

Q.1(a) Explain Complexity classes and polynomial time algorithms

05

Ans.: The Theory of Complexity deals with

- the classification of certain “decision problems” into several classes: the class of “easy” problems, the class of “hard” problems, the class of “hardest” problems;
- relations among the three classes;
- properties of problems in the three classes.

Polynomial Time algorithm :

Definition: An algorithm is polynomial-time if its running time is $O(n^k)$, where k is a constant independent of n , and n is the input size of the problem that the algorithm solves.

Remark: Whether you use n or n^α (for fixed $\alpha > 0$) as the input size, it will not affect the conclusion of whether an algorithm is polynomial time.

This explains why we introduced the concept of two functions being of the same type earlier on. Using the definition of polynomial-time it is not necessary to fixate on the input size as being the exact minimum number of bits needed to encode the input!

Examples of Polynomial-Time Algorithms :

- The standard multiplication algorithm learned in school has time $O(m_1 m_2)$ where m_1 and m_2 are respectively, the number of digits in the two integers.
- DFS has time $O(n + e)$
- Kruskal’s MST algorithm runs in time $O((e + n) \log n)$.

Q.1(b) Explain general method for greedy algorithms.

05

Ans.: **Greedy Method :**

The greedy method is one of the most straight forward algorithm design technique ; and it can be applied to a wide variety of problems.

The greedy method suggests that one can devise an algorithm which works in stages, considering one input at a time. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of next input into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution.

(Most problems have n input and require us to obtain a subset that satisfies some constraints are called a feasible solution. We are required to find a feasible solution that optimizes (minimizes or maximizes) a given objective function. A feasible solution that does this is called an optimal solution.)

Q.1(c) Explain optimal storage of tapes.

05

Ans.: Optimal Storage of Tapes

On a computer tape of length ℓ there are n programs to be stored. With each program, i is a length associated, $1 \leq i \leq n$. If the sum of all the lengths of the programs is at most ℓ , all programs can be stored on the tape. When a program has to be retrieve from this tape, assume the tape is initially positioned at the front. Therefore if the programs are stored in the order $I = i_1, i_2, \dots, i_n$, the time t_j needed to retrieve program i_j is proportional to $\sum_{1 \leq k \leq j} \ell_{i_k}$. The expected or mean retrieval time (MRT) is $\left(\frac{1}{n}\right) \sum_{1 \leq j \leq n} t_j$, if all the programs are retrieved equally often. In the optimal storage an tape problem. We need to find a permutation for the n programs. So that when they are stored on the tape, the MRT is minimized, in this order.

e.g. Let $n = 3$ & $(\ell_1, \ell_2, \ell_3) = (5, 10, 3)$. There are $n! = 6$ possible orderings.

Ordering I	D(I) [respective d values]
1, 2, 3	$5 + 5 + 10 + 5 + 10 + 3 = 38$
1, 3, 2	$5 + 5 + 3 + 5 + 3 + 10 = 31$
2, 1, 3	$10 + 10 + 5 + 10 + 5 + 3 = 43$
2, 3, 1	$10 + 10 + 3 + 10 + 3 + 5 = 41$
3, 1, 2	$3 + 3 + 5 + 3 + 5 + 10 = 29$
3, 2, 1	$3 + 3 + 10 + 3 + 10 + 5 = 34$

The optimal ordering is 312

Q.1(d) Implement binary search and derive its completely.

05

Ans.: Binary Search :

Binary search technique is very fast and efficient. It requires the list of elements to be in sorted order.

In this method, to search an element we compare it with the element present at the centre of the list. If it matches then the search is successful otherwise, the list is divided into two halves—one from the 0th element to the centre element (first half), another from the centre element to the last element (second half). As a result all the elements in first half are smaller than centre element whereas all the elements in second half are greater than the centre element.

The searching will now proceed in either of the two halves depending upon whether the target element is greater or smaller than the centre element. If the element is smaller than the centre element then the searching is done in the first half, otherwise it is done in the second half.

The process of comparing the required element with the centre element and if not found then dividing the elements into two halves is repeated till the element is found or the division of half parts gives one element.

0	1	2	9	10	11	15	20	46	72
---	---	---	---	----	----	----	----	----	----

Sorted List for Binary Search

Let us take an array `arr` that consists of 10 sorted numbers and 46 is the element that is to be searched. The binary search when applied to this array works as follows:

- (a) 46 is compared with the element present at the centre of list (i.e 10) since 46 is greater than 10, the sorting is done in the second half of the array.
- (b) Now, 46 is compared with the centre element of the second half of the array (i.e. 20). Again, 46 is greater than 20, the searching will be done between 20 and the last element 72.
- (c) The process is repeated till 46 is found or no further subdivision of array is possible.

Binary Search can be analyzed with the best, worst, and average case number of comparisons. These analyses are dependent upon the length of the array, so let $N = |\text{arr}|$ denote the length of the Array `arr`.

Best case: $O(1)$ comparisons

In the best case, the item to be found 'X' (in our case, 46) is the middle in the array `arr`. A constant number of comparisons (actually just 1) are required.

Worst case: $O(\log n)$ comparisons

In the worst case, the item X does not exist in the array `arr` at all. Through each recursion or iteration of Binary Search, the size of the admissible range is halved. This halving can be done ceiling $(\log n)$ times. Thus, ceiling $(\log n)$ comparisons are required.

Average case: $O(\log n)$ comparisons

To find the average case, take the sum over all elements of the product of number of comparisons required to find each element and the probability of searching for that element. To simplify the analysis, assume that no item which is not in `arr` will be searched for, and that the probabilities of searching for each element.

Q.2(a) Explain how to find maximum and minimum value in an array using Divide and Conquer. 10

Ans.: In this algorithm, the list of elements is divided at the mid in order to obtain two sublists. From both the sublist maximum and minimum elements are chosen. Two maxima and minima are compared and from them real maximum and minimum elements are determined. This process is carried out for entire list. The algorithm is as given below.

```

Algorithm Max_Min_Val (i, j, max, min)
//Problem Description : Finding min, max elements recursively.
//Input : i, j are integers used as index to an array A. The max and min will
//contain maximum and minimum value elements.
//Output : None
if (i = j) then
{
    max ← A [i]
    min ← A [j]
}
else if (i = j-1) then
{

```

```

if (A [i] < A [j]) then
{
    max ← A [j]
    min ← A [i]
}
else
{
    max ← A [i]
    min ← A [j]
} //end of else
} //end of if
else
{
    mid ← (i+j) / 2 //divide the list handling two lists separately
    Max_Min_Val (i, mid, max, min)
    Max_Min_val (mid + 1, j, max_new, min_new)
    if (max < max_new) then
        max ← max_new //combine solution
    if (min > min_new) then
        min ← min_new //combine solution
}

```

Example : Consider a list of some elements from which maximum and minimum element can be found out.

1	2	3	4	5	6	7	8	9
50	40	-5	-9	45	90	65	25	75

Step 1 :

1	2	3	4	5	Sublist 1
50	40	-5	-9	45	

6	7	8	9	Sublist 2
90	65	25	75	

We have divided the original list at mid and two sublists : Sublist 1 and sublist 2 are created. We will find min and max values respectively from each sublist.

Step 2 :

1	2	3	4	5	Sublist
50	40	-5	-9	45	

6	7	8	9	Sublist
90	65	25	75	

Again divide each sublist and create further sublists. Then from each sublist obtain.

Step 3:

1	2	3	4	5
50	40	-5	-9	45

6	7
90	65

8	9
25	75

It is possible to divide the list (50, 40, -5) further. Hence we have divided the list into sublists and min, max values are obtained.

Step 4 : Now further division of the list is not possible. Hence we start combining the solutions of min and max values from each sublist.

1	2
50	40

3
-5

Combine (1, 2) and (3) Min = -5, Max = 50

1	2	3
50	40	-5

4	6
-9	45

Combine (1, ... 3) and (4, 5) Min = -9, Max = 50

Now we will combine (1, ... 3) and (4, 5) and the min and max values among them are obtained. Hence, min value = -9

max value = 50

Step 5:

6	7
90	65

8	9
25	75

Combine (6, 7) Min = 25, Max = 90

Step 6:

1	2	3	4	5
50	40	-5	-9	45

6	7	8	9
90	65	25	75

Combine the sublists (1, 5) and (6,9). Find out min and max values which are min = -9 max = 90

Thus the complete list is formed from which the min and max values are obtained. Hence final min and max values are Min = -9 and max = 90

Analysis :

There are two recursive calls made in this algorithm, for each half divided sublist. Hence, time required for computing min and max will be

$$T(n) = T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 \quad \text{when } n > 2$$

$$T(n) = 1 \quad \text{when } n = 2$$

If single element is present in the list then $T(n) = 0$.

Now, time required for computing min and max will be :

$$\begin{aligned}
 T(n) &= 2 \underbrace{T(n/2)} + 2 \\
 &\quad \downarrow \\
 &= 2 [2 \underbrace{T(n/4)} + 2] + 2 \\
 &\quad \quad \downarrow
 \end{aligned}$$

$$= 2(2 [2T(n/8) + 2] + 2) + 2$$

$$= 8T(n/8) + 10$$

Continuing in this fashion a recursive equation can be obtained. If we put $n = 2^k$ then

$$T(n) = 2^{k-1}T(2) + \sum_{i=1}^{k-1} 2^i = 2^{k-1} + 2^k - 2$$

$$T(n) = 3n / 2 - 2$$

Neglecting the order of magnitude, we can declare the time complexity is $O(n)$.

Q.2(b) Explain Strassen matrix multiplication. 10

Ans.: In 1969, Strassen published an algorithm whose time complexity is better than cubic in terms of both multiplications and additions/subtractions. The following example illustrates his method.

Suppose we want the product C of two 2×2 matrices, A and B. That is,

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

Strassen determined that if we let

$$m_1 = (a_{11} + a_{22})(b_{11} + b_{22})$$

$$m_2 = (c_{21} + a_{22})b_{11}$$

$$m_3 = a_{11}(b_{12} - b_{22})$$

$$m_4 = a_{22}(b_{21} - b_{11})$$

$$m_5 = (a_{11} + a_{12})b_{22}$$

$$m_6 = (a_{21} - a_{11})(b_{11} + b_{12})$$

$$m_7 = (a_{12} - a_{22})(b_{21} + b_{22})$$

the product C is given by

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

In the exercises, you will show that this is correct.

To multiply two 2×2 matrices, Strassen's method requires seven multiplications and 18 additions/subtractions, whereas the straightforward method requires either multiplications and four additions. We have saved ourselves one multiplication at the expense of doing 14 additional additions or subtractions. This is not very impressive, and indeed it is not in the case of 2×2 matrices that Strassen's method is of value. Because the commutativity of multiplications is not used in Strassen's formulas, those formulas pertain to larger matrices that are each divided into four submatrices. First we divide the matrices A and B, as illustrated in figure . Assuming that n is a power of 2, the matrix A_{11} , for example, is meant to represent the following submatrix of A:

$$\begin{array}{c} \leftarrow n/2 \rightarrow \\ \uparrow \\ n/2 \\ \downarrow \end{array} \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} \dots a_{1,n/2} \\ a_{21} & a_{22} \dots a_{2,n/2} \\ \dots & \dots \\ a_{n/2,1} & \dots a_{n/2,n/2} \end{bmatrix}$$

Using Strassen's method, first we compute

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

Where our operations are now matrix addition and multiplication. In the same way, we compute M_2 through M_7 . Next we compute

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

and C_{12} , C_{21} , and C_{22} . Finally, the product C of A and B is obtained by combining the four submatrices C_{13} . The following example illustrates these steps.

Suppose that $A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}$ $B = \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}$

Figures illustrates of the partitioning in Strassen's method. The computations proceed as follows:

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ &= \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) \times \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right) \\ &= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} \\ &\begin{matrix} \uparrow & \leftarrow 2 & \rightarrow \\ 2 \downarrow \end{matrix} \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} \times \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix} \end{aligned}$$

When the matrices are sufficiently small, we multiply in the standard way. In this example, we do this when $n = 2$. Therefore,

$$\begin{aligned} M_1 &= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} \\ &= \begin{bmatrix} 3 \times 17 + 5 \times 7 & 3 \times 10 + 5 \times 9 \\ 11 \times 17 + 13 \times 7 & 11 \times 10 + 13 \times 9 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix} \end{aligned}$$

After this, M_2 through M_7 are computed in the same way, and then the values of C_{11} , C_{12} , C_{21} , and C_{22} are computed. They are combined to yield C .

Strassen's algorithm is slightly faster than the general matrix multiplication algorithm. The general algorithm's time complexity is $O(n^3)$, while the Strassen's algorithm is $O(n^{2.8})$

Q.3(a) Explain single source (Bellman Ford) Algorithm with an example.

10

Ans. : Algorithms Bellman and Ford algorithm to compute shortest paths

Algorithm BellmanFord(v, cost, dist, n)

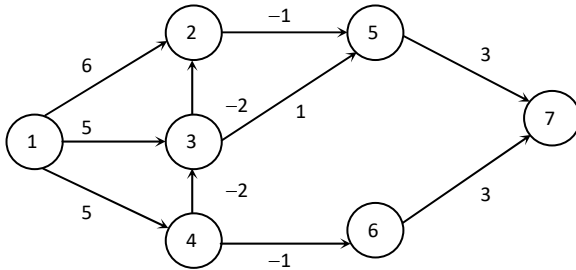
```
// Single-source / all-destinations shortest
// paths with negative edge costs
{
  for i : = 1 to n do // Initialize dist.
    dist[i] := cost[v, i];
  for k : =2 to n - 1 do
    for each u such that u ≠ v and u has
      at least one incoming edge do
      for each (i, u) in the graph do
        if dist[u] > dist[i] + cost[i, u] then
          dist[u] := dist[i] + cost[i, u];
}
```

Example: Figure below gives a seven –vertex graph, together with the arrays $dist^k$, $k = 1, \dots, 6$. These arrays were computed using the equation just given. For instance, $dist^k[1] = 0$ for all k since 1 is the source node. Also, $dist^1[2] = 6$, $dist^1[3] = 5$, and $dist^1[4] = 5$, since there are edges from 1 to these nodes. The distance $dist^1[]$ is ∞ for the nodes 5, 6 and 7 since there are no edges to these from 1.

$$dist^2[2] = \min \{dist^1[2], \min_i \{dist^1[i] + cost[i, 2]\}\}$$

$$= \min \{6, 0 + 6, 5 - 2, 5 + \infty, \infty + \infty, \infty + \infty, \infty + \infty\} = 3$$

Here the terms $0 + 6, 5 - 2, 5 + \infty, \infty + \infty, \infty + \infty, \infty + \infty$, and $\infty + \infty$ correspond to a choice of $i = 1, 3, 4, 5, 6$, and 7 , respectively. The rest of the entries are computed in an analogous manner.



(a) A directed graph

k	dist ^k [1..7]						
	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) dist^k

Q.3(b) Explain flowshop scheduling with example.

10

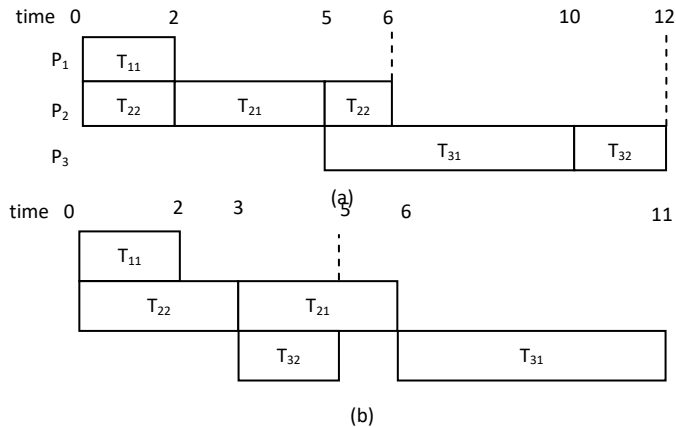
Ans.: A job, requires the performance of several distinct task, to be processed. In flow shop use generally have n jobs each requiring m tasks $T_{1i}, T_{2i}, \dots, T_{mi}$, $1 \leq i \leq n$, to be performed. The time required to complete task T_{ji} is t_{ji} .

A schedule for the n jobs is an assignment of tasks to time intervals on the processors. Task T_{ji} must be assigned to processor P_j . No processor may have more than one task assigned to it in any time interval. Additionally, for any job i the processing of task T_{ji} , $j > 1$, cannot be started until task $T_{j-1,i}$ has been completed.

Example : Two jobs have to be scheduled on three processors. The task times are given by the matrix J

$$J = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$$

Two possible schedules for the jobs are shown below in the figure in the



A nonpreemptive schedule is a schedule in which the processing of a task on any processor is not terminated until the task is complete. A schedule for which this need not be true is called preemptive. The schedule of Figure (a) is a preemptive schedule. Figure (b) shows a nonpreemptive schedule. The finish time $f_i(S)$ of job i is the time at which all tasks of job i have been completed in schedule S . In figure (a), $f_1(S) = 10$ and $f_2(S) = 12$. In Figure (b), $f_1(S) = 11$ and $f_2(S) = 5$. The finish time $F(S)$ of a schedule S is given by

$$F(S) = \max_{1 \leq i \leq n} \{f_i(S)\}$$

The mean flow time MFT (S) is defined to be

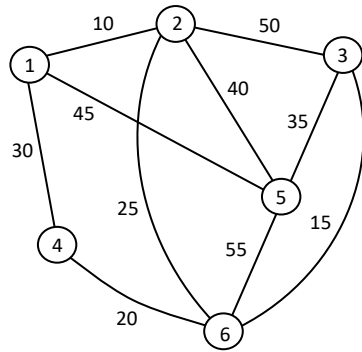
$$MFT(S) = \frac{1}{n} \sum_{1 \leq i \leq n} f_i(S)$$

An optimal finish time (OFT) schedule for a given set of jobs is a nonpreemptive schedule S for which $F(S)$ is minimum over all nonpreemptive schedules S . A preemptive optimal finish time (POFT) schedule, optimal mean finish time schedule (OMFT), and preemptive optimal mean finish (POMFT) schedule are defined in the similar way.

Although the general problem of obtaining OFT and POFT schedules for $m > 2$ and of obtaining OMFT schedules is computationally difficult dynamic programming leads to an efficient algorithm to obtain OFT schedules for the case $m = 2$.

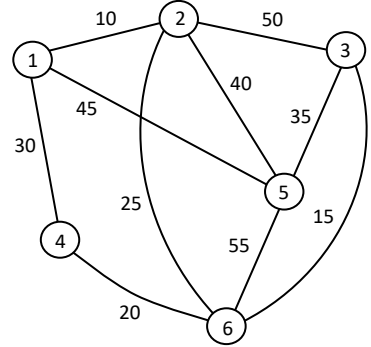
Q.4(a) Find the MCST (Minimum Cost Spanning tree) of given graph.

10



Ans.: Consider the following graph :

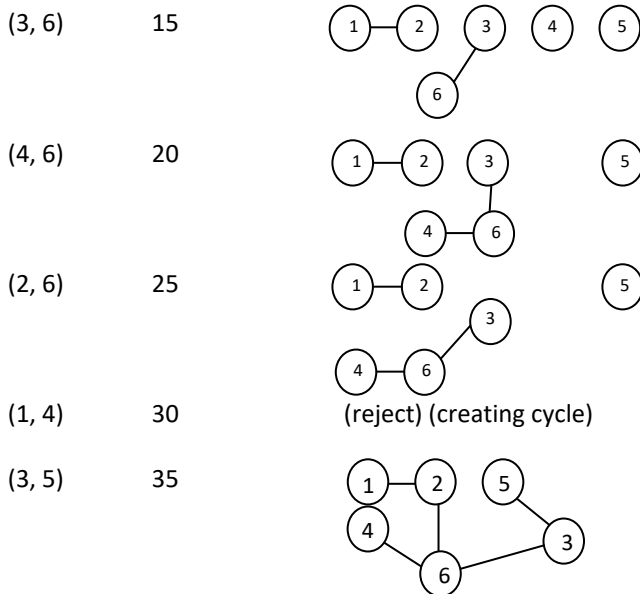
The following is the stages in Prim’s algorithm to find the minimum cost spanning tree of the given graph.



Edge	Cost	Spanning tree
(1, 2)	10	
(2, 6)	25	
(2, 6)	15	
(6, 4)	20	
(1, 4)	reject (cycle)	
(3, 5)	35	

The following is the stages in Kruskal’s algorithm to find the minimum cost spanning tree of the same graph.

Edge	Cost	Spanning tree
(1, 2)	10	



Q.4(b) Explain 8 queens problem with reference to backtracking.

10

Ans.: **Backtracking :**

Sometimes we are facing the task of finding an optimal solution to a problem, there is no applicable theory to help us to find the optimum, except by resorting to exhaustive search. So new systematic, exhaustive technique is used known as 'backtracking'.

In this technique a partial solution is derived at each step and validity of partial solution is checked and if incorrect we backtrack and repair the solution.

Examples: Chess, 8 queen problem, puzzle, tic-tac-toe problem.

Consider the puzzle of how to place eight queens on a chessboard, so that no queen can attack another. One of the rule for chess is that, a queen can take another piece that lie on the same row, same column, or the same diagonal as that of queen.

The 8 queen problem is a classic combinatorial problem to place eight queens on an 8×8 chessboard that no two queens are in direct positioning of attack i.e. no two queens out of eight are on the same row, same column or diagonal. For solving above problem, we are using a design in strategy known as backtracking.

The chessboard has eight rows and eight columns.

In this problem of 8-queens, the solution can be expressed as (x_1, x_2, \dots, x_8) , where x_1 is the position of 1st queen, x_2 is the position at 2nd queen and so on.

Queens are numbered from 1 to 8 and without losing generality. We are assuming that 1st Queen will be placed in 1st row, Queen II will be placed in 2nd row and so on.

So x_1 will be any value from 1 to 8, since there are eight columns in one row even x_2 will be having any value from 1 to 8 and so on.

To solve the above problem, first we are finding a place for the first queen. Let it be 1. i.e. $x[1] = 1$. Then we try to find the place of 2nd queen. Since queen I is directly placed in 1st column the IInd Queen cannot be placed in 1st or 2nd column, otherwise there will be a direct attack between Queen I and Queen II.

Since we are placing Queen I in first column, so we have to place Queen II in column 3. Therefore $x[2] = 3$.

If we are not able to find a proper location for a queen then we will backtrack and try to adjust the position of the previous queen. If it is not possible we will be backtrack once again.

There are many solutions to above problem, out of which two solutions are as shown below.

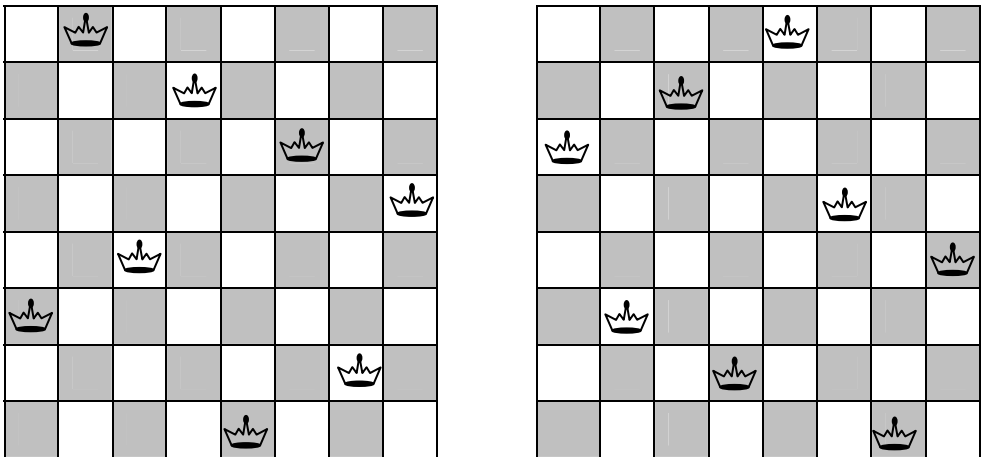


Fig. : Two configurations showing eight non-attacking queens

Consider Simple example for four queens :

Consider 4×4 chessboard and we have to place 4 Queens on it, such that there will no direct attack between them. i.e. No two queens are in same row or same column or diagonal.

We are placing each queen in different rows. Since there are 4 Queens and 4 rows, so we are placing each Queen in different rows.



In 4 Queen problem, the solution can be expressed as (x_1, x_2, x_3, x_4) where x_1 is the position of Ist Queen, x_2 will be the position of IInd Queen and so on.

x_1 will be having any value from 1 to 4 as there are 4 columns in a row.

x_2 will be having any value from 1 to 4 & so on.

We are first placing the Ist Queen in 1st column. So the IInd Queen is going to place in 3rd or 4th column, such that there should not be direct attack between Queen I and Queen II.




The position is as shown below:

	1011	1012	1013	1014
row 1		?	?	?
row 2	X	X		?
row 3	X	X	X	X
row 4				

(a) Dead end





'?' in above figure implies that one more position is possible for the queen to be place. In above shown figure we are not getting the solution for placing 4 Queens on 4×4 boards so we will backtrack and try to adjust the position of IInd Queen.

The figure can be shown as follows:





	?	?	?
X	X	X	
X		X	X
X	X	X	X

(b) Dead end

Even in above figure (b) we are not able to find a proper locations for Queen so we will backtrack. So for Queen III and Queen II, no other possibility of position. So we change the position of Ist Queen.

X		X	X
X	X	X	
	X	X	X
X	X		X

(c) Solution

(d) Solution

The above figure (c) gives the perfect solution of 4 Queens on 4×4 board and no Queen is direct attacking the other Queen.

One more solution is possible which is as shown in figure (d).

Algorithm :

To find a proper position of queen we are using one procedure. PLACE

```

int x[20];
int place(int k)
{
    int i=1,flag = 1;

    while (i<=k-1 && flag)
        if (x[i] == x[k] || abs(x[i]-x[k]) == abs(i-k))
            flag = 0;

```

```
    else
        i++;

return flag;
}
void main()
{
    int k,i;
    k=1; x[k] = 0;
    while (k > 0)
    {
        x[k]++;
        while (x[k] <= 8 && ! place(k))
            x[k]++;
        if (x[k] <= 8)
        {
            if (k == 8)
            {
                for (i=1;i<=8;i++)
                    printf("%d ",x[i]);
                printf("\n");
            }
            else
            {
                k++;
                x[k] = 0;
            }
        }
        else
            k--; /* backtrack */
    }
}
```

Note : $ABS(X[i] - X[K]) = ABS(i - k)$

using this we are checking whether the queen K is coming in diagonal with any of the queens from 1 to K – 1.

X [i] -- column position of ith queen.

X[k] -- column position of kth queen.

i ---- row position of ith queen.

k ---- row position of kth queen.

The above function PLACE will return true if it can find out a proper place for kth queen in the kth row of the 8 × 8 chess board. Otherwise it will return false.

Sum of Subsets :

Suppose n distinct positive numbers (sometimes called weights) is given. To find all combinations of these numbers whose sum is m, is called sum of subsets problem.

A backtracking solution using fixed tuple size strategy can be considered. In this, depending on whether the weight w_i is included or not, the element x_i of the solution vector is either one or zero.

The bounding functions is $B_k(x_1, \dots, x_k) = \text{true}$

$$\text{iff } \sum_{i=1}^k w_i x_i + \sum_{i=k+1}^n w_i \geq m$$

If this condition is not satisfied x_1, \dots, x_k cannot lead to an answer node. If we assume the w_i 's are initially in non-decreasing order, the bounding functions can be strengthened. In this case x_1, \dots, x_k can lead to an answer node if

$$\sum_{i=1}^k w_i x_i + w_{k+1} \leq m$$

Q.5(a) Explain sum of subsets problem and its solution using backtracking.

10

Ans.: Sum of Subsets :

Suppose n distinct positive numbers (sometimes called weights) is given. To find all combinations of these numbers whose sum is m , is called sum of subsets problem.

A backtracking solution using fixed tuple size strategy can be considered. In this, depending on whether the weight w_i is included or not, the element x_i of the solution vector is either one or zero.

The bounding functions is $B_k(x_1, \dots, x_k) = \text{true}$

$$\text{iff } \sum_{i=1}^k w_i x_i + \sum_{i=k+1}^n w_i \geq m$$

If this condition is not satisfied x_1, \dots, x_k cannot lead to an answer node. If we assume the w_i 's are initially in non-decreasing order, the bounding functions can be strengthened. In this case x_1, \dots, x_k can lead to an answer node if

$$\sum_{i=1}^k w_i x_i + w_{k+1} \leq m$$

Recursive backtracking algorithm for sum of subsets problem:

Algorithm SumOfSub(s, k, r)

// Find all subsets of $w[1 : n]$ that sum to m . The values of $x[j]$,

// $1 \leq j < k$, have already been determined. $S = \sum_{j=1}^{k-1} w[j] * I[j]$

// and $r = \sum_{j=k}^n w[j]$. The $w[j]$'s are in nondecreasing order.

// It is assumed that $w[1] \leq m$ and $\sum_{i=1}^n w[i] \geq m$.

{

// Generate left child. Note: $s + w[k] \leq m$ since B_{k-1} is true.

$x[k] := 1$;

if $(s + w[k] = m)$ then write $(x[1 : k])$; // Subset found

// There is no recursive call here as $w[j] > 0, 1 \leq j \leq n$.

else if $(s + w[k] + w[k + 1] \leq m)$

then SumOfSub($s + w[k], k + 1, r - w[k]$);

```
// Generate right child and evaluate Bk.
if ((s + r - w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
{
    x[k] := 0;
    SumOfSub(s, k + 1, r - w[k]);
}
}
```

Q.5(b) Explain 15 puzzle problem with respect to branch and bound.

10

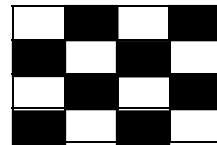
Ans.: The 15–puzzles: An Example

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

(a) An arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(b) Goal arrangement



(c)

Fig. 1: 15–puzzle arrangements

The 15–puzzle (invented by Sam Loyd in 1878) consists of 15 numbered tiles on a square frame with a capacity of 16 tiles (Figure 1). We are given an initial arrangement of the tiles, and the objective is to transform this arrangement into the goal arrangement of Figure 1(b) through a series of legal moves. The only legal moves are ones in which a tile adjacent to the empty spot (ES) is moved to ES. Thus from the initial arrangement of Figure 1(a), four moves are possible. We can move any one of the tiles numbered 2, 3, 5, or 6 to the empty spot. Following this move, other moves can be made. Each move creates a new arrangement of the tiles. These arrangements are called the states of the puzzle. The initial and goal arrangements are called the initial and goal states. A state is reachable from the initial state if there is a sequence of legal moves from the initial state to this state. The state space of an initial state consists of all states that can be reached from the initial state. The most straightforward way to solve the puzzle would be to search the state space for the goal state and use the path from the initial state to the goal state as the answer. It is easy to see that there are $16!$ ($16! \approx 20.9 \times 10^{12}$) different arrangements of the tiles on the frame. Of these only one–half are reachable from any given initial state. Indeed, the state space for the problem is very large. Before attempting to search this state space for the goal state, it would be worthwhile to determine whether the goal state is reachable from the initial state. There is a very simple way to do this. Let us number the frame positions 1 to 16. Position i is the frame position containing tile numbered i in the goal arrangement of Figure 1(b). Position 16 is the empty spot. Let position (i) be the position number in the initial state of the tile numbered i . Then position (16) will denote the position of the empty spot.

For any state let $less(i)$ be the number of tiles j such that $j < i$ and position $(j) >$ position (i) . For the state of Figure 1 (a) we have, for example, $less(1) = 0$, $less(4) = 1$, $less(12) = 6$. Let $x = 1$ if in the initial state the empty spot is at one of the shaded positions of Figure 1(c) and $x = 0$ if it is at one of the remaining positions.

Q.6 Write short notes on : **20**

Q.6(a) Write short note on Rabin Karp string matching algorithm. **05**

Ans.: The Rabin-Karp algorithm :

The Rabin-Karp algorithm is a string searching algorithm created by Michael O. Rabin and Richard M. Karp that seeks a pattern, i.e. a substring, within a text by using hashing. It is not widely used for single pattern matching, but is of considerable theoretical importance and is very effective for multiple pattern matching. For text of length n and pattern of length m , its average and best case running time is $O(n)$, but the (highly unlikely) worst-case performance is $O(nm)$, which is one of the reasons why it is not widely used. However, it has the unique advantage of being able to find any one of k strings or less in $O(n)$ time on average, regardless of the size of k .

One of the simplest practical applications of Rabin-Karp is in detection of plagiarism. Say, for example, that a student is writing an English paper on Moby Dick. A cunning professor might locate a variety of source material on Moby Dick and automatically extract a list of all sentences in those materials. Then, Rabin-Karp can rapidly search through a particular paper for any instance of any of the sentences from the source materials. To avoid easily thwarting the system with minor changes, it can be made to ignore details such as case and punctuation by removing these first. Because the number of strings we're searching for, k_t is very large, single-string-searching algorithms are impractical.

The naive algorithm basically consists of two nested loops—the outermost loop runs through all the $n - m + 1$ possible shifts, and for each such shift the innermost loop runs through the m characters seeing if they match. Rabin and Karp propose a modified algorithm that tries to replace the innermost loop with a single comparison as often as possible.

Features of Rabin-Karp algorithm :

- Rabin-Karp algorithm uses a hashing function.
- Preprocessing phase of Rabin-Karp algorithm in $O(m)$ time complexity and constant space.
- Searching phase of Rabin-Karp algorithm in $O(mn)$ time complexity.
- It uses $O(n + m)$ expected running time.

Q.6(b) Write short note on LCS (Longest Common Subsequence). **05**

Ans.: LCS (Longest Common Subsequence) :

Given two strings X and Y of lengths n and m respectively, we have to find the longest common subsequence. Subsequence of X is any string of the form $X(i_1) X(i_2) X(i_3) \dots X(i_k)$, $i_j < i_{j+1}$ for $j = 1, 2, \dots, k - 1$. Subsequence is a sequence of characters that are not necessarily contiguous but are taken in order. Suppose $X = \text{"aabcdafacd"}$, then "acded" is a subsequence of X . Substring is a subsequence if $i_2 - i_1 = i_3 - i_2 = i_4 - i_3 = \dots = i_k - i_{k-1} = 1$. The problem we are interested here is to find the longest string S that is a subsequence of both X and Y . The brute-force approach for this problem yields an exponential algorithm. By using dynamic programming, we can solve the problem much faster.

Let $L(i, j)$ be the length of the longest common subsequence of both $X(0)X(1) \dots X(i)$ and $Y(0)Y(1) \dots Y(j)$. We distinguish between two cases.

1. The last character is the same in the two strings, that is $X(i) = Y(j) = 'c'$. The longest common subsequence of $X(0)X(1) \dots X(i)$ and $Y(0)Y(1) \dots Y(j)$ ends with 'c'. So we write: $L(i, j) = L(i - 1, j - 1) + 1$
2. The last characters are different in the two strings, that is $X(i) \neq Y(j)$. In this case the common subsequence ends with $X(i)$ or $Y(j)$ or none of these. So we write:

$$L(i, j) = \max\{L(i - 1, j), L(i, j - 1)\}$$

For Example, to make sense in the boundary cases when $i = 0$ or $j = 0$. We write:

$$L(i, -1) = 0 \text{ for } i = -1, 0, \dots, n - 1 \text{ and } L(-1, j) = 0 \text{ for } j = -1, \dots, m - 1.$$

The problem satisfies the principle of optimality. We cannot have the longest common subsequences without also having the longest common subsequences for the sub-problems. We give pseudocode for the problem based on the above formulation.

Procedure LCSS (X, Y, n, m)

Character Array X(n), Y(m)

For $i = -1$ to $n - 1$ Do

$L(i, -1) = 0;$

Endfor

For $j = -1$ to $m - 1$ Do

$L(-1, j) = 0;$

Endfor

For $i = 0$ to $n - 1$ Do

For $j = 0$ to $m - 1$ Do

If $x(i) = Y(j)$ Then

$L(i, j) = L(i - 1, j - 1) + 1;$

Else

$L(i, j) = \text{Max}\{L(i - 1, j), L(i, j - 1)\};$

Endif

Endfor

Endfor

End LCSS

The time complexity of the above procedure is dominated by the two nested for loops. The if statement within the nested loop requires constant time and so the time complexity of the algorithm is $O(\text{length}(X) \times \text{length}(Y)) = O(nm)$.

Example : Find the longest common subsequence of $X = \text{'aabcdacdbb'}$. $Y = \text{'abacdabd'}$ using dynamic programming.

Using Table below we find that the common longest subsequence has length 6. To find a longest subsequence we work back through the table. We start with $L(n - 1, m - 1)$, that is, with the cell (9, 7). We find that the characters are different. We can now move to either the cell (9, 6) or the cell (8, 7). We examine the cell (9, 6) and find that the common character is b. So the last character of a longest common subsequence is b.

From the cell (9, 6), we move to the cell (8, 5) and find that the characters are different. We can now move to the cell (8, 4) or the cell (7, 5). If the characters corresponding to a cell (i, j) are identical, then we move to cell (i - 1, j - 1); otherwise to the cell (i - 1, j) or (i, j - 1) whichever is having the greater value. The possible traces through the table for the example strings are shown in the directed graph of Figure 1. The algorithm produces five longest common subsequences each of length 6: abacdb. abcdab. aacdab,. aacadad, abcdad as shown in Figure 1.

Table : Finding the length of the longest common subsequence

L		a	b	a	c	d	a	b	d
	-1	0	1	2	3	4	5	6	7
-1	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1
0									
a	0	1	1	2	2	2	2	2	2
1									
b	0	1	2	2	2	2	2	3	3
2									
c	0	1	2	2	3	3	3	3	3
3									
d	0	1	2	2	3	4	4	4	4
4									
a	0	1	2	3	3	4	5	5	5
5									
c	0	1	2	3	4	4	5	5	5
6									
d	0	1	2	3	4	5	5	5	6
7									
b	0	1	2	3	4	5	5	6	6
8									
b	0	1	2	3	4	5	5	6	6
9									

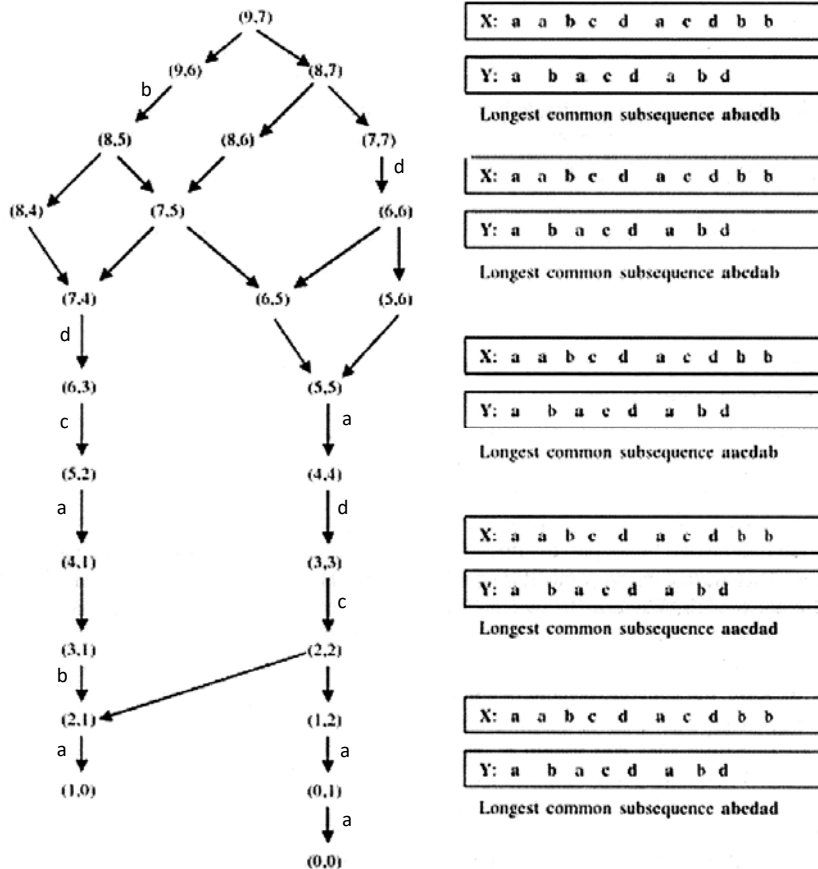


Fig. : Finding the longest common sub-sequences

Q.6(c) Write short note on Merge sort and its complexity. 05

Ans.: Merge sort is based on the divide-and-conquer paradigm. Its worst-case running time has a lower order of growth than insertion sort, Since we are dealing with sub-problems, we state each sub-problem as sorting a sub-array $A[p \dots r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through sub-problems.

To sort $a[p \dots r]$:

Step 1 : Divide Step

If a given array A has zero one element, simply return; it is already sorted. Otherwise, split $A[p \dots r]$ into two sub-arrays $A[p \dots q]$ and $A[q + 1 \dots r]$, each containing about half of the elements of $A[p \dots r]$. That is, q is the halfway point of $A[p \dots r]$.

Step 2 : Conquer Step

Conquer by recursively sorting the two sub-arrays $A[p \dots q]$ and $a[q + 1 \dots r]$.

Step 3 : Combine Step

Combine the elements back in $A[p \dots r]$ by merging the two sorted sub-arrays $A[p \dots q]$ and $A[q + 1 \dots r]$ into a sorted sequence. To accomplish this step, we will define a procedure $MERGE(A, p, q, r)$.

Note that the recursion bottoms out when the sub-array has just one element, so that it is trivially sorted.

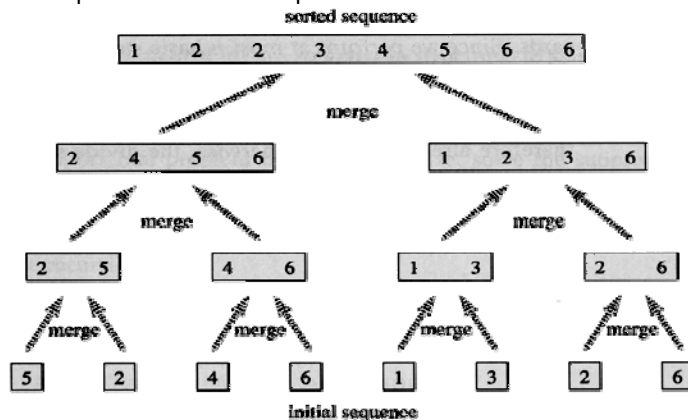
Algorithm for Merge Sort :

To sort the entire sequence $A[1 \dots n]$, make the initial call to the procedure $MERGE-SORT(A, l, n)$.

MERGE-SORT(A, p, r)

1. IF $p < r$ // Check for base case
2. THEN $q = \text{FLOOR}[(p + r)/2]$ // Divide step
3. $MERGE(A, p, q)$ // Conquer step.
4. $MERGE(A, q + 1, r)$ // Conquer step.
5. $MERGE(A, p, q, r)$ // Conquer step.

Example: Bottom-up view of the above procedure for $n = 8$.



Merging :

What remains is the $MERGE$ procedure. The following is the input and output of the $MERGE$ procedure.

INPUT: Array A and indices p, q, r such that $p \leq q \leq r$ and sub-array $A[p \dots q]$ is sorted and sub-array $A[q + 1 \dots r]$ is sorted. By restrictions on p, q, r , neither sub-array is empty.

OUTPUT: The two sub-arrays are merged into a single sorted sub-array in $A[p \dots r]$.

We implement it so that it takes $\Theta(n)$ time, where $n = r - p + 1$, which is the number of elements being merged.

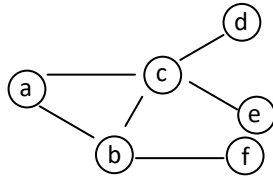
Complexity: The complexity of merge sort is $O(N \log N)$ in the best case/worst case/average case.

Q.6(d) Write short note on Proof of vertex cover problem as NP problem.

05

Ans.: DVC \in NP

Vertex Cover: A vertex cover of a graph G is a set of vertices such that every edge in G is incident to at least one of these vertices.



Find a vertex cover of G of size two

Decision Vertex Cover (DVC) Problem: Given an undirected graph G and an integer k , does G have a vertex cover with k vertices.

Claim: DVC \in NP.

Proof: A certificate will be a set C of $\leq k$ vertices. The brute force method to check whether C is a vertex cover takes time $O(K_e)$. As $K_e < (n + e)^2$, the time to verify is $O((n + e)^2)$. So a certificate can be verified in polynomial time.

