points . If we try comparing points with points outside the rectangle , then they will surely have a distance greater than d. Hence, instead of each point being compared to every other point in the strip. We are comparing with at most only other 7 points in the strip.

Code:

```
float bruteforcesoln(Point P[], int n)
{
    float minimum = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (distance(P[i], P[j]) < minimum)
                minimum = dist(P[i], P[j]);
    return minimum;
}
//A function to calculate the minimum distance between a given set of points
float stripClosest(Point strip[], int size, float d)
{
    float minimum = d;  // Initialize the minimum distance as d
      quicksort(strip, size, sizeof(Point), compareY);
```

```
    // Pick all points one by one and try the next points till the difference
    // between y coordinates is smaller than d.
    // This is a proven fact that this loop runs at most 6 times
    for (int i = 0; i < size; ++i)
        for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
            if (distance(strip[i],strip[j]) < minimum)
                minimum = distance(strip[i], strip[j]);
       return min;
}

// A recursive function to find the smallest distance. The array P contains
// all points sorted according to x coordinate
float closestUtil(Point P[], int n)
{
    // If there are 2 or 3 points, then use brute force
    if (n <= 3)
        return bruteForce(P, n);

    // Find the middle point
    int mid = n/2;
    Point midPoint = P[mid];
```

```
// Consider the vertical line passing through the middle point
// calculate the smallest distance dl on left of middle point and
// dr on right side
float dl = closestUtil(P, mid);
float dr = closestUtil(P + mid, n-mid);

// Find the smaller of two distances
float d = min(dl, dr);

// Build an array strip[] that contains points close (closer than d)
// to the line passing through the middle point
Point strip[n];
int j = 0;
for (int i = 0; i < n; i++)
    if (abs(P[i].x - midPoint.x) < d)
        strip[j] = P[i], j++;

// Find the closest points in strip.  Return the minimum of d and closest
// distance is strip[]
return min(d, stripClosest(strip, j, d) );
```

```
}


// The main function that finds the smallest distance
// This method mainly uses closestUtil()
float closest(Point P[], int n)
{
    qsort(P, n, sizeof(Point), compareX);


    // Use recursive function closestUtil() to find the smallest distance
    return closestUtil(P, n);
}
```

Problem 4: Convex-hull problem:

Convex: A set of points(finite or infinite) in the plane is called convex if for any two points in the set P and Q , the entire line segment belongs to set.

Convex-hull: A convex hull is the smallest convex polygon containing all the given points.

Given the set of points for which we have to find the convex hull. Suppose we know the convex hull of the left half points and the right half points, then the problem now is to merge these two convex hulls and determine the convex hull for the complete set.

This can be done by finding the upper and lower tangent to the right and left convex hulls.

Now the problem remains, how to find the convex hull for the left and right half. Now recursion comes into the picture, we divide the set of points until the number

of points in the set is very small, say 5, and we can find the convex hull for these points by the brute algorithm. The merging of these halves would result in the convex hull for the complete set of points.

CHAPTER-4: DYNAMIC PROGRAMMING

4.1 Definition

Wherever we find a repetitive solution that has repeated calls for a single input, we can optimize it using Dynamic Programming. The idea is simply to save the results of small problems, so that we do not have to calculate them again when needed later. This simple optimization reduces time complexity from exponential to polynomial. For example, if we write simple recursive solutions for Fibonacci Numbers, we get the exponential time complex and if we optimize it by storing the solution of subproblems, the complex time decreases linear.

Problem 4.2:Longest Subsequence Problem

If there is a sequence of numbers a1,a2,....,an. we call the subsequence increasing if all the numbers in the subsequence and the respective indices of the subsequence in the sequence are increasing. (See here 1,2,3,...n are indices).

The longest subsequence of the sequence 5,2,8,6,3,6,9,7 is 2,3,6,9.

We can write the numbers in the sequence and add an edge from a{i} to a{j} if a{i} is less than a{j} where i < j. This is done for every number in the sequence. Now, it will become a linear topological order. From now, it's just the calculation of the longest path in a directed acyclic graph.

Let L(j) be the length of the longest path that ends at j^{th} vertex. Then,

L(j) = 1+max(L(i)) where i < j and i,j has the directed edge from i io j.(or array{i} < array{j})

Other Methods:

We can use recursion to implement the above logic. But the timecomplexity becomes exponential in this case, as repeatedly we have to calculate the same value like in the n^{th} fibonacci number calculation .

```
int func(int array[],int n,int* maximum )

{

  if(n==1) return 1;

  int max=1,ans=1;

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

   {

    ans = func(array,i,maximum)

     if(array[i-1] < array[n-1] && ans+1>max)

     max=ans+1;

   }

   if(max>*maximum)  *maximum=max;

   return max;

}
```

Another method is memoization. We store the values of L(i) calculated before and use them later.

```
int func( int arr[],int n)

{

   Vector L(n,1)

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

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

        if(arr[j] < arr[i] && L[i] < L[j]+1)  L[i] = L[j]+1
```

```
    }
     sort(L,L+n);
     return L[n-1];
}
```

Problem2:

The minimum number of insert or delete or overwrite to convert one word to other.

A natural measure of the distance between two strings is the extent to which they can be aligned , or matched up.The cost of the alignment is the number of columns in which the letters differ and the edit distance between two strings is the cost of their best possible alignment.

First our goal is to find the editing ditance…

Cost = number of blanks + number of mismatched letters.

Let x, y be the strings such that x should be converted to y using minimum cost. let's divide it into smaller subproblems. Let us denote E{i,j} as the editing distance for the substrings x{1,2...i} and y{1,2,...,j}. If we see the last column of alignment there are only three possibilities

x{i} ____ x{i}

____ y{j} y{j}

```
    for i=1,2....,m:
   E{i,0}=i;
   for j=1,2.....,n:
   E{0,j}=j;
```

for i=1,2....,m:

for j=1,2.....,n:

E{i,j}=min(1+E{i-1,j},1+E{i,j-1},diff{i,j}+E{i-1,j-1})

return E{m,n}

so , we can write E{i,j}=min(1+E{i-1,j},1+E{i,j-1},diff{i,j}+E{i-1,j-1}) and diff{i,j}=1 if x{i}==y{j} else 0 -->(1)

E(i,0)=i because all the elements are mismatch as we are not taking any elements from the second string..We should delete all the elements in x{0,1,2...i} to get a empty string. Similarly E(0,j)=j as we have to add y{0,1,2...j} to get the empty substring.These are called the base states

the three arguments in the min function represents the three situations shown before.

If we write as a table with rows and columns headings as the letters of the two strings and the values in it as E{i,j} for ith row and jth column, and then we can see the recursion or the main equation (1) by inspecting the element from the top block, element at the left diagonal, element from the left block.

E{i,j} is the editing cost for substring x{0,1,...i},y{0,1,....j}. If we want to get that value go to the i^th row and j^th column that is E{i,j}. E{i-1,j} is the element attached at the top, E{i,j-1} is the left element attached, E{i-1,j-1} is the left diagonal element. So, we can decide E{i,j} by looking the surrounded values in the table.

Timecomplexity of the algorithm is O(mn); if m,n are the lengths of the two strings given..

Code:

```
min(int x, int y, int z) { return min(min(x, y), z); }

int editDistDP(string str1, string str2, int m, int n)
{
   // Create a table to store results of subproblems
   int dp[m + 1][n + 1];

   // Fill d[][] in bottom up manner
   for (int i  = 0; i <= m; i++) {
```

```
        for (int j = 0; j <= n; j++) {
            // If first string is empty, only option is to
            // insert all characters of second string
            if (i == 0)
                dp[i][j] = j; // Min. operations = j

            // If second string is empty, only option is to
            // remove all characters of second string
            else if (j == 0)
                dp[i][j] = i; // Min. operations = i

            // If last characters are same, ignore last char
            // and recur for remaining string
            else if (str1[i - 1] == str2[j - 1])
                dp[i][j] = dp[i - 1][j - 1];

            // If the last character is different, consider
            // all possibilities and find the minimum
            else
                dp[i][j]
                    = 1
                      + min(dp[i][j - 1], // Insert
                            dp[i - 1][j], // Remove
                            dp[i - 1][j - 1]); // Replace
        }
    }

    return dp[m][n];
}
```

Problem3: Minimum number of coins hat can make a given value

Statement: Given a value V, if we want to make a change for V cents, and we have an infinite supply of each of C = { C1, C2, .., Cm} valued coins, what is the minimum number of coins to make the change? If it's not possible to make a change, print -1.

If V == 0, then 0 coins required.
If V > 0
  minCoins(coins[0..m-1], V) = min {1 + minCoins(V-coin[i])}
                        where i varies from 0 to m-1
                        and coin[i] <= V

Code:

```
// m is size of coins array (number of different coins)
int minCoins(int coins[], int m, int V)
{
    // table[i] will be storing the minimum number of coins
    // required for i value.  So table[V] will have result
    int table[V+1];

    // Base case (If given value V is 0)
    table[0] = 0;

    // Initialize all table values as Infinite
    for (int i=1; i<=V; i++)
        table[i] = INT_MAX;

    // Compute minimum coins required for all
    // values from 1 to V
    for (int i=1; i<=V; i++)
    {
```

```
        // Go through all coins smaller than i
        for (int j=0; j<m; j++)
          if (coins[j] <= i)
          {
              int sub_res = table[i-coins[j]];
              if (sub_res != INT_MAX && sub_res + 1 < table[i])
                  table[i] = sub_res + 1;
          }
    }


     if(table[V]==INT_MAX)
if(table[V]==INT_MAX)
        return -1;


    return table[V];
}
```

The time complexity of the above solution is O(mV).

CHAPTER-5 : BACKTRACKING

Backtracking is an algorithmic-method for solving recurring problems by trying to create a scalable solution, one piece at a time, removing the inconsistent solutions that meet the needs of the problem at any time. (by time, here, refers to. time passed to every level of search engine)

Problem statement:

Find a way to place 8 Queens on 8x8 chess word such that queens are not(in the same row or the same column or the diagonal). Print all the possible considerations.

Explanation:

Backtracking algorithm is used to solve this problem. This algorithm checks all the possible configuration and test whether the required result is obtained or not.

Start

1.Start from the first column

2.If all 8 Queens of pleasure on the chess board, return true print configuration

3.Check for all rows in the current column

a) If queen placed safely mark row and column; and recursively check if we approach in the current configuration, do we obtain a solution or not

b) If placing yields a solution, return true

c) If placing does not yield a solution, unmark and try other rows

4.if all rows tried and solution not obtained, return false and backtrack

End

Code:

```
include <bits/stdc++.h>

using namespace std;

int chessboard [8][8];


/* A  function to check if a queen can be placed on board[row][column]. As if we are keeping the queen at row, column at a given time then in all the columns 0 to column-1 queen are filled and from column+1 to 8 no queens are there */
```

```
bool QueenSafe(int chessboard[8][8], int row, int column)
{
        int i, j;

        for (j = 0; j < col; j++)
                if (chessboard[row][j])
                        return false;
       // checking the upper left diagonal contains queen are not
        for (i = row, j = column; i >= 0 && j >= 0; i--, j--)
                if (chessboard[i][j])
                        return false;
        // Checking the lower diagonal on left side
        for (i = row, j = column; j >= 0 && i < 8; i++, j--)
                if (chessboard[i][j])
                        return false;
        return true;
}

bool solve(int chessboard[8][8], int column)
{
                if (column >= 8){
                        return true;
}
```

```
        //Try keeping  the queen one by one in all the rows
        for (int i = 0; i < 8; i++) {
                if (QueenSafe(board, i, column)) {
                        board[i][column] = 1;


                        // do recursion for placing rest of the queens */
                        if (solve(board, column + 1)){
                                return true;
                        }


// If placing queen in board[i][column] doesn't lead to a solution, then remove queen from board[i][col]
                        board[i][column] = 0; // BACKTRACK
                }
        }


                return false;
}
```

This can be generalised to N*N chessboard problem.

2. M-colouring problem :

Given an undirected graph and an integer m, we have to determine if the graph can be coloured with at most m colours such that no two adjacent vertices of the graph are colored with the same color.

The idea is to give a single color to different vertices, starting with vertex 0. Before assigning color, check for safety by considering the color already assigned to the vertices next to it is to check if the parts on the side has the same color or not.

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

```
bool check(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;

}
```

```
bool m-colouring(  bool graph[V][V], int m, int color[], int v){

  if (v == V)   return true;

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

      // Check if the allocation of color i to vertex v is fine

      if (check(  v, graph, color, c)) {    //check function checks if adjacent colors are same or not

          coloring[v] = i;

          if( m-coluring( graph, m, color, v + 1) == true) return true

         // If  the assignment of  color i not lead to a solution then remove it

           color[v] = 0;

      }

  }
```

```
  //If no color is assigned to the vertex then return false for the function

   return false

}

bool coloring2(bool graph[V][V], int m)

    int color[V];  /

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

 color[j] = 0;

}

  if (  m-colouring( graph, m, color, 0)== false) {

                return false;

    }
```

```
//Print the colour vertex


 return true;


}
```

CHAPTER-6: GRAPH TRAVERSAL PROBLEMS

Important graph algorithms:

Breadth-First Traversal:

1. First, mark all nodes unvisited & start from a given node(or any arbitrary node, if nothing is given) and push it to the queue.
2. Remove the element from the queue, mark it visited, print it.
3. Visit the adjacent unvisited node of the current node, and push them into the queue.
4. Go to step 2, and repeat the step 2-4 until the queue is not empty and all nodes are not visited.
5. If all nodes are visited, then stop.

Code:

```
// BFS algorithm
void bfs(struct Graph* graph, int startVertex) {
  struct queue* q = createQueue();


  graph->visited[startVertex] = 1;
  enqueue(q, startVertex);
```

```
  while (!isEmpty(q)) {
    printQueue(q);
    int currentVertex = dequeue(q);
    printf("Visited %d\n", currentVertex);

    struct node* temp = graph->adjLists[currentVertex];

    while (temp) {
      int adjVertex = temp->vertex;

      if (graph->visited[adjVertex] == 0) {
        graph->visited[adjVertex] = 1;
        enqueue(q, adjVertex);
      }
      temp = temp->next;
    }
  }
}
```

Time complexity is O(V+E).

Recursion:

The method in which a function calls itself is called recursion and the corresponding function is known as recursive function.

Example:

Obtain the n^th fibonacci number.

According to the definition,of fibonacci numbers, f(n) = f(n-1) + f(n-2)

So, we can write the function as,

```
int Fibonacci(int n)
{
  if(n==0) return 0;
        else if(n==1) return 1;
        else return (Fibonacci(n-1)+Fibonacci(n-2));
}
```

Depth-First Traversal:

DFS (Depth-first search) is technique used for traversing the graph. Here backtracking is used for traversal. In this traversal first the deepest node is visited and then backtracks to it's parent node if no sibling of that node exist.

```
void Graph_DFS(int v)
{
   visitedvertex[v] = true;
   printf("%d ",v );
   list<int>iterator j;
   for (int j = adjacent[v].begin(); j != adjacent[v].end(); ++j)
      if (!visited[*j])
```

```
    Graph_DFS(*j);
}
```

Time complexity is given by O(V+E).

GREEDY-APPROACH

Greedy algorithms solve problems by making the choice that seems best at particular moment.

Greedy strategy works on the optimization problems very good. It has the following characteristics.

1. Greedy Choice Property 2. Optimal Substructure.
2. Greedy Choice Property: Global optimum can be achieved by selecting local optimum.
3. Optimal Substructure: Optimal solution to our problem S contains optimal solutions for the subproblems of S.

Problem1: Kruskals Algorithm (Finding MST)

Definition: Minimum Spanning Tree

If we are given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that connects all the vertices together and is a tree. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, undirected,connected, graph is a spanning tree with a weight less than or equal to the weight of every other possible spanning tree. The weight of a spanning tree is the sum of weights of all the edge sof the spanning tree.

Question: How many edges does a minimum spanning tree has?

A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

Approach:

1. Sort the given edges in non-decreasing order of their weight.

2. Select the smallest edge and check whether it forms a cycle or not with the spanning we got so far and if it is not formed include this edge .

3. Repeat 2 till there will be v-1 edges in the set.

Pseudo-code:

```
KRUSKAL(graph G)
 MST = {}
 for each vertex v belonging G.V:
   MAKE-SET(v)
 for each (u, v) in G.E ordered by weight(u, v), increasing:
   if FIND-SET(u) != FIND-SET(v):
      add {(u, v)} to set MST
      UNION(u, v)

return MST
```

Kruskal's algorithm is one of a number of applications that require a dynamic partition of some n element set S into a collection of disjoint subsets $s1, s2, \ldots, sk$. After being initialized as a collection of n one-element subsets, each containing a different element of S, the collection is subjected to a sequence of intermixed union and find operations. (Note that the number of union operations in any such sequence must be bounded above by $n - 1$ because each union increases a subset's size at least by 1 and there are only n elements in the entire set S.) Thus,

we are dealing here with an abstract data type of a collection of disjoint subsets of a finite set with the following operations:

makeset(x) creates a one-element set {x}. It is assumed that this operation can be applied to each of the elements of set S only once.

find(x) returns a subset containing x.

union(x, y) constructs the union of the disjoint subsets Sx and Sy containing x and y, respectively, and adds it to the collection to replace Sx and Sy, which are deleted from it.

Time Complexity**:** O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be at most O(V2), so O(LogV) is O(LogE) the same. Therefore, the overall time complexity is O(ElogE) or O(ElogV).