Algorithm Analysis And Design

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.

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.***

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.

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.

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: DIVIDE-CONQUER

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

Example1:

Consider the exponentiation problem, compute a^n for a nonzero number ***a*** and a nonnegative integer ***n.*** Although this problem might seem trivial, it provides a useful vehicle for illustrating several algorithm design strategies, including the brute force. By the definition of exponentiation,

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;
}
```

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.

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.

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;
        }
        };
                        edges_list[i][j] = 0;
```

}

}

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

CHAPTER-3: DIVIDE-CONQUER

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.

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)

CHAPTER-4: DYNAMIC PROGRAMMING

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.

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 lemlongest 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 in range(1,n)

   {

     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)

{

   vectorL(n,1)

   for i in range(1,n){

      for j in range(0,i)

         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 maximum editing ditance...

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{0,1,2...i} and y{0,1,2,...,j}. If we see the last column of alignment there are only three possibilities

x{i} ____ x{i}

____ y{j} y{j}

for i=0,1,2....,m:

```
E{i,0}=i;

for j=0,1,2.....,n:

E{0,j}=j;

for i=0,1,2....,m:

  for j=0,1,2.....,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..

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 8 x8 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.

The time complexity of approach is O(N!)

Pseudocode

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];


bool isPossible(int n,int row,int column ){

  for(int x=row-1;x>=0;x--){

    if(chessboard [x][column ] == 1){

      return false;

          }

  }


  for(int x=row-1,y=column -1;x>=0 && y>=0 ; x--,y--){

    if(chessboard [x][y] ==1){

      return false;

          }

  }


  for(int x=row-1,y=column +1;x>=0 && y<n ; x--,y++){

    if(chessboard [x][y] == 1){

      return false;

          }

  }
```

```
  return true;
}
void queenHelp(int n,int row){
  if(row==n){
    for(int x=0;x<n;x++){
      for(int y=0;y<n;y++){
        cout << chessboard [x][y] << " ";
      }
          }
    cout<<endl;
    return;

  }

  for(int y=0;y<n;y++){
    if(isPossible(n,row,y)){
      chessboard [row][y] = 1;
      queenHelp(n,row+1);
```

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 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; i++){
```

```
 color[i] = 0;


}


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


                return false;


    }


//Print the colour vertex


 return true;


}
```

CHAPTER-6: GRAPH TRAVERSAL PROBLEMS

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)
{
   visitedverex[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);
}
```

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.

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.