ANALYSIS OF ALGORITHMS

V.C.S.

1 BACKGROUND & DIVIDE AND CONQUER METHOD

CHAPTER IN A NUTSHELL

❑ Algorithm Complexity

The analysis of algorithm focuses on two complexities :

(i) Time Complexity

(ii) Space Complexity

As compared to time analysis, the analysis of space requirement for an algorithm is generally easier, but wherever necessary, both the techniques are used. In this content, space is referred to the storage required in addition to the space required to store the input data. The amount of memory needed by program to run to completion is referred to as **space complexity**. For an algorithm, time complexity depends upon the size of the input, thus, it is a function of input size "n". The amount of time needed by an algorithm to run to completion is referred as **time complexity**.

It should be noted that different time can arise for the same algorithm. Usually, we deal with the best case time, average case time, and worst case time for an algorithm. The minimum amount of time that an algorithm requires for an input of size "n", is referred to as best case time complexity, similarly average case time complexity is the execution of an algorithm having typical input data of size "n". And lastly, the maximum amount of time needed by an algorithm for an input of size, "n", is referred to worst case time complexity. Perhaps, there are many factors which influence the time required by an algorithm.

❑ Binary Search

Given an ordered set of 'N' data items – $D_1, D_2, D_3 \ldots\ldots D_N$ having 'N' distinct keys such as $k_1 < k_2 < k_3 < \ldots. k_N$, the binary search starts with comparing the key 'k_i' with the middle key of the given set. Thus, the result of this checking tells us that which half of the set should be searched next. Every time the key 'k_i' is searched to the middle key of the remaining set of data item until the desired data item is searched or the end of the set is reached. In binary search, the lower limit and upper limit are indicated by two pointers 'l' and 'u'. Thus, 'k_l' and 'k_u' denote the current lower and upper limits for this search.

It can be easily seen that if the desired key 'k_i' is present in the set than it follows the condition as

$$k_l \le k_i \le k_u$$

If the upper limit is less than the lower limit, the search is unsuccessful.

$$u < l$$

The middle key of the given set is obtained by

$$m = \left[\frac{1}{2}(l+u)\right]$$

❑ Merge Sort

The merging of the two sorted arrays can be done so that the resulting array will also be in the sorted form. If we decide to use same technique for creating the sorted array, then the requirement will be the two arrays, which we are going to merge, must be sorted. Merging is the combination of two or more sorted sequences into a single sorted sequence.

Sorting by merging is recursive, and uses divide and conquer strategy. In the base case, we have sequence with exactly one element in it. Since such a sequence is already sorted, there is nothing to be done.

To sort a sequence of $n > 1$ elements.

Divide the sequence into two sequences of length |n/2| and |n/2|; recursively sort each of the two subsequences; and then, merge the sorted subsequences to obtain the final result.

In a two-way merge, two sorted sequences are merged into one. The merge function takes four parameters. The first is a reference to the array to be sorted.

The remaining three, left, middle and right are unsigned integers. It is assumed that

left < middle < right

Furthermore, it is assumed that the two subsequences of the array.

array[left], array[left + 1], array[left + 2], ..., array[middle]

and

array[middle + 1], array[middle + 2], ..., array[right]

are both sorted.

❑ Quick Sort

Quick sort technique is based on the **divide and conquer** design technique. In this, at every step each element is placed

in its proper position. It performs very well on a longer list. It works recursively, by first selecting a random "pivot value" from the list (array). Then it partitions the list into elements that are less than the pivot and greater reduced to the problem of sorting two sublists. It is to be noted that the reduction step in the quick sort finds the final position of particular element, which can be accomplished by scanning that last element of the list from the right to left, and checks with the element. The comparisons of elements with the first element stop when we obtain the elements smaller than the first element. Thus, in this case exchange of both the elements takes place. The whole procedure continues until all the elements of the list are arranged on the left side of the element (pivot). The elements on the right side, are greater than the pivot. Thus, the list is subdivided into two lists. This sorting technique is considered as an in place since it uses no other array storage.

PREVIOUS YEARS QUESTIONS

PART-A

Q.1 ***Determine the frequency counts for all statements in the following segments :***

```
for i = 1 to n do
for j = 1 to i do
for k = 1 to j do
x = x+1
```

[R.T.U. 2013]

Ans. Frequency count for i for loop = n+1
Frequency count for j for loop = n×(i+1)
Frequency count for k for loop = n×i×(j+1)
Frequency count for x = x+1 = n+i×j

Q.2 ***Determine the frequency counts for all statements in the following segments :***

```
i = 1
while (i ≤ n) do
{
x = x+1 ;
i = i+1
}
```

[R.T.U. 2013]

Ans. Frequency count for i = 1 =1
Frequency count for while loop = n+1
Frequency count for x = x + 1 = n
Frequency count for i = i + 1 = n

Q.3 ***What do you mean by space complexity.***

Ans. The amount of memory needed by program to run to completion is referred to as **space complexity**.

Q.4 ***Define time complexity.***

Ans. The amount of time needed by an algorithm to run to completion is referred as **time complexity**.

Q.5 ***Define worst case time complexity.***

Ans. The maximum amount of time needed by an algorithm for an input of size, "n", is referred to worst case time complexity.

PART-B

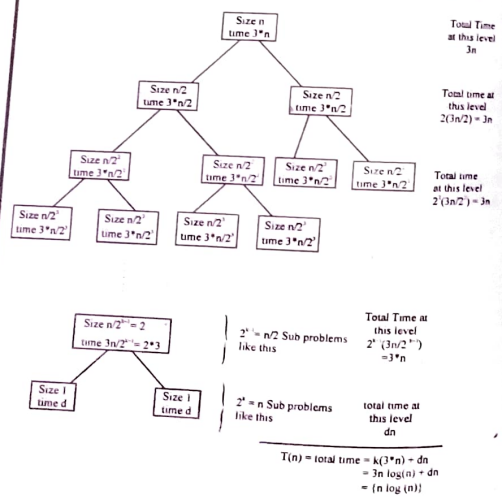
Q.6 ***Solve the recurrence relation for time complexity:***

$T(n) = 2 \quad if\ n = 2$

$T(n) = 2T(n/2)+3n \quad if\ n > 2$ *[R.T.U. 2018, 2016, 2014]*

Ans. Recurrence Tree Method : The Recurrence $T(n) = 2T(n/2) + 3*n$,

$T(1) = d$, when n is a power of $2(n = 2^k, k = \log(n))$



Substitution method

$$T(n) = 2T\left(\frac{n}{2}\right) + 3*n$$

$$T(n) = 2\left(2T\left(\frac{n}{4}\right) + 3*\frac{n}{2}\right) + 3*n = 4T\left(\frac{n}{4}\right) + 3*n + 3*n$$

$$= 4\left(2T\left(\frac{n}{8}\right) + 3*\frac{n}{4}\right) + 3*n + 3*n$$

$$= 8T\left(\frac{n}{8}\right) + 3*n + 3*n + 3*n$$

$$= nT\left(\frac{n}{n}\right) + 3*n + \ldots\ldots + 3*n + 3*n + 3*n$$

$$T(n) = 0n + n + \ldots\ldots + 3*n + 3*n + 3*n$$

$$= n \log n$$

Q.7 ***Explain Strassen's matrix multiplication & derive its complexity also? Justify how is it better than ordinary matrix multiplication.***

[R.T.U. 2018, 2016, 2010]

OR

Describe strassen's method of matrix multiplication.

[R.T.U. 2015]

Ans. Strassen's Matrix Multiplication : Let A and B be two n × n matrices. The product matrix C = AB is also an n × n matrix whose i, j^{th} element is formed by taking the elements in the i^{th} row of A and the j^{th} column of B and multiplying them to get

$$C(i,j) = \sum_{1 \le k \le n} A(i,k)\,B(k,j) \qquad \ldots\ldots(1)$$

for all i and j between 1 and n. To compute C (i, j) using this formula, we need n multiplications. As the matrix C has n^2 elements, the time for the resulting matrix multiplication algorithm, which we refer to as the conventional method is $\Theta(n^3)$.

The divide and conquer strategy suggests another way to compute the product of two n × n matrices. For simplicity, we assume that n is a power of 2, that is, that there exists a non-negative integer k such that $n = 2^k$. In case n is not a power of two, then enough rows and columns of zeros can be added to both A and B so that the resulting dimensions are a power of two.

Imagine that A and B are each partitioned into four square submatrices, each submatrix having dimensions $\frac{n}{2} \times \frac{n}{2}$. Then the product AB can be computed by using the above formula for the product of 2 × 2 matrices : if AB is

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \qquad \ldots\ldots(2)$$

then

$$C_{11} = A_{11}\,B_{11} + A_{12}\,B_{21}$$
$$C_{12} = A_{11}\,B_{12} + A_{12}\,B_{22} \qquad \ldots\ldots(3)$$
$$C_{21} = A_{21}\,B_{11} + A_{22}\,B_{21}$$
$$C_{22} = A_{21}\,B_{12} + A_{22}\,B_{22}$$

If n = 2, then formulas (2) and (3) are computed using a multiplication operation for the elements of A and B. These elements are typically floating point numbers. For n > 2, the elements of C can be computed using matrix multiplication and addition operations applied to matrices of size $\frac{n}{2} \times \frac{n}{2}$. Since n is a power of 2, these matrix products can be recursively computed by the same algorithm we are using for the n × n case. This algorithm will continue applying itself to smaller-sized submatrices until n becomes suitable small (n = 2) so that the product is computed directly.

To compute AB, we need to perform eight multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and four additions of $\frac{n}{2} \times \frac{n}{2}$ matrices. Since two $\frac{n}{2} \times \frac{n}{2}$ matrices can be added in time cn^2 for some constant c, the overall computing time T(n) of the resulting divide and conquer algorithm is given by the recurrence

$$T(n) = \begin{cases} b & ; \ n \le 2 \\ 8T(n/2) + cn^2 & ; \ n > 2 \end{cases}$$

where, b and c are constants.

Since matrix multiplications are more expensive than matrix additions ($O(n^3)$ versus $O(n^2)$), we can attempt to reformulate the equations for C_{ij} so as to have fewer multiplications and possibly more additions.

Volker Strassen has discovered a way to compute the C_{ij}'s of using only 7 multiplications and 18 additions or subtractions. This method involves first computing the seven $\frac{n}{2} \times \frac{n}{2}$ matrices P, Q, R, S, T, U and V. Then the C_{ij}'s are computed using the formulas. As can be seen, P, Q, R, S, T, U and V can be computed using 7 matrix multiplications and 10 matrix additions or subtractions. The C_{ij}'s require an additional 8 additions or subtractions.

$$P = (A_{11} + A_{22})\,(B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22})B_{11}$$
$$R = A_{11}\,(B_{12} - B_{22})$$
$$S = A_{22}\,(B_{21} - B_{11}) \qquad \ldots\ldots(4)$$
$$T = (A_{11} + A_{12})B_{22}$$
$$U = (A_{21} - A_{11})\,(B_{11} + B_{12})$$
$$V = (A_{12} - A_{22})\,(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T \qquad \ldots\ldots(5)$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U$$

The resulting recurrence relation for T(n) is

$$T(n) = \begin{cases} b & ; \ n \le 2 \\ 7T(n/2) + an^2 & ; \ n > 2 \end{cases} \qquad \ldots\ldots(6)$$

where a and b are constants. Working with this formula, we get

$$T(n) = an^2\,[1 + 7/4 + (7/4)^2 + \ldots + (7/4)^{k-1}] + 7^k T(1)$$
$$\le cn^2(7/4)\log_2 n + 7\log_2 n,\ c \text{ is a constant}$$
$$= c_n \log_2 4 + \log_2 7 - \log_2 4 + n \log_2 7$$
$$= O\left(n^{\log_2 7}\right) \approx O\left(n^{2.81}\right).$$

Q.8 ***Solve the following recurrence relations and find their complexities using master method***

(i) $T(n) = 2T(\sqrt{n}) + \log_2 n$

(ii) $T(n) = 4T(n/2) + n^2$ ***[R.T.U. 2018, 2012]***

Ans.(i) $T(n) = 2T(\sqrt{n}) + \log_2 n$

We have a=2, b=1, f(n)= $\log_2 n$ and

$n^{\log_b a} = n^{\log_1 2} = n^{\log 2}$

since $f(n) = \Omega(n^{\log_1 2 - \epsilon})$

where $\epsilon \approx 0.2$ applies if we can show that the regularity condition holds for f(n).

For sufficiently large n,

$$a f(\sqrt{n}) = 2\sqrt{n} \log \sqrt{n} \le 2\sqrt{n} \log n$$
$$= c f(n) \text{ for } c=2$$

Hence solution is

$$T(n) = \theta \log_2 n$$

(ii) $T(n) = 4T(n/2) + n^2$

We have a = 4, b = 3, $f(n)=n^2$

and $n^{\log_b a} = n^{\log_2 4}$

Since $f(n) O(n^{\log_2 4 - \epsilon})$ where $\epsilon = 1$,

Then solution is

$$T(n) = \theta n^2$$

Q.9 ***Consider the following function***

```
int Sequential Search(int A[], int & x, int n)
{
Int i;
For (int i=0, i<n && a[i]!=x;i++)
If (i==n return i;)
}
```

Determine the average and worst case complexity of the function Sequential Search. ***[R.T.U. 2017]***

Ans. The worst case time complexity when element is found at last position

$$T_{\text{worst search}}(n) = n$$
$$= O(n)$$

On average, we will find the item about halfway into the list, we will compare against n/2 items. As n gets large, the coefficient become insignificants in our approximation. So, the complexity is O(n).

$$T_{\text{Average search}}(n) = \frac{\sum_{i=1}^{n}(n-i+1)}{n}$$

$$= \frac{\sum n - \sum i + \sum 1}{n}$$

$$= \frac{n^2 - \left(\frac{n(n+1)}{2}\right) + n}{n}$$

$$= n - \left(\frac{n+1}{2}\right) + 1 = \frac{n+1}{2}$$

$$= \Theta(n)$$

Q.10 ***Show all the steps of Strassen's matrix multiplication algorithm to multiply the following matrices.***

$$X = \begin{bmatrix} 3 & 2 \\ 4 & 8 \end{bmatrix} \text{ and } Y = \begin{bmatrix} 1 & 5 \\ 9 & 6 \end{bmatrix}$$ ***[R.T.U. 2017]***

Ans. Strassen's Multiplication Method

$$X = \begin{bmatrix} 3 & 2 \\ 4 & 8 \end{bmatrix}$$

$$Y = \begin{bmatrix} 1 & 5 \\ 9 & 6 \end{bmatrix}$$

$$= \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$p_1 = a(f - n)$
$= 3(5 - 6)$
$= -3$

$p_2 = (a + b)n$
$= (3 + 2) \times 6$
$= 30$

$p_3 = (c + d)e$
$= (4 + 8) \times 1$
$= 12$

$p_4 = d(g - e)$
$= 8(9 - 1)$
$= 64$

$p_5 = (a+d)(e + n)$
$= (3 + 8)(1 + 6)$
$= 77$

$p_6 = (b - d)(g + h)$
$= (2 - 8)(9 + 6)$
$= -90$

$p_7 = (a - c)(e + f)$
$= (3 - 4)(1 + 5)$
$= -6$

$$XY = \begin{bmatrix} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ p_3 + p_4 & p_5 + p_1 - p_3 - p_7 \end{bmatrix}$$

$$= \begin{bmatrix} 77 + 64 - 30 - 90 & -3 + 30 \\ 12 + 64 & -3 + 77 - 12 + 6 \end{bmatrix}$$

$$= \begin{bmatrix} 21 & 27 \\ 76 & 68 \end{bmatrix}$$

Q.11 ***Explain best-case, average case, worst-case running time of merge sort algorithm.*** ***[R.T.U. 2014]***

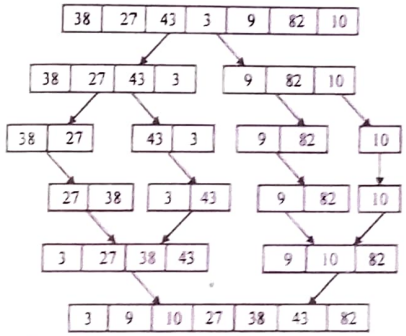
Ans. Analysis of Merge Sort

Merge sort repeatedly divides an array into equal/near equal sub arrays until the size of each sub-array is reduced to 1. Then, starting from the left hand side consecutive two sub arrays are combined together in sorted order, recursively. This process is to be repeated until we get a single array at end.

Merge sort incorporates two main ideas to improve its runtime :

1. A small list will take fewer steps to sort than a large list.
2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists.

Analysis



A recursive merge sort algorithm used to sort an array of 7 integer values. These are the steps a human would take to emulate merge sort (top-down).

In sorting n objects, merge sort has an average and worst case performance of O(n log n). If the running time of merge sort for a list of length n is T(n), then the recurrence $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm.

In the worst case, merge sort does an amount of comparisions equal to or slightly smaller then (n log n - $2^{(\log n)}$ + 1), which is between (n log n - n + 1) and (n log n + n + O (log n)).

1. O(n log n) best, average and worst case complexity because the merging is always linear.
2. Extra O(n) temporary array and back.
3. Extra copying to the temporary array and back.
4. Useful only for external sorting.

Q.12 ***Write an algorithm to search an element from a given array by binary search method. Discuss the time complexity of the algorithm.*** ***[R.T.U. 2014]***

Ans. Binary Search

Binary search is a fast searching algorithm, but it works only on sorted data. It adopts a divide and conquer approach. Every time it reduces the size of list in which search is performed.

The basic idea is very simple :

"If we have a list of elements, already sorted in increasing order, we just have to compare the key we are searching with the middle element." Following situations arise :

- **Key < Middle Element :** Hence, we should restrict our search in first half of the list. All elements in list after the middle element are greater.
- **Key > Middle Element :** We should restrict our search in second half of the list. All elements in list before middle element are smaller.
- **Key = Middle Element :** The search is over. We have found the position of our key in the list.

In first two cases, we may recursively call binary search, with list reduced to half. Thus, in recursive implementation we keep on recursively calling binary search untill there is only one element left in the list. This is the base of recursion. Here only two situations arise :

- **Key = Element :** Search is successful and we report the position at which key was found.
- **Key ≠ Elment :** Search fails. The key does not exist in the list, since there are no more elements to compare with.

The algorithm can be implemented either as recursion or as iteration.

The recursive version of Binary Search

Binary Search1 (Key, A, lb, ub)
// A is the array of elements
// key is value of element to be searched
//lb gives lower bound of array
//ub gives upper bound of array
Step 1 : if (lb > ub)
Step 2 : return "Search fail"
Step 3 : m = [(lb + ub) / 2] ; //position of middle element
Step 4 : if (key = A[m])
Step 5 : return "Search successful at" m
Step 6 : else if (key < A[m])
Step 7 : BinarySearch1 (key, A, lb, m-1);
Step 8 : else BinarySearch (key, A, m + 1, ub);

The lower bound of array gives the position of first accessible element. Upper bound gives the last element accessible in array. Like, if we have a subarray from position 5 to position 9, lb is 5 and ub is 9. Step 1 checks the condition when array has no elements within the bounds. Since there are no elements to compare with, search stops here and key is not found. Hence failure is reported in step 2. step 3 calculates the position of middle element. To have an integer

number as position, we use floor function for case of even sized array. Like, if we have array of eight elements, lb is 1 and ub is 8, there are two middle elements at 4 and 5, but m is 4. If the key is equal to middle element report success at step 5, else go to step 6. Here we compare key and middle element. Step 7 recursively calls Binary Search with first half of array, when key is lesser than middle element. Step 8 recursively calls Binary Search with second half of array, when key is greater than middle element.

First call to this algorithm will be Binary Search1 (key, A, 1, n).

The iterative version of Binary Search

BinarySearch2 (key, A)
//A is array of elements
// key is element to be searched
Step 1 : lb : = 1 ;
Step 2 : ub : = length (A);
Step 3 : while (lb <= ub);
Step 4 : m : = [(lb + ub) / 2] ; //position of middle element
Step 5 : if (key = A[m])
Step 6 : return "Search successful at" m,
Step 7 : else if (key < A[m])
Step 8 : ub : = m–1,
Step 9 : else lb : = m + 1 ,
Step 10 : end while ;
Step 11 : return "Search failed" '

By changing values of lb or ub, we reduce the range of elements in list where w perform search. The failure case is when there is no element in the sublist. It is indicated when lb > ub, for which while loop terminates and the failure message of step 11 is reported.

Analysis

With every iteration the list is divided into half. The comparisions and call overhead is constant. The recurrence relation thus formed is

$T(n) = T(n/2) + c$

This recurrence relation has following solution

$T(n) = O(\log_2 n)$

In worst case of search failure, we will reach it within log n time. For any average case, log n is acceptable limit. A best case is when the key is found at middle position in first call. Obviously, the run time in this case is O(1).

Q.13 ***Express the following function using asymptotic notations:***

(i) $6 \times 2^n + n^2$

(ii) $1/2\, n(n-1)$ *[R.T.U. 2013]*

Ans.

(i) $f(n) = 6 \times 2^n + n^2$

When $n \geq 4$

$n^2 \leq 2n$

So, $f(n) \leq 6 \times 2^n + 2^n = 7 \times 2^n$

Hence $f(n) = O(2^n)$

Where, $C = 7$

$n_0 = 4$

(ii) $F(n) = \dfrac{n(n-1)}{2}$

Then $\dfrac{n(n-1)}{2} \notin O(n)$

$\because$ $F(n) > O(n)$ we get

$$F(n) = \frac{n(n-1)}{2}$$

$$= \frac{n^2 - n}{2}$$

i.e., maximum order is n^2 which is $> O(n)$

Hence $F(n) \notin O(n)$

But $\dfrac{n(n-1)}{2} \in O(n^2)$ as $F(n) \leq O(n^2)$

and $\dfrac{n(n-1)}{2} \in O(n^3)$

Similarly,

$\dfrac{n(n-1)}{2} \in \Omega(n)$ $\because$ $F(n) \geq \Omega(n)$

$\dfrac{n(n-1)}{2} \in \Omega(n^2)$ $\because$ $F(n) \geq \Omega(n^2)$

PART-C

Q.14 ***Illustrate the operation of merge sort on following array 10, 20, 5, 23, 45, 34, 12. Also write the algorithm & its complexity.*** *[R.T.U. 2016]*

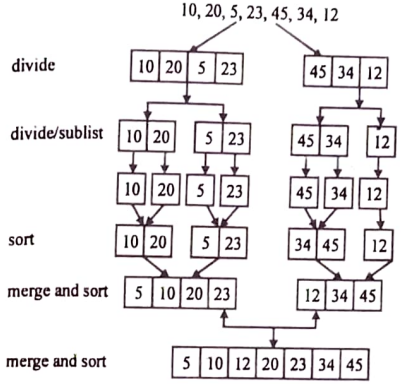
OR

Derive the recurrence relation for merge sort algorithm's time complexity. Also solve it. *[R.T.U. 2018]*

OR

Determine the best case complexity of Merge sort algorithm. *[R.T.U. 2017]*

Ans. Merge sort on 10, 20, 5, 23, 45, 34, 12 sorting the given sequence using merge sort-



Final sorted list: 5 10 12 20 23 34 45

Algorithm used to sort the sequence : MERGE - SORT (p, r, q, A)

1. p ← 1
2. r ← n
3. q ← (p + r)/2
4. MERGE SORT (p, q, A, r)
5. MERGE SORT (q + 1, r, A, p)
6. MERGE (p, q, A, r)

MERGE (p, q, A, r)

1. $n_1 \leftarrow p - q + 1$
2. $n_2 \leftarrow r - q$
3. $L_1[1 \ldots n_1 + 1]$, $L_2[1 \ldots n_2 + 1]$ initialize line arrays
4. for $i = 1$ to n_1
5. do $L_1[i] = A[p - i + 1]$
6. for $j = 1$ to n_2
7. do $L_2[j] = A[r - q + 1]$
8. for k = p to r
9. do for i = 1 to n_1
10. do for j = 1 to n_2
11. if $L_1[i] < L_2[j]$
12. then A [k] = L_1 [i]
13. $i \leftarrow i + 1$
14. else A [k] = L_2 [j]
15. $j \leftarrow j + 1$
16. return A

Algorithm & its Complexity : Algorithms for mergesort is given below :

```
mergesort (int[ ]a, int left, int right)
{
  if (right > left)
  {
    Middle = left + (right – left)/2;
    mergesort (a, left, middle);
    mergesort (a, middle+1, right);
    merge (a, left, middle, right);
  }
}
```

Assuming N is power of two,

For N = 1 : time is a constant (denoted by 1)

else : time to mergesort N elemets = time to mergesort N/2 element + time to merge two arrays each N/2 elements. Time to merge two arrays each N/2 elements is linear, i.e., N.

Thus, we have;

1. $T(1) = 1$
2. $T(N) = 2T(N/2) + N$

Dividing Step (2) by N:

3. $T(N)/N = T(N/2)/(N/2) + 1$

$\because$ N is power of two, so we can write

4. $T(N/2)/(N/2) = T(N/4)/(N/4) + 1$
5. $T(N/4)/(N/4) = T(N/8)/(N/8) + 1$
6. $T(N/8)/(N/8) = T(N/16)/(N/16) + 1$
7. ...
8. $T(2)/2 = T(1)/1 + 1$

Adding Step (3) to (8), the sum of their left hand sides will be equal to sum of RHS.

$T(N)/N + T(N/2)/(N/2) + T(N/4)/(N/4) + \ldots + T(2)/2$
$= T(N/2)/(N/2) + T(N/4)/(N/4) + \ldots + T(2)/2 + T(1)/1$
$+ \log N$

Where log N is sum of 1's on RHS.

9. $T(N)/N = T(1)/1 + \log N$

$T(1)$ is 1, hence

10. $T(N) = N + N \log N = O(N \log N)$

hence complexity of merge sort algorithm is O(N log N)

Q.15 ***Describe various asymptotic notations.*** *[R.T.U. 2018, 2016, 2012]*

Ans. Asymptotic Notations: We represent the complexity of any algorithm as a function of its input size n. Call it f(n). Now, in order to have an estimate of the limits of this function, we take another function g(n), for which we already know the behavior [or it is easy to observe the behavior of g(n)]. The limits can be chosen according to what we desire - overestimate, underestimate, etc.

*"The **Asymptotic Notation** is a representation which describes the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions."*

Depending on the limit applied, the various notations are :

Big-oh Notation (O) : The upper bound for the function 'f' is provided by the Big oh notation (O).

Definition : Considering 'g' to be a function from the non-negative integers into the positive real numbers. Then O(g) is the set of function f, also from the non-negative integers to the positive real numbers, such that for some real constant $c > 0$ and some non-negative integers constant n_0,

$$f(n) < cg(n) \text{ for all } n \geq n_0$$

For all values of $n > n_0$ function 'f' is at most times the function 'g'. It can be noticed that for all 'n' a function may be in O(g) even if $f(n) > g(n)$. Thus, 'g' provides an upper bound by some constant multiple on the value off for all suitably large V(i.e.,$n \geq n_0$).

The set O(g) is usually called as oh of 'g' or big oh of 'g'. As O(g) is explained as a set, it is a good practice to say "f is oh of g" or "f is big oh of g".

Some common asymptotic functions are as follows :

constant : 1,.....
logarithmic : log n .....
linear : n .....
quadratic : n^2, ...
exponential : 2^n,.....
factorial : n!, ...
cubic : n^3, ...
In general,

O(g(n)) = {f(n) : There exists positive constant such that $0 \leq f \leq Cg(n)$ for all n, $n \geq n_0$

Example : For the function $f(n) = 100n + 6$ from the definition of Big oh Notation we can write,

$$0 \leq f(n) \leq cg(n)$$

$$0 \leq 100n + 6 \leq cn$$

Since, big oh notation puts an upper bound on the given function, constant c has value slightly greater than the coefficient of highest order term.

The inequality can be made to hold for any value of $n \geq 6$ by choosing $c \geq 101$.

Thus, for $c = 101$ and $n_1 = 6$, it is verified that

$$100n + 6 = O(n)$$

Big Omega Notation (Ω) : The lower bound for the function 'f' is provided by the big omega notation (Ω).

Definition : Considering 'g' be a function from the non-negative integers into the positive real numbers. The Ω(g)is the set of function 'f' also form the non-negative integers into the positive real numbers, such that for some real constant $c > 0$ and some non-negative integer constant n_0, $f(n) \geq cg(n)$ for all n, $n \geq n_0$.

For all values of $n < n_0$ function 'f' is almost 'c' times the function 'g'. Here, 'g' provides a lower bound, by some constant multiple, on the value of 'n' for all suitable large 'n' (*i.e.*, $n \geq n_0$).

Example : For the function $f(n) = 4n^3 + 2n + 3$, from the definition of big omega notation we can write,

$$0 \leq cg(n) \leq f(n)$$

or

$$0 \leq cn^3 \leq 4n^3 + 2n + 3$$

Since, big omega notation puts lower bound on the given function, constant c has value slightly smaller than or equal to the coefficient of highest order term.

The above inequality can be made to hold for any values of n by choosing $c \leq 4$.

The value of n though needs to be a non-negative integer greater than or equal to zero, i.e., $n \geq 0$.

Thus, for $c = 4$ and $n_0 = 0$, it is verified that

$$4n^3 + 2n + 3 = \Omega(n^3)$$

Thus, the given function is of the order of $\Omega(n^3)$.

Big Theta Notation (θ) : The lower and upper bound for the function 'f' is provided by the Big Theta notation (θ).

Definition : Considering 'g' be a function from the non-negative integers into the positive real numbers. Then $\theta(g) = O(g) \cap \Omega(g)$, that means, the set of functions that are both in O(g) and Ω(g). The θ(g)is the set of function 'f' such that for some positive constants c_1 and c_2 and an n_0 exists such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for all n, $n \geq n_0$. By "$f_{(n)} = \theta(g_{(n)})$" we mean "f is order of g"

find out the order of function $f(n) = 7n + 5$ in Big Theta Notation

From the definition of Big Theta we can write,

$$c_1g(n) \leq c_2g(n)$$

or

$$c_1n \leq 7n + 5 \leq c_2n$$

or

$$c_1 \leq 7n + 5/n \leq c_2$$

The right-handed inequality can be made to hold for any value of $n \geq 5$ by choosing $c_2 \geq 8$. Similarly, the left-handed inequality can be made to hold for any value of $n \geq 5$ by choosing $c_1 \leq 7$.

So,

$$7n \leq 7n+5 \leq 8n$$

Thus, for $c_1 = 7$, $c_2 = 8$ and $n_0 = 5$, it is proved that $7n+5 = \theta(n)$. In other words, f(n) is of the order of θ(n).

Little oh Notation (o) : Big oh notation imposes asymptotically tight bound on function f(n). If we say that $2n + 3 = O(n^2)$, it is not the tighter bound on this function as we have the smaller linear function that also satisfies the Big oh relation, *i.e.*, $2n + 3 = O(n)$.

Little o-notation denotes an upper bound same as Big O notation, but this upper bound is not asymptotically tight Formally, it can be defined as follows.

For a given function g(n), o(g(n)) gives the set of functions f(n) as

o(g(n)) = {f(n) : for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such tht $0 \leq f(n) < c\,g(n)$ for all $n \geq n_0$}

Thus, f(n) = o(g(n)) for any and every constant positive value of c. As n becomes larger and approaches to infinity, f(n) becomes insignificant as compared to g(n).

Mathematically,

$$\lim_{n\to\infty} f(n)/g(n) = 0$$

Example : Examples of few correct bounds for little o notation are as follows :

$3n + 5 = o(n^2)$, as $3n + 5 = O(n^2)$

$10n^2 + 7 = o(n^3)$, as $10n^2 + 7 = O(n^3)$

But if we write $3n + 5 = o(n)$, it would be an incorrect bound as

$$\lim_{n\to\infty} f(n)/g(n) = 0$$

i.e.

$$\lim_{n\to\infty} (3n+5)/n = 3 \neq 0$$

So, $f(n) = 3n + 5 \neq o(n)$

Similarly, $\lim_{n\to\infty} 10n^2 + 7)/n^2 = 10 \neq 0$

Thus, $f(n) = 10n^2 + 7 \neq o(n^2)$

Little Omega Notation (ω)

Big Omega Notation imposes asymptotically tight lower bound on function f(n). To write that $3n + 5 = \Omega(n^2)$ is not the tighter lower bound on the function because it has the smaller linear function that also satisfies the big omega notation, *i.e.*, $3n + 5 = \Omega(n)$. Little Omega denotes the loose lower bound on the function. For above function, $3n + 5 = \omega(n^2)$ is correct bound. Formally it can be defined as follows:

For a given function g(n), ω(g(n)) gives the set of functions f(n) as ω(g(n)) = {f(n) : for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq c\,g(n) < f(n)$ for all $n \geq n_0$}

As the value of n approaches infinity, f(n) becomes very large as compared to g(n).

Mathematically,

$$\lim_{n\to\infty} f(n)/g(n) = \infty$$

Example : Let us try to understand this with the help of examples :

$3n + 5 = \omega(1)$, as $3n + 5 = \Omega(n)$

$10n^2 + 7 = \omega(n)$, as $10n^2 + 7 = \Omega(n^2)$

If we write $3n + 5 = \omega(n)$, it would be an incorrect bound as

$$\lim_{n\to\infty} f(n)/g(n) \neq \infty$$

i.e.

$$\lim_{n\to\infty} (3n+5)/n = 3 \neq \infty$$

So, $f(n) = 3n + 5 \neq \omega(n)$

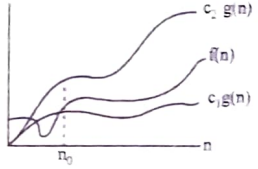
Whereas, for $3n + 5 \neq \omega(1)$,

$$\lim_{n\to\infty} (3n+5)/1 = \infty$$

Thus the above little omega notation for the function 3n+5 is correct.

Graphic examples of the θ,O and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work.

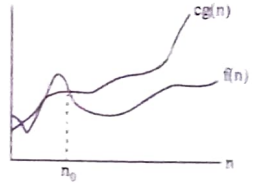
(a) θ-notation bounds a function to within constant factors. We write $f(n) = \theta(g(n))$ if there exist positive constants n_0,c_1, and c_2 such that to the right of n_0, the value of f(n) always lies between $c_1g(n)$ and $c_2g(n)$ inclusive.



f(n)= θ(g(n))

Fig.

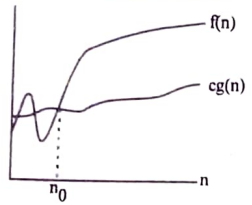
(b) O-notation gives an upper bound for a function to within a constant factor. We write f(n)=O(g(n)) if there are positive constants n_0 and c such that to the right of n_0, the value of f(n) always lies on or below cg(n).



f(n)=O(g(n))

Fig.

(c) Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of f(n) always lies on or above cg(n).



$f(n)=\Omega(g(n))$

Fig.

Q.16 ***(i) Solve the recurrence***
$T(n) = T(n-1)+T(n-2)+1$, *when* $T=0$
$T(1)=1$
(ii) If $f(n) = 100 \times 2^n + n^3 + n$ ***show that*** $f(n)=O(2^n)$ *[R.T.U. 2017]*

Ans. (i) $T(n) = T(n-1) + T(n-2) + 1$, when $T = 0$

$T(1) = 1$

$T(0) = 0$

$T(2) = T(1) + T(0) + 1 = 2$

Recursion Tree

Depth of tree is n and upto the $(n/2)^{th}$ level each level will have 2^k nodes $(0 \le k \le \log_2 (n+1))$. Final base case is F(1) and F(0).

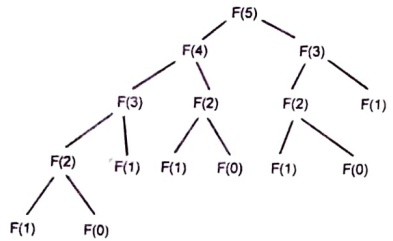


Fig.

∴ Total nodes in recursion tree

$= 1 + 2 + 4 + \ldots + (n-1)$

$= 1(2^n - 1)/(2 - 1)$

$= 2^n - 1$

∴ Recurrence relation $= 0(2^n)$

Using closed form of Fibonacci series

$$F_n = \frac{\phi^n - \varphi^n}{\phi - \varphi} = \frac{\phi^n - \varphi^n}{\sqrt{5}}$$

$$\phi = \frac{1+\sqrt{5}}{2} = 1.618 \quad \text{(Golden Ratio)}$$

$$\varphi = \frac{1-\sqrt{5}}{2} = 1 - \phi = \frac{-1}{\phi} \quad \text{since, } \varphi = \frac{-1}{\phi}$$

$$\therefore \quad F_n = \frac{\phi^n - (-\phi)^n}{\sqrt{5}}$$

Hence, $F_n \approx \phi^n / \sqrt{5}$

$T(n) = \Theta(1.6^n)$

(ii) $f(n) = 100 \times 2^n + n^3 + n$

$f(n) = O(2^n)$

let $g(n) = 2^n$

then $f(n) = O[g(n)]$

$0 \le f(n) \le g(n)$

$\forall \quad n \ge n_0$ for a +ve constant c.

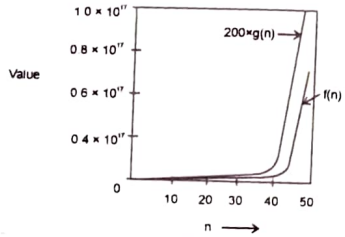


Fig.

Eliminating lower order term

$f(n) = 100 \times 2^n + n^3$

for c = 200

$100 \times 2^n + n^3 + n \le 200 \times 2^n$

$\therefore \quad f(n) = 0(2^n)$

$f'(n) = 100 \times 2^n + 3n^2 + 1$

$200 \times g'(n) = 200 \times 2^n$

since, both are monotonically increasing function with respect to n.

$f(1) = 100 \times 2 + 1 + 1 = 202$

$200 \times g(1) = 200 \times 2^1 = 400$

$\therefore \quad 200 \times g(1) \ge f(1)$, we can say that $f(n) = 0(2^n)$

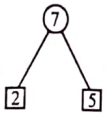
Q.17 ***One file has ten characters (a, b, c, '\n', e, f, g, h, i ') whose frequencies are 27, 31, 11, 5, 72, 51, 89, 33, 2, and 10 respectively.***
(i) Find optimal binary merge pattern tree for above file.
(ii) Generate Huffman code for string "fge\nah".
(iii) Translate given Huffman code 111101110111 1110000110 into text format by above pattern. *[R.T.U. 2014]*

Ans.(i) 27, 31, 11, 5, 72, 51, 89, 33, 2, 10

Sort the elements in increasing order

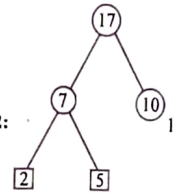
2, 5, 10, 11, 27, 31, 33, 51, 72, 89

Step 1:



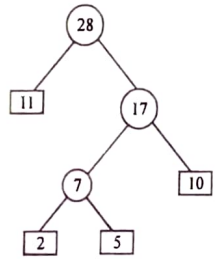
10 11 27 31 33 51 72 89

Step 2:



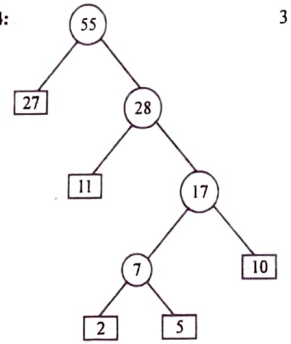
11 27 31 33 51 72 89

Step 3:



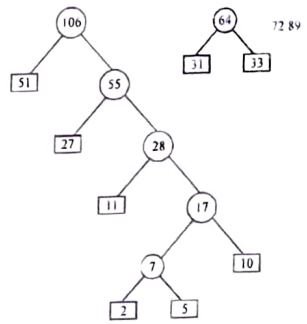
27 31 33 51 72 89

Step 4:

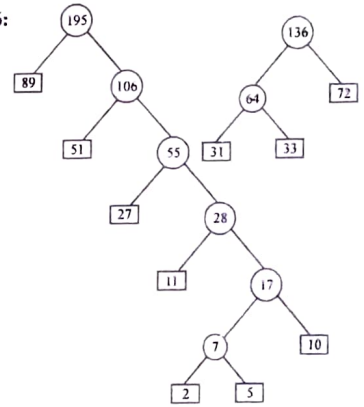


31 33 51 72 89

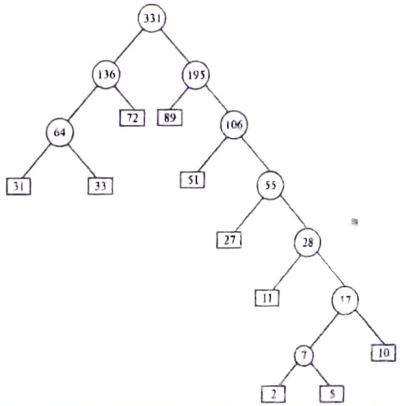
Step 5:



Step 6:



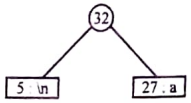
Step 7:



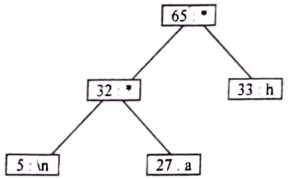
(ii) Frequencies for string (Given)

51 89 72 5 27 33

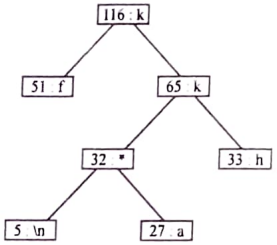
Step 1: Minimum two element



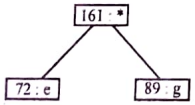
Step 2: 32 33 51 72 89



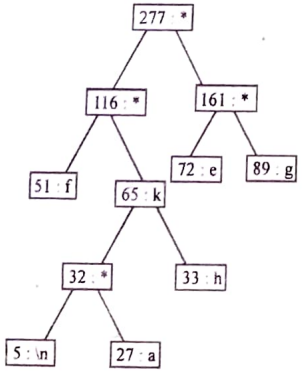
Step 3: 51 65 72 89



Step 4. 72 89 116

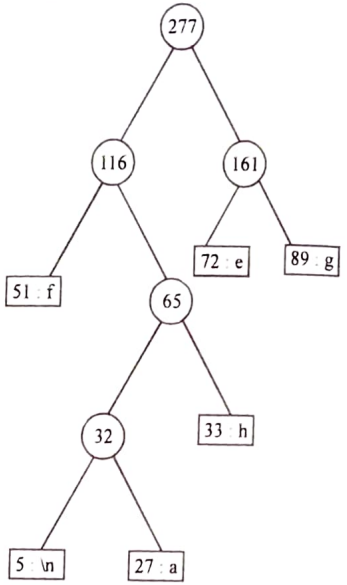


Step 5: 116 161

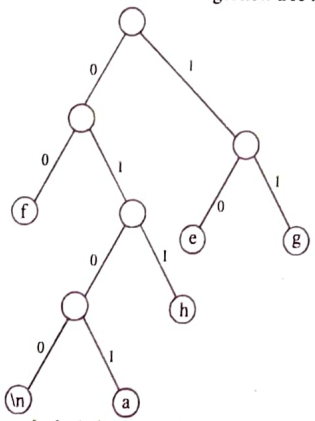


(iii)

The given tree is



Now we consider the left side of the node is bit '0' and right side the node is '1' so we get new tree is



Encode the below given binary pattern is

1111011101111110000110

So we encode this pattern according to given tree first we pick the 4-bit so bit is '1111'. This bit does not exist in the given huffman tree so we take 3-bit, so bit is '111'. This is also not exist so we take 2-bit, so bit is '11'. The value of this

sequence is, we know that '1' means the traversal into right direction so '11' means 'g'.

Now follow the same procedure and decode this binary pattern. So decoded pattern is

g g h e g g g f f g Extra bit (ignored)

11 11 011 10 11 11 11 00 00 11 0

We have seen a extra bit so we ignored this extra bit and stop seeing.

So text of equivalent to the given binary pattern is ggheggggffg.

Q.18 ***Apply Strassen's algorithm to compute, using 2×2matrices, existing the recursion when n = 2***

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

[R.T.U. 2013]

Ans. In Strassen's algorithm, we proceed by partitioning the given matrices into submatrices.

$$\left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \times \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right]$$

$$= \left[\begin{array}{cc|cc} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ \hline 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{array}\right] \times \left[\begin{array}{cc|cc} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ \hline 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{array}\right]$$

Let us complete each A × B matrix by following formula.

$$S_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$

$$= \left(\begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix}\right) \times \left(\begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 4 & 0 \\ 6 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 7 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} (4 \times 1) + (0 \times 7) & (4 \times 2) + (0 \times 1) \\ (6 \times 1) + (2 \times 7) & (6 \times 2) + (2 \times 1) \end{bmatrix}$$

$$S_1 = \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix}$$

$$S_2 = (A_{21} + A_{22}) \times B_{11}$$

$$= \left(\begin{bmatrix} 0 & 1 \\ 5 & 0 \end{bmatrix} + \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix}\right) \times \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 3 & 1 \\ 7 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} (3 \times 0) + (1 \times 2) & (3 \times 1) + (1 \times 1) \\ (7 \times 0) + (1 \times 2) & (7 \times 1) + (1 \times 1) \end{bmatrix}$$

$$S_2 = \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix}$$

$$S_3 = A_{11} \times (B_{12} - B_{22})$$

$$= \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} \times \left(\begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix} - \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 \\ -5 & 4 \end{bmatrix}$$

$$= \begin{bmatrix} (-1 \times 1) + (-5 \times 0) & (1 \times 0) + (0 \times 4) \\ (-1 \times 4) + (-5 \times 1) & (4 \times 0) + (1 \times 4) \end{bmatrix}$$

$$S_3 = \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix}$$

$$S_4 = A_{22} \times (B_{21} - B_{11})$$

$$= \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix} \times \left(\begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} (3 \times 2) + (-1 \times 0) & (-1 \times 3) + (2 \times 0) \\ (2 \times 2) + (-1 \times 1) & (-1 \times 2) + (2 \times 1) \end{bmatrix}$$

$$S_4 = \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix}$$

$$S_5 = (A_{11} + A_{12}) \times B_{22}$$

$$= \left(\begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}\right) \times \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 3 & 1 \\ 5 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} (3 \times 1) + (1 \times 5) & (3 \times 1) + (1 \times 0) \\ (5 \times 1) + (1 \times 5) & (5 \times 1) + (1 \times 0) \end{bmatrix}$$

$$S_5 = \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix}$$

$$S_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$$

$$= \left(\begin{bmatrix} 0 & 1 \\ 5 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix}\right) \times \left(\begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix}\right)$$

$$= \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} 0 & 2 \\ 2 & 5 \end{bmatrix}$$

$$= \begin{bmatrix} (-1 \times 0) + (1 \times 2) & (-1 \times 2) + (1 \times 5) \\ (1 \times 0) + (-1 \times 2) & (1 \times 2) + (-1 \times 5) \end{bmatrix}$$

$$S_6 = \begin{bmatrix} 2 & 3 \\ -2 & -3 \end{bmatrix}$$

$$S_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

$$= \left(\begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix}\right) \times \left(\begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}\right)$$

$$= \begin{bmatrix} -1 & 1 \\ -1 & -1 \end{bmatrix} \times \begin{bmatrix} 3 & 1 \\ 6 & 3 \end{bmatrix}$$

$$= \begin{bmatrix} (-1 \times 3) + (1 \times 6) & (-1 \times 1) + (1 \times 3) \\ (-1 \times 3) + (-1 \times 6) & (-1 \times 1) + (-1 \times 3) \end{bmatrix}$$

$$S_7 = \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix}$$

Now, $C_{11} = S_1 + S_4 - S_5 + S_7$

$$= \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix} + \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix} - \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix} + \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix}$$

$$C_{11} = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}$$

$$C_{12} = S_3 + S_5$$

$$= \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix} + \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix}$$

$$C_{12} = \begin{bmatrix} 7 & 3 \\ 1 & 9 \end{bmatrix}$$

$$C_{21} = S_2 + S_4$$

$$= \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix} + \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 8 & 1 \\ 5 & 8 \end{bmatrix}$$

$$C_{22} = S_1 + S_3 - S_2 + S_6$$

$$= \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix} - \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ -2 & -3 \end{bmatrix}$$

$$= \begin{bmatrix} 3 & 7 \\ 7 & 7 \end{bmatrix}$$

Thus, the final solution is

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix} = \begin{bmatrix} 5 & 4 & 7 & 3 \\ 4 & 5 & 1 & 9 \\ 8 & 1 & 3 & 7 \\ 5 & 8 & 7 & 7 \end{bmatrix}$$

Q.19 ***Give asymptotic upper bound or lower bound on each of the following recurrences. Assume that T(n) is constant and make your bound as tight as possible. Justify your answer (any four).***

(i) $T(n) = 3T(n/2) + n \log n$

(ii) $T(n) = 5T(n/5) + n/n \log n$

(iii) $T(n) = 4T(n/2) + n^2\sqrt{n}$

(iv) $T(n) = 3T\left(\frac{n}{3}+5\right) + n/2$

(v) $T(n) = T(n-2) + 2 \log n$ *[R.T.U. 2011]*

Ans.(i) Comparing given recurrence with the standard recurrence relation used in Master's theorem, we get

$a = 3, b = 2, f(n) = n \log n$

$$n^{\log_b^a} = n^{\log_2^3} = n^{0.7298}$$

Since, order of f(n) can not be directly expressed as a polynomial, we consider

$f(n) = \Omega\left(n^{\log_b^{(a+\varepsilon)}}\right)$, for which

$\log_2^3 = 0.7298$

where $\varepsilon \sqcup 0.2$

So case 3 of Master theorem applies.

Performing the regulatory check, one extra condition should be satisfied. $a.f(n/b) \le c.f(n)$, for some constant $c < 1$.

If $3f(n/2) < cn \log n$

$$\Rightarrow 3.\frac{n}{2}\log\left(\frac{n}{2}\right) \le cn \log n$$

For large values of n, $\left(\frac{3}{2}\right) n \log n \le cn \log n$ or $cf(n) < cn \log n$

where, $c = \frac{3}{2}$

Thus, the solution to the recurrence relation is

$T(n) = \theta(f(n)) = \theta(n \log n)$

(ii) We have

$a = 5, b = 5, f(n) = n/n \log n$

So, $n^{\log_b^a} = \log_5 5 = n$

None of the Master theorem cases may be applied here, since f(n) is neither polynomially bigger or smaller than n and is not equal to $\theta\left(n \log_n^k\right)$ for any $k \ge 0$. Therefore, we solve this problem by algebraic expression.

$$T(n) = 5T(n/5) + n/n \log n$$
$$= 5(5T(n/25) + (n/5)(\log(n/5)) + n/ n \log n$$
$$= 25T(n/25) + n/\log n(n/5) + n/n \log(n)$$
$$= 5^i T(n/5^i) + \sum_{j=1}^{i-1} n/n \log(n/5^j)$$

When $i = \log_5 n$ the first term reduced to $5\log_5^n T(1)$, so we have

$$T(n) = n\theta(1) + n \sum_{j=1}^{\log_5 n - 1} (n/(\log(n/5^{j-1})))$$

$$= \theta(n) + n \sum_{j=1}^{\log_5 n - 1} (1/\log n - (j-1)\log_2 5)$$

$$= \theta(n) + n\left(1/\log_2 5\right) \sum_{j=1}^{\log_5 n - 1} (1/(\log_5 n - (j-1)))$$

$$= \theta(n) + n \log_5 2 \sum_{i=2}^{\log_5 n} \left\{\frac{1}{i}\right\}$$

This is the harmonic sum, so, we have

$T(n) = \theta(n) + C_2 n \ell n (\log_5 n) + \theta(1) = \theta(n \log n)$.

(iii) $T(n) = 4T\left(\frac{n}{2}\right) + n^2\sqrt{n}$

$\therefore \sqrt{n} = \log n$

We have Master theorem case 2 as

$$f(n) = \theta\left(n^{\log_b^a} \log^k n\right) \Rightarrow \theta\left(n^{\log_b^a} \log^{k+1} n\right)$$

for $k \ge 0$

From this theorem $a = 4, b = 2, f(n) = n^2 \log n$

So, $n^{\log_b^a} = n^2$

Since $f(n) = \theta(n^2 \log n)$

$T(n) = \theta(n^2 \log^2 n)$

$$T(n) = 3T\left(\frac{n}{3}+5\right) + \frac{n}{2}$$ **Ans.**

(iv) For $T(n) = O(n \log n)$

We have to show that for some constant c

$T(n) \le cn \log n$

$$T(n) \le cn\left(\frac{n}{3}+5\right)\log\left(\frac{n}{3}+5\right) + \frac{n}{2}$$

$$= cn \log\left(\frac{n}{3}\right) + 10 + \frac{n}{2}$$

$$= cn \log n - cn \log 3 + 10 + \frac{n}{2}$$

$$= cn \log n - cn + 10 + \frac{n}{2}$$

$$= cn \log n - (c - 1/2)n + 10$$

$$= cn \log n - b \le cn \log n$$

if $c \ge 1$, b is constant

Thus, $T(n) = \theta(n \log n)$ **Ans.**

(v) We solve this problem by algebraic subsitution

$$T(n) = T(n-2) + 2 \log n$$
$$= T(n-2) + 2 \log n$$
$$= O(1) + \sum_{i=1}^{n} \log i$$
$$= \theta(1) + \log\left(\prod_{i=1}^{n} i\right)$$
$$= \theta(1) + \log(n!)$$
$$= \theta(n \log n)$$ **Ans.**

❑❑❑

GREEDY METHOD & DYNAMIC PROGRAMMING 2

CHAPTER IN A NUTSHELL

❑ Matrix Chain Multiplication

The matrix chain multiplication problem involves a series of operations to be performed. A dynamic programming approach gives the optimal sequence of operation for this problem. The MCM problem is important in the area of computer design and database, as one wants to achieve code optimization and query optimization.

In dynamic programming approach, we always break down the main problem into several smaller subproblems. Later the solution to these subproblems are combined to give the solution to a main problem. As we know that we cannot change the order of matrices, therefore the only reliable task is to manage the parenthesis.

Let $A_{i,j}$ indicates the multiplication of matrices through i to j. It can be seen that the dimension of $A_{i,j}$ is $p_{i-1} \times p_j$. For optimum matrix multiplication break the problem into several simpler and smaller subproblems of similar structure. We consider the highest level of parenthesizing, where two matrix multiplications are considered for instance, if we consider k, where

$$1 \le k < n-1, \text{ then}$$
$$A_{1,n} = A_{1,k} \times A_{k+1,n}$$

For this, the sequence contains only one matrix if $i = j$, and so the cost of operation is 0. Thus, m mul $[i, j] = 0$.

The multiplication for $A_{i,j}$ is computed if $i < j$. We can split the sequence by considering each k, where $i \le k < j$ as $A_{i,k}$ items $A_{k+1,j}$.

It can be seen that m mul[i, k] and m mul[k − 1, j] are the optimum multiplication needed to compute $A_{i,k}$ are $A_{k+1,j}$ ($A_{i,j}$) respectively.

Also assume that all these values are computed previously and maintained in an array.

Since, p_{i-1} p_k is dimension of $A_{i,k}$ and $p_k \times p_j$ is the dimension of $A_{k+1,j}$, the total number of operations needed to multiply them is p_{i-1} p_j p_j.

From the above discussion we have the following recursive rule for ompputing m mul[i, j]

mmul[i, j]

$$= \begin{cases} 0 & \text{if} \quad i = j \\ \min(m\,mul[i,k] + m\,mul[k+1,j] + p_{i-1} p_k p_j & \text{if} \quad i < j, i \le k < j \end{cases}$$

❑ Longest Common Subsequence

The dynamic programming (DP) technique suggests a breakup of the main problem into several smaller subproblems. There are many ways of breaking up the strings but we shall concentrate on the prefines of the string sequence. A prefix of a sequence indicates an initial string of values, $p_i = < p_1, p_2, p_3 \ldots p_i >$. p_0 indicates the empty sequence. The method is related to compute the LCS for all possible pair of prefixes. Consider LCS[i, j] to be the length of the longest common subsequence of p_i and q_j for example

p_3 = <KLO>

q_4 = <XKLL>

Thus, their longest common subsequence LCS is <KL>. Therefore LCS[3, 4] = 2

The DP method is to compute LCS[i, j] in a manner by assuming that LCS[i', j'] is already computed where $i' < i$ and $j' \le j$, but not equal.

Basis, Either of the sequence is empty : In this case, LCS[i, 0] = LCS[0, j] = 0. That means if either of the sequence is empty, then the longest common subsequence is also empty.

When last character of both the sequence match : Consider $p_i = q_j$ for example

p_i = <LMNL>

q_j = <OLNL>

It is understood that both the sequences end in character 'L', thus we can say that longest common subsequence for the above sequences must end in 'L'.

The overall longest common subsequence can be obtained by removing the last 'L' as we know that it is a part of longest common subsequence. So, we compute LCS of p_{i-1} = <LMN> and q_{j-1} = <OLN> which is <LN>. Later on, 'L' is added at the end of this longest common subsequence <LN>, so as to give the final solution which is <LNL>.

When last character of either sequence does not match: Consider $p_i \ne q_j$ for example

In this case, p_i and q_j cannot be the part of LCS simultaneously as for this they would have be the last character of the LCS.

Thus, either p_i be the part of LCS or q_j be the part of LCS. The condition may also arise where both p_i and q_j are not the parts of LCS.

Considering all the cases we can say

$$LCS[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \ne y_j \end{cases}$$

❑ 0/1 Knapsack Problem

In this problem the list of items are indivisible, that means either we take the item or discard it.

Given a set of n items and Knapsack having capacity w, such that each item has some weight w_i and value v_i then the problem is to pack the Knapsack in such a manner so that a maximum total value is achieved.

The problem is said to be 0/1 Knapsack problem because the item from the list is either rejected or accepted.

We can formulate the subproblems as recursive rule

$$K_p[k,\omega] = \begin{cases} K_p[k-1,\omega] & \text{if } \omega_k > \omega_1 \\ \max\left(K_p[k-1,\omega], K_p[k-1,\omega-\omega_k] + v_k\right) & \text{otherwise} \end{cases}$$

❑ Knapsack Problem

Knapsack problem follows the greedy strategy where the items are arranged in some order which is based on some greedy criterion.

Greedy Criterion I : In this criterion, the items are arranged by their values. Here the item with maximum value is selected first, and process continues till the minimum value.

Greedy Criterion II : In this criterion, the items are arranged by their weights. Here the item with lightest weight is selected first and process continues till the maximum weight.

Greedy Criterion III : In this criterion, the items are arranged by certain ratio, P_i, where P_i is the ratio of value over weight. Here selection proceeds from maximum ratio to minimum ratio.

After taking the greedy criterion, we check whether the weight W_i of item I, from the top of the list is less than the Knapsack capacity, W or not. If W_i is less than or equal to 'W' then the item is selected and the whole process continues with $I_{i+1}, I_{i+2}, \ldots$ in item along with new maximum weight, W − W_i. Otherwise the item having 'W_i' is ignored and process continues with $I_{i+1}, I_{i+2}, \ldots$ in item along with new maximum weight, W.

❑ Job Sequencing

In job scheduling problem, we have to determine the optimum number of activities that are scheduled to the resources.Given a set S = {1, 2, 3, ... n} of 'n' number of activities, each job is scheduled to use some resources, condition might arise where multiple activity has its start and finish time denoted by 'si' and 'fi' respectively. A condition might arise where multiple activities are scheduled to have a common resource, or the start or finish time of activities may overlap. From the given set of activities (jobs), we can say that Ai and Aj are non-interfering activities, if and only if they both have different start and finish times that means their start and finish times do not overlap. $A_i = [s_i, f_i]$ and $A_j = [s_j, f_j]$, so for non-interfering $A_i \cap A_j = \phi$.

Here we are giving the greedy strategy which provides the optimum result. The idea behind this strategy is same as we are discarding that activity which has longer duration $(f_i - s_i)$. In this strategy, we first select the activity with minimum duration $(f_i - s_i)$ and schedule it. Next, we skip all activities that interface with this one, that means we have to select non-interfacing activity having minimum duration and then we have to schedule it. The process is repeated until all the activities are considered. It can be observed that the process of selecting the job becomes faster, if we assume that the jobs are arranged in an increasing order of their finish times.

$$f_1 \le f_2 \le f_3 \ldots \le f_n$$

❑ Optimal Merge Patterns

A greedy attempt to obtain an optimal merge pattern is easy to formulate. Since merging an n-record file and an m-record file requires possibly n + m record moves, the obvious choice for selection criterion is; at each step merge the two smallest size files together. Thus, if we have five files $(x_1, \ldots, x_5)$ with sizes (20, 30, 10, 5, 30), our greedy rule would generate the following merge pattern. Merge x_4 and x_5 to get Z_1 ($|Z_1|$ = 15), merge Z_1 and x_3 to get $Z_2(|Z_2|$ = 35), merge x_2 and x_3 to get $Z_3(|Z_3|$ = 60), and merge Z_2 and Z_3 to get the answer Z_4. The total number of record moves is 205. This merge pattern such as one first described will be referred to as a two-way merge pattern. The two-way merge patterns can be represented by binary merge trees. Example of optimal merge pattern is Huffman codes.

❑ Minimal Spanning Trees

If a weighted graph is considered, then the weight of the spanning tree (T) of graph 'G' can be calculated by summing all the individual weights in the spanning tree T. But we have seen that for any graph 'G' there can be many spanning trees. This is also in the case of weighted graphs from which we can get different spanning trees having different weights. A Minimum Spanning Trees (MST) is a spanning tree of minimum weight.

To compute the minimum cost spanning tree, we are presenting two algorithms :

1) Kruskal's Algorithm 2) Prim's Algorithm

Both are based on greedy technique.

Kruskal's Algorithm : In Kruskal's algorithm, an edge is selected in such a manner that it contains a minimum weight

and upon adding to 'M' does not include any cycle. That means every time the lightest edge is selected first and then is added to 'M' (only if no cycle is obtained).

Prim's Algorithm : For finding the minimum spanning trees Prim's algorithm is yet another algorithm which is also based on the greedy technique. The Prim's algorithm differs from the Kruskal algorithm only in the way of selecting the next safe edge which does not produce cycle upon adding. The running time of Prim's algorithm is essentially same as Kruskal's algorithm $O(V + E) \log V$. The importance of Prim's algorithm is that at loops it is very similar to an algorithm which is known as Dijkstras algorithm, used for finding shortest paths.

PREVIOUS YEARS QUESTIONS

PART-A

Q.1 ***What do you mean by MCM problem.***

Ans. The matrix chain multiplication problem involves a series of operations to be performed. The MCM problem is important in the area of computer design and database, as one wants to achieve code optimization and query optimization.

Q.2 ***Define dynamic programming approach.***

Ans. In dynamic programming approach, we always break down the main problem into several smaller subproblems. Later the solution to these subproblems are combined to give the solution to a main problem.

Q.3 ***Write the steps to develop a dynamic programming algorithm.***

Ans. Steps to develop a dynamic programming algorithm

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up manner.
4. Construct an optimal solution from computed information.

Q.4 ***What is 0/1 Knapsack problem.***

Ans. In this problem the list of items are indivisible, that means either we take the item or discard it.

Given a set of n items and Knapsack having capacity w, such that each item has some weight w_i and value v_i then the problem is to pack the Knapsack in such a manner so that a maximum total value is achieved.

The problem is said to be 0/1 Knapsack problem because the item from the list is either rejected or accepted.

PART-B

Q.5 ***Find the optimal parenthesization of matrix-chain product whose sequence of dimension is (4, 10, 4, 40, 5).*** *[R.T.U. 2018, 2014]*

Ans. The chain (4, 10, 4, 40, 5) means the matrices to be multiplied are of following dimensions.

$m_1 : (4 \times 10)$, $m_2 : (10 \times 4)$, $m_3 : (4 \times 40)$, $m_4 : (40 \times 5)$

All A[i][i] = 0 [base value]

Thus, $A_{11} = 0$, $A_{22} = 0$, $A_{33} = 0$, $A_{44} = 0$

	4	3	2	1
1			160	0
2		1600	0	
3	800	0		
4	0			

A_{12} = operations $(m_1\ m_2) = 4 \times 10 \times 4 = 160$

A_{23} = operations $(m_2\ m_3) = 10 \times 4 \times 40 = 1600$

A_{34} = operations $(m_3\ m_4) = 800$

	4	3	2	1
1		800	160	0
2	1000	1600	0	
3	800	0		
4	0			

A_{13} = min[A_{12} + operations $(m_{12}\ m_3)$, A_{23} + operations $(m_1\ m_{23})$]

= min[160 + (4 × 4 × 40), 1600 + (4 × 10 × 40)]

= [800, 3200] = 800

A_{24} = min [A_{23} + operations $(m_{23}\ m_4)$, A_{34} + operations $(m_2\ m_{34})$]

= min[1600 + (10 × 40 × 5), 800 + (10 × 4 × 5)]

= min [3600, 1000]

= 1000

A_{14} = min[A_{13} + operations $(m_{12}\ m_4)$, A_{24} + operations $(m_1\ m_{24})$]

= min[800 + (4 × 40 × 5), 1000 + (4 × 10 × 5)]

= min[1600, 1200]

= 1200

	4	3	2	j
1	1200	800	160	0
2	1000	1600	0	
3	800	0		
4	0			

$(m_{24}\ m_4) = ((m_{34}\ m_2)\ m_4)$

$= (m_{12}\ m_3) = ((m_1\ m_2)\ m_3)m_4)$

Q.6 ***Solve the following instance of LCS problem through dynamic programming***

$x = ABCDCDBCAD$

$y = BACCDCABBD$ *[R.T.U. 2018, 2015]*

Ans. In our problem

X= ABCDCDBCAD

Y= BACCDCABBD

So the process is like this :

```
LCS-LENGTH(X, Y)
1   m = length[X]
2   n = length[Y]
3   for i = 1 to m
4       do c[i, 0] = 0
5   for j = 1 to n
6       do c[0, j] = 0
7   for i = 1 to m
8       do for j = 1 to n
9           do if Xi == Yj
10              then c[i, j] = c[i – 1, j – 1] + 1
11                   b[i, j] = ARROW_CORNER
12              else if c[i – 1, j] >= c[i, j – 1]
13                   then c[i, j]=c[i-1, j]
14                        b[i, j] = ARROW_UP
15                   else c[i, j]=c[i, j – 1]
16                        b[i, j] = ARROW_LEFT
17  return c and b
```

Input X	ABCDCDBCAD
Input Y	BACCDCABBD

i : 10, j : 10 — X : D, Y : D — Step : $X_i = Y_j$ equal to 'A' = 'D' b[10, 10] = ↑ and c[10, 10] = 6+1=7 See line number 10 and 11

i	j	0	1	2	3	4	5	6	7	8	9	10
		X	B	A	C	C	D	C				
0	x	0	0	0	0	0	0	0	0	0	0	0
1	A	0	↖ 0	↖ 1	← 1	← 1	← 1	↖ 1	← 1	← 1	← 1	← 1
2	B	0	↖ 1	↑ 1	↑ 1	↑ 1	↑ 1	↑ 1	↖ 2	↖ 2	← 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	↖ 2	← 2	↖ 2	← 2	← 2	↑ 2	↑ 2
4	D	0	↑ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3	↖ 3	← 3	↑ 3	↖ 3
5	C	0	↑ 1	↑ 1	↖ 2	↖ 3	↑ 3	↖ 4	← 4	← 4	← 4	← 4
6	D	0	↑ 1	↑ 1	↑ 2	↑ 3	↖ 4	↑ 4	↑ 4	↑ 4	↑ 4	↖ 5
7	B	0	↖ 1	↑ 1	↑ 2	↑ 3	↑ 4	↑ 4	↑ 4	↖ 5	↖ 5	↑ 5
8	C	0	↑ 1	↑ 1	↖ 2	↖ 3	↑ 4	↖ 5	← 5	↑ 5	↑ 5	↑ 5
9	A	0	↑ 1	↖ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6	← 6
10	D	0	↑ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 5	↑ 6	↑ 6	↑ 6	↖ 7

Q.7 ***Explain and write an algorithm for greedy method of algorithm design. Given 10 activities along with their start and finish time as***

$S = \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10}\}$

$S_i = \{1, 2, 3, 4, 7, 8, 9, 9, 11, 12\}$

$F_i = \{3, 5, 4, 7, 10, 9, 11, 13, 12, 14\}$

Compute a schedule where the largest number of activities take place. *[R.T.U. 2017]*

Ans. A greedy algorithm always makes the choice that seems to be best at that moment. This means that is makes a locally optimal choice in the hope that this choice will lead to globally optimal solution.

In many problems, a greedy strategy does not in general procedure an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solution that approximate a global optimal solution in reasonable time.

When we have to decide which choice is optimal, we assume that we have an objective function that needs to be optimized at a given point. A greedy algorithm makes greedy choice at each step to ensure that the objective function is optimized. The greedy algorithm has only one show to compute the optimal solution so that it never goes back and reverses the decision.

For the given example, the greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of previously selected activity. We can sort the activities according to their finishing time so we always consider the next activity as minimum finishing time activity.

$S = \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10}\}$

$S_i = \{1, 2, 3, 4, 7, 8, 9, 9, 11, 12\}$

$F_i = \{3, 5, 4, 7, 10, 9, 11, 13, 12, 14\}$

Sorting according to F_i

$S = \{A_1, A_3, A_2, A_4, A_6, A_5, A_7, A_9, A_8, A_{10}\}$

a. 1. Sort the activities according to their finishing time
2. Select the first activity from sorted array
3. Do following for remaining activities in sorted array

If the start time of this activity is greater than or equal to the finish time of previously selected activity then select this activity.

Schedule

$A_1, A_3, A_4, A_6, A_7, A_9, A_{10}$

7 activities can be performed

Q.8 ***Explain Matrix chain multiplication. Also find the parenthesization for the following matrix A1 = 15×10, A2 = 10×20, A3 = 20×25.*** ***[R.T.U. 2016]***

Ans. Matrix Chain Multiplication : We have sequence (chain) $<A_1, A_2, \ldots, A_n>$ of n matrices to be multiplied and we wish to compute the product

$$A_1, A_2, A_3, \ldots A_n$$

A product of matrix is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix surrounded by parenthesis, matrix multiplication is associative and so all parenthesization yield the same product.

We can multiply two matrices A and B only if the number of column of A is equal to the number of row of B. If A is a $p \times q$ matrix and B is a $q \times r$ matrix the resultant matrix C is a $p \times r$ matrix. The time to compute C is dominated by the number of scalar multiplication.

Suppose that the dimensions of matrices are 10×100, 100×5 and 5×50 respectively. If we multiply according to the parenthesization $((A_1 \cdot A_2) \cdot A_3)$ we perform $10 \times 100 \times 5 = 5000$ scalar multiplication to compute the 10×5 matrix to multiply this matrix by A_3, for a total of 7500 scalar multiplication. similarly, if we multiply according to the parenthesization $(A_1 \cdot (A_2 \cdot A_3))$ we perform total of 75000 scalar multiplication. Thus, computing the product according to the first parenthesization is 10 times faster.

Parenthesization of following matrix

$A_1 = 15 \times 10$

$A_2 = 10 \times 20$

$A_3 = 20 \times 25$

Firstly find vector d as: take row part of each matrix and column parts of last matrix i.e. 25

$\Rightarrow$ $d = (15, 10, 20, 25)$

Where,

$d_0 = 15$

$d_1 = 10$

$d_2 = 20$

$d_3 = 25$

For diagonal s = 1

$m_{i,i+1} = m[i,i] + m[i+1,i+1] + d_{i-1} \cdot d_i \cdot d_{i+1}$, $i = 1, 2, \ldots n-1$

$m_{12} = m_{11} + m_{22} + d_0 d_1 d_2$

$= 0 + 0 + 15 \times 10 \times 20$

$= 3000$

When $i = 2$

$m_{23} = m_{22} + m_{33} + d_1 d_2 d_3$

$= 0 + 0 + 10 \times 20 \times 25$

$= 5000$

Similarly, we find the value for whole table of M

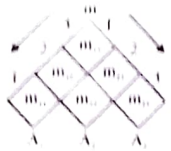


Fig. General structure of table

Now, we will calculate min m for m_{13}

$m_{13} = m_{11} + m_{23} + d_0 d_1 d_3$

$= 0 + 5000 + (15 \times 10 \times 25)$

$= 8750$

At each stage of parenthesization s_2 calculate the minimum scalar multiplication and is added to obtain the final matrix value.

Fig. Table after filling values of m.

Q.9 ***Solve the following optimal merge pattern problem using greedy approach 5, 4, 7, 2, 9, 11, 4, 8*** ***[R.T.U. 2015]***

Ans. Greedy method to solve optimal merge pattern problem:

Step 1 : Sort the files in increasing order of length.

Step 2 : Merge first two files, replace them with resultant file in list.

Step 3 : Repeat from step/till list has only one file.

Step 4 : Exit.

Given : 5, 4, 7, 2, 9, 11, 4, 8

Step 1 : Sorting array : 2, 4, 4, 5, 7, 8, 9, 11
Merge first two : 2 + 4 = 6, 4, 5, 7, 8, 9, 11

Step 2 : Sorting array : 4, 5, 6, 7, 8, 9, 11
Merge first two : 9, 6, 7, 8, 9, 11

Step 3 : Sorting array : 6, 7, 8, 9, 9, 11
Merge first two : 13, 8, 9, 9, 11

Step 4 : Sorting array : 8, 9, 9, 11, 13
Merge first two : 17, 9, 11, 13

Step 5 : Sorting array : 9, 11, 13, 17
Merge first two : 20, 13, 17

Step 6 : Sorting array : 13, 17, 20
Merge first two : 30, 20
Merge the Last two : 50
Total no of operations : 6 + 9 + 13 + 17 + 20 + 30 + 50 = 145

Q.10 ***Consider a knapsack of capacity 10 and items prices as (40, 30, 20, 50) and weight (5, 4, 6, 3). What is the maximum profit that can be earned if fractional items are allowed.*** ***[R.T.U. 2015]***

Ans. v = (40, 30, 20, 50)
w = (5, 4, 6, 3)
Capacity = 10
Number of items, n = 4
Initializing, x = {0, 0, 0, 0,}, Profit = 0

$s = \frac{v_i}{w_i} = \left\{\frac{40}{5}, \frac{30}{4}, \frac{20}{6}, \frac{50}{3}\right\}$

= {8, 7.5, 3.33, 16.66}
Arranging in descending order
s = {16.66, 8, 7.5, 3.3}
According v = {20, 40, 30, 50}
w = {6, 5, 4, 3}
or i = 1, check w[i] <= M
6 <= 10 yes.
x[i] = 1, M = M – w[i] = 10 – 6 = 4
for i = 2, w[2] <= M
5 <= 4, no.
Iteration stops
Check, is i <= n,
2 <= 4, yes

$x[i] = \frac{M}{w[i]} = \frac{4}{5} = 0.8$

M = 0
Hence, vector is [1, 0.8, 0, 0]
Total profit = 20 × 1 + 40 × 0.8 + 0 + 0
= 20 + 32 = 52

Q.11 ***Compare dynamic programming and divide and conquer approach.*** ***[R.T.U. 2015]***

OR

What is the difference between divide and conquer and dynamic programming method? Explain with example. ***[R.T.U. 2014]***

Ans. Difference between divide and conquer and dynamic programming method

1. Divide and conquer work by dividing problem into sub-problems, conquer each sub problem recursively and combine these solutions.

Dynamic programming is a technique for solving problems with overlapping subproblems. Each sub-problem is stored only once and the result of each sub-problem is stored in a table (generally implemented as an array or a hash table) for future references. These sub solutions may be used to obtain the original solution and the technique of storing the sub-problem solution is known as memorization.

Dynamic programming = recursion + reuse

2. In divide and conquer, the sub problems are independent of each other while in case of Dynamic programming, the sub-problems are not independent of each other (Solution of one sub-problem may be required to solve another sub-problem)
3. Divide and conquer does more work on the sub-problems and hence consumes more time. Dynamic programming solve the sub-problems only once and then stores it in the table.
4. Example of divide and conquer is merge short, binary search. Example of dynamic programming is matrix chain multiplication.

Example : A classical example of divide and conquer is the merge sort algorithm, where you divide the array into two halves (disjoint halves so they are independent) then sort and merge them. At each level of division of the array, the parent array is divided into two disjoint children arrays, so the elements in any division never overlap, or the arrays to be stored never overlap.

For the dynamic programming example, a classic example would be a little tedious to understand, so we would rather put up a minimal but simplistic one.

Consider the N^{th} term of the fibonacci sequence given by the relation : f(n) = f(n – 1) + f(n – 2) with [f(1) = 0, f(2) = 1]. In order to compute recursively, f(4) the algorithm break the problem into two sub-problems namely f(3) and f(2). Now we know that f(2) = 1, so this result is stored in a table. Further, in order to recursively compute f(3) = f(2) + f(1), again f(2) is required to be computed, an overlapping sub-problem. This time instead of computing it again, the result is looked from the table and returned.

Q.12 ***Find optimal solution to the knapsack (0,1) instance n = 7, m = 15, (p1 p2,...p7) = (10, 5, 15, 7, 6, 18, 3) and (w1 w2,...w7) = (2, 3, 5, 7, 1, 4, 1)*** ***[R.T.U. 2013]***

Ans. $n = 7$

$m = 15$

$(p_1, p_2, p_3 \ldots p_7) = (10, 5, 15, 7, 6, 18, 3)$

$(w_1, w_2, w_3 \ldots w_7) = (2, 3, 5, 7, 1, 4, 1)$

Optimal solution using knapsack (0, 1) = ?

Step 1 : Find $\frac{p_i}{w_i}$ for all instance $i = 1$ to 7

$$\frac{p_1}{w_1} = \frac{10}{2} = 5$$

$$\frac{p_2}{w_2} = \frac{5}{3} = 1.66$$

$$\frac{p_3}{w_3} = \frac{15}{5} = 3$$

$$\frac{p_4}{w_4} = \frac{7}{7} = 1$$

$$\frac{p_5}{w_5} = \frac{6}{1} = 6$$

$$\frac{p_6}{w_6} = \frac{18}{4} = 4.5$$

$$\frac{p_7}{w_7} = \frac{3}{1} = 3$$

Step 2 : Arrange in decreasing order

6	5	4,5	3	3	1.6	1	P_i/W_i (in decreasing order)
1	2	4	5	1	3	7	Weight

Step 3 : Include items in above order (generated in step2) till now capacity < = m

Weight		Profit
1	P_7	3
5	P_2	15
4	P_6	18
2	P_1	10
1	P_5	6

Capacity utilized = 1 + 2 + 4 + 5 + 1 = 13

Total $\text{profit}_{max} = 6 + 10 + 18 + 15 + 3 = 52$

M after capacity utilized = 15 – (1 + 2 + 4 + 5 + 1) = 2

Q.13 ***Let w = {4, 6, 7, 8} and m = 18. Find all possible subsets of w that sum to m. Draw the state space tree that is generated.*** *[R.T.U. 2013]*

Ans. The subset of w which makes the sum 18 are (4, 6, 8)

The solution vector in fixed-size for these solutions is (1, 1, 0, 1). The part of search tree, i.e. the actual tree for the ution vector is as shown in figure.

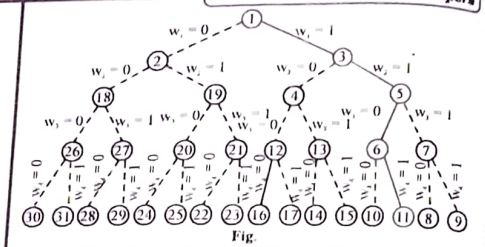


Fig.

The solution vector includes the node 1, 3, 5, 6, 11. This solution vector is shown by thin line and rest of the edges are dash line in the search tree.

Similarly, other solutions can be found out by tracing the search tree.

Q.14 ***Find the optimal merge pattern for the given values 35, 15, 20, 40, 10.*** *[R.T.U. 2012]*

Ans. (35 | 15 | 20 | 40, 10)
a[1] a[2] a[3] a[4] a[5]

Now elements a[1] and a[2] are merged to yield
(15, 35 | 20| 40, 10)
Then a[3] is merged with a [1 : 2]
(15, 20, 35 | 40, 10)
Next, elements a[1:3] and a[4:5] are merged
(10, 15, 20, 35, 40)
The algorithm is given by

```
1 merge sort (low, high)
2 // a [low: high] is a global array to be sorted.
3 // small (p) is true if there is only one element
4 // to sort. In this case the list is already sorted
5 {
6 if (low < high) then // if there are more than one element
7 {
8 // Divide p into sub problems
9 // Find where to split the set
10 mid : = [(low+high)/2],
11 // Solve the sub problems;
12 Merge sort (low, mid);
13 Merge sort (mid + 1, high);
14 // Combine the solutions.
15 Merge (low, mid, high);
16 }
17 }
```

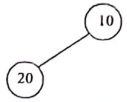
Q.15 ***Illustrate the operation of construction of min heap tree for given values 10, 20, 30, 1, 2, 3, 4, 11, 21, 31, 41.*** *[R.T.U. 2012]*

Ans. Given values : 10, 20, 30, 1, 2, 3, 4, 11, 21, 31, 41

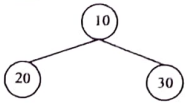
To construct the min heap steps are as follows

(i) Insert 10

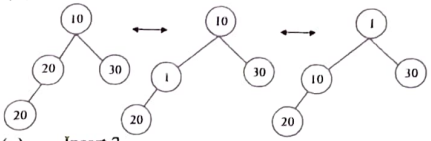
(ii) Insert 20



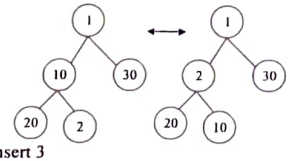
(iii) Insert 30



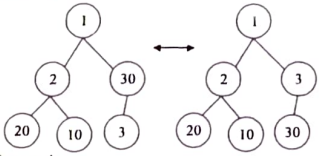
(iv) Insert 1



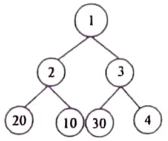
(v) Insert 2



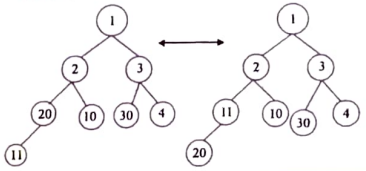
(vi) Insert 3



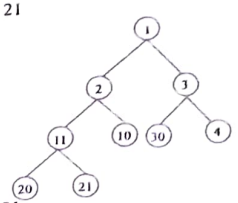
(vii) Insert 4



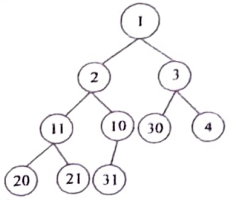
(viii) Insert 11



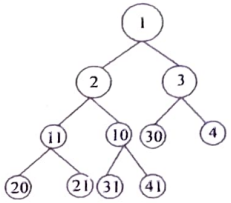
(ix) Insert 21



(x) Insert 31



(xi) Insert 41



Q.16 ***Find optimal solution for given data by Knapsack problem.***
Consider n =5, (W_1, W_2, W_3, W_4, W_5) = (5, 4, 6, 3, 1) (P_1, P_2, P_3, P_4, P_5) = (5, 2, 2, 4, 5) and M = 12. *[R.T.U. 2012]*

Ans. To solve this problem, we use some strategy to determine the fraction of weight which should be included so as to maximize the profit and fill the Knapsack.

$(x_1\ x_2\ x_3\ x_4 x_5)$	$\Sigma x_i w_i$	$\Sigma p_i x_i$
(i) $\left(\frac{1}{1}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}\right)$	3.51	10.76

Now taking maximum profit 5 with weight 5 and 1.

$x = x_5 = 1, \quad \Sigma w_i x_i < m$

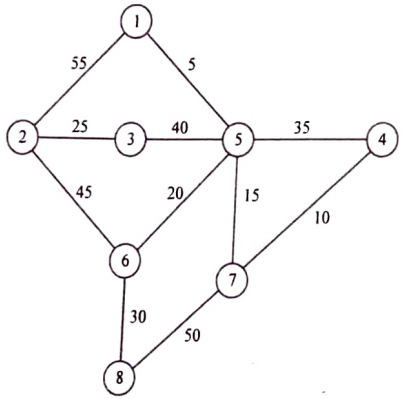
(ii) $\left(1, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, 1\right)$	5.51	20.19
(iii) $\left(1, \frac{1}{3}, \frac{1}{4}, 1, 1\right)$	8.6	29.44

(iv) $\left(1, \frac{1}{3}, 1, 1, 1\right)$ 12 36.66

At each step, we try to get maximum profit. The maximum profit we get by

(iv) $x_1=1$, $x_2=1/3$, $x_3=1$, $x_4=1$, $x_5=1$. These fractions of weight provide maximum profit.

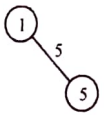
Q.17 ***Trace the Kruskal's algorithm to obtain minimum spanning tree from the graph.***



[R.T.U. 2011]

Ans. To obtain the minimum spanning tree form the graph, steps are as follows:

Step 1 : Edge with minimum weight is {1, 5}. So this edge can be added to set A.

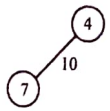


So, $A = A \cup \{1, 5\}$

Thus, at this step updated sets are

{1, 5}, {2}, {3}, {4}, {6}, {7}, {8}

Step 2 : Next edge with minimum weight is {4, 7}

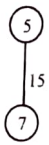


So, $A = A \cup \{4, 7\}$

Thus, at this step updated sets are

{(1, 5), (4, 7), (2), (3), (6), (8)}

Step 3 : Next edge with minimum weight is {5, 7}



So, $A = A \cup \{5, 7\}$

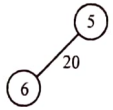
Thus, at this step updated sets are :

{(1, 5), (4, 7), (5, 7), (2), (3), (6), (8)}

So, this can also be written as

{(1, 5, 7, 4), (2), (3), (6), (8)} because their addition does not form a cycle

Step 4 : Next edge with minimum weight is {5, 6}

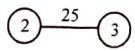


So, $A = A \cup \{5, 6\}$

Thus, at this step updated sets are :

{(1, 5), (4, 7), (5, 7), (5, 6), (2), (3), (8)}

Step 5 : Next edge with minimum weight is {2, 3}

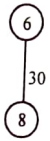


So, $A = A \cup \{2, 3\}$

Thus, at this step updated sets are :

{(1, 5), (4, 7), (5, 7), (5, 6), (2, 3), (8)}

Step 6 : Next edge with minimum weight is {6, 8}

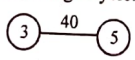


So, $A = A \cup \{6, 8\}$

Thus, at this step updated sets are :

{(1, 5), (4, 7), (5, 7), (5, 6), (2, 3), (6, 8)}

Step 7 : Next edge with minimum weight is {5, 4} but adding it will form a cycle so cant be added. Next edge {5, 3} which can be added without forming a cycle.



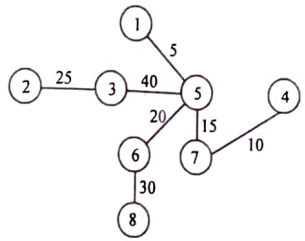
So, $A = A \cup \{5, 3\}$

Thus, at this step updated sets are

{(1, 5), (4, 7), (5,7), (5, 6), (2, 3), (6, 8), (5, 3)}

Adding all other edges namely, {2, 6}, {7, 8} and {2, 1} will result in cycle formation hence not added to set A.

So, the tree obtained is a minimum spanning tree.



$$\text{Minimum Cost} = \sum_{u,v \in T} W(T)$$

$$= 5 + 25 + 40 + 20 + 30 + 15 + 10$$

$$= 145$$

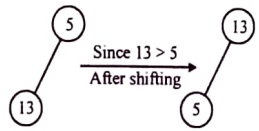
Q.18 ***Illustrate the operation of heap on following array:***

A = <5, 13, 2, 25, 7, 17, 20, 8, 4> [R.T.U. 2011]

Ans. Creation of a Maximum heap

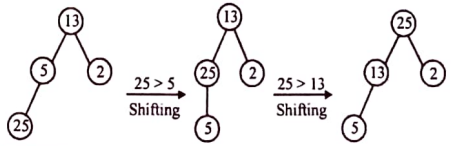
(i) First element is 5, take it as root.

(ii) Next element is 13. Make it as left child of root node and compare that whether the element inserted is smaller than the parent node or not, then this element should be shifted with the parent node.

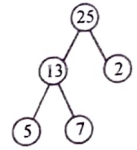


(iii) Next element is 2

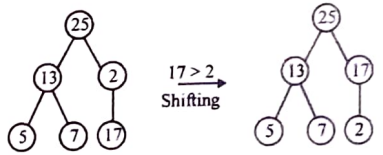
(iv) Next element is 25.



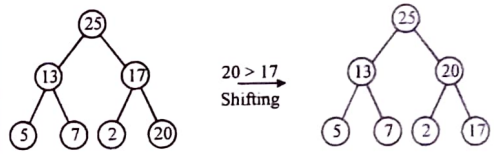
(v) Next element is 7



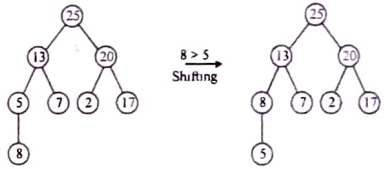
(vi) Next element is 17



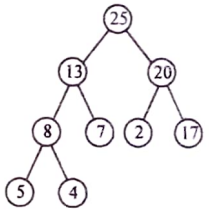
(vii) Next element is 20



(viii) Next element is 8



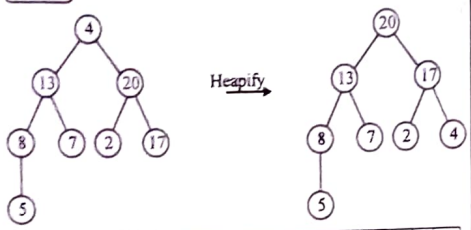
(ix) Next element is 4



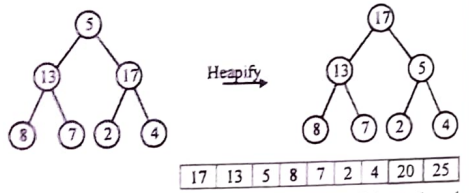
After performing maximum heap operations, the array looks like this

25	13	20	8	7	2	17	5	4

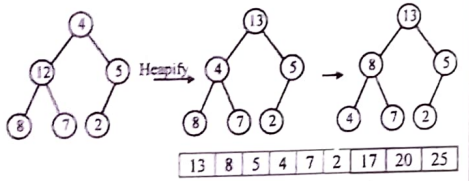
Now, we exchange the first element with the last element. Heap size decreases, it contains only 8 nodes now Heapify as root



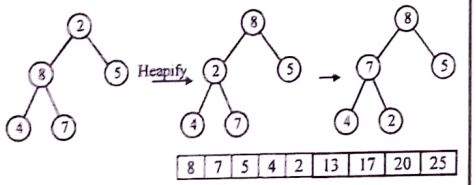
The root value floats down to node 7 and second largest element reaches the root. Remove this by exchanging with last leaf (which is now at position 8). Heap size decreases, now it is 7.



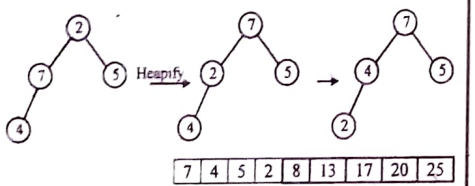
Now next we exchange node 1 with node 7. Reduced heap size to 6.



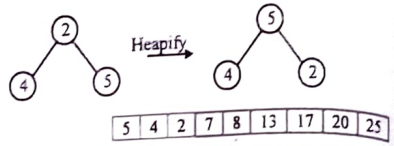
Next iteration exchange node 1 with node 6



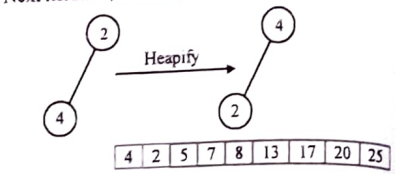
Next iteration, exchange node 1 with node 5.



Next iteration, exchange node 1 with node 4



Next iteration, exchange node 1 with node 3



In the last iteration, node 1 is exchanged with node2. It is now included in sorted array. We do not need any iteration for last node since it is already in its sorted position.

2	4	5	7	8	13	17	20	25

Q.19 ***X = <a, a, b, a, b>, Y = <b, a, b, b>. If Z is an LCS of X and Y, then find Z using dynamic programming.*** *[R.T.U. 2011]*

Ans. Here X = <a, a, b, a, b> and Y = <b, a, b, b>

m = length (X) and n = length (Y)

m = 5, n = 4

Now, filling in the m × n table with the value of c[i, j] and the appropriate arrow for the value of b[i, j]. Initialize top row and left column to 0.

Work across the row starting at the row1 and column from 1 till end.

For every box, check $x_i = y_j$

- If yes, then fill in the value equal to diagonal neighbour value + 1 and mark the box with the arrow "↖".
- If no, then compare values in the box above and the box to the left and fill in the box with the maximum. Put arrows according to from where the value is derived. If c[i – 1, j] ≥ c[i, j – 1] then b[i, j] entry is "↑" otherwise "←"

Here

$x_1 = a$ $y_1 = b$

$x_2 = a$ $y_2 = a$

$x_3 = b$ $y_3 = b$

$x_4 = a$ $y_4 = b$

$x_5 = b$

Now, fill the value of c[i, j] in m × n table.

Initially,

for i = 1 to 5, c[i, 0] = 0

for j = 0 to 4, c[0, j] = 0

That is

		0	1	2	3	4
i	j		y_1 b	a	b	b
0	x_0	0	0	0	0	0
1	a	0				
2	a	0				
3	b	0				
4	a	0				
5	b	0				

Now, for i = 1 and j = 1, we check x_1 and y_1, we get $x_1 \neq y_1$ *i.e.* a ≠ b

and $c[i-1, j] = [0, 1] = 0$

$c[i, j-1] = [1, 0] = 0$

That is $c[i-1, j] = c[i, j-1] = 0$ and b[1, 1] "↑"

Now, i = 1 and j = 2

Check x_1 and y_2, we get $x_1 = y_2$

$c[i-1, j] = [1-1, 2] - 1$

$= 0 + 1 = 1$

c[1, 2] = 1, b[1, 4] = "↖"

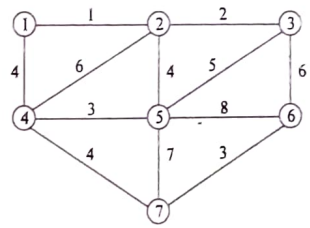
Similarly we fill all values of c[i, j] and finally we get,

		0	1	2	3	4
		y_j	b	a	b	b
0	x_i	a	0	0	0	0
1	a	0	↖1	←1	←1	
2	a	0	←0	↖2	←1	←1
3	b	0	↖1	↑1	↖2	↑3
4	a	0	↑1	↖2	↑2	←3
5	b	0	↖2	↑2	↖3	↖4

The entry 4 in c [5, 4] is the length of the Z. and the final output of Z is

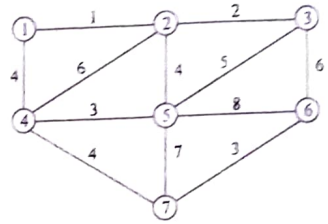
Z = < a ,b, b>

Q.20 ***Find minimum spanning tree of the following graph using Prim's and Kruskal's method.***

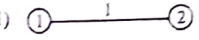


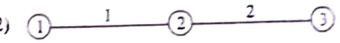
[R.T.U. 2010, Raj. Univ. 2006,2004]

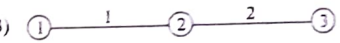
Ans. Kruskal's Method : Minimum spanning tree using Kruskal's method

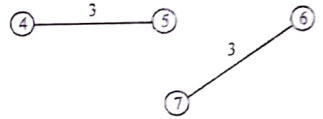


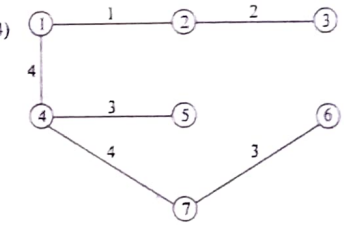
Edge is selected in such a manner that it contains a minimum weight and adding to 'M' does not includes any cycle.

(1) 

(2) 

(3) 



(4) 

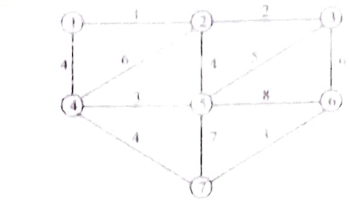
(5) We will not add next 4 weight edge 2 to 5 to 'M' because by adding this will get the closed paths.

(6) We will not add 5 weight edge 5 to 3 to 'M'.

(7) We will not add 6 weight edge 4 to 2 to 'M'.

(8) So as above we will not add 6, 7 and 8 weight edges to 'M'.

Prim's Method : We start from source node and add the edge to 'M' which is having the least weight among the edges connected to that node. And that will not make a closed path.



(1)

(2)

(3)

(4)

(5)

(6)

PART-C

Q.21 ***What is Dynamic programming? How it gives the optimal solution?***
Consider n = 3, consider M = 6, (w1, w2, w3) = (2,3,3)
(p1, p2, p3) = (1,2,4)
Find optimal solution for given knapsack problem. ***[R.T.U. 2016]***

OR

What is dynamic programming ? How it gives optimal solutions? ***[R.T.U. 2018]***

OR

Discuss Knapsack problem with respect to dynamic programming approach. Find optimal solution for given problem, w(weight set)={5, 10, 15, 20} and size of knapsack is 8. ***[R.T.U. 2017]***

OR

Consider n = 3, $(w_1, w_2, w_3) = (2, 3, 3)$, $(p_1, p_2, p_3) = (1, 2, 4)$ and m = 6. Find optimal solution for given data. ***[R.T.U. 2011]***

Ans. Dynamic programming : Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.

Dynamic programming is the most powerful design technique for optimization problems. The solutions for the dynamic programming are based on multistage optimizing decisions, on a few common elements.

Dynamic programming is closely related to divide and conquer technique, where the problem breaks down into smaller subproblems and each subproblem is solved recursively. The dynamic programming differs from divide and conquer in a way that instead of solving subproblem recursively, it solves each of the subproblem only once and stores the solution to the subproblems in a table. Later on, the solution to the main problem is obtained by these subproblem's solutions.

Optimal Solution for knapsack problem : For m = 6 and n = 3, table will contain n rows and w columns.

i will vary from to 1 to n
w will vary from 1 to m

Step 1.

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0						
2	0						
3	0						

$c[0, w] \leftarrow 0$ for w = 1 to m
$c[i, 0] \leftarrow 0$ for i = 1 to n

Step 2. For i = 1, check for each value of w
For $w = 1$, $w_1 = 2$ and $w_1 > w$
So, $c[1, 1] = c[i - 1, w] = c[0, w] = 0$
For $w = 2$, $w_1 = 2$
and $w_1 = w$, $w - w_1 = 0$
Check if $v_1 + c[0, 0] > c[0, 2]$
Or $1 + 0 > 0$
So, $c[1, 2] = v_1 + c[0, 0] = 1$
For $w = 3$, $w_1 < w$, $w - w_1 = 0$
$v_1 + c[0, 1] > c[0,3]$
$1 + 0 > 0$

So, $c[1, 3] = v_1 + c[0, 1] = 1$
For $w = 4$, $w_1 < w$, $w - w_1 = 2$
So, $c[1, 4] = v_1 + c[0, 2]$
$= 1 + 0 = 1$
For $w = 5$, $w_1 < w$, $w - w_1 = 3$
So, $c[1, 5] = v_1 + c[0, 3]$
$= 1 + 0 = 1$
For $w = 6$, $w_1 < w$, $w - w_1 = 4$
So, $c[1, 6] = v_1 + c[0, 4]$
$= 1 + 0 = 1$
So, the updated table for row i = 1 is

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1
2	0						
3	0						

Step 3. For i = 2 check for each value of w
For $w = 1$, $w_2 = 3$, $w_2 > w$, $w - w_2 = -2$
So, $c[2, 1] = c[1, 1] = 0$
For $w = 2$, $w_2 = 3$, $w_2 > w$, $w - w_2 = -1$
So, $c[2, 2] = c[1, 2] = 1$
For $w = 3$, $w_2 = w$, $w - w_2 = 0$
$v_2 + c[i - 1, w - w_2] > c[i - 1, w]$
or $2 + c[1, 0] > c[1, 3]$
$2 + 0 > 1$
So, $c[2, 3] = v_2 + c[1, 0]$
$= 2 + 0 = 2$
For $w = 4$, $w_2 < w$ and $w - w_2 = 1$
$v_2 + c[1, 1] > c[1, 4]$
or $2 + 0 > 1$
So, $c[2, 4] = v_2 + c[1, 1]$
$= 2 + 0 = 2$
For $w = 5$, $w_2 < w$ and $w - w_2 = 2$
$v_2 + c[1, 2] > c[1, 5]$
or $2 + 1 > 1$
So, $c[2, 5] = v_2 + c[1, 2]$
$= 2 + 1 = 3$
For $w = 6$, $w_2 < w$ and $w - w_2 = 3$
$v_2 + c[1, 3] > c[1, 6]$
or $2 + 1 > 1$
So, $c[2, 6] = v_2 + c[1, 3]$
$= 2 + 1 = 3$
So, the updated table for row i = 2 is

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	2	2	3	3
3	0						

Step 4. For i = 3 check for each value of w
For $w = 1$, $w_3 = 3$ and $w_3 > w$
So, $c[3, 1] = c[i - 1, w] = c[2, 1] = 0$
For $w = 2$, $w_3 > w$
So, $c[3, 2] = c[i - 1, w] = c[2, 2] = 1$
For $w = 3$, $w_3 = w$ and $w - w_3 = 0$
$v_3 + c[i - 1, w - w_3] > c[i - 1, w]$
or $v_3 + c[2, w - w_3] > c[2, w]$
or $4 + c[2, 0] > [2, 3]$
or $4 + 0 > 2$
So, $c[3, 3] = v_3 + c[2, 0]$
$= 4 + 0 = 4$
For $w = 4$, $w_3 < w$ and $w - w_3 = 1$
$v_3 + c[2, w - w_3] > c[2, w]$
or $4 + c[2, 1] > c[2, 4]$
or $4 + 0 > 2$
So, $c[3, 4] = v_3 + c[2, 1]$
$= 4 + 0 = 4$
For $w = 5$, $w_3 < w$ and $w - w_3 = 2$
$v_3 + c[2, w - w_3] > c[2, w]$
or $4 + c[2, 2] > c[2, 5]$
or $4 + 1 > 3$
So, $c[3, 5] = v_3 + c[2, 2]$
$= 4 + 1 = 5$
For $w = 6$, $w_3 < w$ and $w - w_3 = 3$
$v_3 + c[2, w - w_3] > c[2, w]$
or $4 + c[2, 3] > c[2, 6]$
or $4 + 2 > 3$
So, $c[3, 6] = v_3 + c[2, 3]$
$= 4 + 2 = 6$
The updated table of is

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	2	2	3	3
3	0	0	1	2	4	5	6

Step 5. After computing the table, all we need to do is find out the optimal solutions *i.e.*, the items to be put into knapsack.
Check starting from [3, 6]
$c[3, 6] \neq c[2, 6]$ so, item 3 is part of Knapsack.
Now check for $c[i - 1, w - w_i]$, *i.e.*,
$c[2, 6 - w_3]$ or $c[2, 3]$
we can see that $c[2, 3] \neq c[1, 3]$
So item 2 is part of Knapsack
Now, $w = w - w_2 = 3 - 3 = 0$
So, $w = 0$ and $i = i - 1 = 0$
Thus, $w = 0$ and $i = 0$ and algorithm ends.
Finally we have item 2 and 3 in the Knapsack with value = 2 + 4 = 6. This is an optimal solution.

Optimal Solution : Given weights and values of n items, we put these items in a knapsack of capacity w to get maximum total values in knapsack.

Ex. Value [] = {60, 100, 120}
Weight [] = {10, 20, 30}
w = 50
w = 10, value = 60
w = 20, value = 100
w = 30, value = 120
w = (20 + 10), value = (60 + 100) = 160
w = (30 + 10), value = (120 + 60) = 180
w = (30 + 20), value = (100 + 120) = 220
w = (10 + 20 + 30) > 50
Solution = 2 < 0

In the given problem, only weight array is given, not value array, so we cannot solve the given knapsack problem.

Q.22 ***Discuss Dynamic programming solution to longest common subsequence problem. Write an algorithm to compute an LCS of two given strings.*** ***[R.T.U. 2017]***

OR

Write the short note on Longest common subsequence. ***[R.T.U. 2010, Raj. Univ. 2004]***

OR

Write an algorithm for finding LCS.

Ans. In the **longest common subsequence problem**, we are given two sequences $X = (x_1, x_2 \ldots x_m)$ and $Y = (y_1, y_2, \ldots y_n)$ and we have to find a longest common string S. Common to both the sequence using dynamic programming.

Step 1: Characterizing a longest common subsequence : A brute-force approach to solve the LCS problem is to enumerate all subsequences of X and check each subsequence to see if it is also a subsequence of Y, keeping track of the longest subsequence found. Each subsequence of X corresponds to a subset of the indices {1, 2, ... m} of X. There are 2^m subsequences of X, so this approach requires exponential time, making it impractical for long sequences.

The LCS problem has an optimal-substructure property, as the following theorem shows. As we shall see, the natural classes of subproblems correspond to pairs of "prefixes" of the two input sequences. To be precise, given a sequence $X = (x_1, x_2, \ldots, x_m)$, we define the i^{th} prefix of X, for $i = 0, 1, \ldots m$, as $X_i = (x_1, x_2, \ldots, x_i)$. For example, if $X = (A, B, C, B, D, A, B)$, then $X_4 = (A, B, C, B)$ and X_0 is the empty sequence.

Theorem (Optimal substructure of a LCS) : Let $X = (x_1, x_2, \ldots x_m)$ and $Y = (y_1, y_2, \ldots, y_n)$ be sequences, and let $Z = (z_1, z_2, \ldots, z_k)$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is a LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is a LCS of X_{m-1} and Y.

3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is a LCS of X and Y_{n-1}.

Proof (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length k + 1, contradicting the subposition that Z is a longest common subsequence of X and Y. Thus, we must have $z_k = x_m = y_n$.

Now, the prefix Z_{k-1} is a length (k – 1) common subsequence of X_{m-1} and Y_{n-1}. We wish to show that it is LCS. Suppose for the purpose of contradiction that there is a common subsequence W of X_{m-1} and Y_{n-1} with length greater than K – 1. Then appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction.

(2) If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is LCS of X and Y.

(3) The proof is symmetric to (2).

The characterization of theorem shows that LCS of two sequences contains within it a LCS of prefixes of the two sequences. Thus, the LCS problem has an optimal-substructure property. A recursive solution also has the overlapping-subproblems property.

Step 2 : A recursive solution : Theorem implies that there are either one or two subproblems to examine when finding a LCS of $X = (x_1, x_2, \ldots, x_m)$ and $Y = (y_1, y_2, \ldots, y_n)$. If $x_m = y_n$, we must find a LCS of X_{m-1} and Y_{n-1}. Appending $x_m = y_n$ to this LCS yields LCS of X and Y. If $x_m \neq y_n$, then we must solve two subproblems: finding LCS of X_{m-1} and Y and finding LCS of X and Y_{n-1}.

Whichever of these two LCSs is longer is the LCS of X and Y. Because these cases exhaust all possibilities, we know that one of the optimal subproblem solutions must be used within LCS of X and Y.

We can readily see the overlapping-subproblems property in the LCS problem. To find LCS of X and Y, we may need to find the LCS of X and Y_{n-1} and of X_{m-1} and Y. But each of these subproblems has the subproblem of finding the LCS of X_{m-1} and Y_{n-1}.

As in the matrix-chain multiplication problem, our recursive solution to the LCS problem involves establishing a recurrence for the value of an optimal solution. Let us define c[i, j] to be the length of the LCS of the sequences X_i and Y_j. If either i = 0 or j = 0, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula.

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases} \quad (1)$$

Observe that in this recursive formulation, a condition in the problem restricts which subproblems we may consider. When $x_i = y_j$, we can and should consider the subproblem of finding the LCS of X_{i-1} and Y_{j-1}. Otherwise, we instead consider the two subproblems of finding the LCS of X_i and Y_{j-1} and of X_{i-1} and Y_j.

Finding the LCS is not the only dynamic programming algorithm that rules out subproblems based on conditions in the problem. For example, the edit-distance problem has this characteristics.

Step 3 : Computing the length of LCS : Based on equation (1), we could easily write an exponential-time recursive algorithm to compute the length of the LCS of two sequences. Since there are only $\Theta(mn)$ distinct subproblems, however, we can use dynamic programming to compute the solutions bottom up.

Procedure LCS-LENGTH takes two sequences $X = (x_1, x_2, \ldots, x_m)$ and $Y = (y_1, y_2, \ldots, y_n)$ as inputs. It stores the c[i, j] values in a table c[0, .. m, 0 .. n] whose entries are computed in row-major order. (That is, the first row of c is filled in from left to right, then the second row and so on). It also maintains the table b[1...m, 1...n] to simplify construction of an optimal solution. Intuitively, b[i, j] points to the table entry corresponding to the optimal subproblem solution chosen when computing c[i, j]. The procedure returns the b and c tables; c [m, n] contains the length of the LCS of X and Y.

```
LCS-LENGTH (X, Y)
1   m ← length[X]
2   n ← length[Y]
3   for i ← 1 to m
4       do c [i, 0] ← 0
5   for j ← 0 to n
6       do c[0, j] ← 0
7   for i ← 1 to m
8       do for j ← 1 to n
9           do if x_i = y_j
10                then c[i, j] ← c[i – 1, j – 1] + 1
11                     b[i, j] ← "↖"
12                else if c[i – 1, j] ≥ c[i, j – 1]
13                     then c [i, j] ← c[i – 1, j]
14                          b [i, j] ← "↑"
15                     else c[i, j] ← i, j – 1]
16                          b[i, j] ← "←"
17  return c and b
```

The c and b tables computed by LCS-LENGTH on the sequences X = (A, B, C, B, D, A, B) and Y = (B, D, C, A, B, A). The square in row i and column j contains the value of c[i, j] and the appropriate arrow for the value of b[i, j]. The entry 4 in c[7, 6] - the lower right-hand corner of the table is the length of an LCS {B, C, B, A} of X and Y. For i, j > 0, entry c[i, j] depends only on whether $x_i = y_j$ and the values in entries c[i – 1, j], c[i, j – 1], and c[i – 1, j – 1], which are computed before c[i, j]. To reconstruct the elements of a LCS, follow the b[i, j] arrows from the lower right-hand corner; the path is shaded. Each "↖" on the path corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

		j	1	2	3	4	5	6
			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	0↑	0↑	0↑	1↖	1←	1↖
2	B		0	1↖	1←	1←	1↑	2↖	2←
3	C		0	1↑	1↑	2↖	2←	2↑	2↑
4	B		0	1↖	1↑	2↑	2↑	3↖	3←
5	D		0	1↑	2↖	2↑	2↑	3↑	3↑
6	A		0	1↑	2↑	2↑	3↖	3↑	4↖
7	B		0	1↖	2↑	2↑	3↑	4↖	4↑

Fig.

Figure shows the tables produced by LCS-LENGTH on the sequences X = {A, B, C, B, D, A, B} and Y = {B, D, C, A, B, A}. The running time of the procedure is O(mn), since each table takes O(1) time to compute.

Step 4 : Constructing a LCS : The b table returned by LCS-LENGTH can be used to quickly construct LCS of $X = (x_1, x_2, \ldots, x_m)$ and $Y = (y_1, y_2, \ldots, y_n)$. We simply being at b [m, n] and trace through the table following the arrows. Whenever we encounter a "↖" in entry b[i, j], it implies that $x_i = y_j$ is an element of the LCS. The elements of the LCS are encountered in reverse order by this method. The following recursive procedure prints out LCS of X and Y in the proper, forward order. The initial invocation is PRINT-LCS (b, X, length[X], length[Y]).

```
PRINT-LCS(b, X, i, j)
1   if i = 0 or j = 0
2       then return
3   if b[i, j] = "↖"
4       then PRINT-LCS(b, X, i – 1, j – 1)
5            print x_i
6   elseif b[i, j] = "↑"
7       then PRINT-LCS(b, X, i – 1, j)
8   else PRINT-LCS(b, X, i, j – 1)
```

For the table in this procedure prints "BCBA." The procedure takes time O(m + n), since at least one of i and j is decremented in each stage of the recursion.

Q.23 ***Explain Prim's algorithm for finding minimum spanning tree.*** ***[R.T.U. 2016]***

OR

Find minimum cost spanning tree by implementing prim's algorithm for given weighted graph.

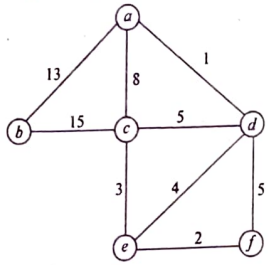


Fig. ***[R.T.U. 2014]***

Ans. Minimum Spanning Tree

The MST problem is to find a tree T of a given graph G that contain all the vertices of G and has the minimum total weight of the edges of G overall such tree.

Prim's algorithms

Prim_MST (G, V, r)

// Given a graph G , vertex set V and root vertex r

// Uses array key and π

Step 1 : For each vertex v in V;

Step 2 : Key [v] : = ∞; // set key to initial value;

Step 3 : Π [v] : = null; // set parent of every vertex to NULL;

Step 4 : End for;

Step 5 : Key [r] : = 0; // key of root vertex is zero;

Step 6 : Make a priority queue Q of all vertices;

Step 7 : Select vertex with minimum key field

u : = min(Q);

Step 8 : For each vertex v adjacent to u;

Step 9 : If $u \in Q$ and w(u, v) < key [v];

Step 10 : Π [v] : = u;

Step 11 : Key [v] : = w(u, v);

Step 12 : End if;

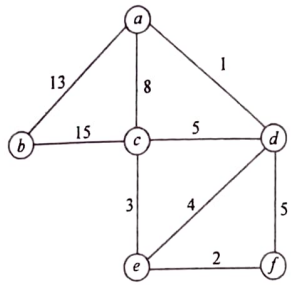
Step 13 : End for;

Step 14 : If Q not empty, goto Step 4;

Step 15 : Exit.

Working Rules for Prim's Algorithm

1. Start with an vertex u (assumed).
2. Now select another vertex such that edge is created from u and v and is of minimum weight then connect uv and add it to set of vertex v.
3. Now among the set of all vertices find other vertex V, that is not included such that (v_i, v_j) is minimum labeled and add it to V.
4. Continue the above steps till we get a MST.



Step 1: Start with a vertex, here for the given graph we will start from vertex a.

(a) Starting vertex

Step 2: Now, select another vertex v such that edge is created from u and v and is of minimum weight

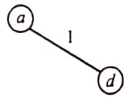
$u = a$

$v = b$ then weight =13

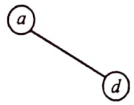
$v = c$ then weight =8

$v = d$ then weight = 1

that is we get minimum weight 1, therfore connect uv and add it to set of vertex v.



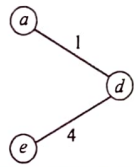
Step 3: Among the set of all vertices find other vertex v_i that is not included such that (v_i, v_j) is minimum labeled and add it to v.



$\min(ac, dc, de, df)$

$= \min(8, 5, 4, 5)$

$= 4 = de$

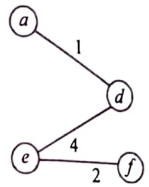


Step 4: Continue till we get MST.

$\min(dc, df, ef, ec)$

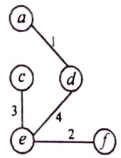
$\min(5, 5, 2, 3)$

$= 2 = ef$

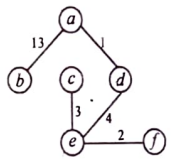


As we have seen that the above tree does not have all connected vertices so we have to move further to get minimum edge,

Step 5: In that stage we have a option that e → c have minimum cost so



Step 6 : As we have seen in the above tree, the edge b is not connected to that tree so we connect a → b as if we reached the edge b, a to b have minimum cost so the resultant MST is



Hence minimum cost is = 13 + 1 + 3 + 4 + 2 = 23

Q.24 ***Find the optimal parenthesization of multiplication of a matrix chain specified by (4, 16, 10, 8, 20). Show all tables and decision steps involved.*** ***[R.T.U. 2015]***

Ans. Given : (4, 16, 10, 8, 20)

Let $P_0 = 4$

$P_1 = 16$

$P_2 = 10$

$P_3 = 8$

$P_4 = 20$

For $m_{ij} = \{m(i, k) + m(k + 1, j) + P_{i-1}, P_k P_3\}$ for $i \le k < j$

Now,

$1 \le k < 2$

$m_{12} = m[1, 1] + m[2, 2] + P_0P_1P_2$

$= 0 + 0 + (4 \times 16 \times 10)$

$= 640$

$2 \le k < 3$

$m_{23} = m[2, 2] + m[3, 3] + P_1P_2P_3$

$= 0 + 0 + (16 \times 10 \times 8)$

$= 1280$

$3 \le k < 4$

$m_{34} = m[3, 3] + m[4, 4] + P_2P_3P_4$

$= 0 + 0 + (10 \times 8 \times 20)$

$= 1600$

Now, $1 \le k < 3$

$k = 1 \quad m_{13} = m[1, 1] + m[2, 3] + P_0P_1P_3$

$= 0 + 1280 + (4 \times 16 \times 8)$

$= 1280 + 512$

$= 1792$

$k = 2 \quad m_{13} = m[1, 2] + m[3, 3] + P_0P_2P_3$

$= 640 + 0 + (4 \times 10 \times 8)$

$= 640 + 320$

$= 960$

Now, $2 \le k < 4$

$k = 2 \quad m_{24} = m[2, 2] + m[3, 4] + P_1P_2P_4$

$= 0 + 160 + (16 \times 10 \times 20)$

$= 160 + 3200$

$= 3360$

$k = 3 \quad m_{24} = m[2, 3] + m[4, 4] + P_1P_3P_4$

$= 1280 + 0 + (16 \times 8 \times 20)$

$= 1280 + 0 + 2560$

$= 3840$

Now, $1 \le k < 4$

$k = 1 \quad m_{14} = m[1, 1] + m[2, 4] + P_0P_1P_4$

$= 0 + 3360 + (4 \times 16 \times 20)$

$= 3360 + 1280$

$= 4640$

$k = 2 \quad m_{14} = m[1, 2] + m[3, 4] + P_0P_2P_4$

$= 640 + 160 + (4 \times 10 \times 20)$

$= 640 + 160 + 800$

$= 1600$

$k = 3 \quad m_{14} = m[1, 3] + m[4, 4] + P_0P_3P_4$

$= 960 + 0 + (4 \times 8 \times 20)$

$= 960 + 0 + 640$

$= 1600$

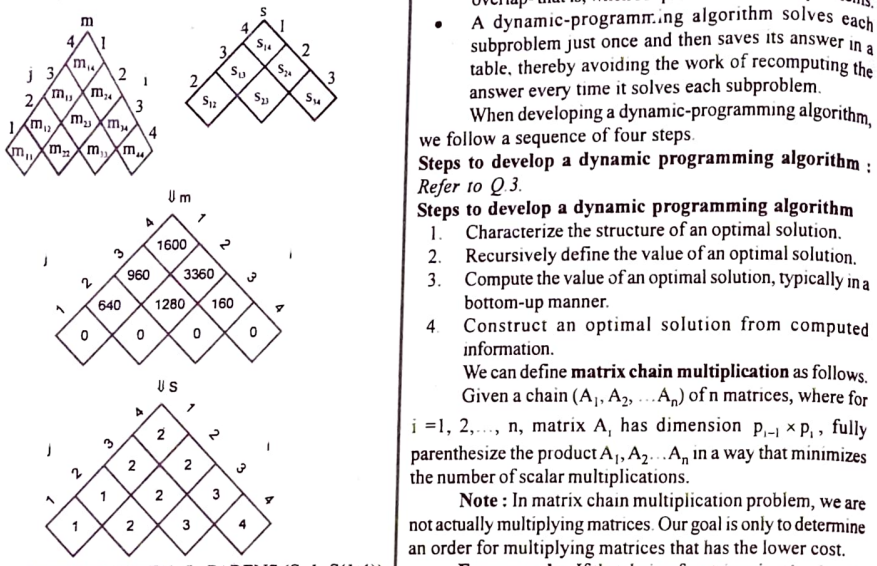
Now, initially i = 1, j = 4

Since we have all the values of auxilary table M and S, we have final optimal parenthesiation.

Now if i = j then m[i, j] = 0

so $m_{11} = m_{22} = m_{33} = m_{44} = 0$

Auxiliary table m and s need to be constructed



So printf ("PRINT_OPTIMAL_PARENS (S, 1, S(1,4)) PRINT_OPTIMAL_PARENS (S, S(1, 4)+1)Printf)"

So,

$\Rightarrow$ ((S, 1, 2) (S, 3, 4))

$\Rightarrow$ ((S, 1, 1) (S, 2, 2)) ((S, 3, 3) (S, 4, 4))

$((A_1, A_2)(A_3, A_4))$

Q.25 ***When and how dynamic programming approach is applicable? Discuss matrix chain multiplication with reference to dynamic programming technique and also write a subroutine for matrix chain multiplication and apply it on the following array.***

30	1	40	10	25

[R.T.U. 2013]

Ans. Dynamic programming typically applies to optimization problems in which we make a set of choice in order to arrive at an optimal solution.

Dynamic programming is effective when a given subproblem may arise from more than one partial set of choices;

- The key technique is to store the solution to each such subproblem in case it should reappear.
- Dynamic programming, like the divide and conquer method, solves problems by combining the solutions to subproblems.
- Dynamic programming applies when the subproblems overlap- that is, when subproblems share subproblems.
- A dynamic-programming algorithm solves each subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subproblem.

When developing a dynamic-programming algorithm, we follow a sequence of four steps.

Steps to develop a dynamic programming algorithm : *Refer to Q.3.*

Steps to develop a dynamic programming algorithm

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up manner.
4. Construct an optimal solution from computed information.

We can define **matrix chain multiplication** as follows.

Given a chain $(A_1, A_2, \ldots A_n)$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1, A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

Note : In matrix chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lower cost.

For example : If the chain of matrices is $<A_1, A_2, A_3, A_4>$, then we can fully parenthesize the product A_1, A_2, A_3, A_4 in five distinct ways:

$(A_1(A_2(A_3A_4)))$
$(A_1((A_2A_3)A_4))$
$((A_1A_2)(A_3A_4))$
$((A_1(A_2A_3))A_4)$
$(((A_1A_2)A_3)A_4)$

To illustrate the different cost incurred by different parenthesization of a matrix product,

Consider the problem of a chain $<A_1, A_2, A_3>$ of three matrices.

Suppose dimentions of the matrices are

10×100 for A_1
100×5 for A_2
5×50 for A_3

Now

$((A_1A_2)A_3)$

$10.100.5 = 5000$ (A_1A_2)

$10.5.50 = 2500$ $((A_1A_2)A_3)$

$5000 + 2500 = 7500$ Scaler multiplications

$(A_1(A_2A_3)$

$100.5.50 = 2500$ (A_2A_3)

$10.100.50 = 50000$ $((A_1(A_2A_3)$

$2500 + 50000 = 75000$ Scaler multiplications

Thus, Computing the product according to the first parenthesization is 10 times faster.

Counting the number of parenthesizations

$$p(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} p(k)\, p(n-k) & \text{if } n \geq 2 \end{cases}$$

Applying dynamic programming : We shall use the dynamic-programming method to determine how to optimally parenthesize a matrix chain using four steps :

Refer to Steps to develop a dynamic programming algorithm

Subroutine for matrix chain multiplication

Algo-matrix-chain order (p)

1. $n \leftarrow$ length [p] -1
2. Let m [1..n, 1..n] be new tables
3. For i$\leftarrow$1 to n
4. m [i,j]=0
5. for $l \leftarrow 2$ to $n = l$ is the chain length
   (i) do for $i \leftarrow 1$ to $n - l + 1$
   (ii) Set $j = i + l - 1$
   (iii) m [i, j] $= \infty$
6. for k$\leftarrow$i to j$-$1
   (i) Set q$\leftarrow$m [i, k]+m[k+1, j]+$p_{i-1}p_kp_j$
   (ii) if q < m[i, j]
   (iii) m[i, j] $\leftarrow$ q then
   (iv) s[i, j] $\leftarrow$ k
   (v) return m and s

Algo-print-parenthesize (s, i, j)

1. if i = j
2. print "A_i"
3. else
4. print "("
5. Print_optimal_parenthesis (s, i, s [i, j])
6. Print_optimal_parenthesis (s, s[i, j]+1, j)
7. Print ")"

Now we have to apply it on given array,

30	1	40	10	25

$p_0 = 30$, $p_1 = 1$, $p_2 = 40$, $p_3 = 10$, $p_4 = 25$

The length of matrix chain is length [P]=5.

So, n=length [p]$-$1=4, and matrices are $A_1 = 30 \times 1$, $A_2 = 1 \times 40$, $A_3 = 40 \times 10$, $A_4 = 10 \times 25$

Our matrix is

	0	1	2	3
0	0			
1		0		
2			0	
3				0

Step 1 : Auxiliary table m and s need to be constructed

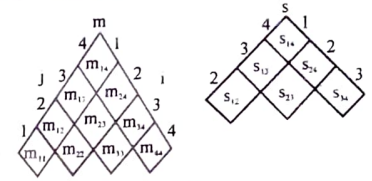


Fig.

We are considering the table only for above the main diagonal for [i ≤ j], so only that part of table m and s need to be constructed following the algorithm matrix chain order:

If i = j, m[i, j] = 0

So, $m_{11}, m_{22}, m_{33}, m_{44} = 0$

Step 2 : In all other cases in table we can see i<j. (For example m_{12}, where i=1 and j=2).

So, initially m[i, j]= ∞ for i < j

Now, we have to calculate minimum value of m[i, j] and put those value in table m

$$m_{12} = \min_{1 \leq k < j} \{m[i,k] + m[k+1, j] + p_{i-1}p_kp_j\}$$

Here $1 \leq k < 2$

So, the only value of k=1 that satisfies $1 \leq k < 2$

So, $m_{12} = m[1, 1] + m[2, 2] + p_0\, p_1\, p_2$

$\Rightarrow$ $0 + 0 + 30 \times 1 \times 40$

$= 1200$

So, m[1, 2]=1200 and s[1, 2]=k=1

Similarly, m_{23}=m[2, 2]+m[3, 3]+ $p_1p_2p_3$

$\Rightarrow$ 0+0+0×40×10

=400

Where the only value of k=2 which satisfies $2 \leq k < 3$

So, m[2, 3]= 400

And s[2, 3]= k = 2

And m_{34}= m[3, 3]+m[4, 4]+ $p_2p_3p_4$

$\Rightarrow$ $0 + 0 + 40 \times 10 \times 25$

$= 10000$

Where, the value of k=3

So, m[3, 4]=10000

And s[3, 4]=k=3

Step 3 : For the value of m_{13} we have to select minimum of 2 values for $i \leq k < j$ or $1 \leq k < 3$. Actually there are two values of k which satisfies $1 \leq k < 3$; k=1 and k=2.

So, $$m[1,3] = \min_{1 \leq k < 3} \{m[i,k] + m[k+1, j] + p_{i-1}p_kp_j\}$$

For k = 1

$m[1, 3] = m[1, 1] + m[2, 3] + p_0p_1p_3$

$= 0 + 400 + 30 \times 1 \times 10$

$= 700$

For k = 2

$m[1, 3] = m[1, 2] + m[3, 3] + p_0p_2p_3$

$= 1200+0+30\times40\times10$
$= 1200+12000$
$= 13200$

If we compare 13200 and 700, we can easily see that m[1, 3] is minimum for k=1

So, m[1, 3]=700
s[1, 3]=k=1

Similarly, $m[2, 4] = \min_{2\le k<4}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}$

For k = 2,
$m[2, 4] = m[2, 2]+m[3, 4]+p_1p_2p_4$
$= 0+10000+1\times40\times25$
$= 10000+1000$
$= 11000$

For k = 3,
$m[2, 4] = m[2, 3]+m[4, 4]+ p_1p_3p_4$
$= 400+0+1\times10\times25$
$= 650$

[2, 4] is minimum for k=3

So, m[2, 4]=650
s[2, 4]=k=3

Step 4 : Next we have the values of m_{14} and there are three possible values for k which satisfies $1 \le k < 4$; k=1, 2 and 3

$m_{14} = \min_{1\le k<4}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}$

For k=1,
$m[1, 4] = m[1, 1]+m[2, 4]+p_0p_1p_4$
$= 0 + 650 + 30 \times 1 \times 25$
$= 1400$

For k = 2,
$m[1, 4] = m[1, 2]+m[3, 4]+p_0p_2p_4$
$= 1200+10000+30\times40\times25 = 41200$

For k = 3,
$m[1, 4] = m[1, 3]+m[4, 4]+ p_0p_3p_4$
$= 700+0+30\times10\times25$
$= 8200$

m[1, 4] is minimum for k=1

so, m[1, 4] = 1400
s[1, 4] = k = 1

Step 5 : Since we have all the value of auxiliary tables m and s, we have find optimal parenthesization.

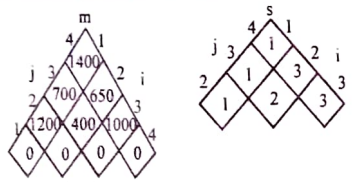


Fig.

Step 6 : Using "PRINT_OPTIMAL_PARENS" method and output, the tabels. We have to find optimal parenthesization.

Initially value of i=1 and j=4

So, Print "("
PRINT_OPTIMAL_PARENS (s, 1, s[1, 4])
PRINT_OPTIMAL_PARENS (s, s[1, 4]+1, 4)
PRINT ")"

So, we have parenthesization (by putting the value from s-table in above given formula).

$\Rightarrow$ ((s, 1, 1) (s, 2, 4)
$\Rightarrow$ ((A,)(s, 2, 3)(s, 4, 4))
$\Rightarrow$ ((A,)(s, 2, 2)(s, 3, 3)(s, 4, 4))
$\Rightarrow$ $\boxed{((A_1)(A_2A_3)(A_4))}$

Q.26 ***Define how Knapsack problem is solved by using dynamic programming approach?*** ***[R.T.U. 2011]***

OR

What is dynamic programming ? What is principle of optimality ? Explain knapsack problem using dynamic programming. ***[Raj. Univ. 2006]***

Ans. Dynamic Programming : *Refer to Q.21*

Principle of Optimality : The principle of optimality states that for solving the problem optimally its subproblem should be solved optimally. Not all the times each subproblem is solved optimally.

One way to solve problems for which it is not possible to make a sequence of stepwise decisions leading to an optimal decision sequence is to try all possible decision sequences. We could enumerate all decision sequences and then pick out the best. But the time and space requirements may be prohibitive. Dynamic programming often drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences, that cannot possibly be optimal. In dynamic programming an optimal sequence of decisions is obtained by making explicit appeal to the principle of optimality.

Knapsack Problem : Given a list of items, we have to pack the Knapsack (bag) in such a manner so as to get the maximum (optimum) total value. Each item of the list has associated weight and value. It should be noted that total weight that we can carry is not more than the fixed weight W. So in order to pack the Knapsack we have to consider the weight of item as well as its value. We assume that all the weights and values are positive integers.

$w = <w_1, w_2, w_3 \ldots\ldots w_n>$

$v = <v_1, v_2, v_3 \ldots\ldots v_n>$.

The **Knapsack** problem has two variants :

- **0/1 Knapsack Problem**

In this problem the list of items are indivisible; that means either we take the item or discard it. We will discuss this problem using DP approach.

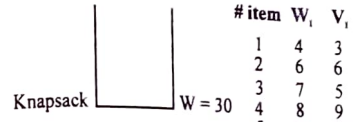
- **Fractional Knapsack Problem**

In this, the list of items are divisible; that means we can take any fraction of an item. We describe this problem using Greedy approach.

0/1 Knapsack Problem

The 0/1 Knapsack problem is stated as follows : Given a set of n items and a Knapsack having capacity w, such that each item has some weight w_i and value v_i then the problem is to pack the Knapsack in such a manner so that a maximum total value is achieved.

The problem is said to be 0/1 Knapsack problem because the item from the list is either rejected or accepted.



# item	W_i	V_i
1	4	3
2	6	6
3	7	5
4	8	9
5	10	12

Fig. : Knapsack with w = 30

Simple Approach

It can be seen that there can be 2^n possible combination of items as there are 'n' items in a set. One method is to find all the possible combinations and select the best one in such a manner which will provide maximum value and weight less than or equal to Knapsack capacity, w. Well, it can be observed that for this the running time complexity will be $O(2^n)$.

Dynamic Programming Approach

Again according to DP approach we have to break the problem into several (simple) smaller subproblems. For this consider that the items in the given set 'S' are labelled from 1 to n, then a subproblem would be to obtain an optimal solution for

S_k = <item labelled 1, 2, 3, ..., k>

The question arises whether we can obtain the final soltuion (S_n) in terms of the subproblem (S_k), as described above or not. The answer is 'no', this is illustrated below :

Maximum weight for Knapsack : w = 30

For S_4 :

Total weight : 25 <w_1, w_2, w_3, w_4>

Maximum value : 23 <v_1, v_2, v_3, v_4>

For S_5 :

Total weight : 30 <w_1, w_2, w_3, w_4>

Maximum value : 32 <v_2, v_3, v_4, v_5>

It can be observed that solution for S_4 is not the part of the solution for S_5. So, we include an additional parameter, 'W' which will signify the exact weight for each subset of the items. For this subproblem we have to compute $K_p[k, w]$. From the above explaination we can formulate the subproblems as recursive rule :

$$K_p[k, w] = \begin{cases} K_p[k-1, w] & \text{if } w_k > w_i \\ \max(K_p[k-1, w], K_p[k-1, w-w_k]+v_k), & \text{otherwise} \end{cases}$$

From the above recursive formula we can depict that the best subset of S_k that has the total weight ω, may either contain item K or not. Following two cases arise in this manner :

Case I : When $w_k > W$

In this case item 'k' cannot become the part of the solution, as the weight is greater than 'W' which is unacceptable.

Case II : When $w_k \le W$

In this case the item k can become the part of the solution, and the item 'k' is chosen such that it has the maximum value.

The algorithm for the 0/1 Knapsack problem is given below :

Algorithm 0/1 Knapsack

The above algorithm solves the 0/1 Knapsack problem. Given a Knapsack of weight capacity 'W', and a set of items with weight 'w_i' and value 'v_i'. This algorithm determines the packing of the Knapsack in such a manner that a maximum value is achieved and the respective weight is less than or equal to W.

Step 1 : Loop, initialization

for $w \leftarrow 0$ to $W \Rightarrow$ till the total capacity
set $K_p[0, w] \leftarrow 0$

Step 2 : Loop

for $w \leftarrow 0$ to $n \Rightarrow$ set of items
set $K_p[i, 0] \leftarrow 0$

Step 3 : Loop, checking 'K' whether part of solution or not

for $i \leftarrow 1$ to n
for $w \leftarrow 0$ to W
if $(w_i <= w)$ then $\Rightarrow$ item 'i' can be the part of solution
if $(v_i + K_p [i - 1, w - w_i] > K_p[i - 1, w])$ then
set $K_p[i, w] \leftarrow v_i + K_p[i - 1, w - w_i]$
else
set $K_p[i, w] \leftarrow K_p[i - 1, w]$
else set $K_p[i, w] \leftarrow K_p[i - 1, w] \Rightarrow w_i > w$

Step 4 : Finished

Exit.

Example : Consider a Knapsack having weight capacity w = 3 and number of items are three such that,

$S = 3$
$w_i = <1, 2, 3>$
$v_i = <2, 3, 4>$

Ans. On applying algorithm **0/1 Knapsack**, we have

# item	w_i	v_i	Max weight = 3
1	1	2	
2	2	3	
3	3	4	

Initially

i/w	0	1	2	3
0				
1				
2				
3				

⇓

i/w	0	1	2	3
0	0	0	0	0
1				
2				
3				

for w ← 0 to W
set K_p [0, w] ← 0

⇓

i/w	0	1	2	3
0	0	0	0	0
1	0			
2	0			
3	0			

for i ← 1 to n
set K_p [i, 0] ← 0

i/w	0	1	2	3
0	0	0	0	0
1	0	2		
2	0			
3	0			

#item	w_i	v_i
1	1	2
2	2	3
3	3	4

i = 1
v_i = 2
w_i = 1
w = 1
w − w_i = 0

if ($w_i \le w$) code

⇓

i/w	0	1	2	3
0	0	0	0	0
1	0	2	2	
2	0			
3	0			

i = 1
v_i = 2
w_i = 1
w = 2
w − w_i = 1

if ($w_i \le w$) code

⇓

i/w	0	1	2	3
0	0	0	0	0
1	0	2	2	2
2	0			
3	0			

i = 1
v_i = 2
w_i = 1
w = 3
w − w_i = 2

if ($w_i \le w$) code

⇓

i/w	0	1	2	3
0	0	0	0	0
1	0	2	2	2
2	0	2		
3	0			

i = 2
v_i = 3
w_i = 2
w = 1
w − w_i = −1

if ($w_i > w$) code

⇓

i/w	0	1	2	3
0	0	0	0	0
1	0	2	2	2
2	0	2	3	
3	0			

i = 2
v_i = 3
w_i = 2
w = 2
w − w_i = 0

if ($w_i \le w$) code

⇓

i/w	0	1	2	3
0	0	0	0	0
1	0	2	2	2
2	0	2	3	5
3	0			

i = 2
v_i = 3
w_i = 2
w = 3
w − w_i = 1

if ($w_i \le w$) code

⇓

i/w	0	1	2	3
0	0	0	0	0
1	0	2	2	2
2	0	2	3	5
3	0	0		

i = 3
v_i = 4
w_i = 3
w = 2
w − w_i = −2

if ($w_i > w$) code

⇓

i/w	0	1	2	3
0	0	0	0	0
1	0	2	2	2
2	0	2	3	5
3	0	0	2	

i = 3
v_i = 4
w_i = 3
w = 2
w − w_i = −1

if ($w_i > w$) code

⇓

i/w	0	1	2	3
0	0	0	0	0
1	0	2	2	2
2	0	2	3	5
3	0	0	2	5

i = 3
v_i = 4
w_i = 3
w = 3
w − w_i = 0

The algorithm presented above only computes the maximum possible value that can be taken in the Knapsack *i.e.*, the value in K_p[n, w]. To choose the items that take part in making this maximum value we have to perform some more steps :

Steps for obtaining actual Knapsack items.

- The table has all the information that we require.
- The maximum value that we can place in the Knapsack is given by K_p[n, w].
- Consider i = n and k = w, then following condition arises.

Whether K_p[i, k] is equal to K_p[i − 1, k] or not. If the condition is true then we have to mark the i^{th} item as in the Knapsack. The detailed algorithm for finding the actual Knapsack items is given below.

Algorithm FindKnapsack

The above algorithm finds the actual Knapsack item. 'K_p' stores the final solution as the maximum value and the weight of the item which is less than or equal to w.

Step 1 : Initialization

set i ← n.

set k ← w.

Step 2 : Loop, checking 'K' as a part of the Knapsack

while (i > 0 and k > 0)

if (K_p [i, k] ≠ K_p [i − 1, k]) then

mark the i^{th} item as in the Knapsack

set i ← i − 1

set k ← k − w_i

else

set i ← i − 1

Step 3 : Finished

Exit.

For finding the actual Knapsack items we are running **Findknapsack** Algorithm on the previous example :

i/w	0	1	2	3
0	0	0	0	0
1	0	2	2	2
2	0	2	3	5
3	0	0	2	5

i = 3
k = 3
v_i = 4
w_i = 3
K_p[i, k] = 5
K_p[i − 1, k] = 5

i ← i − 1

⇓

i/w	0	1	2	3
0	0	0	0	0
1	0	2	2	2
②	0	2	3	5
3	0	0	2	5

i = 2
k = 3
v_i = 3
w_i = 2
K_p[i, k] = 5
K_p[i − 1, k] = 2

i ← i − 1

⇓

i/w	0	1	2	3
0	0	0	0	0
①	0	2	2	2
②	0	2	3	5
3	0	0	2	5

k − w_i = 1
k = k − w_i
i = 1
k = 1
v_i = 2
w_i = 1
K_p[i, k] = 2
K_p[i − 1, k] = 0
k − w_i = 0
k = k − w_i
i = 0
k = 0

⇓

i/w	0	1	2	3
0	0	0	0	0
①	0	2	2	2
②	0	2	3	5
3	0	0	2	5

The optimal Knapsack contains
S = {1, 2}
w_i = {1, 2 } = {3} ≤ W
c_i = {2, 3} = {5}. Optimum value.

Q.27 ***What is the Greedy approach? Does it always give optimal solution? Give two examples in which Greedy algorithm gives optimal solution.***

[R.T.U. 2010, Raj. Univ. 1997]

Ans. Greedy algorithms make good local choices in the hope that they result in an optimal solution :

- They result in feasible solutions.
- Not necessarily an optimal solution.
- A proof is needed to show that the algorithm finds an optimal solution.

A counter example shows that the greedy algorithm does not provide an optimal solution.

Knapsack Problem : The knapsack problem is a problem in combinatorial optimization. It derives its name from the maximization problem of choosing possible essentials that can fit into one bag (of maximum weight) to be carried on a trip. Given a set of items, each with a cost and a value, then determine the number of each item to include in a collection so that the total cost is less than some given cost and the total value is as large as possible.

The different kinds of knapsack problems are :

Consider the problem

In the following, we have n kinds of items, x_1 through x_n. Each item x_j has a value p_j and a weight w_j. The maximum weight that we can carry in the bag is C.

0-1 knapsack problem : The 0-1 knapsack problem restricts the number of each kind of item to zero or one.

Mathematically the 0-1knapsack problem can be formulated as :

Fractional knapsack problem

maximize $\sum_{i=1}^{n} V_i x_i$

subject to $\sum_{i=1}^{n} W_i x_i \le W$

Where W_i and x_i can take any value between 0 and 1.

Although the problems are similar but the Greedy approach does not guarantee to give the optimal solution in case of 0-1 knapsack problem ,it can be a sub-optimal solution.

The greedy approach guarantees to give an optimal solution to the fractional knapsack problem which we will be seeing.

There are three approaches to the fractional knapsack problem by using greedy strategies :

(i) Greedy by Profit : At each step select from the remaining items the one with the highest profit (provided the capacity of the knapsack is not exceeded). This approach tries to maximize the profit by choosing the most profitable items first.

(ii) Greedy by Weight : At each step select from the remaining items the one with the least weight (provided the capacity of the knapsack is not exceeded). This approach tries to maximize the profit by putting as many items into the knapsack as possible.

(iii) Greedy by Profit Density : At each step select from the remaining items the one with the largest profit density, (provided the capacity of the knapsack is not exceeded). This approach tries to maximize the profit by choosing items with the largest profit per unit of weight.

The first two approaches also does not necessarily gives an optimal solution to the problem. But the 3rd approach gives the optimal solution .

We apply the greedy method to solve the knapsack problem. We are given n objects and a knapsack or bag. Object i has a weight w_i and the knapsack has a capacity m.

If a fraction x_i, $0 \le x_i \le 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m. Formally, the problem can be stated as

maximize $\sum_{1 \le i \le n} p_i x_i$ ... (1)

subject to $\sum_{1 \le i \le n} w_i x_i \le m$ ... (2)

and $0 \le x_i \le 1, 1 \le i \le n$ ... (3)

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying (2) and (3) above. An optimal solution is a feasible solution for which (1) is maximized.

Example : Consider the following instance of the knapsack problem :

$n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are :

(x_1, x_2, x_3)	$\Sigma w_i x_i$	$\Sigma p_i x_i$
1. (1/2, 1/3, 1/4)	16.5	24.25
2. (1, 2/15, 0)	20	28.2
3. (0, 2/3, 1)	20	31
4. (0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit.

When one applies the greedy method to the solution of the knapsack problem; there are at least three different measures one can attempt to optimize when determining which object to include next. These measures are total profit, capacity used, and the ratio of accumulated profit to capacity used. Once an optimization measure has been chosen

```
1   Algorithm GreedyKnapsack (m, n)
2   // p[1 : n] and w [1 : n] contain the profits and weights
    respectively
3   // of the n objects ordered such that p [i] / w[i] ≥ p
    [i + 1] / w [i + 1].
4   // m is the knapsack size and x [1 : n ] is the solution
    vector.
5   {
6   for i = 1 to n do x[i] : = 0.0; // Initialize x.
7   U = m,
8   for i  = 1 to n do
9   {
10  if (w [i] > U) then break;
11  x[i] = 1.0; U = U – w [i];
12  }
13  if (i ≤ n) then x[i] = U / w [i];
14  }
```

Algorithm : Algorithm for greedy strategies for the knapsack problem method suggests choosing objects for inclusion into the solution in such a way that each choice optimizes the measure at that time. Thus, a greedy method using profit as its measure will at each step choose an object that increases the profit the most. If the capacity measure is used, the next object included will increase this the least. Although greedy-based algorithms using the first two measures do not guarantee optimal solutions for the knapsack problem.

❑❑❑

3 Branch and Bound & Pattern Matching Algorithms

Chapter In A Nutshell

❑ **KMP Matcher**

The working of KMP algorithm is similar to the finite automation matches; in this, pattern string P, and text string T both are scanned from left to right, and are compared. If any mismatch occurs then the algorithm searches the largest suffix of the "wrong start" which is also a prefix of a pattern P. Thus, it determines the shifting of the pattern P for possible match.

The KMP algorithm has a linear running time of $O(n + m)$, which is achieved by using the auxiliary function P_i(prefix function), pre-processed from the pattern P in time O(m).

The function P computes the shifting of pattern matches within itself.

If for a given pattern P, a mismatch is detected at some position j then we know that we have already matched $j - 1$ characters successfully. By this idea, we can decide the possible shifts, so that we can restart our matching.

Consider the string "10100" which is mismatched at 5th character, then we already know that the text so far matched consists of "1010X", where X is not known. Thus, we can restart matching at the 3rd character which is '1' against the unknown character 'X'. Given a pattern string P, each string position j can be pre-processed. Thus, a position can be obtained from where to restart the matching when a mismatch occurs immediately after j^{th} position. It can be accomplished in the following manner.

- From the original string consider a substring 0..j.
- Starting at the end of the substring, "slide" it along the original string.
- We have a potential restart position for a mismatch after j^{th} position, where all overlapping character matches.
- Take an auxiliary array - "next", and store that largest potential restart position for j.

❑ **Boyer Moore Algorithms :** The Boyer-Moore-algorithm can give substantially faster searches for the long strings.

The basic idea behind the algorithm is that the pattern is scanned from left-to-right when proceeding through the text, which is also referred to as "looking glass heuristic".

The strategy for BM algorithm is :

- For finding the right most mismatch, it starts searching at the right of the pattern.
- It uses information about the possible alphabet of the text; as well as the characters in the pattern.

For determining the smallest possible shifts, BM works with two different preprocessing strategies each time when a mismatch occurs the algorithm.

❑ **Bad Character Heuristic :** As the name suggests it concentrates on the "bad character" in the text where the mismatch has occurred. If that character is not contained in P, then the pattern is shifted after "bad character" in the string for further matching. But if the "bad character" is somewhere in the pattern then we search for the right most appearance of the "bad character" in the pattern and match it against the text.

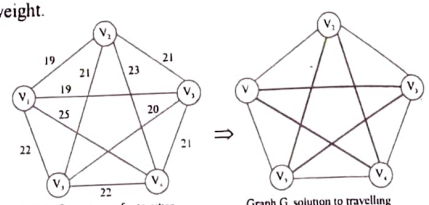
❑ **Good Suffix Heuristic :** As the name symbolises, it works for the suffix, suppose for a given pattern P if we have already matched some suffix S, but there is a mismatch with the preceding character of x then, we shift the pattern to right along the string so that the matched port is occupied by the same suffix 'S'. It is to be noted that no complete match of the suffix 'S' is possible if 'S' does not occur elsewhere in P. In this case, we have to match the largest prefix of P.

❑ **Travelling Salespersons Problem**

In this problem, salesperson needs to visit 'n' cities in such a manner that all the cities much be visited only once and in the end he returns to the city from where he started, with minimum cost.

Given a group of cities $C = (C_1, C_2, C_3, ... C_n)$, where cost i, j denotes the cost of travelling from city C_i to city C_j. The travelling salesperson problem is to find a route starting and ending at C_i that will take in all the cities with the minimum cost. According to the greedy strategy :

- Start with any arbitrary city, say C_1.
- Choose the minimum cost city from C_1.
- At any step i we will be left with n – i cities.
- The method is repeated until all the cities are visited and at every step we have to select the city with the minimum weight.



Graph G, vertices refer to cities, edges refer to links between cities and weights refer to cost of the links

Graph G, solution to travelling sales person problem according to greedy strategy. Dark lines are showing the path

PREVIOUS YEARS QUESTIONS

PART-A

Q.1 ***What is Travelling salespersons problem.***

Ans. In this problem, salesperson needs to visit 'n' cities in such a manner that all the cities much be visited only once and in the end he returns to the city from where he started, with minimum cost.

Q.2 ***Explain the working of KMP matcher.***

Ans. The working of KMP algorithm is similar to the finite automation matches; in this, pattern string P, and text string T both are scanned from left to right, and are compared. If any mismatch occurs then the algorithm searches the largest suffix of the "wrong start" which is also a prefix of a pattern P. Thus, it determines the shifting of the pattern P for possible match.

Q.3 ***What is the basic idea behind of Boyer Moore Algorithm.***

Ans. The basic idea behind the algorithm is that the pattern is scanned from left-to-right when proceeding through the text, which is also referred to as "looking glass heuristic".

Q.4 ***Write the strategy for Boyer Moore Algorithm.***

Ans.

- For finding the right most mismatch, it starts searching at the right of the pattern.
- It uses information about the possible alphabet of the text; as well as the characters in the pattern.

Q.5 ***What do you mean by bad character heuristic.***

Ans. As the name suggests it concentrates on the "bad character" in the text where the mismatch has occurred. If that character is not contained in P, then the pattern is shifted after "bad character" in the string for further matching. But if the "bad character" is somewhere in the pattern then we search for the right most appearance of the "bad character" in the pattern and match it against the text.

PART-B

Q.6 ***Using Rabin Karp algorithm to solve the feet T = 2359023141526739921 and P = 31415 and modulo q = 13, m = 5.*** ***[R.T.U. 2013, 2011]***

OR

Given the text T = {2, 3, 5, 9, 0, 2, 3, 1, 4, 1, 5, 2, 6, 7, 3, 9, 9, 2, 1} and P = {3, 1, 4, 1, 5} and modulo q = 13, m = 5. Choose the pattern matching with average case complexity and explain the search process. Justify the answer for choosing such algorithm. ***[R.T.U. 2018]***

OR

Explain Rabin Karp method with suitable example. Also give the algorithm for the same. ***[R.T.U. 2010]***

Ans. The Rabin Karp algorithm involves both the steps of pattern matching, *i.e.*, preprocessing and matching. For better explanation, we assume that string character contain

$\Sigma = \{0, 1, ...9\}$ thus, each character is a decimal digit. Our process start with the calculation of decimal value for the pattern and the sub string of given text.

For the given pattern P [1... m], p denotes the decimal value and for the given test T [1 ...n], t_s denotes the decimal value for length m substring T [S + 1... S + m] where, $0 \le S \le n - m$.

For the given pattern, S is valid shift if and only if $p = t_s$ which means,

$$P [1....m] = T [S + 1 .... S + m]$$

p is commuted time θ (m) and t_s values are computed in time θ (n – m + 1). So we compare p with each value of t_s and determine valid shift s in time $[\theta (m) + \theta (n - m + 1)] = \theta (n)$.

p is calculated using Horner's rule as:

$p = P[m] + 10(P[m - 1] + 10 (P[m - 2] + ... 10(P[2] + 10 P [1])....))$

Similarly, t_{s+1} is computed using t_s as:

$$t_{s+1} = 10(t_s - 10^{m-1}T [S + 1]) + T [S + m + 1]$$

t_{s+1} calculation shifts the pattern by one digit.

Subtracting the term $10^{m-1}T [S + 1]$ removes the higher order digit from t_s and multiplying it by 10, shifts it one position towards left. Adding the term T [S + m + 1] brings lower order digit to the number.

Pattern P = <3, 1, 4, 1, 5>

So, m = length [P] = 5

We have to check if T [S + 1 ..... S + m] = P [1... m]

Where, $0 \le S \le n - m$

Step 1

p = P and q = 31415 mod 13 = 7

For, S = 0, t_{s+1} = ?

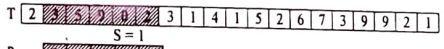
There is no shift initially so,

t_s = 23590 mod 13

= 8 ≠ 7

So, pattern unmatched and shift is applied.

Step 2



$$t_{s+1} = (d(t_s - T [S + 1]h) + T [S + m + 1]) \bmod 13$$

T[S + 1] is higher-order digit for older T[S + 1] and T [S + m + 1] is the new lower-order digit.

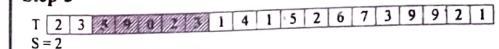
h = 10000 for this example.

So, for S = 1, $t_{s+1} = t_2 = (10(23590 - 2 \times 10000) + 2) \bmod 13$

= 9 ≠ 7

So, pattern unmatched and shift is applied.

Step 3



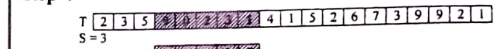
For S = 2, $t_{s+1} = t_3 = (10(35902 - 3 \times 10000) + 3) \bmod 13$

= 59023 mod 13

= 3 ≠ 7

Pattern unmatched and shift is applied.

Step 4



For S =3, $t_{s+1} = t_4 = (10(59023 - 5 \times 10000) + 1) \bmod 13$

= 90231 mod 13

= 11 ≠ 7

Pattern unmatched and shift is applied.

Step 5

For S = 4, $t_{s+1} = t_5 = (10(90231 - 9 \times 10000) + 4) \bmod 13$

= 2314 mod 13

= 0 ≠ 7

Pattern unmatched and shift is applied.

Step 6

For S = 5, $t_{s+1} = t_6 = (10(2314 - 0 \times 10000) + 1) \bmod 13$

= 23141 mod 13

= 1 ≠ 7

Pattern unmatched and shift is applied.

Step 7

For S = 6, $t_{s+1} = t_7 = (10(23141 - 2 \times 10000) + 5) \bmod 13$

= 31415 mod 13

= 7

Here, P = t_{s+1}

Pattern matched will shift S = 6.

Step 8

This is done till S = n – m = 19 – 5 = 14. If any other match is found, it is also a pattern match. In this, the only pattern is at S = 6.

Q.7 ***Explain and write Knuth Morris Pratt algorithm for pattern matching and also comment on its running time.*** ***[R.T.U. 2017]***

OR

Write the KMP string matching algorithm and also find the prefix function for the following pattern: a b a b b a b a a and implement the algorithm for one step. *[R.T.U. 2013]*

Ans. Knuth-Morris-Pratt-Algorithm : In Brute-Force-Algorithm, we have to backup the text pointer for every mismatch. Some characters are examined twice or thrice depending upon the shifts for the given pattern. The finite automata is used to eliminate the needless shifts, but the complicated preprocessing for the computation of transition function pays off. The working of KMP algorithm is similar to the finite automaton matches; in this, pattern string P, and text string T both are scanned from left to right, and are compared. If any mismatch occurs then the algorithm searches the largest suffix of the "wrong start" which is also a prefix of a pattern P. Thus, it determines the shifting of the pattern P for possible match.

The KMP algorithm has a linear running time of O(n + m), which is achieved by using the auxiliary function Pf(prefix function), pre-processed from the pattern P in time O(m). The function P computes the shifting of pattern matches within itself. It can be observed that if we know that how the pattern matches shifts against itself, then we can slide the pattern more characters towards right than just one character as we have seen in the Brute-Force algorithm.

```
KMP MATCHER (T, P)
1. n ← length [T]
2. m ← length [P]
3. π ← COMPUTE-PREFIX-FUNCTION(P)
4. q ← 0                    ▷ Number of characters matched.
5. for i ← 1 to n           ▷ Scan the text from left to right
6. do-while q > 0 and P[q +1] ≠ T[i]
7. do q ← π[q]              ▷ Next character does not match.
8. if P [q +1] = T[i]
9. then q ← q + 1           ▷ Next character matcher
10. if q = m                ▷ Is all of P matched ?
11. then print "Pattern occurs with shift" i – m
12. q ← π[q]                ▷ Look for the next match.
```

For instance : If for a given pattern P, a mismatch is detected at some position j then we know that we have already matched j – 1 characters successfully. By this idea we can decide the possible shifts, so that we can restart our matching.

Consider the string "10100" which is mismatched at 5th character, then we already know that the text so far matched consists of "1010X", where X is not known. Thus, we can restart matching at the 3rd character which is '1' against the unknown character 'X'.

Given a pattern string P, each string position j can be pre-processed. Thus, a position can be obtained from where to restart the matching when a mismatch occurs immediately after j^{th} position. It can be accomplished in the following manner.

- From the original string consider a substring 0..j.
- Starting at the end of the substring, "slide" it along the original string.
- We have a potential restart position for a mismatch after j^{th} position, where all overlapping character matches.
- Take an auxiliary array - "next", and store that largest potential restart position for j.

i	1	2	3	4	5	6	7	8	9
P(i)	a	b	a	b	b	a	b	a	a

Step 1. m ← length [P]
i.e., length [P] = 9, so m = 9
Step 2. π [1] ← 0 and also k ← 0
Step 3. For q ← 2 to m, i.e., q ← 2 to 9 and now q = 2
Step 4. do while k > 0 and P [k + 1] ≠ P [q], i.e., 0 > 0 and P [1] ≠ P [2] (false)
Step 5. It will not execute.
Step 6. If P [k + 1] = P [q], i.e., P [1] = P [2] (false)
Step 7. It will not execute
Step 8. π [q] ← k, i.e., π [2] ← 0
Again Step 3. Now q = 3
Step 4. 0 > 0 and P [1] ≠ P [3] (false)
Step 6. If P [1] = P [3] (true)
Step 7. then k ← k + 1, i.e., k ← 1
Step 8. π [3] ← 1
Again Step 3. Now q = 4
Step 4. 1 > 0 and P [2] ≠ P [4] (false)
Step 6. If P [2] = P [4] (true)
Step 7. then k ← k + 1, i.e., k ← 2
Step 8. π [4] ← 2
Again Step 3. Now q = 5
Step 4. 2 > 0 and P [3] ≠ P [5] (true)
Step 5. then k ← 0
Step 8. π [5] ← 0
Again Step 3. Now q = 6
Step 4. 0 > 0 and P [1] ≠ P [6] (false)
Step 6. If P [1] = P [6] (true)
Step 7. then k ← k + 1, i.e., k ← 1
Step 8. π [6] ← 1
Again Step 3. Now q = 7
Step 4. 1 > 0 and P [2] ≠ P [7] (false)
Step 6. If P [2] = P [7] (true)
Step 7. then k ← k + 1, i.e., k ← 2
Step 8. π [7] ← 2
Again Step 3. Now q = 8
Step 4. 2 > 0 and P [3] ≠ P [8] (false)
Step 6. If P [3] = P [8] (true)
Step 7. then k ← k + 1, i.e., k ← 3
Step 8. π [8] ← 3

Again Step 3. Now q = 9
Step 4. 3 > 0 and P [4] ≠ P [9]
Step 5. then k ← 1 (true)
Step 8. π [9] ← 1

Thus, the final table is

i	1	2	3	4	5	6	7	8	9
P(i)	a	b	a	b	b	a	b	a	a
π(i)	0	0	1	2	0	1	2	3	1

Running Time : The algorithm compute_prefix_function has running time of θ(m). The matching time for KMP-matcher is θ(n).

Q.8 ***Let p = rrllrrll be a pattern and T = lrrrlrrllllrrrlrrlrrllrlrrlrrrlr be a text in a string matching problem :***
(i) How many shifts (both valid and invalid) will be made by the Naive string matching algorithm?
(ii) Provide the algorithm to compute the transition function for a string matching automation.
(iii) Find out the state transition diagram for the automation to accept the pattern P given above. *[R.T.U. 2017]*

Ans. p = rrllrrll
T = lrrrlrrllllrrrlrrlrrllrlrrlrrrlr
(i) Total shift = T(n) – p(m) + 1
= 30 – 8 + 1
= 23
Valid Shift = 0
Inward Shift = 23

(ii) The algorithm is to get the next state from the current state for every possible character.

Given a character x and a state k, we can get the next state by considering the string "patt [0 .... k–1]x" which is basically concarnation of pattern characters patt[0], patt[1], ..., patt[k – 1] and the character x. The idea is to get larger of the longest prefix of given pattern such that the prefix is also suffix of "patt [0 .... k -1]x". The value of length gives us next length.

(iii)

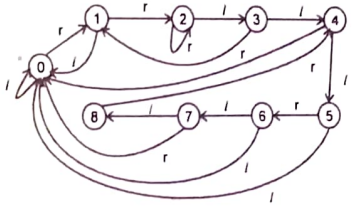


Fig.

Q.9 ***Suggest an approximation algorithm for traveling sales person problems using minimum spanning tree algorithm. Assume that the cost function satisfies the triangle inequality.*** *[R.T.U. 2016]*

Ans. If for the set of vertices a,b,c, ∈ V, it is true that t (a,c) ≤ t(a, b) + t(b, c) where t is the cost function, we say that t satisfies the triangle inequality.

First create a minimum spanning tree the weight of which is a lower bound on the cost of an optimal traveling salesman tour. Using this minimum spanning tree let us create a tour the cost of which is at most 2 times the weight of the spanning tree.

Approximation – Traveling Sales Person Problem

Input: A complete graph G (V. E)
Output: A Hamiltonian cycle

1. Select a "root" vertex r ∈ V [G].
2. Use MST-Prim (G, c, r) to compute a minimum spanning tree from r.
3. Assume L to be the sequence of vertices visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H that visits the vertices in the order L.

The next set of figures show the working of the proposed algorithm.

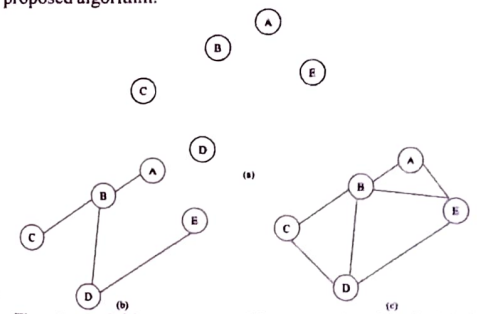


Fig. : A set of cities and the resulting connection after the MST-Prim algorithm has been applied

In figure (a) a set of vertices is shown. Part (b) illustrates the result of the MST-Prim thus the minimum spanning tree MST-Prim constructs. The vertices are visited like {A, B, C, D, E, A) by a preorder walk, part (c) shows the tour, which is returned by the complete algorithm.

Now, assume H* to be an optimal tour for a set of vertices. A spanning tree is constructed by deleting edges from a tour. Thus, an optimal tour has more weight than the minimum-spanning tree, which means that the weight of the minimum spanning tree forms a lower bound on the weight of an optimal tour.

$$c(t) \le c(H^*) \quad \ldots(1)$$

...l walk of T be the complete list of vertices ...are visited regardless if they are visited for the ... or not. The full walk is W.

W = A, B, C, B, D, B, E, B, A

The full walk crosses each edge exactly twice. Thus, we can write:

$$c(W) = 2c(T) \quad ...(2)$$

From equations (1) and (2) we can write that

$$c(W) \le 2c(H^*) \quad ...(3)$$

Which means that the cost of the full path is at most 2 time worse than the cost of an optimal tour. The full path visits some of the vertices twice which means it is not a tour. We can now use the triangle inequality to erase some visits without increasing the cost. The fact we are going to use is that if a vertex a is deleted from the full path if it lies between two visits to b and c the result suggests going from b to c directly.

We are left with the tour: A, B, C, D, E, A. This tour is the same as the one we get by a preorder walk. Considering this preorder walk let H be a cycle deriving from this walk. Each vertex is visited once so it is a Hamiltonian cycle. We have derived H deleting edges from the full walk so we can write:

$$c(H) \le c(W) \quad ...(4)$$

From (3) and (4) we can imply:

$$c(H) \le 2\, c(H^*) \quad ...(5)$$

This last inequality completes the proof.

Q.10 ***State lower bound theory.***

[R.T.U. 2015, Raj. Univ. 2008, 2007, 2006,2005]

Ans. Lower Bound Theory : If two algorithms for solving the same problem are discovered and their times differed by any order of magnitude, then the one with the smaller order is generally regarded as superior. A function g(n) that is a lower bound on the time that any algorithm must take to solve the given problem. If we have an algorithm whose computing time is the same order as g(n), then we know that asymptotically we cannot do anything better.

If f(n) is the time for some algorithm, then we write $f(n) = \Omega(g(n))$ to mean that g(n) is a lower bound for f(n). Formally this equation can be written, if there exist positive constant c and n_0 such that $|f(n)| \ge c|g(n)|$ for all $n > n_0$. In addition to developing lower bounds to within a constant factor, we are also concerned with determining more exact bounds whenever this is possible.

Deriving good lower bounds is often more difficult than devising efficient algorithms. Perhaps this is because a lower bound states a fact about all possible algorithms for solving a problem. Usually, we cannot enumerate and analyze all these algorithms, so lower bound proofs are often hard to obtain.

However, for many problems it is possible to easily observe that a lower bound identical to n exists, where n is the number of inputs (or possibly outputs) to the problem. For example, consider all algorithms that find the maximum of an unordered set of n integers. Clearly every integer must be examined at least once, so $\Omega(n)$ is a lower bound for any algorithm that solves this problem. Or, suppose we wish to find an algorithm that efficiently multiplies two n × n matrices. Then $\Omega(n^2)$ is a lower bound on any such algorithm since there are $2n^2$ inputs that must be examined and n^2 outputs that must be computed. Bounds such as these are often referred to as trivial lower bounds because they are very easy to obtain. We know how to find the maximum of n elements by an algorithm that uses only n – 1 comparisons so there is no gap between the upper and lower bounds for this problem. But for matrix multiplication the best-known algorithm requires $O(n^2 + \epsilon)$ operations ($\epsilon > 0$), and so there is no reason to believe that a better method cannot be found.

Q.11 ***Find the pattern ABCBC in the next ACABABCABCBCA using KMP matcher.***

[R.T.U. 2015]

Ans. The given string is ACABABCABCBCA

We have to find : ABCBC

Step-I

A C A B A B C A B C B C A

A B C B C

i	Fail Value	Failed at position 1, fail (1) = 0, so we can jump by (1) – (0) = 1
0	–1	
1	0	
2	0	
3	0	
4	0	

Step-II

A C A B A B C A B C B C A

A B C B C

i	Fail Value	Failed at posit.on 0, fail (0) = –1, so we can jump by (0) – (–1) = 1
0	–1	
1	0	
2	0	
3	0	
4	0	

Step-III

A C A B A B C A B C B C A

A B C B C

i	Fail Value	Failed at position 2, fail (2) = 0, so we can jump by (2) – (0) = 2
0	–1	
1	0	
2	0	
3	0	
4	0	

Step-IV

A C A B A B C A B C B C A

A B C **B** C

i	Fail Value	Failed at position 3, fail (3) = 0, so we can jump by (3) – (0) = 3
0	–1	
1	0	
2	0	
3	0	
4	0	

Step-V

A C A B A B C A B C B C A

A B C B C

i	Fail Value	Match Found
0	–1	
1	0	
2	0	
3	0	
4	0	

String found.

Q.12 ***Solve the Travelling Salesman Problem (TSP) for the following graph by using the branch and bound algorithm, the tour must be start from vertex 1 and generate only tour in which 2 is visited before 3.***

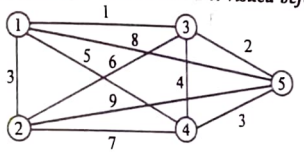


Fig.

[R.T.U. 2013]

Ans. Step 1 : First we have to make the cost matrix from the given graph. Rows and columns denotes the number of cities in the graph.

From city →	To city → 1	2	3	4	5
1	∞	3	1	5	8
2	∞	∞	6	7	9
3	∞	∞	∞	4	2
4	∞	∞	∞	∞	3
5	∞	∞	∞	∞	∞

Cost of going from city i to city i is ∞

edge (i, i) is ∞

The edges (1, 1), (2, 2), (3, 3) (4, 4) and (5, 5) all is ∞.

So, weight of other edges are as provided in the cost matrix.

Step 2 : Now, we have to reduced cost matrix, R. This is done by selecting the smallest element from each row and column and subtracting it from all other elements.

The aim is to have at least one zero in each row and column. The other entries of the matrix must be non-negative. Reduced cost matrix is constructed to get better minimum cost value.

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

⇒

	1	2	3	4	5
1	∞	2	0	4	7
2	∞	∞	0	1	3
3	∞	∞	∞	2	0
4	∞	∞	∞	∞	0
5	∞	∞	∞	∞	∞

–2 –1

⇒

	1	2	3	4	5
1	∞	0	0	3	7
2	∞	∞	0	0	3
3	∞	∞	∞	1	0
4	∞	∞	∞	∞	0
5	∞	∞	∞	∞	∞

Now every column and rows are at least 1 zero.

Total amount of subtracted in the process of finding reduced cost matrix = 1 + 6 + 2 + 3 + 2 + 1 = 15

This total amount will become the root of the search tree corresponding to given graph.

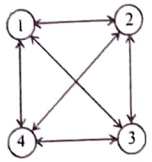
Step 3 : If 1 is the root node of parent node and 2 is child node there is an edge between the two nodes as (1, 2). Change all entries in row 1 and column 2 of the reduced cost matrix to ∞. This prevents the use of any more edges leaving vertex 1 or entering vertex 2.

Q.13 ***Solve the TSP problem for the following cost matrix***

	x	y	z	w
x	A	10	15	20
y	5	A	9	10
z	6	13	A	12
w	8	8	9	A

[R.T.U. 2012]

Ans.



$g(2, \phi) = c_{21} = 5$

$g(3, \phi) = c_{31} = 6$

$g(4, \phi) = c_{41} = 8$

$\therefore\ g(2, \{3\}) = c_{23} + g(3, \phi) = 15$

$g(2, \{4\}) = 8$

$g(3, \{2\}) = 18$

$g(3, \{4\}) = 20$

$g(4, \{2\}) = 13$

$g(4, \{3\}) = 15$

Next, we compute g(i, s) with $|s| = 2$, $i \neq 1$, $i \notin s$, and $i \notin s$.

$$g(2,\{3,4\}) = \min\{c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\}\} = 25$$

$$g(3,\{2,4\}) = \min\{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\} = 23$$

$$g(4,\{2,3\}) = \min\{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\} = 23$$

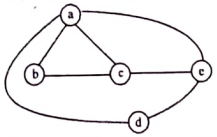
Finally

$$g(1,\{2,3,4\}) = \min\{c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\})\}$$

$$= \min\{35, 40, 43)$$

$$= 35$$

Q.14 ***Find a Hamiltonian circuit using backtracking method for the following graph G = (V, E)***



[R.T.U. 2012]

Ans. For finding Hamiltonian circuit Backtracking algorithm is used.

First algorithm gives next vertex and second algorithm gives all Hamiltonian circuit.

Algorithm Next Value (k)

```
// x [ 1:k – 1] is a path of k–1 distinct vertices.
// If x [k] = 0, then no vertex has as yet been
// assigned to x [k]. After execution, x [k] is
// assigned to the next highest numbered vertex
// which does not already appear in x [1 : k–1]
// and is connected by an edge to x [k–1]
// Other wise x [k] = 0. If k = n, then in addition
// x [k] is connected to x [1].
{
  repeat
    {
      x [k]: = (x[k] + 1) mod (n + 1); // Next vertex
      if (x [k] = 0) then return;
      if (G [x[k–1], x [k]] ≠ 0) then
      { // Is there an edge ?
    for j : = 1 to k–1 do if (x [j] = x [k]) then break;
                  // Check for distinctness
if (j = k) then // If true, then the vertex is distinct
if (c < n) or ((k = n) and G [x[n], x[1]] ≠ 0))
    then return;
    }
}  until (false);
}
```

Algorithm to find all Hamiltonian Cycles

```
// This algorithm uses the recursive formulation
// of backtracking to find all the Hamiltonian
// cycles of a graph. The graph is stored as an
// adjacency matrix G [1:n, 1:n]. All cycles begin
// at node 1.
{
   repeat
   { // Generate values for x[k].
   Next Value (k); // Assign a legal next value to
                  // x [k].
   if (x [k] = 0) then return;
   if (k = n) then write (x [1 : n]);
   else Hamiltonian (k + 1);
   } until (false);
}
```

According to these above algorithms the Hamiltonian circuit for the given graph are

(1) a, b, c, e, d, a

(2) b, c, a, e, d, a, b

(3) c, e, d, a, b, c

(4) e, d, a, b, c, e

(5) d, e, c, b, a, d

PART-C

Q.15 ***Solve the TSP problem having the following cost matrix using branch and bound technique.***

	A	B	C	D
A	X	5	2	3
B	4	X	2	3
C	4	2	X	3
D	7	6	8	X

[R.T.U. 2018, 2011]

Ans. According to cost matrix

	A	B	C	D
A	X	5	2	3
B	4	X	2	3
C	4	2	X	3
D	7	6	8	X

Subtract the minimum quantity from each row

Row A = Row A – 2

Row B = Row B – 2

Row C = Row C – 2

Row D = Row D – 6

The reduced matrix is now

$$RC = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & 3 & 0 & 1 \\ 2 & \infty & 0 & 1 \\ 2 & 0 & \infty & 1 \\ 1 & 0 & 2 & \infty \end{bmatrix} \end{array}$$

Still column A and D do not have any zero. So, we reduce them properly.

Column A = Col A – 1

Column D = Col D – 1

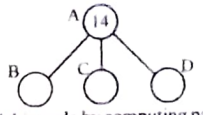
Hence, the reduced matrix is

$$RC = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & 3 & 0 & 0 \\ 1 & \infty & 0 & 0 \\ 1 & 0 & \infty & 0 \\ 0 & 0 & 2 & \infty \end{bmatrix} \end{array}$$

The lower bound is calculated as sum of all quantities subtracted.

Lower bound = 2 + 2 + 2 + 6 + 1 + 1 = 14

The tree constructed for this problem will start a root vertex of cost 14. The root node corresponds to the vertex A.



We expand this node by computing path values from A to B, C and D

Considering path A → B, we formulate a matrix FC by setting row A and column B to infinity (∞). Also, entry [B, A] = ∞, so that the path does not trace back to vertex A.

$$FC = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 0 \\ 1 & \infty & \infty & 0 \\ 0 & \infty & 2 & \infty \end{bmatrix} \end{array}$$

In these matrices either a row or column should have all values ∞ or have at least one zero. As we can see this matrix is already reduced.

Total reduced quantity, r = 0

Node value, $I_B = I_A + r + RC\ [A, B]$

$= 14 + 0 + 0$

$= 14$

Now, we can also take path A → C, for which the corresponding matrix will have row A and column C set to ∞ And the entry [C, A] = ∞, so that the path does not trace back to A.

$$FC = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 0 \\ \infty & 0 & \infty & 0 \\ 0 & 0 & \infty & \infty \end{bmatrix} \end{array}$$

This matrix is also already reduced. Hence r = 0

Node value, $I_C = I_A + r + RC\ [A, C]$

$= 14 + 0 + 0 = 14$

Considering path A → D, we have a matrix with row A and column D set to ∞ and the entry [D, A] = ∞.

$$FC = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty \\ 1 & 0 & \infty & \infty \\ \infty & 0 & 2 & \infty \end{bmatrix} \end{array}$$

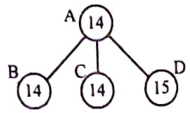
Subtract 1 from column A.

$$FC = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & 0 & \infty \\ 0 & 0 & \infty & \infty \\ \infty & 0 & 2 & \infty \end{bmatrix} \end{array}$$

Total reduced quantity, r = 1

Node value, $I_D = I_A + r + RC [A, D]$
$= 14 + 1 + 0$
$= 15$

The tree looks like,



Choosing the minimum node to expand, we select node B of value 14. Upon expansion it will lead to following paths. Taking path A → B → C we formulate the matrix FC by setting row A, row B and column C to ∞. Also the entries [C, A] and [C, B] are set to ∞. Thus,

$$FC = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ 1 & 0 & \infty & 0 \\ 0 & 0 & \infty & \infty \end{bmatrix} \end{array}$$

That is already reduced so, r = 0

Node value, $I_C = I_B + r + RC [C, B]$
$= 14 + 0 + 0 = 14$

Taking path A → B → D, we formulate the matrix FC by setting row A, row B and column D to ∞. Also the entries [D, A] and [D, B] are set to ∞.

$$FC = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ 1 & 0 & \infty & \infty \\ \infty & \infty & 2 & \infty \end{bmatrix} \end{array}$$

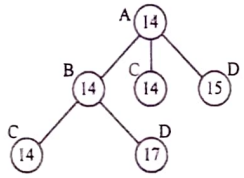
Subtract 1 from column A and 2 from column C

$$FC = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \end{bmatrix} \end{array}$$

So, $r = 1 + 2 = 3$

Node value, $I_D = I_B + r + RC [C, D]$
$= 14 + 3 + 0 = 17$

The tree now looks like,



Since, there are no different paths now to explore we have found the solution. The solution path is

$A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$

Total path cost $= C [A, B] + C [B, C] + C [C, D] + C [D, A]$
$= 5 + 2 + 3 + 7 = 17$

Note that this is same as value of last node in tree, where we found the solution.

Q.16 ***Explain Naive string matching algorithm using suitable example.*** ***[R.T.U. 2016]***

OR

Describe Naive string matching algorithm in detail? ***[R.T.U. 2018, 2012]***

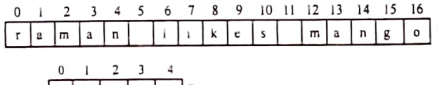
Ans. String matching algorithms are normally used in text processing. This text processing is often done during compilation of source program. In software design or in system design, text processing is an important activity. String matching means finding one or more of all the occurrences of a string in the text. These occurrences are called as *pattern*. String matching algorithms are also called as *Pattern Matching Algorithms*.

One of the pattern matching algorithm is *Naive Method*.

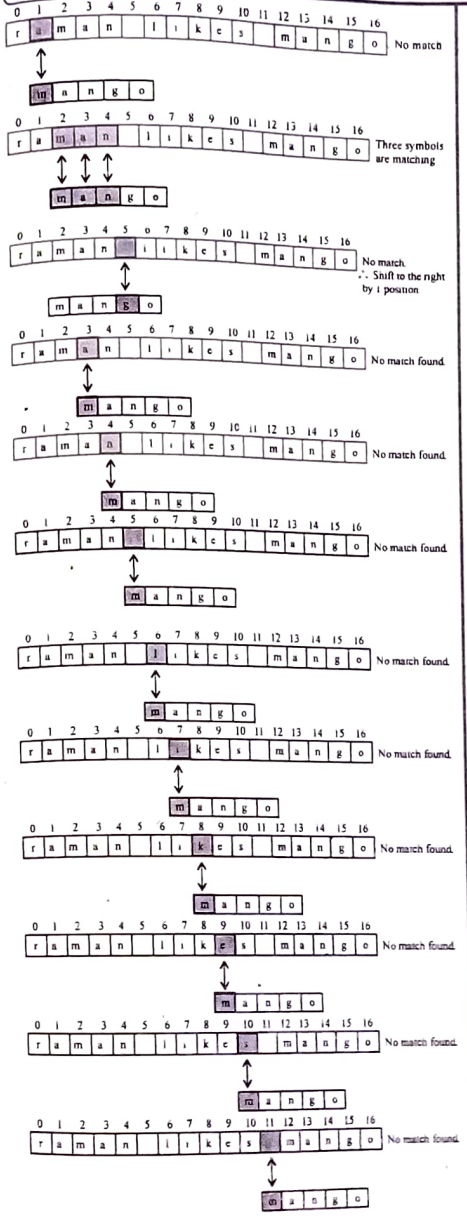
Naive Method : This is the simplest method which works using *Brute Force* approach. The *Brute Force* is a straight forward approach of solving the problem. This method has "Just do it" approach. This algorithm performs checking at all positions in the text between 0 to n–m, whether an occurrence of the pattern starts there or not. Then after each attempt, it shifts the pattern by exactly one position to the right. If the match is found then it returns otherwise the matching process is continued by shifting one character to the right. If there is no match at all in the text for the given pattern even then we have to do n comparisons. This method can be explained with the help of some example.

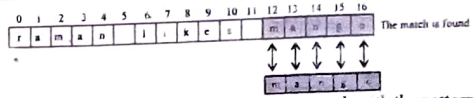
Example

Consider the text and pattern as given below :



We will start finding match for pattern from 0th location in *text*. If the match is not found then shift to the right by 1 position.





Hence return index 12, because a match with the pattern is found from that location in *text*.

Algorithm

```
Algorithm Naive (T [1 . . . . n], P [1 . . . . m])
{
// Problem Description : This algorithm finds
// the string matching using Naive method.
// Input : The array text T and pattern P
for (s ← 0 to n – m) do
{
if (P [1 . . . . . m] = T [s+1 . . . . . s+m] then
print ("pattern finding with shift", s) ;
}
} // end of algorithm
```

Analysis

In the given example, if a match is not found then shift the pattern to right by 1 position *i.e.* almost always we are shifting the pattern to the right. The worst case occurs when we have to make all the m comparisons. This results in **worst case** time complexity of **Θ (mn)**. For a typical word search in natural language the **average case** efficiency is **Θ (n)**.

Basic Notations and Terminologies

$\sum^*$: It is pronounced as 'sigma star'. This notation denotes the set of all finite length strings formed using input set $\sum$
e.g. :
$\sum = \{a\}$ then $\sum^* = \{\varepsilon, a, aa, aaa, \ldots\}$

Zero length string : It is called empty or null string. It is denoted by ε (epsilon)

$|a|$: Means length of a string a.

Concatenation : The concatenation of strings x and y is denoted by xy with length $|x|+|y|$. The concatenation means characters from x are followed by all the characters from y.

Prefix of a string : The prefix means previously occurring strings if w is prefix of some string a then it is denoted as $w[a$.

Suffix of a string : The suffix means the string that are occurring *after* particular string. It is denoted by]. If w is a suffix of some string a then it is denoted as $w]a$.

These notations are used in string operations.

Q.17 ***Explain the prefix function for a string with an example and write KMP matcher algorithm?*** ***[R.T.U. 2012]***

OR

Write short note on Prefix function for string matching. ***[R.T.U. 2018]***

Ans. Given a pattern P[1 ...m], the prefix function for the pattern P is the function

$$\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m-1\}$$

Such that

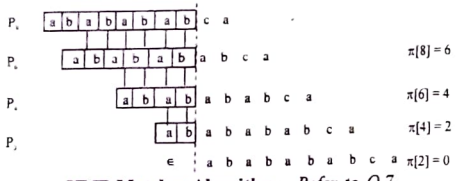
$$\pi[q] = \max\{k : k < q \text{ and } P_k \rceil P_q\}$$

i.e. $\pi[q]$ is the length of the longest prefix of P that is proper suffix of P_q.

i	1	2	3	4	5	6	7	8	9	10
P[i]	a	b	a	b	a	b	a	b	c	a
π[i]	0	0	1	2	3	4	5	6	0	1

P= abababacba and q=8. The π function for the given pattern. Since $\pi[8] = 6, \pi[6] = 4, \pi[4] = 2$ and $\pi[2] = 0$ by iterating π we obtain

$$\pi[8] = \{6, 4, 2, 0\}$$



KMP Matcher Algorithm : *Refer to Q 7.*

Q.18 ***(a) Write an algorithm for solving n-queen problem. Trace it for N=6 using backtracking approach.***

(b) Describe Travelling salesman problem. Show that a TSp can be solved using backtracking method in the exponential time. ***[R.T.U. 2017]***

Ans. (a) Algorithm for Solving n-queen Problem :

Queens Problem

In a game of chess, a queen is a game-piece which can attack in all directions that is vertically, horizontally, or diagonally (both rising and falling).

Problem Definition

The n-queens problem is the problem of placing n chess queens on an n×n chessboard such that none of them are in attacking position with respect to any other using the standard chess queen's moves. Hence, it is also called the Peaceful Queens Problem.

The eight-queens problem is defined as - "Given a chess board, of 8×8 fields, is it possible to place 8 queens on this board, so that no two queens can attack each other?"

Similarly, four-queens problem could also be defined.

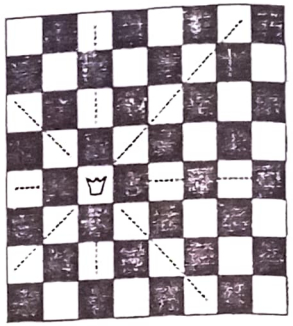


Fig. 1 : Directions of Attack by a Queen

Thought Process

A simple approach to solution requires that no two queens share the same row, column, or diagonal. This is so because placing a queen on any square of the board blocks the row and column where it is placed, as well as the two diagonals (rising and falling) at whose intersection the queen was placed.

Naive approach says we can form all possible permutations of board positions and check them. But this means checking n^n candidate solutions. Imagine the large amount of time it will take.

We can use the following tricks to reduce the number of candidate solutions

- **Checking for Same Row :** Place queens row by row, like queen 1 in row 1, queen 2 in row 2 and so on till queen n. In this way, no two queens are placed in same row automatically.
- **Checking for Same Column:** Try all columns for a queen i except the column in which queen i-1 is placed. Thus, we reduce a column. But, new queens must not be attacked by the ones at the top that have already been placed on the board. If we can remember columns of all previous queens, then we can exclude them also from the candidate solution. This reduces the number of candidate solutions to ,n!.

We have found a solution if we placed a queen in the bottom most row, that is if we reach a leaf node and the board position is still valid.

Not all the, n-queens problems have proper solutions. So first we find whether a solution exists or not? If yes, then

what is a possible solution? Also, there is more than one possible solution.

Two-Queens Problem

Let us start at n=2, and increase further.

The two-queens problem does not have a solution. It is best illustrated by the Fig. 2 given below.

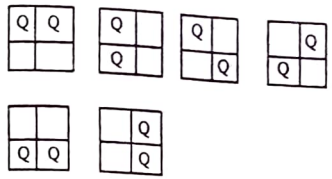


Fig. 2 : Possible Placing of Two Queens

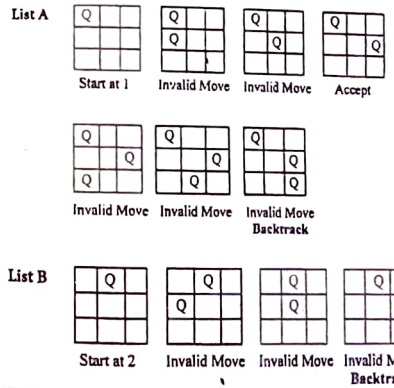
Three-Queens Problem

The three-queens problem is illustrated in the Fig.2 Possible situations of these queens are shown in lists and discussed below.

List A: Place the first queen in first row, first column. Try placing a queen in second row. It is finally placed in second row, third column. Now try placing third queen, no place is found correct. So we backtrack. But second row queen is in last position, so backtrack one level up. We come to situation list B.

List B : We start by placing first queen in second column. Then place second queen in second row. But no place is valid. Hence backtrack.

List C : Place the first queen in third column. Try placing second queen. Second row first con is good place. Now for third queen in third row no column is valid place. So backtrack. Try another position for second queen, which is not found. So there is no solution.



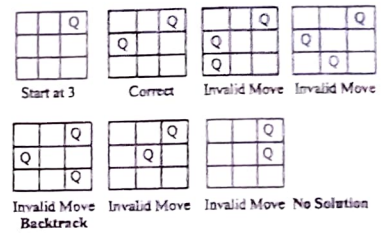


Fig. 3 : Possible Placing of Three Queen's

Hence we can conclude the three-queen's problem does not have a solution.

Four-Queen's Problem

The four-queen's problem is solvable. We explain here a detailed solution using backtracking. First we construct the search tree for four-queens problems, as shown in Fig.4

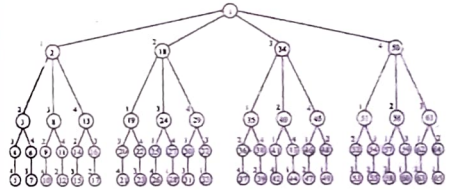
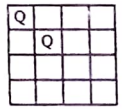


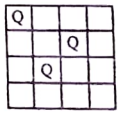
Fig. 4 : Search Tree for Four-Queen's Problem

Now we conduct the depth-first traversal of this tree.

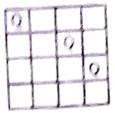
Traversing the path node 1 → node 2 → node3, we come to invalid position, as shown in the grid.So, prune the tree below node 3 and backtrack to node 2.



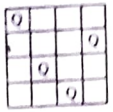
Traversing the path node 1 → node 2 → node 8 → node 9, we come to invalid position, as shown in the grid.So, backtrack to node 8, since there are still other paths from node 8. The, subtree at node 9 is pruned.



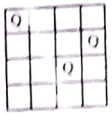
The path node 1 → node 2 → node 8 → node 11 also gives invalid position. And, now the, paths from node 8 are exhausted. So, backtrack till node 2. Subtree rocted at node 11 is pruned.



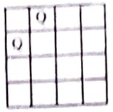
Taking the path node 1 → node 2 → node 13 → node 14 → node 15 we come to invalid position, as shown in the grid Backtrack till node 13.



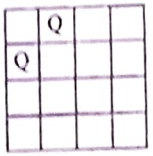
The path node 1 → node 2 → node 13 → node 16 gives an invalid position, as shown in the grid. And, no further paths are left till node 2. So, backtrack to node 1. Subtree at node 16 skipped.



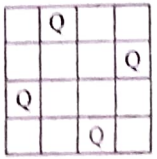
Traversing the path node 1 → node 18 → node 19 also gives an invalid position. So, backtrack. The subtree at node 19 is removed.



Traversing the path node 1 → node 18 → node 24 also gives an invalid position. So, backtrack to node 18. The subtree at node 19 is pruned.



Taking up the path node 1 → node 18 → node 29 → node 30 → node 31 gives a valid position. It is shown in the grid. Hence solution found.



The pruned tree and the path to a solution leaf are shown in the Fig.5. By symmetry there can be other solutions also. This is not a unique solution.

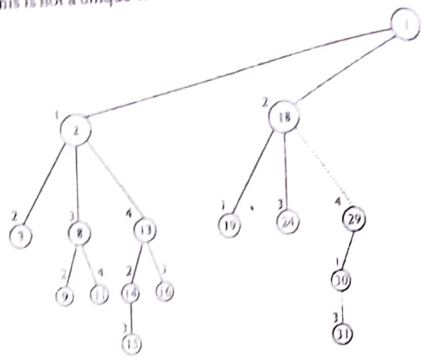


Fig. 5 : Path to a Solution of Four-Queen's Problem

Eight-Queen's Problem

First we have to construct the search tree for eight-queens problems, which would we very large to build. After this, conduct a depth-first traversal of this tree. Backtrack as soon as invalid board position is found and prune any subtree below that point. There are many solutions possible. One such solution is shown in Fig.6.

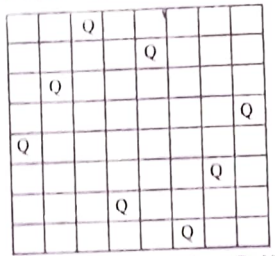


Fig. 6 : A Solution of Eight-Queens Problem

As a tuple the solution can be reported as (3, 5, 2, 8, 1, 7, 4, 6).

To implement an algorithm for the eight-queens, or in general n-queens problem, we can adopt a mathematical model to check that no two queens appear in same row, column or diagonal.

Checking for same row is already handled by placing one queen in each row. Check for same column is handled by decreasing the number of leaves in every level of the tree. Checking for diagonal includes the following calculation.

If two queens are at positions (i,j) and (k,l), than they are on same diagonal if one of the following two conditions hold :

I. $(i - j) = (k - l) \Rightarrow (i - k) = (j - l)$

II. $(i + j) = (k + l) \Rightarrow (i - k) = (l - j)$

That is the difference between rows is same as difference between columns, which is a diagonal Position.

Trace it for N = 6 :

i : 3

1 0 0 0 0 0
0 0 0 1 0 0
0 1 0 0 0 0
= 0 0 0 0 0 0
0 0 1 0 0 0
0 0 0 0 0 0

i : 1

1 0 0 0 0 0
0 0 0 0 0 0
0 1 0 0 0 0
= 0 0 0 0 0 0
0 0 1 0 0 0
0 0 0 0 0 0

i : 4

1 0 0 0 0 0
0 0 0 0 0 0
0 1 0 0 0 0
= 0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0

i : 1

1 0 0 0 0 0
0 0 0 0 0 0
0 1 0 0 0 0
= 0 0 0 0 0 0
0 0 0 0 0 0
0 0 1 0 0 0

i : 5

1 0 0 0 0 0
0 0 0 0 0 0
0 1 0 0 0 0
= 0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0

i : 2

1 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
= 0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0

i : 2

1 0 0 0 0 0
0 0 1 0 0 0
0 0 0 0 0 0
= 0 1 0 0 0 0
0 0 0 1 0 0
0 0 0 0 0 0

i : 4

1 0 0 0 0 0
0 0 1 0 0 0
0 0 0 0 0 0
= 0 1 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0

i = 1

1 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
= 0 1 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0

i : 2

1 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
= 0 1 0 0 0 0
0 0 0 0 0 0
0 0 1 0 0 0

i : 5

$$= \begin{matrix} 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{matrix}$$

i : 3

$$= \begin{matrix} 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{matrix}$$

i : 2

$$= \begin{matrix} 1&0&0&0&0&0\\ 0&0&1&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&0&1&0&0 \end{matrix}$$

i : 5

$$= \begin{matrix} 1&0&0&0&0&0\\ 0&0&1&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&0&0&0&0 \end{matrix}$$

i : 1

$$= \begin{matrix} 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&0&0&0&0 \end{matrix}$$

i : 4

$$= \begin{matrix} 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{matrix}$$

i : 4

$$= \begin{matrix} 1&0&0&0&0&0\\ 0&0&1&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&1&0&0&0&0 \end{matrix}$$

i : 1

$$= \begin{matrix} 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&1&0&0&0&0 \end{matrix}$$

i : 1

$$= \begin{matrix} 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&1&0&0&0\\ 0&0&0&0&0&0\\ 0&1&0&0&0&0 \end{matrix}$$

i : 3

$$= \begin{matrix} 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&1&0&0&0&0 \end{matrix}$$

i : 5

$$= \begin{matrix} 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{matrix}$$

i : 0

$$= \begin{matrix} 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{matrix}$$

i : 4

$$= \begin{matrix} 0&0&1&0&0&0\\ 1&0&0&0&0&0\\ 0&0&0&1&0&0\\ 0&1&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{matrix}$$

i : 2

$$= \begin{matrix} 0&0&1&0&0&0\\ 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{matrix}$$

i : 0

$$= \begin{matrix} 0&0&0&0&0&0\\ 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{matrix}$$

$$= \begin{matrix} 0&0&0&1&0&0\\ 1&0&0&0&0&0\\ 0&0&0&0&1&0\\ 0&1&0&0&0&0\\ 0&0&0&0&0&1\\ 0&0&1&0&0&0 \end{matrix}$$

Final Solution

			Q		
Q					
				Q	
	Q				
					Q
		Q			

Ans. (b) Travelling Salesman Problem : A travelling salesperson has to travel a group of cities, which are connected to each other.

There is a cost associated with travelling from one city to other city. The salesperson wants to travel in such a way that he visits every city exactly once and returns to the city where he started. Also, cost of this tour needs to be minimized.

Now, this network of cities can be represented through a directed graph with weighted edges, where a vertex denotes a city. Weight of an edge (i,j) denotes the cost of travelling from vertex i to vertex j. Since cost of travelling from city i to city j can be different from the cost of travelling from city j to city i, we have directed edges.

Problem Definition

Given a graph G with a cost function C(i,j) defined for each edge (i, j) in G, we want to find the minimum cost tour that visits every vertex in G and returns to the starting vertex.

We define a function x(i,j), such that

$$x(i,j) = \begin{cases} 1, & \text{if city j comes directly after city i in the tour} \\ 0, & \text{otherwise} \end{cases}$$

The objective is to minimize $\sum_i \sum_j x_{ij} \cdot c_{ij}$, subject to the constraint that x(i,j) should be so chosen that no city is visited twice except the starting one.

Algorithm

Upper bound for all nodes is assumed to be ∞. Hence, $u[i] = \infty$, for all i.

Lower bound is calculated by a bounding function.

We can compute for each edge (i,j) the minimum cost path beginning at i, ending at j, while visiting all vertices of G along the path. We generate this path by augmenting current path by one vertex at a time (that is with each iteration of the algorithm).

- At any point we have a partial path P starting at vertex 1, we consider only those vertices which are not in P for augmenting the path. This is the branching step.
- A path has a dead end if the vertices not in P are disconnected in G + e. This is used to stop branching.
- A lower bound is defined as sum of path cost and cost of new edge. That is, total cost of edges already in path P added to cost of edge e. This is the bounding algorithm. Mathematically,

$$\hat{f}(j) = \hat{f}(i) + RC[i,j] + e$$

where, $\hat{f}(i)$ is lower bound of parent node

$\hat{f}(j)$ is lower bound of current node

$RC[i,j]$ is the (i,j) entry of RC matrix which is obtained by reducing the cost matrix of graph G. e is a factor introduced by augmenting the path.

After we have run the algorithm for one edge in G, we can use the best path (path with minimum value) found so far over all tested edges for next iteration of the algorithm, using this path as current best solution partial). Thus, we apply the algorithm recursively.

We have found a solution when all vertices have been visited.

TSP Solved Using Backtracking Method : We are given a set of n cities, with the distances between all cities. A traveling salesman, who is currently staying in one of the cities, wants to visit all other cities and then return to his starting point, and he is wondering how to do this. Any tour of all cities would be a valid solution to his problem, but our traveling salesman does not want to waste time: he wants to find a tour that visits all cities and has the smallest possible length of all such tours. So in this case, optimal means: having the smallest possible length.

In Branch and Bound method, cost of any tour can be written as below

Cost of a tour T = (1/2) × Σ(Sum of cost of two edges adjacent to u and in the tour T)

where u ∈ V

For every vertex u, if we consider two edges through it in T, and sum their costs. The overall sum for all vertices would be twice of cost of tour T (We have considered every edge twice.)

(Sum of two tour edges adjacent to u) >= (sum of minimum weight two edges adjacent to u)

Cost of any tour >= 1/2) × Σ (Sum of cost of two minimum weight edges adjacent to u)

where u ∈ V

```
Algorithm TSP Backtrack( A, currLength, lengthSoFar, minCost)
1.  n ← length( A )  // number of elements in the array
    A
2.  if i = n
3.  then minCost ← min(minCost, lengthSoFar +
    distance( A[1], A[n]))
4.  else for j ← i + 1 to n
5.  do Swap A[i + 1] and A[j]  // select A[j] as the next
    city
6.  newLength ← lengthSoFar + distance
    (A[i], A[i + 1])
7.  if newLength < minCost  // this will never be a better
    solution
8.  then skip  // prune
9.  else minCost ←
10. min( minCost, TSP Backtrack ( A, i + 1, newLength,
    minCost))
11. Swap A[i + 1] and A[j]  // undo the selection
12. return minCost
```

The worst case complexity of Branch and Bound remains same as that of the Brute Force clearly because in worst case, we may never get a chance to prune a node. Whereas, in practice it performs very well depending on the different instance of the TSP. The complexity also depends on the choice of the bounding function as they are the ones deciding how many nodes to be pruned.

Q.19 ***Discuss Boyer moore pattern matching algorithm with appropriate example of good prefix and bad character.*** ***[R.T.U., 2017, 2015]***

OR

Explain Boyer Moore Algorithms with suitable example. ***[R.T.U., 2016]***

OR

Explain both the heuristics of Boyer-Moore Algorithm with suitable examples. ***[R.T.U., 2014]***

Ans. The Boyer–Moore Algorithm : If the pattern P is relatively long and the alphabet string S is reasonably large, then an algorithm by Robert S. Boyer and J. Strother Moore is likely to be the most efficient string–matching algorithm.

```
Boyer–Moore–Matcher (T, P, Σ)
     //T is text
     //P is pattern
1    n ← length[T]
2    m ← length[P]
3    λ ← Compute–Last–Occurrence–Function (P, m, Σ)
4    γ ← Compute–Good–Suffix–Function (P, m)
5    s ← 0
6    while s <= n – m
7    do j ← m
8    while j > 0 and P[j] = T[s + j]
9    do j ← j – 1
10   if j = 0
11   then print "Pattern occurs at shift" s
12   s ← s + γ[0]
13   else s ← s + max(γ[j], j + λ[T[s + j]])
```

Aside from the mysterious–looking λ's and γ's, this program looks remarkably like the naive string–matching algorithm.

```
12   s ← s + 1
13   else s ← s + 1
```

The while loop beginning on line 6 considers each of the n – m + 1 possible shifts 's' in turn, and the while loop beginning on line 8 tests the condition P[1..m] = T[s + 1 .... s + m] by comparing P[j] with T[s + j] for j = m, m – 1, ..., 1. If the loop terminates with j = 0, a valid shift Σ_j = s has been found, and line 11 prints out the value of s. At this level, the only remarkable features of the Boyer–Moore algorithm are that it compares the pattern against the text from right to left and that it increases the shift s on lines 12–13 by a value that is not necessarily 1.

The Boyer–Moore algorithm incorporates two heuristics that allow it to avoid much of the work that our previous string–matching algorithms performed. These heuristics are so effective that they often allow the algorithm to skip altogether the examination of many text characters. These heuristics, known as the "bad–character heuristic" and the "good – suffix heuristic". They can be viewed as operating independently in parallel. When a mismatch occurs, each heuristic proposes an amount by which s can safely match without missing a valid shift. The Boyer–Moore algorithm chooses the larger amount and increases s by that amount : when line 13 is reached after a mismatch, the bad–character heuristic proposes increasing s by j – λ[T(s + j)], and the good–suffix heuristic proposes increasing s by γ[j].

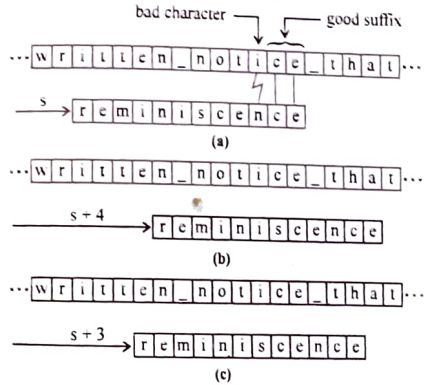


Fig.

Fig. shows the example of Boyer–Moore heuristic. (a) Matching the pattern "reminiscence" against a text by comparing characters in a right to left manner. The shift s is invalid, although a good suffix "ce" of the pattern matched correctly against the corresponding characters in the text (matching characters are shown shaded), the bad character "i", which didn't match the corresponding character "a" in the pattern, as discovered in the text. (b) The bad–character heuristic proposes moving the pattern to the right, if possible, by the amount that guarantees that the bad text character will match the rightmost occurrence of the bad character in the pattern. (c) With the good–suffix heuristic, the pattern is moved to the right by the least amount that guarantees that any pattern characters that align with the good suffix "ce" previously found in the text will match those suffix characters.

The bad–character heuristic : When a mismatch occurs, the bad–character heuristic uses information about where the bad text character T[s + j] occurs in the pattern (if it occurs at all) to propose a new shift. In the best case, the mismatch occurs on the first comparison (P[m] ≠ T[s + m] and the bad–character T[s + m] does not occur in the pattern at all, (imagine searching for a^m in the text string b^n). In this case, we can increase the shift s by m, since any shift smaller than s + m will align some pattern character against the bad–character, causing a mismatch. If the best case occurs repeatedly, the Boyer–Moore algorithm examines only a fraction 1/m of the text characters, since each text character examined yields a mismatch, thus causing s to increase by m.

In general, the bad–character heuristic works as follows. Suppose we have just found a mismatch : P[j] ≠ T[s + j] for some j, where 1 ≤ j ≤ m. We then let b be the largest index in the range 1 ≤ b ≤ m such that T[s + j] = P[k], if any such k exists. Otherwise, we let k = 0. k = 0, the bad–character T[s + j] didn't occur in the pattern at all, and so we can safely increase s by j without missing any valid shifts. k < j : the rightmost occurrence of the bad–character is in the pattern to the left of position j, so that j – k > 0 and the pattern must be moved j – k characters to the right before the bad text character matches any pattern character. Therefore, we can safely increase s by j – k without missing any valid shifts. k > j, j – k < 0, and so the bad–character heuristic is essentially proposing to decrease s. This recommendation will be ignored by the Boyer–Moore algorithm, since the good–suffix heuristic will propose a shift to the right in all cases.

The following simple program defines λ[α] to be the index of the right–most position in the pattern at which character α occurs, for each α ∈ Σ. If α does not occur in the pattern, then λ[α] is set to 0. We call λ the last occurrence function for the pattern. The expression j – λ[T[s + j]], implements the bad–character heuristic. (Since j – λ[T(s + j)]) is negative if the right most occurrence of the bad–character T[s + j] in the pattern is to the right of position; we rely on the positivity of γ[j].

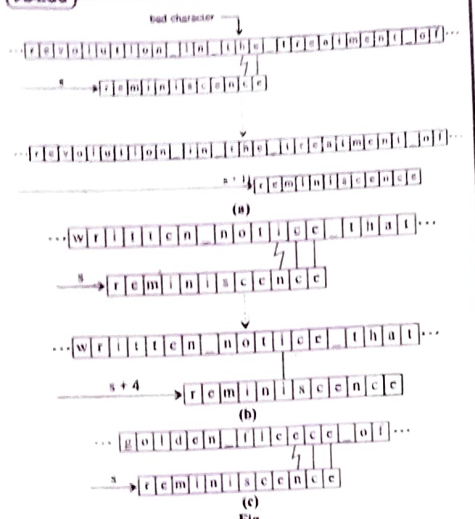


Fig.

Fig. shows the cases of the bad–character heuristic. (a) The bad–character occurs nowhere in the pattern, and so the pattern can be advanced by j = n characters until it has passed over the bad–character. (b) The rightmost occurrence of the bad–character in the pattern is at position k < j, and so the pattern can be advanced by j – k characters. Since j = 10 and k = 6 for the bad–character i, the pattern can be advanced 4 positions until the i's line up. (c) The rightmost occurrence of the bad–character in the pattern is at position k > j. In this example, j = 10 and k = 12 for the bad character e. The bad–character heuristic proposes a negative shift, which is ignored.

Compute–Last–Occurrence–Function (P, m, Σ)

```
1   for each character α ∈ Σ
2       do λ[α] = 0
3   for j ← 1 to m
4       do λ[P[j]] ← j
5   return λ
```

The running time of procedure Compute–Last–Occurrence–Function is O(|Σ| + m).

The good–suffix heuristic : Let us define the relation Q ~ R (read "Q is similar to R') for strings Q and R to mean that Q ⊐ R or R ⊐ Q. If two strings are similar, then we can align them with their rightmost characters matched, and no pair of aligned characters will disagree. The relation "~" is symmetric : Q ~ R if and only if R ~ Q.

Q ⊐ R and S ⊐ R imply Q ~ S. ....(1)

If we find that P[j] ≠ T[s + j], where jm , then the good–suffix heuristic says that we can safely advance s by

y[j] = m – max {k : 0 ≤ k m and P[j + 1 ..m] ~ P_k}

That is, Y[j] is the least amount we can advance s and not cause any characters in the "good suffix" T[s + j + 1 .. s + m] to be mismatched against the new alignment of the pattern. The function y is well defined : for all j, since P[j + 1..m] ~ P_0 for all j : the empty string is similar to all strings. We call y the good–suffix function for the pattern P.

We first observe that y[j] ≤ m – π[m] for all j, as follows. If w = π[m], then P_w ⊐ P by the definition of π. Furthermore, since P[j + 1 .. m] ⊐ P for any j, we have P_w ~ P[j + 1 .. m]. Therefore, y[j] ≤ m – π[m] for all j.

Definition of y

y[j] = m – max {k : π[m] and P[j + 1..m] ~ P_k}.

To simplify the expression for Y further, we define P' as the reverse of the pattern P and π' as the corresponding prefix function. That is P'[i] = P[m – i + 1] for i = 1, 2, ..., m, and π'(t) is the largest u such that u < t and P'_u ⊐ P'_t.

If k is the largest possible value such that P[j + 1..m] ⊐ P_k, then we claim that π'[l] = m – j.

where l = (m – k) + (m – j). To see that this claim is well defined, note that P(j + 1 ... m) ⊐ P_k implies that m – j ≤ k, and thus l ≤ m. Also j < m and k ≤ m, so that l ≥ 1. We prove this claim as follows. Since P[j + 1..m] ⊐ P_k, we have P'_{m-j} ⊐ P'_l. Therefore, π'[l] ≥ m – j. Suppose now that p > m – j, where p = π'[l]. Then, by the definition of π', we have P"p' ⊐ P'_l or equivalently, P'[1..p] = P'[l – p + 1..l].

Rewriting this equation in terms of P rather than P', we have P(m – p + 1 ..... m) = ... P[m – l + 1..m – l + p]. Substituting for l = 2m – k – j, we obtain P(m – p + 1..m] = P[k – m + j + 1..k – m + j +p], which implies P[m – p +1..m] ⊐ $P_{k-m+j+p}$, since p > m – j, we have j + 1 > m – p + 1, and so P[j + 1..m] ⊐ P[m – p + 1..m], implying that P[j + 1..m] ⊐ P_{km+j+p} by the transitivity of ⊐. Finally, since p > m – j, we have k' > k, where k' = k – m + j + p, contradicting our choice of k as the largest possible value such that P[j + 1..m] ⊐ P_k. This contradiction means that we can't have p > m – j, and thus p = m – j, which proves the claim. π'[l] = m – j implies that j = m – π'[l] and k = m – l + π'[l], we can rewrite our definition of y still further.

y[j] = m – max({π[m]}
U{m – l + π'[l] , 1 ≤ l ≤ M m and j = m – π'[l]})
= min({m – π[m]}
U {l – π'[l] : 1 ≤ l ≤ m and j = m – π'[l]}).

Again, the second set may be empty.

Compute–Good–Suffix–Function (P, m)

```
1   π ← Compute–Prefix–Function (P)
2   P' ← reverse (P)
3   π' ← Compute–Prefix–Function (P')
4   for j ← 0 to m
5       do y[j] ← m – π [m]
6   for l ← 1 to m
```



```
7       do j ← m – π'[l]
8          if y[j] > l – π'[l]
9             then y[j] ← l – π'[l]
10  return y
```

The procedure Compute–Good–Suffix–Function is a straight forward implementation of equation. Its running time is O(m).

The worst–case running time of the Boyer–Moore algorithm is clearly O(n – m + l)m + |Σ|, since *Compute–Last–Occurrence–Function* takes time O(m + |Σ|), *Computer–Good–Suffix–Function* takes time O(m), and the Boyer–Moore algorithm (like the Rabin–Karp algorithm) spends O(m) time validating each valid shift s.

Q.20 ***What is backtracking? Explain 8-queens problem, also write algorithm for the same.*** ***[R.T.U. 2016]***

OR

What is backtracking. ***[R.T.U. 2015]***

OR

Write the short note on backtracking algorithms. ***[R.T.U. 2010, 2009]***

Ans. Backtracking : Backtracking is a technique of solving a problem by trial and error. However, it is a well organized trial and error. We make sure that we never try the same thing twice. We also make sure that if the problem is finite we will eventually try all possibilities (assuming there is enough computing power to try all possibilities).

Backtracking imposes a tree structure on the solution space. Backtracking does a preorder traversal of this tree, while processing the leaves. It saves time by using pruning, that is, by skipping those internal nodes that do not promise useful leaves.

Backtracking Algorithm

General steps involved in a backtracking algorithm are :

Step - 1: Choose a basic object, like strings, combinations and permutations.

Step - 2: Apply divide and conquer approach to generate these basic objects. Implement through recursion.

Step - 3: The test of desired property is placed at the base of recursion, that is with the leaves.

Step - 4: To include pruning, place the code for pruning before each recursive call

Fig. 1 shows that how during traversal if we do not find a solution in a branch, then we backtrack to visit other branch. This visiting of other branch by coming up one level in the tree then following a level down in other branch is called backtracking. It is done till a good solution is found.

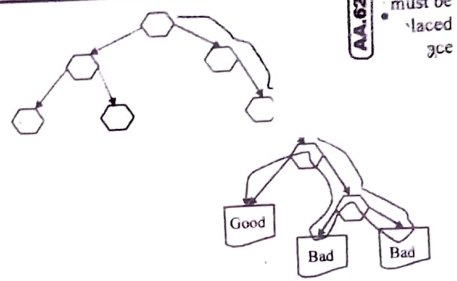


Fig. 1 : Illustration of Backtracking

In general case, we assume our solution to be a vector $v = (a_1, a_2, \ldots a_n)$, where each element a_i is selected from a finite set S_i. And follow the steps given in the Fig. 2. It is a general algorithm for backtracking. We always begin with empty solution, augment it at each stage and check whether we are going in the direction of feasible solution or not.

Step-1 : Build a partial solution of length k, that is $v = (a_1, a_2, \ldots a_k)$

Step-2 : Try to extend it by adding one more element from set S_k.

Step-3 : Test whether this augmented vector is still a possible solution. If no then delete last element. If yes and k=n, report the solution and exit.

Step-4 : If there are elements left in S_k, go to step 2, else return.

Fig. 2 : A General Backtracking Algorithm

To generate all solutions, the algorithm is slightly modified. Instead of exiting after reporting a solution, we continue till all solution vectors have been tested. Each time we find a possible solution we report it.

Types of Backtracking Algorithms : Actually, two versions of backtracking algorithms exist :

1. The algorithms in which the solution needs only to be feasible (the solution must satisfy problem constraints). Backtracking stops as soon as a feasible solution is found. For example, the problem of finding sum of subsets, n-queens problem.
2. The algorithms in which solution needs to be optimal. It should be feasible also. This occurs when we apply backtracking to optimization problems. For example, knapsack problem, assignment problem, etc.

State Space Tree : A given problem has a set of constraints and possibly an objective function. The solution optimizes an objective function, and/or is feasible. We can represent the solution space for the problem using a state space tree, where

- The root of the tree represents 0 choices.

- Nodes at depth 1 represent first choice.
- Nodes at depth 2 represent the second choice and so on.

In this tree, a path from a root to a leaf represents a candidate solution.

A node is called Non-promising if it cannot lead to a feasible or optimal solution. Otherwise if it can lead to a feasible or optimal solution then it is called Promising.

Backtracking consists of doing a Depth First Search of the state space tree, checking whether each node is promising. If the node is non promising, then backtrack to the node's parent and continue from next child.

For optimization problems, the state space tree is same but definitions are modified. To deal with optimization, we compute two functions simultaneously :

Best to store value of best solution achieved so far; and *Value*(v) to calculate the value of the solution at node v.

Best is initialized to a value that is equal to a candidate solution or worse than any possible solution. Best is updated to value(v) if the solution at v is "better". By "better" we mean larger in the case of maximization and smaller in the case of minimization.

A node is promising when it is feasible and can lead to a feasible solution and there is a chance that a better solution than best can be achieved by exploring it. Otherwise it is nonpromising.

8-queens problem : Place eight queens on an 8 × 8 chessboard so that no queen attacks another queen.

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

Identify data structures to solve the problem

- **First pass :** Define the chessboard to be an 8 × 8 array.
- **Second pass :** Since each queen is in a different row, define the chessboard solution to be an 8-tuple $(x_1, \ldots, x_8)$ where x_i is the column for i^{th} queen.
- Identify explicit constraints using 8-tuple formulation as $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}, 1 \le i \le 8$.

Solution space of 8 × 8 - tuples

- Identify implicit constraints.
- No two x_i can be the same, or all the queens must be in different columns.

All solutions are permutations of the 8 - tuple (1, 2, 3, 4, 5, 6, 7, 8)

reduces the size of solution space from 8×8 to $8!$ tuples.

- No two queens can be on the same diagonal.
- The solution above is expressed as an 8 - tuple as 4, 6, 8, 2, 7, 1, 3, 5

The game of chess is played on an 8 by 8 board with 64 squares. There are a number of pieces used in the game. The most powerful one is called a queen. The eight queens problem refers to a configuration which cannot occur in an actual game, but which is based on properties of the queen. The problem is: place 8 queens on the board in such a way that no queen is attacking any other.

The queen attacks any other piece in its row, column, or either of its two diagonals. In Fig. 1 shown below, the queen (Q) is attacking all squares marked X. Squares not under attack are not marked.

			X				X
X			X			X	
	X		X		X		
		X	X	X			
X	X	X	Q	X	X	X	X
		X	X	X			
	X		X		X		
X			X			X	

Fig. 1 : The eight queens problem

The power of the queen is such that there is some difficulty in placing all eight queens.

A solution to the problem is an array of eight integer values stored in the variable result: for each column i, result[i] is the row in which the queen is placed.

Fig. 2 A solution to the eight-queens problem

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

Fig. 2 : The eight-queens problem


```
1    byte result[8];
2    bool a[8];
3    bool b[15];
4    bool c[15];
5
6    active proctype Queens ( ) {
7        byte col = 1,
8        byte row;
9    do
10   : : choose ( ) ;
11           !a[row-1];
12           !b[row+col-2];
13           !c[row-col+7];
14           a[row-1] = true;
15           b[row+col-2] = true;
16           c[row-col+7] = true;
17           result[col-1] = row;
18           if
19           : : col == 8 → break
20               else → col++
21           fi
22           od;
23           Write( )
23   }
```

So for the solution in Figure 2, result contains 1, 5, 8, 6, 3, 7, 2, 4. The algorithm works by nondeterministically choosing a row for each column in sequence, and then checking that a queen placed on that square cannot capture a queen that has already been placed on the board.

To facilitate checking for captures, three auxiliary Boolean arrays are used: a [i] is true if there is a queen in row I; b[i] is true if there is a queen on the positive diagonal I; c[i] is true if there is a queen on the negative diagonal i. the positive diagonals go from the lower left to the upper right of the board, while the negative diagonals go from the upper left to the lower right. It is easy to check that there are fifteen diagonals of each type, that the squares on the positive diagonals are those with equal values of the sum of the row and column numbers, and that the squares on the negative diagonals are those with equal values of the difference between them.

The process starts out by initializing col to 1. It then proceeds with a do-statement that places a queen on each column in sequence, terminating when a queen has been placed on the eighth column (lines 18-21).

Once a row number is chosen for a queen, lines 11-13 check the row and diagonal data structures to see if a capture is possible. If so, the execution blocks, and since there is only one process, the computation terminates at this point. If a capture is not possible, the data structures are updated (lines 14-16) and the row number stored in result. Since array indices in PROMELA start from zero, offsets to the indices must be computed in lines 11-17. Once all queens have been placed successfully, the solution is written out in the inline sequence Write.

Q.21 ***What is importance of Rabin Karp string matching algorithm.*** ***[R.T.U. 2015]***

OR

The Rabin-Karp Algorithm is suitable for string pattern matching. Justify the answer. ***[R.T.U. 2012]***

Ans. The Rabin-Karp Algorithm

A string search algorithm which compares a string's hash values, rather than the strings themselves. For efficiency, the hash value of the next position in the text is easily computed from the hash value of the current position.

How Rabin-Karp Works

- Let characters in both arrays T and P be digits in radix-S notation, (S=(0,1, ...,9).
- Let p be the value of the characters in P.
- Choose a prime number *q* such that fits within a computer word to speed computations.
- Compute (p mod q)
  - The value of p mod q is what we will be using to find all matches of the pattern P in T.
- Compute (T[s+1, .., s+m] mod q) for s = 0 .. n-m
- Test against P only those sequences in T having the same (mod q) value.

(T[s+1, .., s+m] mod q) can be incrementally computed by subtracting the high-order digit, shifting, adding the low-order bit, all in modulo q arithmetic.

Complexity

- The running time of the Rabin-Karp algorithm in the worst-case scenario is O(n–m+1)m but it has a good average-case running time.
- If the expected number of valid shifts is small O(1) and the prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time O(n+m) plus the time to required to process spurious hits.

Applications

- Bioinformatics

– Used in looking for similarities of two or more proteins; i.e. high sequence similarity usually implies significant structural or functional similarity.

Example

Hb A_human

GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKL

G– –VK–HGKKV A–––––– AH– D–– — ––––LS–LH

KL

Hb B_human

GNPKVKAHGKKVLGAFSDGLAHLDNLKGTF ATLSELHCDKL + similar amino acids

- Alpha hemoglobin and beta hemoglobin are subunits that make up a protein called hemoglobin in red blood cells. Notice the similarities between the two sequences, which probably signify functional similarity.
- Many distantly related proteins have domains that are similar to each other, such as the DNA binding domain or cation binding domain. To find regions of high similarity within multiple sequences of proteins, local alignment must be performed. The local alignment of sequences may provide information of similar functional domains present among distantly related proteins.

Rather than pursuing more sophisticated skipping, the Rabin-Karp algorithm seeks to speed up the testing of equality of the pattern to the substrings in the text by using a hash function. A hash function is a function which converts every string into a numeric value, called its hash value; for example, we might have hash("hello")=5. Rabin-Karp exploits the fact that if two strings are equal, their hash values are also equal. Thus, it would seem all we have to do is compute the hash value of the substring we are searching for, and then look for a substring with the same hash value.

However, there are two problems with this. First, because there are so many different strings, to keep the hash values small we have to assign some strings the same number. This means that if the hash values match, the strings might not match, we have to verify that they do, which can take a long time for long substrings. Luckily, a good hash function promises us that on most reasonable inputs, this would not happen too often, which keeps the average search time good.

Q.22 ***Explain backtracking method by using the example of 4-Queen problem with flow diagram.*** ***[R.T.U. 2014]***

Ans. Backtracking

Back tracking is a systematic way to go through all the possible configuration of a search space. In general case, we assume our solution is a vector $v = (a_1, a_2, \ldots, a_n)$ where each element a_i is selected from a finite ordered set S.

We build from a partial solution of length $k, v = (a_1, a_2, \ldots, a_k)$ and try to extend it by adding another element. After extending it, we will test whether what we have so far is still possible as partial solution. If not, we delete a_k and try the next element from S_k.

Compute S_1, the set of candidate first element of v.

k = 1

while k > 0 do

while $S_k \neq 0$ do /* advance */

$a_k = a_n$ element in S_k

$S_k \leftarrow S_k - a_k$

if $(a_1, a_2 \ldots a_k)$ is solution

Print K = K + 1

compute S_k, the candidate K^{th}

elements given v

K = K - 1 */ Back tracking*/

Recursive Back-Tracking

Recursion can be used for elegant and easy implementation of backtracking.

Backtrack (a, K)

1. if a is a solution
2. then print(a)
3. else
4. K ← K + 1
5. compute S_k
6. while $S_k \neq 0$ do
7. $a_k \leftarrow$ an element in S_k
8. $S_k \leftarrow S_k - a_k$
9. Backtrack (a, K)

• Backtracking can be easily used to iterate through all subsets or permutations of a set.

• Backtracking ensures correctness by enumerating all possibilities.

• For backtracking to be efficient, we must prune the search space.

4-Queens Problem : *Refer to Q 18(a).*

Q.23 ***For given Travelling salesman problem matrix:***

∞	6	0	2
5	∞	7	0
0	4	∞	14
8	0	0	∞

(i) What is the reduced cost matrix?

(ii) Find the optimal tour of given Travelling Salesman Problem. ***[R.T.U. 2014]***

Ans. (i) Suppose a salesman wants to visits a certain number of cities alloted to him. He knows the distance (or cost or time) of journey between every pair of cities usually denoted by C_{ij} i.e., from city i to city j. His problem is to select such a route that starts from his home city, passes through each city once and only once, and returns to his home city in the shortest possible distance (or at the least cost or in least time).

Formulation of Travelling-Salesman Problem

Suppose C_{ij} is the distance (or cost or time) from city i to city j and $x_{ij} = 1$ if the salesman goes directly from city i to city j, and $x_{ij} = 0$ otherwise.

The minimizes $\sum_i \sum_j x_{ij} C_{ij}$ with the additional restriction that the x_{ij} must be so chosen that no city is visited twice before the tour of all cities is completed.

In a particular he cannot go directly from city i to j itself. This possibility may be avoided in the minimization process by adopting the convention $C_{ij} = \infty$ which ensures that x_{ij} can never be unity.

Alternatively, omit the variable x_{ij} from the problem specification. It is also important to note that only single $x_{ij} = 1$ for each of i and j. The distance (or cost or time) matrix for this problem is given in table.

From City \	A_1	A_2	....	A_n
A_1	∞	C_{12}	....	C_{1n}
A_2	C_{21}	∞	....	C_{2n}
:	:	:	:	:
A_n	C_{n1}	C_{n2}	....	∞

The given travelling salesman problem is

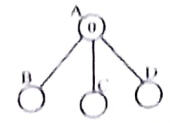
$$C = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & 6 & 0 & 2 \\ 5 & \infty & 7 & 0 \\ 0 & 4 & \infty & 14 \\ 8 & 0 & 0 & \infty \end{bmatrix} \end{array}$$

As seen in the above given matrix every row and every column have zero and entire row or column have value ∞. Hence this matrix is already reduced so no further action is needed.

$$RC = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & 6 & 0 & 2 \\ 5 & \infty & 7 & 0 \\ 0 & 4 & \infty & 14 \\ 8 & 0 & 0 & \infty \end{bmatrix} \end{array}$$

Lower Bound (LB) = 0

(ii) The Construction tree for this problem will start at a root vertex of cost 0 (zero). The root node corresponds to vertex A.



Expand this node by computing path value from A to B, C and D.

Considering path A → B, we formulate a matrix M by setting row A and column B to infinity (∞). Also entry [B, A] = ∞, so that the path does not track back to A.

$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 7 & 0 \\ 0 & \infty & \infty & 14 \\ 8 & \infty & 0 & \infty \end{bmatrix} \end{array}$$

As seen in the above matrix every row & every column have zero and entire row & column have value ∞. Hence this matrix is already reduced so r = 0.

Node value, l_b = LB + r + RC(A, B)

= 0 + 0 + 6

= 6

Now consering the path A → C, for which the corresponding matrix will have row A and column C set to ∞. And the entry [C, A] = ∞, so that the path does not track back to A.

$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ 5 & \infty & \infty & 0 \\ \infty & 4 & \infty & 14 \\ 8 & 0 & \infty & \infty \end{bmatrix} \end{array}$$

As seen in the above matrix the row C and column A does not have zero. So we require to subtract the minimum value from each row.

Row C = Row C − 4

Column A = Column A − 5

So reduced matrix is

$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 0 \\ \infty & 0 & \infty & 10 \\ 3 & 0 & \infty & \infty \end{bmatrix} \end{array}$$

So r = 4 + 5 = 9, them

Node value, l_c = LB + r + RC (A, C)
= 0 + 9 + 0
$l_c = 9$

Now considering the path A → D, we have a matrix with row A and column D set to ∞, and the entry [D, A] = ∞. It is shown below

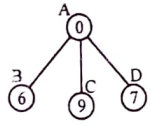
M	A	B	C	D
A	∞	∞	∞	∞
B	5	∞	7	∞
C	0	4	∞	∞
D	∞	0	0	∞

As seen in above matrix, the row B does not have zero. So we require to subtract the minimum value by that row, so
Row B = Row B – 5

M	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	2	∞
C	0	4	∞	∞
D	∞	0	0	∞

So r = 5, then
Node value, l_d = LB + r + RC(A, D)
= 0 + 5 + 2
= 7

Hence the constructed tree



At this stage we have
$l_b = 6$ (minimum)
$l_c = 9$
$l_d = 7$

Thus, at this stage, we find that the path (A, B) is the most promising with the minimum cost 6. Therefore, we expand the node B.

Now Taking path A → B → C, we formulate the matrix M by setting row A, row B and column C to ∞. Also the entries [B, A] and [C, A] are set to ∞. Thus

M	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	∞
C	0	4	∞	14
D	8	0	∞	0

As seen the above matrix, the row C, column A and umn D does not have zero, so we first subtract 4 from row C then subtract 10 from column D and then subtract 8 from column A.

So the reduced matrix is

M	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	∞
C	∞	0	∞	0
D	0	0	∞	∞

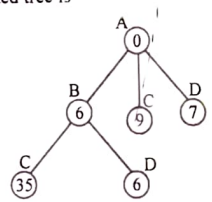
So r = 8 + 4 + 10 = 22, then
Node value, $l_c = l_b$ + r + RC(B, C)
= 6 + 22 + 7
= 35

Now we take the path A → B → D, we formulate the matrix M by setting row A, row B and column D to ∞. Also the entries [B, A] and [D, A] are set to ∞. Thus

M	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	∞
C	∞	4	∞	∞
D	∞	0	0	∞

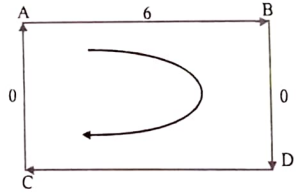
As seen in the above matrix, the matrix is already reduced. So r = 0, then
Node value, $l_d = l_b$ + r + RC (B, D)
= 6 + 0 + 0
= 6

So the constructed tree is



Since there is no different path now to explore, we found the solution. The minimum distance node is D with value 6, therefore the solution path is

A → B → D → C → A



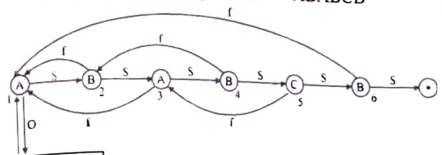
Total cost = C[A, B] + C[B, D] + C[D, C] + C[C, A]
= 6 + 0 + 0 + 0
= 6

Note that this is same as value of last node in tree, where we found the solution.

Q.24 ***Explain and construct the KMP flow chart for pattern P = "ABABCB" and also show the actions perform of KMP flow chart with given text T = "ACABAABABA"*** *[R.T.U. 2014]*

Ans. KMP pattern matching algorithm calculates the prefix code. This prefix code calculations scans the pattern first and calculates the prefix of each character in the pattern. The prefix function calculates the shifting of pattern matches within itself. If we know how the pattern matches shifts against itself then pattern an slide more than one character towards right. Then according to this prefix code generated it is decided that by how many positions the pattern to be shifted further if any mismatch occurs instead of shifting the pattern only by one character

KMP flow chart for pattern P = "ABABCB"



Cell number	Text index	Second character	Sucess or Failure
1	1	A	S
2	2	C	f
1	2	C	f
0	2	C	get next character
1	3	A	S
2	4	B	S
3	5	A	S
4	6	A	f
2	6	A	f
1	6	A	S
2	7	B	S
3	8	A	S
4	9	B	S
5	10	A	f
3	10	A	S
4	11	none	failure

Now action perform of KMP flow chart with given text
T = "ACABAABABA"
n = length [T] = 10
m = length [P] = 6
q = 0

(i) i = 1, q = 0, S = 0
Comparing P[1] with T[1]
T → A C A B A A B A B A
↕
P → A B A B C B
P[1] matches T[1]. Since there is a match, P is not shifted

(ii) i = 2, q = 1
Comparing P[2] with T[2]
T → A C A B A A B A B A
↕
P → A B A B C B
P[2] does not match T[2], 'P' will be shift one positions to the right.

(iii) i = 3, q = 0, S = 1
Comparing P[1] with T[2]
T → A C A B A A B A B A
↕
P → A B A B C B
P[1] does not T[2], 'P' will be shift one position to the right.

(iv) i = 4, q = 0, S = z
Comparing P[1] with T[3]
T → A C A B A A B A B A
↕
P → A B A B C B
P[1] matches T[3]. Since there is a match, P is not shifted

(v) i = 5, q = 1, S = 2
Comparing P[2] with T[4]
T → A C A B A A B A B A
↕ ↓
P → A B A B C B
P[2] matches T[4]. Since there is a match, P is not shifted

(vi) i = 6, q = 2, S = 2
Comparing P[3] with T[5]
T → A C A B A A B A B A
↕
P → A B A B C B
P[3] matches T[5]. Since there is a matches, P is not shifted.

(vii) i = 7, q = 3, S = 2
Comparing P[4] with T[6]
T → A C A B A A B A B A
↕
P → A B A B C B

P[4] does not match T[6], 'P' will be shift one position to the right

(viii) i = 8, q = 0, S = 3

Comparing P[1] with T[4]

T → A C A B A A B A B A

↕

P → A B A B C B

P[1] does not match with T[4]. 'P' will be shift one position to the right

(ix) i = 9, q = 0, S = 4

Comparing P[1] with T[5]

T → A C A B A A B A B A

↕

P → A B A B C B

P[1] match T[5], Since there is a match, P is not shifted.

(x) i = 10, q = 1, S = 4

Comparing P[2] with T[6]

T → A C A B A A B A B A

↕

P → A B A B C B

P[2] does not match T[6], and now it is not possible to shift one position to the right

Thus, we get the pattern not match at any shifting.

Q.25 *What is the use of prefix function in KMP string algorithm? Explain with example.* ***(R.T.U. 2011)***

OR

What is prefix function and how is it computed for KMP–matching algorithm ? Give KMP–matching algorithm and compare it with naive string–matching algorithm.

(Raj. Univ. 2006, 2005, 2003, 2001, 1996)

Ans. For the given pattern, we first have to compute the prefix functions.

Step 1 : Compute-prefix-function

m = length [P] = 5

For q = 1,

π[1] = 0

initialize k = 0

For q = 2,

P[k + 1] = P[1] = a

P[q] = P[2] = a

∴ P[k + 1] = P[q],

So, k = k + 1 = 1

So, π[q] = π[2] = k = 1

For q = 3, k > 0,

P[k + 1] = P[2] = a

P[q] = P[3] = b

∴ P[k + 1] ≠ P[q]

So, k = π[k] = π[1] = 0

Now, P[k + 1] = P[1] = a

P[q] = P[3] = b

P[k + 1] ≠ P[q]

So, π[q] = π[3] = k = 0

For q = 4,

P[k + 1] = P[1] = a

P[q] = P[4] = a

⇒ P[k + 1] = P[q]

So, k = k + 1 = 1

and π[q] = π[4] = k = 1

For q = 5, k > 0,

P[k + 1] = P[2] = a

P[q] = P[5] = b

⇒ P[k + 1] ≠ P[q]

So, k = π[k] = π[1] = 0

Now, P[k + 1] = P[1] = a

P[q] = P[5] = b

P[k + 1] ≠ P[q]

So, π[5] = k = 0

Thus, we have the prefix function for the pattern as follows

q	1	2	3	4	5
P[q]	a	a	b	a	b
π[q]	0	1	0	1	0

Step 2 : After getting the prefix function we perform the matching of text T and pattern P.

n = length [T] = 17

m = length [P] = 5

q = 0

For i = 1,

P[q + 1] = P[1] = a

T[i] = T[1] = a

⇒ P[q + 1] = T[i]

So, q = q + 1 = 1

and q ≠ m

For i = 2,

P[q + 1] = P[2] = a

T[i] = T[2] = a

⇒ P[q + 1] = T[i]

So, q = q + 1 = 2

For i = 3,

P[q + 1] = P[3] = b

T[i] = T[3] = a

⇒ P[q + 1] ≠ T[i] and q > 0

So, q = π[q] = π[2] = 1

Still q > 0 and P[q + 1] = a = T[i]

So, q = q + 1 = 2

For i = 4,

P[q + 1] = P[3] = b

T[i] = T[4] = b

⇒ P[q + 1] = T[i]

So, q = q + 1 = 3

For i = 5,

P[q + 1] = P[4] = a

T[i] = T[5] = a

⇒ P[q + 1] = T[i]

So, q = q + 1 = 4

For i = 6,

P[q + 1] = P[5] = b

T[i] = T[6] = b

⇒ P[q + 1] = T[i]

So, q = q + 1 = 5

Here, q = m

So, pattern occurs with shift i – m = 1

and q = π[q] = π[5] = 0

For i = 7,

We have to search for another pattern match in the rest of the text

q = 0 from the last iteration

P[q + 1] = P[1] = a

T[i] = T[7] = a

⇒ P[q + 1] = T[i]

So, q = q + 1 = 1

For i = 8,

P[q + 1] = P[2] = a

T[i] = T[8] = a

⇒ P[q + 1] = T[i]

So, q = q + 1 = 2

For i = 9,

P[q + 1] = P[3] = b

T[i] = T[9] = b

⇒ P[q + 1] = T[i]

So, q = q + 1 = 3

For i = 10,

P[q + 1] = P[4] = a

T[i] = T[10] = a

⇒ P[q + 1] = T[i]

So, q = q + 1 = 4

For i = 11,

P[q + 1] = P[5] = b

T[i] = T[11] = a

⇒ P[q + 1] ≠ T[i] and q > 0

So, q = π[q] = π[4] = 1

Now P[q + 1] = P[2] = a

⇒ P[q + 1] = T[i]

So, q = q + 1 = 2

For i = 12,

P[q + 1] = P[3] = b

T[i] = T[12] = b

⇒ P[q + 1] = T[i]

So, q = q + 1 = 3

For i = 13,

P[q + 1] = P[4] = a

T[i] = T[13] = a

⇒ P[q + 1] = T[i]

So, q = q + 1 = 4

For i = 14,

P[q + 1] = P[5] = b

T[i] = T[14] = b

⇒ P[q + 1] = T[i]

So, q = q + 1 = 5

Here, q = m

So, pattern occurs with shift i – m = 14 – 5 = 19

and q = π[q] = π[5] = 0

For i = 15,

P[q + 1] = P[1] = a

T[i] = T[15] = a

⇒ P[q + 1] = T[i]

So, q = q + 1 = 1

For i = 16,

P[q + 1] = P[2] = a

T[i] = T[16] = a

⇒ P[q + 1] = T[i]

So, q = q + 1 = 2

For i = 17,

P[q + 1] = P[3] = b

T[i] = T[17] = b

⇒ P[q + 1] = T[i]

So, q = q + 1 = 3

This is the end of text, but q ≠ m so, no more patterns match the text. Thus, finally we have pattern matching the text with shift S = 1 and S = 9.

The basic idea is to slide the pattern towards the right along the string so that the longest prefix of P that we have matched, matches the longest suffix of T that we have already matched. If the longest prefix of P that matches a suffix of T is nothing, then we slide the whole pattern towards right. The algorithm computing prefix function is given as

Function Compute–Prefix (P) : The above function computes the prefix function, which determines the shifting of pattern matches within itself.

Step 1 : Initialization

set m ← length [P]
set P[0] ← 0
set k ← 0

Step 2 : Loop, computation of possible shifts

for q ← 2 to m, while (k > 0 and P[k + 1] ≠ P[q])
set k ← P[k]
if (P[k + 1] = P[q]) then
set k ← k + 1
set P[q] ← k

Step 3 : Return value at the point of call : return (P)

The above algorithm runs in O(m) amortized time. The Knuth-Morris-Pratt algorithm for matching the string is given below.

Procedure KMP Matcher (T, P) : The above procedure computes whether or the given pattern P is present in the text string T or not. It uses auxiliary function 'compute-prefix' to compute Pi.

Step 1 : Initialization

set n ← length [T] , set m ← length [P]
: **Calling Compute-Prefix**
Set Pi ← Compute-prefix (P)
: **Numbers of characters matched**
set q ← 0

Step 2 : Loop, scanning from left to right along text

for i ← 1 to n while (q > 0 and P [q + 1] ≠ T[i])
set q ← Pi [q]
: **No matching of next character**
if (P [q + 1] = T[i] then set q ← q + 1
: **Matching of next character**
if (q = m) then
: **Whether all character of P matched ?**
display "pattern occur with shift" i – m set q ← Pi[q]
: **look for next match**

The above algorithm has O(n + m) total running time.

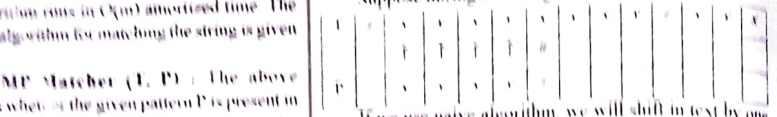
Knuth-Morris-Pratt Algorithm versus Naive

Linear time string matching was first discovered by Knuth, Morris and Pratt. They performed a rigorous analysis of naive algorithm and suggested a method to use the properties of string to be searched (pattern). Their method stores the information, which naive approach wasted, while scanning the text. To record the information in useful form it makes use of an auxiliary function π.

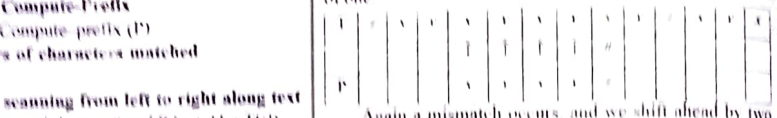
The basic idea is to observe the string to be matched (called pattern) and find if it has some repeated substrings (or prefixes specifically). Repeated prefix allows shifting of pattern by larger distance than naive approach during matching. To illustrate this concept, let us consider

P = x y x y z x y x

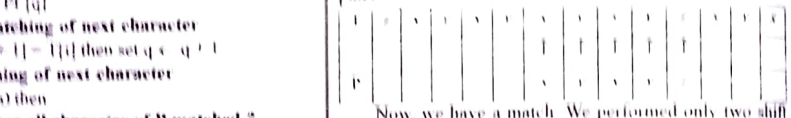
Suppose during matching, a mismatch occurs at 'z'.



If we use naive algorithm, we will shift in text by one character ahead. But here, till z, we had matched 'x y' then 'x y'. So we can align the first pair of x y in pattern with next pair of x y in text. Thus, we move ahead by two places instead of one.



Again a mismatch occurs, and we shift ahead by two places.



Now, we have a match. We performed only two shift operations instead of four. Thus, this technique can save time.

Obviously, the achievement over naive approach depends highly on the pattern and text. A pattern with longer prefix repetition will give a faster search if text also contains high frequency of that prefix.

❑❑❑

4 Assignment Problems & Randomized Algorithms

Chapter In A Nutshell

❑ Randomized Algorithms

A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation. A randomized algorithm uses a random number at least once during the computation to make a decision.

Randomized algorithms belong to the class of probabilistic algorithms. It uses a degree of randomness as part of its logic. This randomness can be in the input – implying a random running time. If we fix the running time then randomness is in the output. There is certain probability of its correctness, or a small probability of error in output.

❑ Min-Cut Problem

In this, we present a randomized algorithm for solving the min-cut problem of graphs. To understand the problem, we need some terms :

- A cut of a connected graph is obtained by dividing the vertex set V of a graph G into two sets V_1 and V_2, such that
  (i) There are no common vertices in V_1 and V_2. That is the two sets are disjoint.
  (ii) $V_1 \cup V_2 = V$. Every vertex of V is in V_1 or V_2.
- Any edge from V_1 to V_2 or V_2 to V_1 is said to be crossing the cut, if we remove all the edges which cross the cut, then the graph is divided into two graphs.
- All the edges which cross the cut collectively form the cut-set of the graph.
- A cut-set of minimum cardinality is called the minimum cutset of an unweighted graph. The cardinality of minimum cut set is called min-cut value of the graph G.

Though, definition of min-cut varies slightly for weighted graph.

❑ 2-SAT PROBLEM

It is another example of randomized algorithms – the algorithm for 2-Sat problem.

Satisfiability Problem

It is the problem of answering whether a boolean formula is **satisfiable** or not and boolean formula is in **Conjunctive Normal Form** (CNF). It is a collection of clauses in conjunction, each consisting of the disjunction of several literals. A conjunction is logical AND, and a disjunction is logical OR. It is similar to the product of sums (POS) form. A **literal** is either a boolean **variable** (such as x) or the negation of one (such as $\bar{x}$). Hence, there are two literals per variable.

A boolean formula is said to be satisfiable if there exists a truth assignment for which value of formula is true. A **truth assignment** is an assignment of false or true to each variable so that every clause contains a literal whose value is true. The SAT problem is about finding a satisfying truth assignment or else reporting that none exists.

Following is a boolean formula

$$(\bar{x}_1 + x_2)(x_3 + x_4 + \bar{x}_2)(x_1 + x_3 + \bar{x}_2)(x_3 + \bar{x}_4)$$

Sometimes, $\vee$ symbol is used to show logical-OR operation, and the symbol $\cup$ is used to show logical-AND operation. Like, following formula

$$(\bar{x}_1 \vee x_2) \wedge (x_3 \vee x_4 \vee \bar{x}_2) \wedge (x_1 \vee x_3 \vee \bar{x}_2) \wedge (x_3 \vee \bar{x}_4)$$

Generally, there is no constraint on number of literals a clause can contain.

If a boolean formula, written in CNF contains exactly two literals per clause, then it is called **2-CNF formula**. Its corresponding satisfiability problem is called **2-SAT problem**.

PREVIOUS YEARS QUESTIONS

PART-A

Q.1 What do you mean by randomized algorithms

Ans. A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

Q.2 What is satisfiability problem

Ans. It is the problem of answering whether a boolean formula is **satisfiable** or not and boolean formula is in **Conjunctive Normal Form** (CNF), if it is a collection of clauses in conjunction, each consisting of the disjunction of several literals.

Q.3 Write the advantages of randomized algorithms

Ans

- Randomized algorithms are better than probabilistic analysis.
- In randomized algorithm, we randomize in the algorithm, not in the input distribution.
- For most of randomized algorithms, no particular input elicits its worst - case behaviour.

Q.4 Define Literals

Ans. A **literal** is either a boolean **variable** (such as x_2) or the negation of one (such as $\bar{x}_2$). Hence, there are two literals per variable.

Q.5 Write disadvantages of randomized algorithms

Ans. Disadvantages of Randomized Algorithm

- The randomized algorithm performs badly when the random number generator produces an "unlucky" permutation.
- Randomizing the input takes some additional time.

PART-B

Q.6 Write short note on Quadratic assignment problem. *(R.T.U. 2018, 20...)*

OR

Explain the quadratic assignment problem with suitable example. *(R.T.U. 2016, 20...)*

Ans. Quadratic Assignment Problem (QAP)

The Quadratic Assignment Problem (QAP) is one of fundamental combinatorial optimization problems in the field of optimization or operations research in mathematics, from the category of the facilities location problems.

The problem models the following real life problem:

There are a set of n facilities and a set of n locations. For each pair of locations, a distance is specified and for each pair of facilities a weight or flow is specified (e.g., the amount of supplies transported between the two facilities). The problem is to assign all facilities to different locations with the goal of minimizing the sum of the distances multiplied by the corresponding flows.

Intuitively, the cost function encourages factories with high flows between each other to be placed close together.

The problem statement resembles that of the assignment problem; only the cost function is expressed in terms of quadratic inequalities, hence the name.

It is stated as:

"If there are n locations and n facilities and each facility is assigned to only one location at a time then the quadratic assignment problem is to obtain minimum cost. This cost can be computed using distance between two locations and flow between two facilities" e.g.

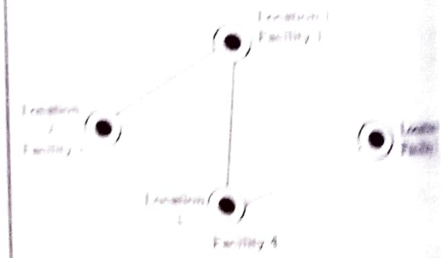


Fig.

Here

- Facility 1 is assigned to location 1
- Facility 2 is assigned to location 2
- Facility 3 is assigned to location 4
- Facility 4 is assigned to location 3

The lines between two facilities represent the flow between these two facilities. Suppose

dist (1, 2) = 10 dist (1, 4) = 21

dist (1, 3) = 40 dist (3, 4) = 15

The required flow between the facilities is

flow (1, 2) = 1

flow (1, 4) = 4

flow (1, 3) = 1

flow (3, 4) = 2

Then total cost $Z = \sum_{i,j} \text{dist}(i, j) * \text{flow}(i, j)$

$Z = 10 \times 1 + 40 \times 1 + 21 \times 4 + 15 \times 2$

$= 10 + 40 + 84 + 30 = 164$

The objective is to find best possible permutation in order to obtain minimum cost.

The Quadratic Assignment Problem (QAP) is NP hard problem and there is no algorithm for solving this problem in polynomial time.

Q.7 Explain Randomized min cut theorem with suitable example. *(R.T.U. 2018, 2016)*

OR

State the Randomized min cut theorem. *(R.T.U. 2014)*

Ans. Randomized Algorithm

Algorithm min_cut_randomized

1 Input a multi graph G

2 While |V| > 2 do

3 Pick any edge e randomly and contract it

4 Remove self loop

5 End while

6 Return the set of the edges |E|

Explanation :

Let G=(V, E) be a multigraph with n vertices and m edges. And we know that a cut is a set of edges which cuts the graph into two connected components.

The minimum cut is the cut of minimum size.

The minimum cut has size at most the minimum degree of any node. The minimum degree can be much larger than the size of the minimum cut.

To determine the min-cut

(a) We pick an edge uniformly and merge the two vertices at its end points.

(b) There are several edges between some pairs of newly formed vertices.

(c) Edges between vertices that merged are removed, so that there is no any self loop.

(d) With each contraction the no. of vertices of G decreased by one.

Note

- In the process of determining the min cut removing self loops does not have any effect on the size of the min cut.
- Edge contraction can only increase the size of the min cut.

Examples

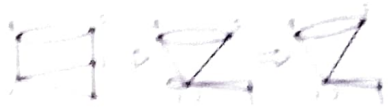


Fig.

Q.8 Explain Las Vegas algorithm with example. *(R.T.U. 2016, 2015)*

OR

Compare Las vegas and Monte carlo algorithm approaches *(R.T.U. 2017, 2015)*

OR

What do you mean by randomized algorithms. Explain Las Vegas algorithms and Monte Carlo algorithms with suitable example. *(R.T.U. 2016, 2015)*

Ans. Randomized Algorithm : In order to use probabilistic analysis, we can use probability and randomness as a tool for algorithm design and analysis, by making the behavior of part of the algorithm random.

For example, in hiring problem, it may seem as if the candidates are being presented to us in a random order, but we have no way of knowing whether or not they really are.

I.e. we call an algorithm randomized if its behavior is determined not only by its input but also by values produced by a random-number generator.

If RANDOM (a,b) is a random number generator then

RANDOM (0,1) returns either 0 or 1, each with probability 1/2.

Example

Randomized_Hire_Algo(n)

//n is total no. of candidates

1 Randomly permute the list of candidates

2. Best = 0 //candidate 0 is a least-qualified dummy candidate.
3. For i = 1 to n
4. Interview candidate i
5. If candidate i is better than candidate best
6. Best = i
7. Hire candidate i

Las Vegas Algorithm

A randomized algorithm is called Las Vegas if its output is always correct but its running time is a random variable. Randomized quicksort is an example of Las Vegas algorithm. Its output is always a sorted table, but the running time is random.

Usually the analysis of a Las Vegas algorithm tries to bind the expected running time, or bound the running time with high probability.

Example

- Average running time analysis assumes some distribution of problem instances.
- Robinhood effect

LV "steal" time from the "rich" instance – instances that were solved quickly by deterministic algo -- to give it to the "poor" instance.

Reduce the difference between good and bad instances.

In computing, a Las Vegas algorithm is a randomized algorithm that always gives correct results; that is, it always produces the correct result or it informs about the failure. In other words, a Las Vegas algorithm does not gamble with the verity of the result; it gambles only with the resources used for the computation. A simple example is randomized quicksort, where the pivot is chosen randomly, but the result is always sorted. The usual definition of a Las Vegas algorithm includes the restriction that the expected run time always be finite, when the expectation is carried out over the space of random information, or entropy, used in the algorithm.

Las Vegas algorithms were introduced by László Babai in 1979, in the context of the graph isomorphism problem, as a stronger version of Monte Carlo algorithms. Las Vegas algorithms can be used in situations where the number of possible solutions is relatively limited, and where verifying the correctness of a candidate solution is relatively easy while actually calculating the solution is complex.

Monte Carlo Algorithm

Randomized algorithms are those in which we consider some variables for time and resources and are called randomized so that we could compute the desired result.

Monte Carlo and Las Vegas are such algorithms which uses the randomized algorithm to calculate the result.

In Las Vegas we are sure to get a correct output but we have to keep in mind that the expected running time is finite. Las Vegas does not gamble with the correctness of result it only gambles with the resources used.

In Monte Carlo there is a boundation on running time and we are not sure to get a 100% correct result. It gambles with the result but it have to keep in mind the time allotted.

Randomized Quick Sort (S)

(1) We choose an element y from S. In this, each element in S have equal probability of being chosen.

(2) Now we divide the S sequence in two parts one containing elements smaller than y (S_1) and other (S_2) containing element greater than y.

(3) We recursively call random sort (S) for sequences S_1 and S_2.

(4) We place the elements after sorting like elements of S_1, y and then elements of S_2.

(5) Exit.

Randomized algorithm deals or gambles with the value choosen, it could be time as well as resources.

Q.9 ***Solve $f = (x_1 \vee \bar{x}_2)(x_3 \vee \bar{x}_4)(\bar{x}_1 \vee x_3)(x_4 \vee x_6)$ using randomized algorithm.*** ***[R.T.U. 2017]***

Ans. Using Randomized Algorithm

$f = (x_1 \vee \bar{x}_2)(x_3 \vee \bar{x}_4)(\bar{x}_1 \vee x_3)(x_4 \vee x_6)$

Pick x_1 at random, so $T = \{x_1\}$

Put x_1 = True

Remove clauses centering x_1, which is $(x_1 \vee \bar{x}_2)$

Now, $f = (x_3 \vee \bar{x}_4)(\bar{x}_1 \vee x_3)(x_4 \vee x_6)$

due to $\bar{x}_1$, force variables are x_3

$T = \{x_3\}$

Put x_3 = True

Remove clauses centering x_3, which $(\bar{x}_1 \vee x_3) \equiv (x_3 \vee \bar{x}_4)$

Now, $f = (x_4 \vee x_6)$

There are no forced variable due to $\bar{x}_3$

Pick x_4 at random

$T = \{x_4\}$

Remove clauses centering x_4, which $(x_4 \vee x_6)$

Now f = ture

So, stop truth assignment is

{False, True, True, True, False}

which represent $\{x_1, x_2, x_3, x_4, x_6\}$.

Q.10 ***State the assignment problem and solve the following assignment problem using branch and bound for which cost matrix is given below.***

$$Cost = \begin{matrix} 4 & 7 & 5 \\ 2 & 6 & 1 \\ 3 & 9 & 8 \end{matrix}$$

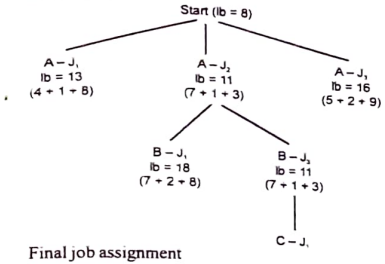
[R.T.U. 2017]

Ans.

	J_1	J_2	J_3
A	4	7	5
= B	$\bar{2}$	6	1
C	3	9	$\bar{8}$

lb = 4 + 1 + 3 = 8

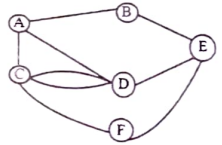
lower bound



Final job assignment

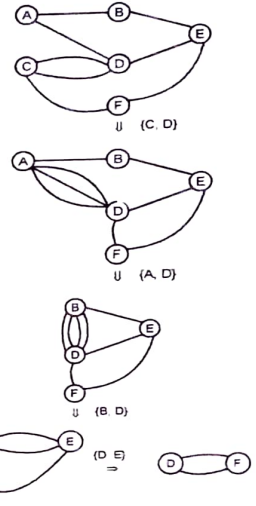
	J_1	J_2	J_3
A	4	$\boxed{7}$	5
B	2	6	$\boxed{1}$
C	$\boxed{3}$	9	8

Q.11 ***Give randomized algorithm for min cut of the following graph.***



[R.T.U. 2017]

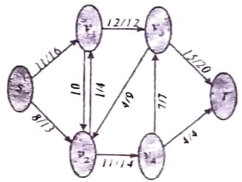
Ans. Randomized algorithm for min cut



Min Cut = 2

V = {F} ∪ {A, B, C, D, E}

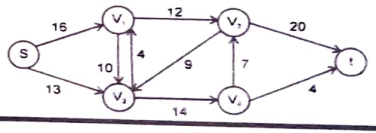
Q.12



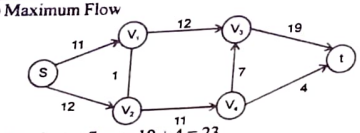
(a) Find Maximum flow in above network.

(b) Find the corresponding minimum cut and check that its capacity is same as that value of maximum flow found in a) part. ***[R.T.U. 2017]***

Ans.

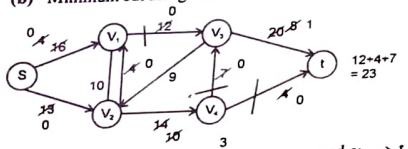


(a) Maximum Flow



Maximum flow = 19 + 4 = 23

(b) Minimum cut using ford fulkerson's algorithm



In residual graph, $v_1 \rightarrow v_3$, $v_3 \rightarrow v_4$ and $v_4 \rightarrow t$ has residual capacity of 0.

Hence minimum cut

$= v_1 \rightarrow v_3, v_3 \rightarrow v_4, v_4 \rightarrow t$

Flow Through min cut = 12 + 7 + 4 = 23

Q.13 ***Solve the given assignment problem by branch and bound method.***

	Job 1	*Job 2*	*Job 3*	*Job 4*
Person 1	*9*	*2*	*7*	*8*
Person 2	*6*	*4*	*3*	*7*
Person 3	*5*	*8*	*1*	*8*
Person 4	*7*	*6*	*2*	*4*

[Note: Consider person a, person b, person c, person d.]

[R.T.U. 2016]

Ans.

	Job 1	Job 2	Job 3	Job 4
Person a	9	2	7	8
Person b	6	4	3	7
Person c	5	8	1	8
Person d	7	6	2	4

Lower bound : Any solution to this problem will have total cost at least: 2 + 3 + 1 + 2 (or 5 + 2 + 1 + 4)

First two level of the state-space tree-

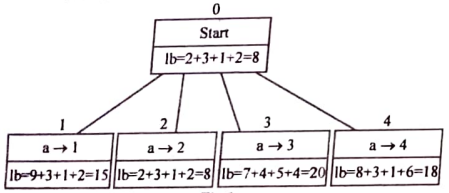


Fig.1

Figure 1 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.

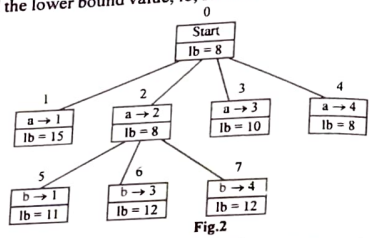


Fig.2

Figure 2 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.

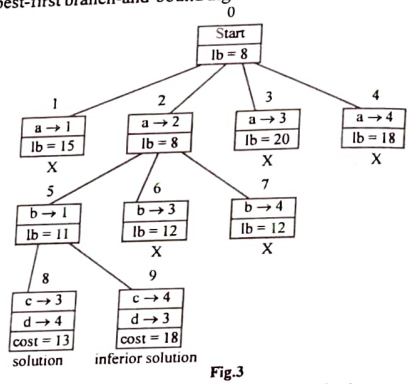


Fig.3

Figure 3 complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

Q.14 ***Give a randomized solution for Min-cut of following graph.***

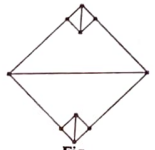
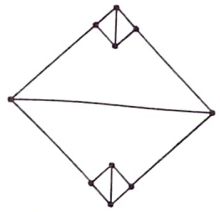
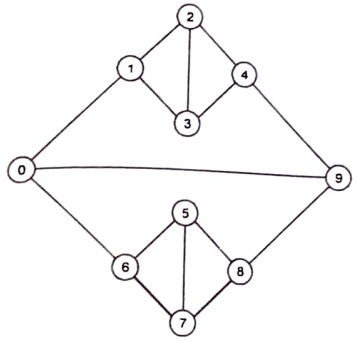


Fig. ***[R.T.U. 2015]***

Ans. Given graph is :

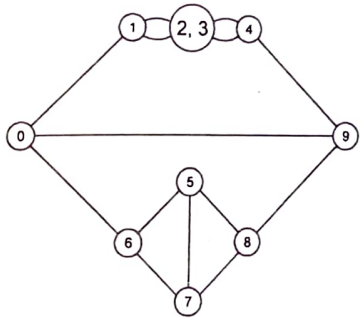


To solve this we have to assign a number to each vertex.

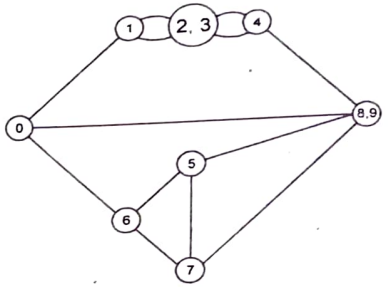


To solve this by randomization we have to reduce random edges and also remove self loops if aapeared. Now we reduce the modified graph as follows :

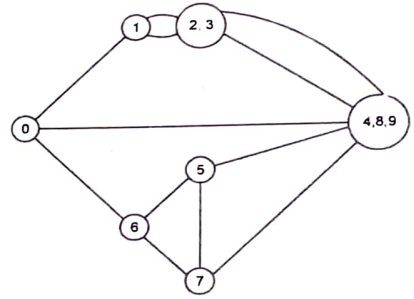
1. Contracting Edge 2 – 3 so graph is



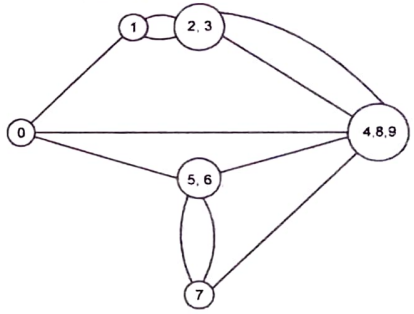
2 Contracting Edge 8 – 9



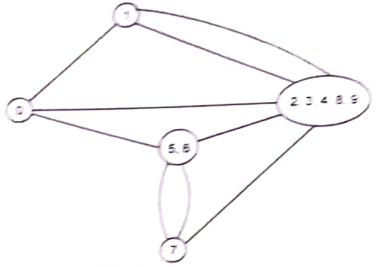
3. Contracting Edge 4 – 9



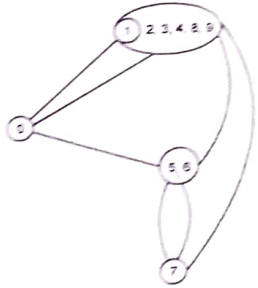
4. Contracting Edge 6 – 5



5 Contracting Edge 2 – 4

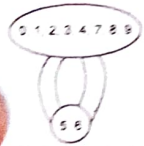


6 Contracting Edge 1 – 2



7 Contracting Edge 0 – 9

8 Contracting Edge 7 – 8



This is the final graph after applying Randomized Karger's Algorithm.

Q.15 *Solve* $f = (x_1 \vee \bar{x}_2)(x_2 \vee \bar{x}_4)(\bar{x}_1 \vee x_2)(x_2 \vee \bar{x}_5)(x_4 \vee x_6)(x_4 \vee \bar{x}_6)$ *using a randomized algorithm.* *[R.T.U. 2015]*

Ans. $f = (x_1 \vee \bar{x}_2)(x_2 \vee \bar{x}_4)(\bar{x}_1 \vee x_2)(x_2 \vee \bar{x}_5)(x_4 \vee x_6)(x_4 \vee \bar{x}_6)$

Pick x_1 at random. So, $T = \{x_1\}$

Put x_1 = True

Remove clauses containing x_1, which is $(x_1 \vee \bar{x}_2)$

Now, $f = (x_2 \vee \bar{x}_4)(\bar{x}_1 \vee x_2)(x_2 \vee \bar{x}_5)(x_4 \vee x_6)(x_4 \vee \bar{x}_6)$

due to $\bar{x}_1$, forced variables are, x_2

$T = \{x_2\}$

Put x_2 = True

Remove clauses containing x_2, which is $(\bar{x}_1 \vee x_2)$ and $(x_2 \vee \bar{x}_4)$

Now, $f = (x_2 \vee \bar{x}_5)(x_4 \vee x_6)(x_4 \vee \bar{x}_6)$

There are no forced variables due to $\bar{x}_5$

Pick x_4 at random

$T = \{x_4\}$

Remove clauses containing x_4, which is $(x_4 \vee x_6).(x_4 \vee \bar{x}_6)$

$f = (x_2 \vee \bar{x}_5)$

Taking x_2 as true, $T = \{x_2\}$

Now f = true

So stop, truth assignment is {False, True, True, True, False, False} which represent $\{x_1, x_2, x_3, x_4, x_5, x_6\}$

Q.16 *Show Randomized min cut operation for the following given graph:*

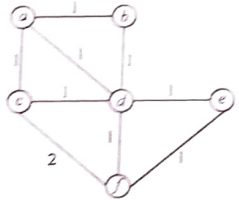


Fig.

[R.T.U. 2014]

Ans. Now, we apply given algorithm to find randomized min-cut.

Min-Cut(G)

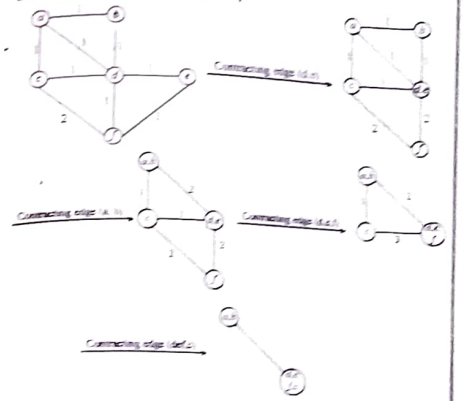
Step 1 : We first apply a while loop as follows

While | V | > 2

do selected an edge (x, y) belonging to E uniformly at random. That is, we apply a while loop if vertex are greater than two for the given graph

While | V | > 2 true since there are more then two vertex in the given graph.

Step 2 : Here, we contract the edge (x, y) which we select after applying step 1 while loop



Randomized Min-Cut

Q.17 *Solve the given assignment problem by branch and bound method.*

Tasks ⇒ / Agents ⇓	1	2	3
A	14	17	13
B	12	16	11
C	13	19	14

[R.T.U. 2014]

Ans.

Step 1: Calculate upper bound by sequential assignment

Assign task 1 to Agent A cost = 14

Assign task 2 to agent B cost = 16

Assign task 3 to agent C cost = 14

Upper bound = 14 + 16 + 14 = 44

Step 2: Lower bound = 12 + 16 + 11 = 39

Step 3: Reduce the cost matrix

Row a = Row a – 13

Row b = Row b – 11

Row c = Row c – 13

	1	2	3
a	1	0	0
b	1	1	0
c	0	2	1

Middle column does not have a zero

So column 2 = column 2 – 4

The resulting reduced matrix is

RC =		1	2	3
	a	1	0	0
	b	1	1	0
	c	0	2	1

The total amount reduced = 13 + 11 + 13 + 4 = 41

We construct a tree with root labeled with bound value 41 from here we have choices agent A, task 1, 2 or 3

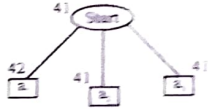
bound values are

41 + RC[a, 1] = 41 + 1 = 42

41 + RC[a, 2] = 41 + 0 = 41

41 + RC[a, 3] = 41 + 0 = 41

tree looks like

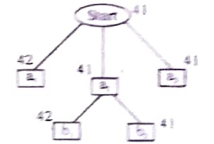


select minimum leaf node to expand which is a_2 and a_3

We have two choices left for b, either or 3 bound values are

$lb(a_2)$ = RC[b, 1] = 41 + 1 = 42

$lb(a_2)$ = RC[b, 3] = 41 + 0 = 41



Selecting the minimum node we get the solution, because it is at lowest level. The solution is assign task 2 to agent A, task 3 to agent B and task 1 to agent C.

X is (corresponding solution matrix)

X =		1	2	3
	a	1	0	0
	b	0	0	1
	c	1	0	0

Q.18 What is flow shop scheduling problem? Schedule two jobs on 4 machines using flow shop scheduling technique. The time required by each operation of these jobs is given by following matrix.

[illegible]

Ans. Flow shop scheduling problems, are a class of scheduling problems with a work shop or group shop in which the flow control shall enable an appropriate sequencing for each job and for processing on a set of machines or with other resources [illegible] in compliance with given processing orders. Especially the maintaining of a continuous flow of processing tasks is desired with a minimum of idle time and a minimum of waiting time. Flow shop scheduling is a special case of job shop scheduling where there is strict order of all operations to be performed on all jobs. Flow shop scheduling may apply as well to production facilities as to computing designs.

A special type of flow shop scheduling problem is the permutation flow shop scheduling problem in which the processing order of the jobs on the resources is the same for each subsequent step of processing.

As given in problem, there are 4 machines. The flow shop scheduling for these operations is as shown below.

Fig.

Fig.

Q.19 Explain the flow networks and augmenting paths.

Ans. A flow network [illegible] is a directed graph in which each edge [illegible] has a non-negative capacity [illegible].

If [illegible], we assume that [illegible]. We distinguish two vertices in a flow network, a source s and a sink t. For convenience, we assume that every vertex lies on some path from the source to the sink. That is, for every vertex [illegible], there is a path [illegible]. The graph is therefore connected, and [illegible].

We are now ready to define flows more formally. Let [illegible] be a flow network with a capacity function c. Let s be the source of the network and let t be the sink. A flow in G is a real-valued function [illegible] that satisfies the following three properties:

Capacity constraint: For all [illegible], we require [illegible]

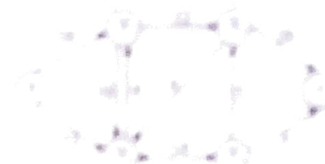
Skew symmetry: For all [illegible], we require [illegible]

Flow conservation: For all [illegible], we require [illegible]

The quantity f(u, v), which can be positive, zero or negative, is called the flow from vertex u to vertex v. The value of a flow f is defined as

[illegible]

That is, the total flow out of the source.



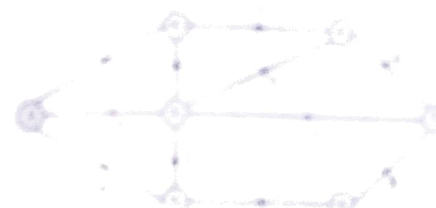
Given a flow network [illegible] and a flow f, an augmenting path p is a simple path from s to t in the residual network [illegible]. By the definition of the residual network, each edge [illegible] on an augmenting path admits some additional positive flow from u to v without violating the capacity constraint on the edge.

The shaded path in following figure is an augmenting path. Treating the residual network [illegible] in the figure as a flow network, we can increase the flow through each edge of this path by up to [illegible] units without violating a capacity constraint, since the smallest residual capacity on this path is [illegible]. We call the maximum amount by which we can increase the flow on each edge in an augmenting path p the residual capacity of p, given by

[illegible]

Q.20 Construct the Min-Cut for the following network.

Ans.

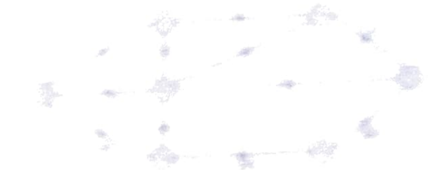


Step 1 : Consider path [illegible] from s to t

Calculate [illegible] (flow augmented path)

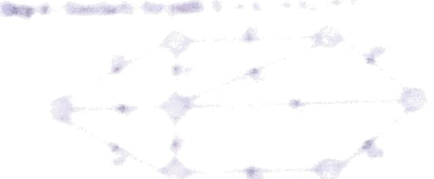
[illegible]

Step 2 : [illegible]

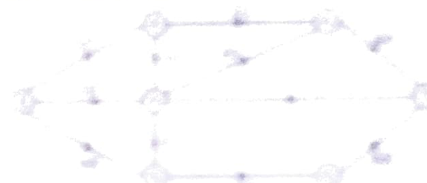


Step 3 : Similarly for path [illegible]

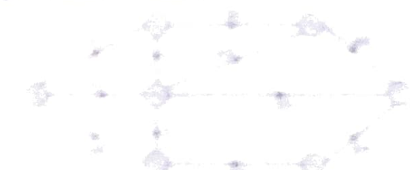
Step 4 : Similarly for path [illegible]



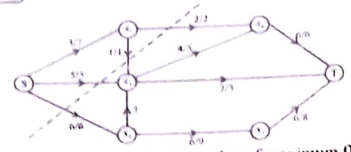
Step 5 : Similarly for path [illegible]



Step 6 : Similarly for path [illegible]



Now for minimum cut, we will identify all arcs that are carrying a flow equal to their maximum flow capacity. These are [illegible]. Then, the min-cut will be [illegible]



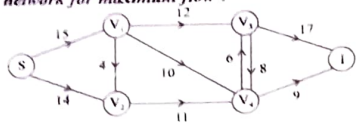
Also, capacity of minimum cut = value of maximum flow

6 + 5 + 1 + 2 = 6 + 5 + 3

(min-cut capacity) (max-flow value)

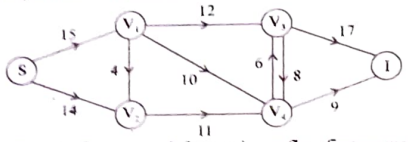
14 = 14

Q.21 ***Define flow networks and solve the following flow network for maximum flow :***



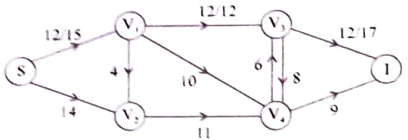
[R.T.U. 2011, Raj. Univ. 2006]

Ans. Flow Networks : Flow networks define the flow of the network. Where forward arrow defines the forward flow (source to destination) and backward arrow defines backword flow (destination to source).

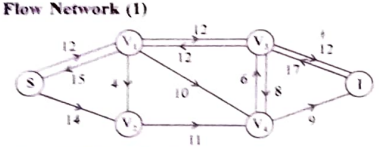


To make flow network for maximum flow first we make residual network.

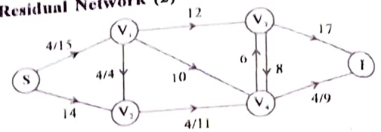
Residual Network (1)



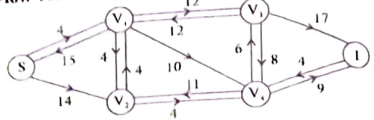
Flow Network (1)



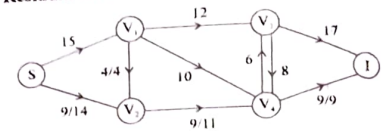
Residual Network (2)



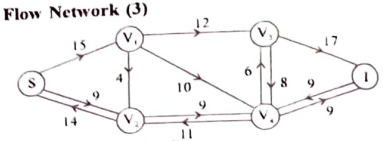
Flow Network (2)



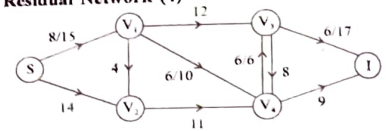
Residual Network (3)



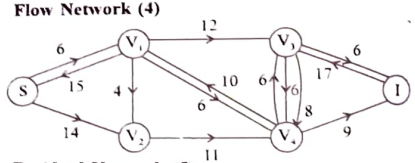
Flow Network (3)



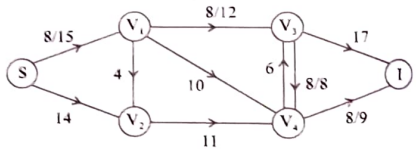
Residual Network (4)



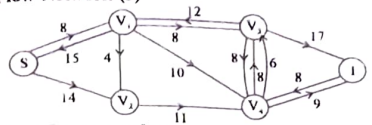
Flow Network (4)



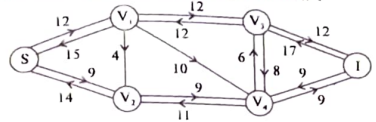
Residual Network (5)



Flow Network (5)



Final flow network with maximum flow :



Q.22 ***Write short note on Bi-quadratic Assignment Problem.*** *[R.T.U. 2011]*

Ans. Biquadratic Assignment Problem : A generalization of the QAP is the Biquadratic assignment problem denoted BiQAP, which is essentially a quadratic assignment problem with cost coefficient formed by the products of two four-dimensional arrays. More specifically, consider two $n^4 \times n^4$ arrays, $F = (f_{ijkl})$ and $D = (d_{mpst})$. The BiQAP can then be stated as

$$\min \sum_{i,j=1}^{n} \sum_{k,\ell=1}^{n} \sum_{m,p=1}^{n} \sum_{s,t=1}^{n} f_{ijkl} d_{mpst} x_{im} x_{jp} x_{ks} x_{lt}$$

$$\text{such that } \sum_{i=1}^{n} x_{ij} = 1,\ j = 1, 2, \ldots, n$$

$$\sum_{j=1}^{n} x_{ij} = 1,\ i = 1, 2, \ldots, n$$

$$x_{ij} \in \{0, 1\},\ i, j = 1, 2, \ldots, n$$

The major application of the BiQAP arises in Very Large Scale Integrated (VLSI) circuit design. The majority of VLSI circuits are sequential circuits and their design process consists of two steps : first, translate the circuit specifications into a state transition table by modeling the system using finite state machines and secondly, try to find an encoding of the states such that the actual implementation is of minimum size. Equivalently, the BiQAP can be stated as :

$$\min_{\phi \in S_n} \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} \sum_{\ell=1}^{n} f_{ijkl} d_{\phi(i)\phi(j)\phi(k)\phi(\ell)}$$

where, $\int n$ denotes the set of all permutations of $N = \{1, 2, \ldots, n\}$. All different formulations for the QAP can be extended to the BiQAP, as well as most of the linearizations that have appeared for the QAP. The computational results showed that these bounds are weak and deteriorate as the dimension of the problem increases. This observation suggests that branch and bound methods will only be effective on very small instances. For larger instances, efficient heuristics, that find good-quality approximate solutions, are needed. Several heuristics for the BiQAP have been developed, in particular deterministic improvement methods and variants of simulated annealing and tabulated search algorithms.

Computational experiments on test problems of size up to n = 32, with known optimal solutions suggest that one version of simulated annealing is best among those tested. The GRASP heuristic has also been applied to the BiQAP and produced the optimal solution for all the test problems.

PART-C

Q.23 ***Discuss the formulation of simple assignment problem of size n.*** *[R.T.U. 2018, 2015]*

Ans. Assignment problems deal with the question how to assign n objects to m other objects in an injective fashion in the best possible way. An assignment problem is completely specified by its two components: the assignments which represent the underlying combinatroial structure, and the objective function to be optimized which models "the best possible way".

In the classical assignment problem one has m = n and most of the problem with m > n can be transformed or are strongly related to a analogous problems with m = n.

From the mathematical point of view assignment is a objective mapping of a finite set $N = \{1, 2, \ldots, n\}$ into itself, i.e., a permutation ϕ assigning some $j = \phi(i)$ to each $i \in N$. The set of all permutations (assignments) of n items will be denoted by S_n and n! elements. Every permutation ϕ of the set $N = \{1, , n\}$ corresponds uniquely to a permutation matrix $X_\phi = (x_{ij})$ with $x_{ij} = 1$ for $j = \phi(i)$ and $x_{ij} = 0$ for $j \neq \phi(i)$. Thus a permutation matrix $X = (x_{ij})$ can be defined as a matrix which fulfills the following conditions, so called assignment constraints.

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \text{for all } j = 1, \ldots, n$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \text{for all } i = 1, \ldots, n$$

$x_{ij} \in \{0, 1\}$ for all i, j = 1, . . . , n

By replacing the conditions $x_{ij} \in \{0, 1\}$ by $x_{ij} \geq 0$ in (1), we get doubly stochastic matrix. The set of all doubly matrices forms the assignment polytope P_A. Due to a famous result of (Birkhoff, 1946), the assignment polytope P_A is the convex hull of all assignments, or equivalently, every doubly stochastic matrix can be written as convex combination of permutation matrices.

The concept of an assignment is strongly related to another well known concept in graph theory and in combinatorial optimization, matching in bipartite graphs. A bipartite graph G is a triple (V, W, E), where the vertex sets V and W have no vertices in common and the edge set E is called a matching, if every vertex of G is incident with at most one edge from M. The cardinality of M is called cardinality of matching. The maximum matching problem asks for a matching with as many edges as possible. A matching M is called a perfect matching, if every vertex of G is incident with exactly one edge from M. Evidently, every perfect matching is a maximum matching. A perfect matching in a bipartite graph $G = (V, W, E)$ with $V = \{v_1, v_2, \ldots, v_n\}$, $W = \{w_1, w_2, \ldots, w_n\}$ can be represented by a permutation ϕ_m of $\{1, 2, \ldots, n\}$ such that $\phi_m(i) = j$ if and only if $(v_i, w_j) \in M$. Hence, a perfect matching in a bipartite graph is an assignment.

(Hopcroft and Karp, 1973) gave an $O(|E|\sqrt{|V|})$ algorithm which constructs a perfect matching if it exists. (Even and Tarjan, 1975) gave an $O(|E|\sqrt{|V|})$ algorithm for the maximum flow problem on unit capacity simple net works, algorithm which can also be applied to find a matching of maximum cardinality in a bipartite graph. (Alt et al., 1991) gave an $O\left(|V|^{1.5}\sqrt{|E|/\log|V|}\right)$ implementation for the Hopcroft-Karp algorithm. Based on ideas similar to theose in (Hopcroft adn and Karp, 1973), a fast randomized Monte-Carlo algorithms is given by (Mulmuley et al., 1087).

LAP (Linear Assignment Problem)

The Linear Assignment Problem (LAP) is one of the oldest and most studies problems in combinatorial optimization. Many different algorithms have been developed to solve this problem. Also other aspects of the problem as the asymptotic behavior or special cases have been thoroughly investigated. The reader is referred to (Dell'Amico and Martello, 1997) for a comphrehensive annotated bibligraphy and to (Burkard and Cela, 1999) for a recent review on assignment problems.

Recall the original model where n items are to be assigned to n other objects in the best possible way. Let c_{ij} be the cost incurred by the assignment of object i to object j. We are looking for an assignment ϕ which minimizes the overall cast $\sum_{i=1}^{n} c_{i\phi(i)}$. Thus, the Linear Assignment Problem (LAP) is given as follows

$$\min_{\phi \in S_n} \sum_{i=1}^{n} c_{i\phi(i)}$$

Where S_n is the best of permutations of {1, 2, ..., n} Based on the description (1) of the set of all assignment, the LAP can also be formulated as follows:

min $\sum_{ij} c_{ij} x_{ij}$ over all matrix $X = (x_{ij})$ which fulfill (1).

Due to Birkhoff's result, we can relax the conditions $x_{ij} \in \{0, 1\}$ to $x_{ij} \geq 0$ and obtain the linear programming formulation of the LAP Any basic solution of this linear program corresponds to a permutation matrix.

$$\min \sum_{i=1}^{n} c_{ij} x_{ij}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad j = 1, \ldots, n$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, \ldots, n$$

$$x_{ij} \geq 0 \qquad i, j = 1, \ldots, n$$

As we will mention in the next section many algorithms for the LAP are based on linear programming techniques and consider often the dual linear program.

$$\max \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} u_j$$

$$u_i + u_j \leq c_{ij} \qquad i, j = 1, \ldots, n$$

$$u_i + u_j \in \mathbb{R} \qquad i, j = 1, \ldots, n$$

where u_i and u_j, $1 \leq i, j \leq n$, are dual variables.

Among the numerous applications of the LAP the so-called personal assignments are the most typical. In the personal assignment we want to assign people to objects, e.g. jobs, machines, rooms, to other people etc. Each assignment has a "cost" and we want to make the assignment so as to minimize the overall sum of the costs. For example one company might want to assign graduated to vacant jobs. In this case, the cost c_{ij} is given by $c_{ij} = -p_{ij}$ where p_{ij} is the proficiency index for placing candidate i to job j, and the goal is to assign each candidate i to some vacancy $\phi(i)$ such that the overall cost $\sum_i c_{i\phi(i)}$ is minimized, or equivalently, the overall proficiency $\sum_i p_{i\phi(i)}$ is maximized.

There are many other applications of the linear assgnment problem e.g. in locating and tracing objects in space, scheduling on parallel machines, inventory planning, vehicle and crew scheduling, wiring of typewriters etc. The reader is referred to (Ahuja et al., 1995) and (Burkard adn Cela, 1999) for a detailed description of some applications of the LAP and literature pointers to other applications.

So an assignment problem is a particular case of transportation problem where the objective is to assign a number of resources to an equal number of activities so as to minimize total cost or maximize total profit of allocation. The problem of assignment arises because available resources such as men, machines, etc. have varying degrees of efficiency for performing different activities. Therefore, cost, profit or time of performing the different activities isdifferent. Thus, the problem is how the assignments should be made so as to optimize the given objective. The assignment problem is one of the fundamental combinatorial optimization problems in the branch of optimization or operations research in Mathematics. It consists of finding amaximum weight matching in a weighted bipartite graph.

Solution of the Assignment Problem

So far, in the literature it is available that an assignment problem can be solved

by the following four methods.

I. Enumeration method
II. Simplex method
III. Transportation method
IV. Hungarian method

Here, we discuss each of four, one by one

I. Enumeration method

In this method, a list of all possible assignments among the given resources (likemen, machines, etc.) and activities (like jobs, sales areas, etc.) is prepared. Then an assignment involving the minimum cost (or maximum profit), time or distance is selected. If two or more assignments have the same minimum cost (ormaximum profit), time or distance, the problem has multiple optimal solutions.

In general, if an assignment problem involves n workers/jobs, then there are in total n! Possible assignments. As an example, for an n=3 workers/jobs problem, we have to evaluate a total of 3! or 6 assignments. However, when n is large, the method is unsuitable for manual calculations. Hence, this method is suitable only for small n.

II. Simplex Method

Since each assignment problem can be formulated as a 0 or 1 which becomes integer linear programming problem. Such a problem can be solved by the simplex method also. As can be seen in the general mathematical formulation of the assignment problem, there are n×n decision variables and n+n or 2n equalities. In particular, for a problem involving 5 workers/jobs, there will be 25 decision variables and 10 equalities. It is, again, difficult to solve manually.

III. Transportation Method

Since an assignment problem is a special case of the transportation problem, it can also be solved by transportation methods. However, every basic feasible solution of a general assignment problem having a square payoff matrix of ordern should have more m+n-1= n+n-1= 2n-1 assignments. But due to the special structure of this problem, any solution cannot have more than n assignments. Thus, the assignment problem is inherently degenerate. In order to remove degeneracy, (n-1) number of dummy allocations will be required in order to proceed with the transportation method. Thus, the problem of degeneracy teach solution makes the transportation method computationally inefficient for solving an assignment problem.

IV. Hungarian Method

Assignment problems can be formulated with techniques of linear programming and transportation problems. As it has a special structure, it is solved by the special method called Hungarian method. This method was developed by D. Konig, a Hungarian mathematician and is therefore known as the Hungarian method of assignment problem. In order to use this method, one needs to know only the cost of making all the possible assignments. Each assignment problem has a matrix (table) associated with it. Normally, the objects (or people) one wishes to assign are expressed in rows, whereas the columns represent the tasks (or things) assigned to them. The number in the table would then be the costs associated with each particular assignment.

Q.24 ***Write and explain Ford Fulkerson algorithm.*** *[R.T.U. 2017]*

OR

What do you mean by Multi-Commodity flow in the network? Find the max flow path by Ford-Fulkerson method for given network.

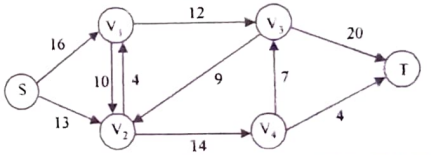


Fig. *[R.T.U. 2018, 2014]*

OR

Describe problem definition of Multicommodity flow in the network. State and prove the Ford Fulkerson's theorem. *[R.T.U. 2016]*

OR

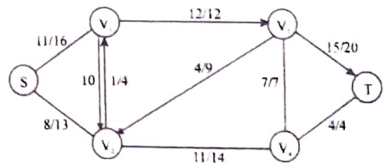
State multicommodity flow problem. *[R.T.U. 2015]*

OR

Explain multi commodity flow problem with some suitable example. *[R.T.U. 2013]*

OR

Show the formation of cuts, augmentation path, min-flow-max-cut in the following graph.



[R.T.U. 2011]

Ans. Multicommodity flow : Multi Commodity Flow (MCF) problem is characterized by a set of commodities to be routed through a network at a minimum cost. It yields formulation of optimization problems that arise in industrial application such as transportation or tele-communications.

Where commodities may represent messages in telecommunications or vehicles in transportation.

- Each commodity has to be transported from one or several origin nodes to one or several destination nodes.
- Given a network represented by a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a non negative capacity $c(u, v) \geq 0$
- Each commodity has to shipped from a set of supply nodes to a set of demand nodes.
- We are given k different commodities, $k_1, k_2 \ldots k_k$

Where

Commodity i is specified by the triple

$$k_i = (s_i, t_i, d_i)$$

Where

s_i = source of commodity i

t_i = sink of commodity

d_i = demand which is the desired flow value for commodity i from s_i to t_i

- Flow for commodity i, denoted by f_i is defined by a real-valued function that satisfies the three constraints of flow problems namely Capacity constraints, skew symmetry and flow conservation.
- $f_i(u, v)$ is the flow of commodity i from vertex u to vertex v

The aggregate flow $f(u, v)$ is the sum of the various commodity flows.

$$i.e.\ f(u, v) = \sum_{i=1}^{k} f_i(u, v)$$

Note : This aggregate flow must not be more than the capacity of edge (u, v)

$$i.e. \sum_{i=1}^{k} f_i(u, v) \leq c(u, v) \text{ for each } u, v \in V$$

$$f_i(u, v) = f_i(v, u) \qquad \text{for } i = 1, 2, \ldots k$$

Example

Fig. : Three commodity flow problem

As shown in above figure, there are three commodities flowing through the network.

- The nodes are the source and sink for different commodities.
- Edges are labeled with capacities that can be routed through them.
- There are three commodities k_1, k_2 and k_3

For k_1

Source = node 1

Sink = node 3

i.e.

k_1 must originate only at node 1 and terminate only at node 3

Similarly,

For k_2

Source = node 2

Sink = node 1

For k_3

Source = node 3

Sink = node 2

Now, we know that the sum of all commodities following on an edge should not exceed edge capacity say 1.

So for k_1 the only path from source to sink 3 is $1 \to 2 \to 3$

To restrict the flow value to the capacity of the edge, the only option is to block one of the edges $1 \to 2$ or $2 \to 3$ like this the total flow would be 1 which is equal to the capacity of the edge.

Similarly,

For k_2

Path from source 2 to sink 1 is $2 \to 3 \to 1$

For k_3

Path from source 3 to sink 2 is $3 \to 1 \to 2$

In each case, one edge is allowed and other is blocked to restrict the flow value equal to the capacity of the edge mentioned in the flow network.

Ford Fulkerson's theorem : In each iteration of the Ford-FullKerson method, we find some augmenting path p and increase the flow f on each edge of p by the residual capacity $C_f(p)$. The following implementation of the method computes the maximum flow in a graph $G = (V, E)$ by up dating the flow $f[u, v]$ between each pair u, v of vertices that are connected by an edge. If u, v are not connected by an edge in either direction, we assume implicitly that $f[u, v] = 0$. The capacities $C(u, v)$ are assumed to be given along with the graph, and $C(u, v) = 0$, if $(u, v) \notin E$.

Ford-FulKerson (G, s, t)

1 For each edge $(u, v) \in E[G]$

2 do $f[u, v] \leftarrow 0$

3 $f[v, u] \leftarrow 0$

4 While there exists a path p from s to t in the residual network G_f

5 do $C_f(p) \leftarrow \min\{C_f(u, v) : (u, v) \text{ is on } p\}$

6 for each edge (u, v) in p

7 do $f[u, v] \leftarrow f[u, v] + C_f(p)$

8 $f[v, u] \leftarrow -f[u, v]$

we can apply Ford-fulkerson algorithm to get the maximal flow through network.

Analysis : The total running time for Ford-fulkerson algorithm is $O(E f^*)$.

Example : Show the step by step implementation of Ford-fulkerson algorithm.

Solution : Here, we show the step by step implementation procedure of Ford fullKerson algorithm.

FORD-FULKERSON (G, s, t)

Step 1 : Find edge (u, v) which are in $E(G)$ i.e., edge $(u, v) \in E(G)$ and set $f[u, v] = 0$, $f[v, u] = 0$

$$f_0(s, P_1) = f_0(P_1, P_2) = f_0(P_2, t) = f_0(s, P_3) = f_0(P_3, P_2) = f_0(s, P_4) = 0$$

Also, $f_0(P_1, P_3) = f_0(P_3, P_1) = f_0(P_2, P_4) = f_0(P_4, P_2) = 0$

$$f_0(P_4, t) = 0$$

Step 2 : Consider path $s \to P_1 \to P_2 \to P_3 \to P_4 \to t$ from s to t. Calculate FAP (Flow augmented path)

$$\omega(s, P_1) = f_0(s, P_1) = 16 - 0 = 16 > 0$$

$$\omega(P_1, P_2) = f_0(P_1, P_2) = 12 - 0 = 12 > 0$$

$$\omega(P_2, P_3) = f_0(P_2, P_3) = 9 - 0 = 9 > 0$$

$$\omega(P_3, P_4) = f_0(P_3, P_4) = 14 - 0 = 14 > 0$$

$$\omega(P_4, t) = f_0(P_4, t) = 4 - 0 = 4 > 0$$

$$\Rightarrow \quad C_f(p) = \min\{C_f(u, v) : (u, v) \in p\}$$

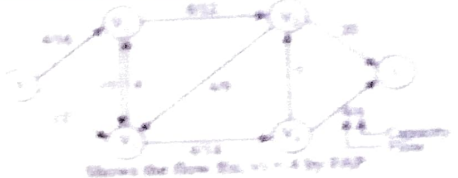
$$= \min\{16, 12, 9, 14, 4\}$$

$$= 4$$

Step 3 : For each edge $(u, v) \in p$ (flow augmented path)

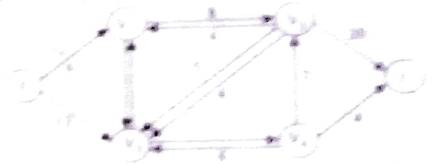
Set $f[u, v] \leftarrow f[u, v] + C_f(p)$

$$\Rightarrow \quad f[u, v] = 0 + 4 = 4$$



Shows the flow fm. $v_1 = 4$ by FAP

Step 4 : Set $f[v, u] \leftarrow -f[u, v]$



Since path $s \to P_1 \to P_2 \to P_3 \to P_4 \to t$ does not contain other edges, so there are not in flow augmented path. Again use FAP with new flow, to find out new path.

Step 5 : Consider path $s \to P_1 \to P_2 \to P_3 \to P_4 \to t$ from s to t. Calculate flow augmented path, with new flow

$$\omega(s, P_1) = f_0(s, P_1) = 16 - 4 = 12 > 0$$

$$\omega(P_1, P_2) = f_0(P_1, P_2) = 12 - 0 = 12 > 0$$

$$\omega(P_2, P_3) = f_0(P_2, P_3) = 14 - 4 = 10 > 0$$

$$\omega(P_3, P_4) = f_0(P_3, P_4) = 7 - 0 = 7 > 0$$

$$\omega(P_4, t) = f_0(P_4, t) = 20 - 0 = 20 > 0$$

Step 6 : Now find C_f as

$$C_f = \min\{C_f(u, v) : (u, v) \in p\}$$

$$= \min\{12, 10, 10, 7, 20\}$$

$$= 7$$

That is minimum capacity of flow C_f is 7

Step 7 : For each edge $(u, v) \in p$ (flow augmented path) find $f(u, v)$ as

$$f(u, v) \leftarrow f(u, v) + C_f(p)$$

$$f_1(s, P_1) = f(s, P_1) + C_f(p) = 4 + 7 = 11$$

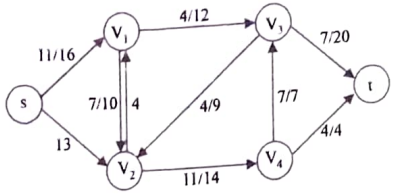
$$f(P_1, P_2) = f(P_1, P_2) + C_f(p) = 0 + 7 = 7$$

$$f_1(P_2, P_3) = f(P_2, P_3) + C_f(p) = 4 + 7 = 11$$

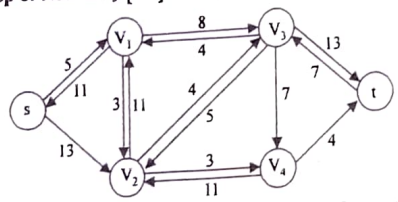
$$f(P_3, P_4) = f(P_3, P_4) + C_f(p) = 0 + 7 = 7$$

$$f_1(P_4, t) = f_0(P_4, t) + C_f(p) = 0 + 7 = 7$$

Now, we show that flow in the graph as follows:



Step 8: Now set $f[v,u] \leftarrow -f[u,v]$



Step 9: Consider path $s \rightarrow V_2 \rightarrow V_1 \rightarrow V_3 \rightarrow t$ from s to t, calculate flow augmented path, with new flow.

$$\omega(s,V_2) - f_0(s,V_2) = 13 - 0 = 13 > 0$$

$$\omega(V_2,V_1) - f(V_2,V_1) = 11 - 3 = 8 > 0$$

$$\omega(V_1,V_3) - f(V_1,V_3) = 12 - 4 = 8 > 0$$

$$\omega(V_3,t) - f(V_3,t) = 20 - 7 = 13 > 0$$

Step 10: Now again find C_f as:

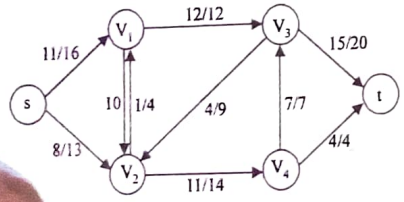
$$C_f = \min\left[C_f(u,v)\right]$$

$$= \min[13,8,8,13] = 8$$

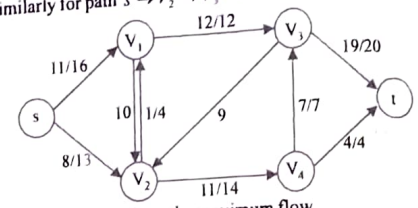
That is, minimum capacity of flow C_f is 8.

Step 11: For each edge $(u,v) \in p(FAP)$ we find f(u, v) as:

$$f(u,v) \leftarrow f(u,v) + C_f(p)$$

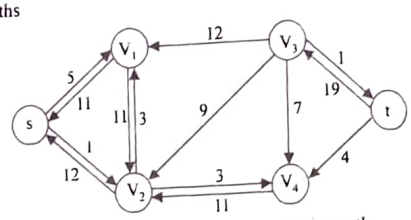


Similarly for path $s \rightarrow V_2 \rightarrow V_3 \rightarrow t$



Show the maximum flow

Below, we show the residual network which has no augmenting paths



Residual network with no augmenting paths

Q.25 ***Explain Flow shop scheduling with suitable example.*** *[R.T.U. 2016]*

OR

Briefly describe flow shop scheduling and network capacity assignment problem. *[R.T.U. 2015]*

OR

Write Flow shop scheduling algorithm. *[R.T.U. 2014]*

Ans. Flow Shop Scheduling : A flow shop problem exists when all the jobs share the same processing order on all the machines. In flow shop, the technological constraints demand that the jobs pass between the machines in the same order. Hence, there is a natural processing order (sequence) of the machines characterized by the technological constraints for each and every job in flow shop. Frequently occurring practical scheduling problems focus on two important decisions:

- The sequential ordering of the jobs that will be processed serially by two or more machines
- The machine loading schedule which identifies the sequential arrangement of start and finish times on each machine for various jobs.

Managers usually prefer job sequence and associated machine l: ading schedules that permit total facility processing time, mean flow time, average tardiness, and average lateness to be minimized. The flow shop contains **m** different machines arranged in series on which a set of **n** jobs are to be processed. Each of the **n** jobs requires **m** operations and each operation

is to be performed on a separate machine. The flow of the work is unidirectional, thus every job must be processed through each machine in a given prescribed order. In other words, if machines are numbered from 1,2,3 ....m then operations of job j will correspondingly be numbered $(1, j), (2, j), (3, j), \ldots (m, j)$. In this context, each job has been assigned exactly **m** operations where as in real situations a job may have a fewer operations. Nevertheless, such a job will still be treated as processing **m** operations but with zero processing times correspondingly. The general **n** jobs, **m** machine flow shop scheduling problem is quite formidable. Considering an arbitrary sequence of jobs on each machine, there are $(n!)^m$ possible schedules which poses computational difficulties. Therefore, efforts in the past have been made by researchers to reduce this number of feasible schedules as much as possible without compromising on optimality condition. Literature on flow shop process indicates that it is not sufficient to consider only schedules in which the same job sequence occurs on each machine with a view to achieving optimality. On the other hand, it is not always essential to consider $(n!)^m$ schedules in search for an optimum. The following two dominance properties will indicate the amount of reduction possible in flow shop problems

Theorem 1

When scheduling to optimize any regular measure of performance in a static deterministic flow shop, it is sufficient to consider only those schedules in which the same job sequence exists on machine 1 and machine 2.

Theorem 2

When scheduling to optimize makespan in the static deterministic flow shop, it is sufficient to consider only those schedules in which the same job sequence exists on machine 1 and 2, and the same job sequence on machines **m-1** and **m**. The implications of above dominance can be interpreted as follows:

- For any regular measure of performance, by virtue of the fact that the same job sequence on the first two machines is sufficient for achieving optimization, it is $(n!)^{m-1}$ schedules that constitute a dominant set.
- For makespan problems, by virtue of the fact that the same job sequence on machine **m-1** and **m** besides on machine 1 and 2 is sufficient for achieving optimization, it is $(n!)^{m-2}$ schedules that constitute a dominant set for $m > 2$.

A permutation schedule is that class of schedule which may be completely identified by single permutation of integers. For a flow shop process, the permutation schedule is, therefore, a schedule with the same job sequence on all machines. Interpreting the above results yet in another way, it is observed that:

- In a two machine flow shop, permutation schedule is the optimum schedule with regard to any regular measure of performance.
- In a three machine flow shop, permutation schedule is the optimum schedule with respect to makespan criterion. Unfortunately, this second dominance property is confined to makespan only. This neither extends to other measures of performance nor does it take into account large flow shops with $m > 3$. However, no stronger general results than the above two concerning permutation schedules are available in the literature.

Network Capacity Assignment Problem

Capacity Assignment problem which focuses on finding the best possible set of capacities for the links that satisfies the traffic requirements in a prioritized network while minimizing the cost. Unlike most approaches, which consider a single class of packets flowing through the network, we base our study on the more realistic assumption that different classes of packets with different packet lengths and priorities are transmitted over the networks. Apart from giving a brief overview of the problem and a report of the existing schemes, we present a new continuous learning automata strategy for solving the problem. This strategy is analogous to the discretized version already reported except that it operates in a continuous probability space, and is thus easier to comprehend since it is more akin to the well-studied families of learning automata.

Data networks are divided into three main groups which are characterized by their size, these are Local Area Networks (LANs), Metropolitan Area Networks(MANs) and Wide Area Networks (WANs). An Internet work is comprised of several of these networks linked together, such as the Internet. Most applications of computer networks deal with the transmission of logical units of information or messages, which are sequences of data items of arbitrary length. However, before a message can be transmitted it must be subdivided into packets. The simplest form of a packet is a sequence of binary data elements of restricted length, together with addressing information sufficient to identify the sending and receiving computers and an error correcting code.

There are several tradeoffs to be considered when designing a network system. Some of these are difficult to quantify since they are criteria used to decide whether the overall network design is satisfactory. This decision is based on the designer's experience and familiarity with the requirements of the individual system. As there are several components to this area, a detailed examination of the pertinent factors, which are primarily cost and performance.

In the process of designing computer networks, the designer is confronted with a trade-off between costs and

performance. Some of the parameters effecting the cost and performance parameters used in a general design process are listed above, but, inpractice, only a subset of these factors are considered in the actual design. In this paper, we study scenarios in which the factors considered include the location of the nodes and potential links, as well as possible routing strategies and link capacities.

The Capacity Assignment (CA) problem specifically addresses the need for a method of determining a network configuration that minimizes the total cost while satisfying traffic requirements across all links. This is accomplished by selecting the capacity of each link from a discrete set of candidate capacities that have individual associated cost and performance attributes. Although problems of this type occur in all networks, in this paper, we will only examine the capacity assignment forprioritized networks. In prioritized networks, packets are assigned to a specific priority class which indicates the level of importance of their delivery. Packets of lower priority will be given preference and separate queues will be maintained for each class.

The currently acclaimed solutions to the problem are primarily based on heuristics that attempt to determine the lowest cost configuration once the set of requirements are specified. These requirements include the topology, the average packet rate, or the routing, for each link, as well as the priorities and the delay bounds for each class of packets. The result obtained is a capacity assignment vector for the network, which satisfies the delay constraints of each packet class at the lowest cost.The primary contribution of this paper is to present a continuous learning

Automation (LA) solution to the CA problem. Apart from this fundamental contribution of the paper, the essential idea of using LA which have actions in a"meta-space" (i.e., the automata decide on a strategy which in turn determines the physical action to be taken in the real-life problem) is novel to this paper and its earlier counterpart.

Assumptions and Delay Formulae

The model used for all the solutions presented have the following features

1. Standard Assumptions : (a) The message arrival pattern is Poissonly distributed,and (b) The message lengths are exponentially distributed.
2. Packets : There are multiple classes of packets, each packet with its own (a)Average packet length, (b) Maximum allowable delay and (c) Unique priority level, where a lower priority takes precedence.
3. Link capacities are chosen from a finite set of predefined capacities with an associated fixed setup cost, and variable cost/km.
4. Given as input to the system are the (a) Flow on each link for each messageclass, (b) Average packet length measured in bits, (c) Maximum allowable delayfor each packet class measured in seconds, (d) Priority of each packet class, (e) Link lengths measured in kilometers, and (f) Candidate capacities and their associated cost factors measured in bps and dollars respectively.
5. A non-preemptive FIFO queuing system is used to calculate the average link delay and the average network delay for each class of packet.
6. Propagation and nodal processing delays are assumed to be zero.

Based on the standard network delay expressions, all the researchers in the field have used the following formulae for the network delay cost:

$$T_{jk} = \frac{\eta_j \left(\sum_i \frac{\lambda_{ji}m_i}{\eta_i c_j} \right)}{(1-U_{r-1})(1-U_r)} + \frac{m_k}{c_j}$$

$$U_r = \sum_{i \in V_r} \frac{\lambda_{ji}m_i}{c_j}$$

$$Z_k = \frac{\sum_j T_{jk}\lambda_{jk}}{\gamma_k}$$

In the above, T_{jk} is the Average Link Delay for packet class k on link j, U_r is the Utilization due to the packets of priority 1 through r (inclusive), V_r is the set of classes whose priority level is in between 1 and r (inclusive), z_k is the Average Delay for packet class, $\eta_j = \sum_i \lambda_{ji}$ is the Total Packet Rate on link j. $\gamma_k = \sum_j \lambda_{jk}$.

Total Rate of packet class k entering the network, ljk is the Average Packet Rate for class k on link j, mkis the Average Bit Length of class k packets, and Cjis the capacity of link j. As a result of the above, it can be shown that the problem reduces to an integer programming problem.

Q.26 ***Solve the assignment problem using Hungarian algorithm for which the following cost matrix***

$$\begin{bmatrix} 15 & 5 & 9 & 7 \\ 2 & 13 & 6 & 5 \\ 7 & 8 & 3 & 11 \\ 2 & 4 & 6 & 10 \end{bmatrix}$$

[R.T.U. 2013]

Ans. First of all we subtract the minimum of each row form their row to have at least one zero in every row of the matrix.

$$\begin{array}{c|cccc|c} & M_1 & M_2 & M_3 & M_4 & \\ J_1 & 15 & 5 & 9 & 7 & -5 \\ J_2 & 2 & 13 & 6 & 5 & -2 \\ J_3 & 7 & 8 & 3 & 11 & -3 \\ J_4 & 2 & 4 & 6 & 10 & -2 \end{array}$$

$$\downarrow$$

$$\begin{array}{c|cccc} & M_1 & M_2 & M_3 & M_4 \\ J_1 & 10 & 0 & 4 & 2 \\ J_2 & 0 & 11 & 4 & 3 \\ J_3 & 4 & 5 & 0 & 8 \\ J_4 & 0 & 2 & 4 & 8 \end{array}$$

-2

M_4 does not contain any zero. Now we have to subtract the minimum number from each element of M_4.

$$\begin{array}{c|cccc} & M_1 & M_2 & M_3 & M_4 \\ J_1 & 10 & 0 & 4 & 0 \\ J_2 & 0 & 11 & 4 & 1 \\ J_3 & 4 & 5 & 0 & 6 \\ J_4 & 0 & 2 & 4 & 6 \end{array}$$

Now we try to cover all the zero with minimum number of horizontal (or vertical) lines.

Junction elements

$$\begin{array}{c|cccc} & M_1 & M_2 & M_3 & M_4 \\ J_1 & \cancel{10} & 0 & \cancel{4} & 0 \\ J_2 & 0 & 11 & 4 & 1 \\ J_3 & \cancel{4} & 5 & 0 & 6 \\ J_4 & 0 & 2 & 4 & 6 \end{array}$$

Since the number of lines are 3 which is not equal to the order of matrix ($3 \neq 4$), we take minimum of uncovered element *i.e.* 1. This 1 is subtracted from all the uncovered elements and added to the junction elements (i.e. 10 and 4). Now we try to cover all the zeros with minimum number of horizontal (or vertical) lines.

$$\begin{array}{c|cccc} & M_1 & M_2 & M_3 & M_4 \\ J_1 & \cancel{11} & 0 & \cancel{5} & 0 \\ J_2 & 0 & 10 & 4 & 0 \\ J_3 & \cancel{4} & 4 & 0 & 5 \\ J_4 & 0 & 1 & \cancel{4} & 5 \end{array} \quad \text{or} \quad \begin{array}{c|cccc|c} & M_1 & M_2 & M_3 & M_4 & \\ J_1 & \cancel{11} & 0 & \cancel{5} & 0 & R_0 \\ J_2 & \cancel{0} & \cancel{10} & \cancel{4} & \cancel{0} & R_1 \\ J_3 & \cancel{4} & \cancel{4} & 0 & \cancel{5} & R_2 \\ J_4 & \cancel{0} & \cancel{1} & \cancel{4} & \cancel{5} & R_3 \end{array}$$

Now, number of lines are 4 which is equal to the order of matrix (4 = 4).

Now, we have to see the rows where the number of zeros are single. Row R_2 and R_3 is like that one. So corresponding to this zero, we assign job $R_2 \rightarrow J_3$ and $R_3 \rightarrow J_4$ to machines M_3 and M_1 respectively, rest of the jobs are to be assigned to machine M_2 and M_4. We can see that job J_1 can be assigned to machine M_2 and M_4, Job J_2 can be assigned to machine M_1 and M_4. By this we can see that machine M_1 is already acquire by the job J_4 so job J_2 must be assign to machine M_4 and the rest of machine that is M_2 is acquire the job J_1.

Here, readers are encouraged to check all the possible combinations whether they lead to minimum cost. One of the possible combination is :

$$\begin{array}{cc|cccc} & & M_1 & M_2 & M_3 & M_4 \\ J_4 \rightarrow M_1 & J_1 & 11 & \boxed{0} & 5 & 0 \\ J_1 \rightarrow M_2 & J_2 & 0 & 10 & 4 & \boxed{0} \\ J_3 \rightarrow M_3 & J_3 & 4 & 4 & \boxed{0} & 5 \\ J_2 \rightarrow M_4 & J_4 & \boxed{0} & 1 & 4 & 5 \end{array}$$

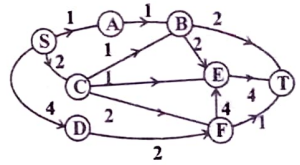
Now we calculate the total cost using cost matrix given initially.

Cost $= (J_4 \rightarrow M_1) + (J_1 \rightarrow M_2) + (J_3 \rightarrow M_3) + (J_2 \rightarrow M_4)$

$\Rightarrow = 2 + 5 + 3 + 5$

$= 15$ **Ans.**

Q.27 ***Find the maximum flow for the following flow network using fordfulkerson method.***

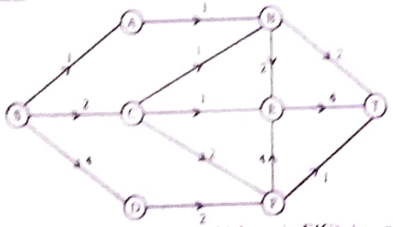


[R.T.U. 2012]

Ans. The Ford-Fulkerson method depends on three important ideas that transcend the method and are relevant to many flow algorithms and problems: residual networks, augmenting paths, and cuts.

The Ford-Fulkerson method is iterative

We start with $f(u, v) = 0$ for all $u, v, \in V$, giving an initial flow of value 0. At each iteration, we increase the flow value by finding on "augmenting path" which we can think of simply as a path from the source s to the sink t along which we can send more flow and then augmenting the flow along this path. We repeat this process until no augmenting path can be found. The max-flow min-cut theorem will show that upon termination, this process yields a maximum flow.



Step 1 : Find edge (u, v) which are in E(G), *i.e.*, edge $(u, v) \in E(G)$ and set f(u, v) = 0, f(v, u) = 0

$f_0(S, A) = f_0(A, B) = f_0(B, T) = f_0(S, C) = 0$

$f_0(C, B) = f_0(C, E) = f_0(C, F) = f_0(B, E) = f_0 = (F, E) = 0$

$f_0(E, T) = f_0(S, D) = f_0(D, F) = f_0(F, T) = 0$

Step 2 : Consider path $S \to A \to B \to E \to T$ from S to T. Calculate FAP (Flow Augmented Path)

$w(S, A) = f_0(S, A) = 1 - 0 = 1$

$w(A, B) = f_0(A, B) = 1 - 0 = 1$

$w(B, E) = f_0(B, E) = 2 - 0 = 2$

$w(E, T) = f_0(E, T) = 4 - 0 = 4$

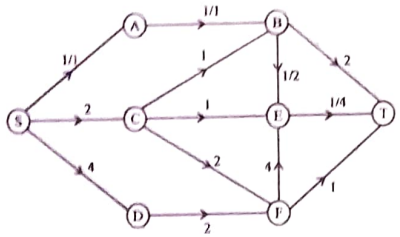
$C_f(p) = \min[C_f(u, v) : (u, v) \in p]$

$= \min\{1, 1, 2, 4\} = 1$

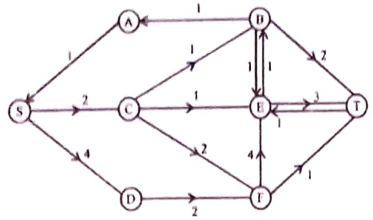
Step 3 : For each edge $(u, v) \in p$ (Flow Augmented Path)

Set $f[u, v] \to f[u, v] + C_f(p)$

$f[u, v] = 0 + 1 = 1$



Step 4 : Set $f[v, u] \leftarrow -f[u, v]$



Since path $S \to A \to B \to E \to T$ does not contain other edges, so they are not in flow augmented path. Again test for FAP with new flow, to find out new path.

Step 5 : Consider path $S \to C \to B \to E \to T$ from S to T. Calculate, flow augmented path with new flow.

$w(S, C) = f_0(S, C) = 2 - 0 = 2$

$w(C, B) = f_0(C, B) = 1 - 0 = 1$

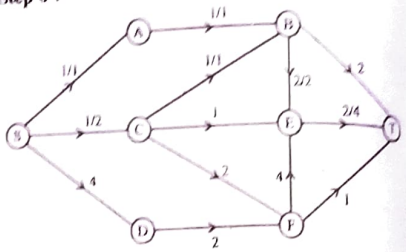
$w(B, E) = f_1(B, E) = 2 - 1 = 1$

$w(E, T) = f_1(E, T) = 4 - 1 = 3$

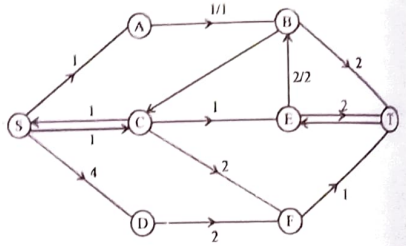
$C_f = \min\{2, 1, 1, 3\} = 1$

That is minimum capacity of flow $C_f = 1$

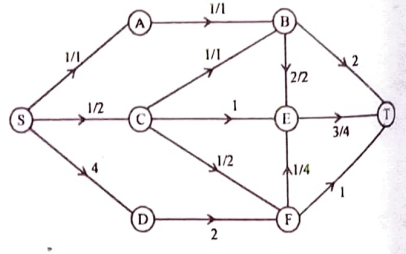
Step 6 :



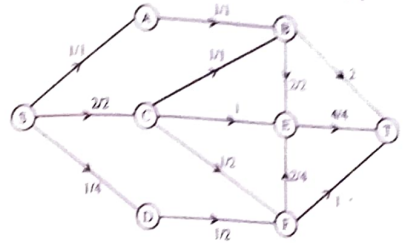
Step 7 : Now set $f[v, u] \leftarrow -f[u, v]$



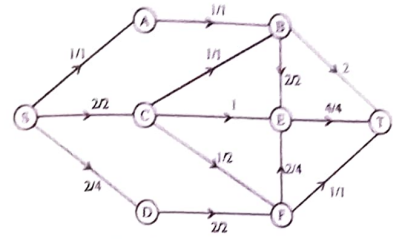
Step 8 : Similarly for path $S \to C \to F \to E \to T$



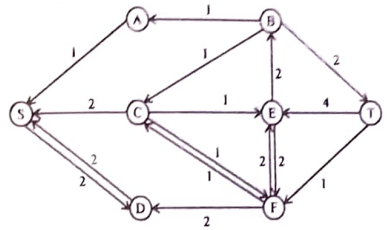
Step 9 : Similarly for path $S \to D \to F \to E \to T$



Step 10 : Similarly for path $S \to D \to F \to T$



Below, we show the residual network which has no augmenting paths.



Q.28 ***Solve the randomized algorithm N-Queens problem.*** ***[R.T.U. 2011]***

Ans. A common problem domain for programming exercise, the N-queens problem is an extension of the classic 8-queens puzzle. This problem involves placing eight queens on an eight by eight chess board in such a way that no one queen can attack another. A queen can move any number of spaces horizontally, vertically or diagonally to attack, thus for the arrangement to be a solution, no queen can be placed on the same row, column or diagonal as another. Following these constrains there are twelve non symmetrical solutions. When extended to include any board size greater than three the problem remains simple but is nontrivial making it a good candidate for exploring different algorithm design methods. Thus, it is a common problem to use an exercise in algorithm analysis. This explains the outcomes of two very different approaches to the problem : *randomization* and *recursion*.

Algorithm : Random

The first algorithm's design revolves around the randomization of queen placement. It works by moving row by row choosing a random open space on the board and placing a queen there.

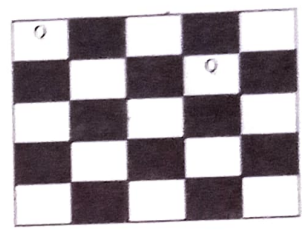


Fig. 1 : Sample Chessboard

If no open space can be found on the current row the algorithm starts over with the first row. A board space is defined as open (not threatened by any previously set queen) if the column, row and diagonals of the cell do not contain another queen. To check for this the column above the cell, the row to the right of the cell and the diagonals above the cell are checked for queens. If a queen is found in any other space checked the current space is not open for placing a queen. For example, if the current square being checked was (3, 2) on the board in figure 1 it would be deemed open but, (3, 4) and (3, 1) would not.

As it iterates through each row the algorithm checks each space in that row keeping track of every open cell it finds in a vector. A random space in that collection is then chosen to place a queen. The queen is then placed and the algorithm moves on to the next row. If the algorithm successfully places a queen on each row, it exits with the current arrangement of queens. However, if the collection of open cells in a row is found to be empty the algorithm starts over with row one, erasing all previously set queens from the board. This algorithm is guaranteed to find a solution for any N value greater than there if given enough time. In order for N queens to be placed on a N × N board every row and column must have a queen on it, thus there is no change that the algorithm will get stuck in continually placing a queen in a row that should not have a queen. In addition, the random way in which the spaces for placing queens are chosen ensure that at some point, if no solution is found, that every arrangement of queens will be tried. Thus, eventually the random choices will lead to an arrangement of queens that agrees with the restrictions of the problem.

Analysis

On analysis the randomized algorithm's big-O evaluation is equal to O(EN^2). The N^2 of this maximum bound was derived from the process that returns the collection of open spaces on a given row being done N times as the algorithm runs through each row on the board. Since the checking process for free cells has a big-O evaluation of O(N), the total executions time for a single run through of the algorithm is equal to N * N which equals N^2. For example, a four by four board will require the algorithm to check for open spaces in four rows. This will result in a total runtime maximum bound of sixteen (four from the checking * four from the row iteration). The factor portion of this analysis comes from the fact that this algorithm is randomized and will not necessarily find a solution to the problem on the first run through. While algorithm will at some point come upon a solution, it is unknown how many iteration of the functions it will take. Therefore, the factor (or number of times the algorithm will be executed) by which the N^2 will need to be multiplied by to find the upper bound of the entire functions cannot be specified.

This algorithm's evaluation could have easily been O(EN^2), but considerable pruning was done in order to avoid this. Since, by definition of the problem, it is known that no one row can have more than one queen on it the algorithm is able to stop over checking the spaces in a row after queen has been placed on that row. Thus allows for the removal of a for loop that would check each space in each row with each iteration. Therefore, instead of iterating by space the algorithm is able to iterate by row, reducing its maximum time bound by a factor of N.

Performance

The random algorithm was tested for N values from four to twenty-five. Each test was done ten times to accommodate for the random factor of how many tries it took to find the solution. The average of the trials was taken for the numbers used below. In addition, since the algorithm will find the same solution for each iteration of the board size only three trials were done per N.

Random algorithm is far more efficient then the second, recursive algorithm. As shown in the figure 2, the average times it took to find a solution for N values four through twenty-five were all below a quarter of a second. As expected from the big-O evaluation, the time increased with a larger N values roughly following f(N^2). Since the upper time bound of the algorithm if it exited after its first try (its first guess is correct) is N^2, the lower bound of the algorithm should be around N^2 since the function will always be run at least once.

The up and down pattern of the graph can be explained by the varying number of solutions that each N value has. For example, when N equals eight there are ninety-two solutions, but when N equals ten there are 724 solutions. It with only an addition fourty spaces to search and keep track of there are an additional 612 solutions the algorithm correctly guess. Thus, the algorithm will probabilistically a solution faster for N values with more solutions than N values nearby with fewer possible solutions. This the sometimes vastly different runtimes between N twenty-one through twenty-three seen in figure 2.

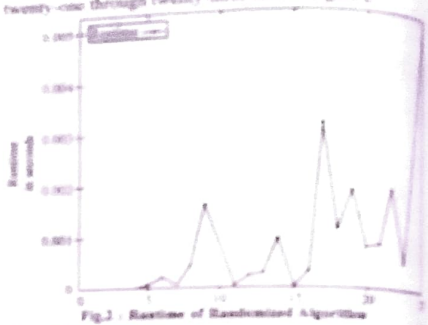


Fig. 2 : Runtime of Randomized Algorithm

Algorithm : Randomized

```
Void Board :: findRandSolution ()
{
    srand (time (NULL));
    int randC;
    // iterate through each row
    for (int r = 0; r < board_size; r++)
    {
        // find open places current row
        vector<int> possC_col
        openC_col (r, possC_col);
        // choose a random open place
        if (! possC_col.empty ())
        {
            randC = rand () % possC_col.size ();
            placeQueen (r, possC_col [randC]);
        }
        // no open spaces      start over
        else
        {
            r = -1;
            clearBoard ();
        }
    }
}
```


PROBLEM CLASSES NP, NP-HARD AND NP-COMPLETE 5

CHAPTER IN A NUTSHELL

❑ **Definitions**

P : P is the set of decision problems with a yes-no answer that is polynomial bound.

A problem is said to be polynomial-bound if there exists a polynomial bound algorithm for it. It is also to be noted that not for all the problems, the class P has "acceptably efficient" algorithm. Also, if a problem does not belong to class P then it is intractable.

An algorithm is said to be polynomial bounded if its worst-case complexity is bound by a polynomial function P of input size n. That means, for each input of size n, the algorithm terminates after atmost P(n) steps.

For instance, $n^3 + 2n^2 + 65$

NP-Hard : If a language L_1 defines some decision problem then it reduces to the language L_2 in polynomial time only.

If there exists a function F which can be computed in polynomial time, the function F takes an input x to L_1 and converts it to an input F(x) of L_2, in such a manner that $x \in L_1$ if and only if $F(x) \in L_2$.

The notation $L_1 \propto_{poly} L_2$ refers to the language L_1 which is polynomial time reducible to language L_2.

We can also say that language L_2, defining some decision problem, is NP-hard if for every $L_1 \in NP$, where language $L_1 \propto NP$ is polynomial-time reducible to language L_2.

NP Complete : If a language L_2 is NP-hard and it also belongs to the class NP, then language L_2 is said to be NP-complete. If any problem that belongs to NP-complete is solved in polynomial time then all the problems belonging to the class NP can be solved in polynomial time. It can be observed that if anyone shows a deterministic polynomial-time algorithm for even one NP-complete problem, then P = NP.

❑ **Decition Problems**

The problems under this class have the single bit output which shows 0 or 1 i.e., the answer for the problem is either zero or one. For instance, some decision problems are

- Given two sets of strings S_1 and S_2, does S_1 is a substring of S_2 ?
- Given two sets of elements S_1 and S_2, does both the sets contain same number of elements ?

Any problem that involves the identification of an optimum (either minimum or maximum) value of a given cost function is known as **optimization problem.** For solving optimization problem, an optimization algorithm is used. For instance, the optimization problem is as follows

- Given a weighted graph G, and an integer t, does G have a minimum spanning tree of weight at most t ?
- Given S, does there exist a subset of elements that fits in the knapsack, and has total profit of at least S ?

❑ **Cook's Theorem**

Cook's theorem states that any NP problem can be converted to SAT in polynomial time. In order to prove this, we require a uniform way of representing NP problems. What makes a problem NP is the existence of a polynomial-time algorithm.

Thus, under Cook's theorem we will try to prove that CIRCUIT-SAT is NP-Complete. For this, CIRCUIT-SAT should be in NP and every problem in NP should be reducible to CIRCUIT-SAT.

CIRCUIT-SAT, that is, circuit-satisfiability problem says that, for a given Boolean combinational circuit consisting of AND, OR and NOT gates, is it satisfiable or not.

In other words, for given input to the circuit, does there exists a truth assignment that causes the output of the circuit to be 1. Some of the various combination elements are shown in fig.

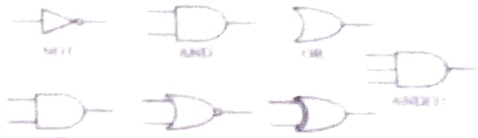


Fig. Combinational elements

PREVIOUS YEARS QUESTIONS

PART-A

Q.1 ***Define the term polynomial bound.***

Ans. An algorithm is said to be polynomial bounded if its worst-case complexity is bound by a polynomial function P of input size n. That means, for each input of size n, the algorithm terminates after atmost P(n) steps.

For instance, $n^7 + 24n^2 + 65$

Q.2 ***What do you mean by NP-complete?***

Ans. If a language L_2 is NP-hard and it also belongs to the class NP, then language L_2 is said to be NP–complete

Q.3 ***Define optimization problem.***

Ans. Any problem that involves the identification of an **optimal** (either minimum or maximum) value of a given cost function is known as **optimization problem.**

Q.4 ***What is NP-hard problem.***

Ans. NP-hard refers to a problem as hard as any NP problem. Formally, a problem is called NP-hard if it cannot be decided, or does not belong to NP class, but all NP problems are reducible to it in polynomial time.

Q.5 ***What do you mean by interactable problems.***

Ans. Certain problems which can be theoretically solved by computational means, yet are infeasible because they require, large number of resources. These can be called infeasible or intractable problems.

PART-B

Q.6 ***Prove that Hamilton cycle problem in NP complete.*** *[R.T.U. 2018, 2016, 2014, 2011, 2010]*

Ans. Let us define a nondeterministic algorithm A that takes, as input, a graph G encoded as an adjacency list in binary notation, with the vertices numbered 1 to N. We define A to first iteratively call the choose method to determine a sequence S of N + 1 numbers from 1 to N. Then, we have A check that each number from 1 to N appears exactly once in S (for example, by sorting S), except for the first and last numbers in S, which should be the same. Then, we verify that the sequence S defines a cycle of vertices and edges in G. A binary encoding of the sequence S is clearly of size at most n, where n is the size of the input. Moreover, both of the checks made on the sequence S can be done in polynomial time in n.

Observe that if there is a cycle in G that visits each vertex of G exactly once, returning to its starting vertex, then there is a sequence S for which A will output "yes." Likewise, if A outputs "yes," then it has found a cycle in G that visits each vertex of G exactly once, returning to its starting point. That is, A non-deterministically accepts the language HAMILTONIAN-CYCLE. In other words, Hamiltonian-Cycle is in NP.

Our next example is a problem related to circuit design testing. A Boolean circuit is a directed graph where each node, called a logic gate, corresponds to a simple boolean function, AND, OR, or NOT. The incoming edges for a logic gate correspond to inputs for its boolean function and the outgoing edges correspond to outputs, which will all be the same value, of course, for that gate. (see fig.) Vertices with no incoming edges are input nodes and a vertex with no outgoing edges is an output node.

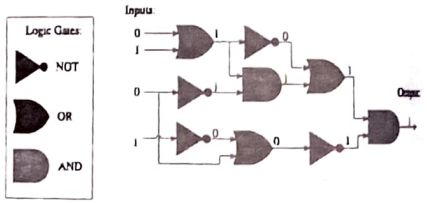


Fig. : An example of Boolean circuit.

Circuit-Sat is the problem that takes as input a boolean circuit with a single output node, and asks whether there is an assignment of values to the circuit's inputs so that its output value is "1." Such an assignment of values is called a satisfying assignment.

Q.7 ***Prove that TSP problem is NP-complete.*** *[R.T.U. 2018, 2012, 2011, 2010]*

Ans. In the TRAVELLING-SALES PERSON problem, we have given a graph G and integer parameter I, such that each edge of graph is associated with certain integer cost w, and we are asked if there is a cycle such that it visits all the vertices in G and has total cost almost I.

Given an instance of problem, we use as a certificate the sequence of n vertices in the tour. The verification algorithm checks that this sequence contain each vertex exactly once, sum up the edge costs, and checks whether the sum is at most K. The process can certainly be done in polynomial time.

To prove that Travelling- Salesman Problem (TSP) is NP hard, we show that Hamilton cycle $\leq_p$ TSP. Let G = (V, E) be an instance of Hamiltonian. We construct an instance of TSP as follow.

We form the complete graph G'=(V,E), where E' = {(i, j) : i, j ∈ V} and we define the cost function c by

$$c(i, j) = \begin{cases} 0 \text{ if } (i, j) \in E \\ 1 \text{ if } (i, j) \notin E \end{cases}$$

The instance of TSP is then (G', c, 0), which is easily formed in polynomial time.

We now show that graph G has a Hamiltonian cycle if and only if graph G' has a tour of cost at most 0. Suppose that graph G has a Hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'. Thus, h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0, since the cost of edges in E' are 0 and 1, concluded that h is Hamiltonian cycle in graph G.

Q.8 ***Prove that circuit satisfiability problem belongs to the class NP.*** *[R.T.U. 2017, 2014]*

Ans. We construct a non deterministic algorithm for accepting circuit-sat in polynomial time. We first use the choose method to "guess" the values of the input nodes as well as the output value of each logic gate. Then, we simply visit each logic gate g in C, that is, each vertex with at least one incoming edge. We then check that the "guessed" value for the output of g is in fact the correct value for g's boolean function, be it an AND, OR, or NOT, based on the given values for the inputs for g. This evaluation process can easily be performed in polynomial time. If any check for a gate fails, or if the "guessed" value for the output is 0, then we output "no." If, on the other hand, the check for every gate succeeds and the output is "1," the algorithm outputs "yes." Thus, if there is indeed a satisfying assignment of input values for C, then there is a possible collection of outcomes to the choose statements so that the algorithm will output "yes" in polynomial time. Likewise, if there is a collection of outcomes to the choose statements so that the algorithm outputs "yes" in polynomial time algorithm, there must be a satisfying assignment of input values for C. Therefore, circuit-sat is in NP.

Q.9 ***Write short note on Cook's theorem and its applications.*** *[R.T.U. 2017, 2015]*

OR

State the Cook's theorem. What is significance of this algorithm? *[R.T.U. 2014]*

Ans. Cook's Theorem : Cooks modeled a NP-problem (an infinite set) to an abstract turing machine. Then he developed a polytransformation from the machine (i.e., all NP-Class problems) to a particular decision problem, namely, the boolean satisfiability (SAT) problem.

Satisfiability is in P if and only if NP = P

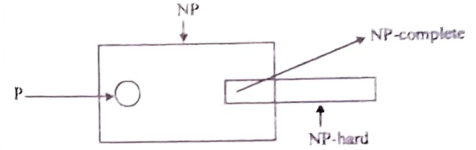


Fig.

Definition : "A problem L is NP-hard if and only if satisfiability reduces to L(satisfiability $\propto$ L). A problem L is NP-complete if and only if L is NP-hard and L ∈ NP.

Significance of Cook's Theorem : If one can find a poly-algorithm for SAT, then by using cook's poly-transformation one can solve all NP-Class problems in poly-time (Consequently, P-class = NP – class would be proved).

SAT is the historically first identified NP-hard problem.

Further Significance of Cook's Theorem : If you find 2 poly-transformation from SAT to another problem Z, then Z becomes another NP-hard problem. That is, if anyone finds a poly algorithm for Z, then by using your polytransformation from SAT to Z, anyone will be able to solve any SAT problem-instance in poly-time, and hence would be able to solve all NP-class problems in poly-time (by cook's theorem).

Q.10 ***Obtain a nondeterministic algorithm of complexity O(n) to determine whether there is a subset of n number a; $1 \leq i \leq n$, that sum to m.*** *[R.T.U. 2013]*

Ans. This is a knapsack decision problem.

Optimization problem : Find the largest total profit of any subset of objects that fits in the knapsack, it also identify optimal value of a given cost.

Decision problem : Given k, is there a subset of objects that fits in the knapsack and has total profit at least k? Is there a subset of the objects whose sizes add up to exactly C?

Algorithm : Nondeterministic knapsack algorithm

```
1   Algorithm DKP (p, w, n, m, r, x)
2   {
3   w := 0; p := 0;
4   for i := 1 to n do
5   {
6   x(i) := Choice (0, 1);
7   w := w + x(i) * w(i); p := p + x(i) * p(i);
8   }
9   if ((w > m) or (p < r)) then Failure ();
10  else Success();
11  }
```

The **for loop** of lines 4 to 8 assigns 0/1 values to x(i), $1 \le i \le n$. It also computes the total weight and profit corresponding to this choice of x(). Line 9 checks to see whether this assignment is feasible and whether the resulting profit is at least r. A successful termination is possible if the answer to the decision problem is yes. The time complexity is O(n). If q is the input length using a binary representation, the time is O(q).

Q.11 ***Prove Knapsack problem are NP-complete.*** *[R.T.U. 2011, 2010, Raj. Univ. 2008, 2006, 2005, 2003]*

Ans. Knapsack Problem : Given a set of items, S number from 1 to n. In the set S, each item i has an integer size l_i and worth w_i. We are also given two integer parameters, l and w, and are asked for a subset s, of S in such a manner that

$$\sum_{i \in s} l_i \le l = \sum_{i \in s} w_i \ge w$$

It can be easily shown that KNAPSACK problem is NP. For this we have to construct a non-determinstic polynomial time algorithm that computes the items to include in the subset s and then checks that they don't violate the l and w constraints, respectively.

It is to be noted that KNAPSACK is also NP-hard. For proving it to be NP-hard, we consider any instance of numbers given, for the SUBSET-SUM problem that can correspond to the items for an instant of KNAPSACK with each $l_i = w_i$ set to a value in the SUBSET-SUM instance and the target for the size l worth w both equal to K (where K is the target integer we wish to sum for the SUBSET-SUM problem). By considering the restriction proof, we have that KNAPSACK is NP-complete.

PART-C

Q.12 ***Explain the Cook's theorem with suitable example.*** *[R.T.U. 2018, 2016]*

OR

What is the use of Cook's theorem? Prove it with an example. *[R.T.U. 2012, 2011, 2010, Raj. Univ. 2004]*

OR

What is Cook's theorem? Explain. [R.T.U. 2013, 2009]

Ans. Cook's theorem states that satisfiability is in P if and only if P = NP. If P = NP, then satisfiability is in P. It remains to be shown that if satisfiability is in P, then P = NP.

Proof : To do this, we show how to obtain from any polynomial time nondeterministic decision algorithm A and input I is a formula Q (A, I) such that Q is satisfiable if A has a successful termination with input I.

If the length of I is n and the time complexity of A is p(n) for some polynomial p(), then the length of Q is $O(p^3(n) \log n) = O(p^3(n))$. The time needed to construct Q is also $O(p^3(n) \log n)$. A deterministic algorithm Z to determine the outcome of A on any input I can be easily obtained.

Algorithm Z simply computes Q and then uses a deterministic algorithm for the satisfiability problem to determine whether Q is satisfiable.

If O (p(m)) is the time needed to determine whether a formula of length m is satisfiable, then the complexity of Z is $O(p^3(n)\log n + q(p^3(n)\log n))$. If satisfiability is in P, then q(m) is a polynomial function of m and the complexity of Z becomes O(r(n)) for some polynomial r(). Hence, if satisfiability is in P, then for every nondeterministic algorithm A in NP we can obtain a deterministic Z in P. So, the above construction shows that if satisfiability is in P, then P = NP.

Before going into the construction of Q from A and I, we make some simplifying assumptions on our nondeterministic machine model and on the form of A. These assumptions do not in any way alter the class of decision problems in NP or P. The simplifying assumptions are as follows :

1. The machine on which A is to be executed is word oriented. Each word is w bits long. Multiplication, addition, subtraction, and so on, between numbers one word long take one unit of time. If numbers are longer than a word, then the corresponding operations take at least as many units as the number of words making up the longest number.

2. A simple expression is an expression that contains at most one operator and all operands are simple variables (*i.e.*,

no array variables are used). Some simple expression are I, I + C, D or I – and I. We assume that all assignments in A are in one of the following forms :

(a) (simple variable) = (simple expression)
(b) (array variable) = (simple variable)
(c) (simple variable) = (array variable)
(d) (simple variable) = Choice(S)

where S is a finite set $\{S_1, S_2, \ldots, S_k\}$ or I, u. In the latter case the function chooses an integer in the range [I, u].

Indexing within an array is done using a simple integer variable and all index values are positive. Only one-dimensional arrays are allowed. Clearly, all assignment statements not falling into one of the above categories can be replaced by a set of statements of these types. Hence, this restriction does not alter the class NP.

3. All variables in A are of type integer or boolean.

4. Algorithm A contains no read or write statements. The only input to A is via its parameters. At the time A is invoked, all variables (other than the parameters) have value zero (or false if boolean).

5. Algorithm A contains no constants. Clearly, all constants in any algorithm can be replaced by new variables. These new variables can be added to the parameter list of A and the constants associated with them can be a part of the input.

6. In addition to simple assignment statements, A is allowed to contain only the following types of statements :

(a) The statement goto k, where k is an instruction number.
(b) Success(), Failure()
(c) Algorithm A may contain type declaration and dimension statements. These are not used during execution of A and so need not be translated into Prob. The dimension information is used to allocate array space. It is assumed that successive elements in an array are assigned to consecutive words in memory. It is assumed that the instructions in A are numbered sequentially from 1 to I (if A has I instruction).

Every statement in A has a number. The goto instructions in (a) and (b) use this numbering scheme to effect a branch. It should be easy to see how to rewrite repeat-until, for and so on statements in terms of goto and if b then goto a statement. Also, note that the goto k statement can be replaced by the statement if true then goto k. So, this may also be eliminated.

7. Let p(n) be a polynomial such that A takes no more than p(n) time units on any input of length n. Because of the complexity assumption of 1, A cannot change or use more than p(n) words of memory. We assume that A uses some subset of the words indexed 1, 2, 3, ...p(n). This assumption does not restrict the class of decision problems in NP. To see this, let f (1), f(2), ...,f(k), $1 \le k \le p(n)$, be the distinct words used by A while working on input I. We can construct another polynomial time nondeterministic algorithm A' that uses 2p(n) words index 1, 2, ..., 2p(n) and solves the same decision problem as A does. A' simulates the behaviour of A. However, A' maps the addresses f (1), f(2), ..., f(k) onto the set {1, 2, ..., k}. The mapping function used is determined dynamically and is stored as a table in words p(n) + 1 through 2p(n).

If the entry at word p(n) + i is j, then A' uses word i to hold the same value that A stored in word j. The simulation of A proceeds as follows. Let k be the number of distinct words referenced by A upto this time. Let j be a word referenced by A in the current step. A' searches its table to find word p(n) + i, $1 \le i \le k$, such that the contents of this word is j. If no such i exists, then A' sets k := k + 1; i := k, and word p(n) + k gives the value j. A' makes use of the word i to do whatever A would have done with word j. Clearly, A' and A solve the same decision problem.

The complexity of A, is $O(p^2(n))$ as it takes A'p(n) time to search its table and simulate a step of A. Since $p^2(n)$ is also a polynomial in n, restricting our algorithms to use only consecutive words that does not alter the classes P and NP.

Q.13 ***Write algorithm for approximation for set cover problem with suitable example.*** *[R.T.U. 2018, 2014]*

OR

Explain approximation algorithm for vertex cover. *[R.T.U. 2017, 2013]*

OR

Explain Approximation Algorithms for Vertex and Set Cover problem. *[R.T.U. 2016]*

OR

Explain set cover problem in detail? *[R.T.U. 2012]*

OR

Explain vertex and set cover problem. *[R.T.U. 2011, 2010, Raj. Univ. 2008, 2006, 2005, 2003]*

Ans. Vertex : The vertex cover takes a graph G and integer K as input and asks whether there exists a vertex cover for G which contains at most K vertex or not. It is already noticed that vertex cover is in NP. Now we have to show that it is NP-hard. For this we have to reduce the 3-SAT problem in polynomial time. This reduction is accomplished in two steps :

- First, it represents an example in which a logic problem is reduced to a graph problem.
- Second, it describes an application of the component design proof technique.

Let us consider that 'B_{fg}' be a given instance of the 3-SAT problem, that is, a CNF Boolean formula, where each clause has exactly three literals. Now, we create a graph G and an integer K such that G has a vertex cover of size at

most K if and only if B_{fg} is satisfiable. For this we add the following :

- For each input operand I in the Boolean formula 'B_{fg}', we add two vertices in G, one of which is labelled as I, and other as $\bar{I}$. After this we add the edge $(I_i, \bar{I}_i)$.
- For each clause C = (m + n + z) in B_{fg}, we form a triangle consisting of three vertices and three edges.
- At least two vertices per triangle must be in the cover for the edges in the triangle, for a total of at least 2C vertices.
- Lastly, we create a flat structure where each literal is connected to the corresponding vertices in the triangle which shares the same literal.

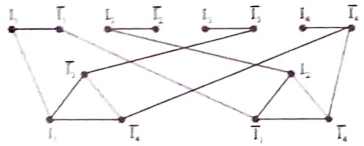


Fig.

The above graph will have a vertex cover of size n + 2C if and only if the expression is satisfiable. Every cover must have at least n + 2C vertices. For showing that our reduction is correct, we have to show the following.

For every satisfying truth assignment there exists a cover : For this select the n vertices that correspond to the true literals to be in the cover. As it is a satisfying truth assignment, atleast one of the three cross edges associated with each clause must already be covered. Now, select the other two vertices to complete the cover.

There exists a satisfying truth assignment for every vertex cover : For this, every vertex cover must contain n first level vertices and 2C second level vertices. Let the truth assignment be defined by the first level vertices. To get the cover at last, one cross-edge must be covered, so that the truth assignment satisfies.

It can be noticed that for a cover to have n + 2C vertices, all the cross edges must be incident on a selected vertex. Let us consider that the n selected vertices from the first level corresponds to true literals. If there exists a satisfying truth assignment, then that means atleast one of the three cross edges from each triangle is incident on a true literal vertex. It is to be noted that by adding the other two vertices to the cover, we cover all the edges associated with the clause.

Vertex-cover problem is to find a vertex cover of minimum size. Using approximation algorithm we have to find a sub-optimal solution to the problem. As a result of this algorithm, we will get a vertex-cover with size no more than twice the size of an optimal vertex cover.

Algorithm Approx_ vertex_ cover

Input to the algorithm is the graph G.

Step 1. Initialize the vertex-cover D to be null.

C ← ϕ

Step 2. The set of edges in G is E.

Step 3. Repeat steps 4 to 6 till the set of edges E is empty.

Step 4. Choose an arbitrary edge (u, v) of E.

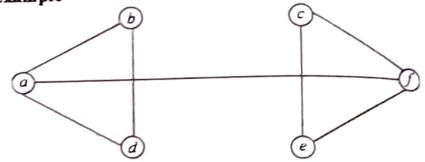
Step 5. Add the endpoints u, v to vertex cover C.

Step 6. Remover every edge incident on either u or v from the set of edges E.

Step 7. return C and Exit.

The running time of this algorithm is O(V + E).

Example



C = ϕ

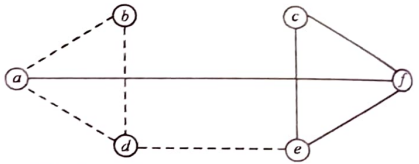
E = {ab, ad, bd, de, af, cf, ef, ce}

pick bd aribitrarily

C = {b, d}

Remove the edges associated with b or d, that is ab, bd, ad and de.

Now E = {af, cf, ef, ce}



Pick cf at random

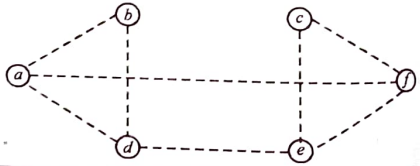
C = {b, d, c, f}

Remove edges associated with c or f, that is af, cf, ef and ce

Now E = ϕ

So, stop

The cover is c



Set Cover problem : Though set cover is a generalized vertex cover, the algorighm is slightly different from that developed above. A set cover is a set of subsets that collectively include all elements of the parent set. The problem can be approached straightforwardly. We can arbitrarily pick a subset and put in the cover to initiate. After this we can look for a subset which has maximum uncovered elements. We may require an auxiliary set of elements covered till now for this purpose. If we perform intersection of the covered elements set and subset, we can choose the subset with minimum intersection. The process can be continued till all elemetns are covered. If we maintain a set of uncovered elements, and select the subset with maximum intersection, we can use it for initialization also.

Thus, the algorithm developed is shown in Fig., which takes input the parent set S and family of subsets S_n. The set cover C is intialized in step 1 as empty set. U is the set of uncovered elements, hence it is initialized with all elements of S, as in step 2. We repeat the loop of step 3 till there are no elements left uncovered. Step 4 selects that subset which has maximum uncovered elements, by performing intersection of every remaining subset S_i with U, and picking the one which has maximum intersection. Since these elements are covered now, remove them from U by set difference operation in step 5. The selected subset in included in the cover, through step 6.

Set_Cover (S, S_n)

// S is the set to be covered

Step 1 : C = ϕ.

Step 2 : U = S.

Step 3 : Until U is empty, repeat step 4 to 6.

Step 4 : Pick set S_i with max $|S_i \cap U|$.

Step 5 : $U = U - S_i$.

Step 6 : $C = C \cup \{S_i\}$.

Step 7 : return C

Example

S = {a, b, c, d, e, f, g, h}

S_1 = {a, b, c}

S_2 = {b, d, f, g}

S_3 = {a, e, f, g}

S_4 = {c, d, e, h}

S_5 = {a, h}

S_6 = {b, c, f, h}

Initially,

C = ϕ; U = S = {a, b, c, d, e, f, g, h}

Pick S_i with max $|S_i \cap U| = S_2$

$\therefore$ $C = C \cup \{S_2\} = \{S_2\}$

$U = U - S_2 = \{a, c, e, h\}$

Pick S with max $|S_i \cap U| = S_1$

$C = C \cup \{S_1\} = \{S_1, S_2\}$

$U = U - S_1 = \{e, h\}$

Pick S_i with max $|S_i \cap U| = S_4$

$C = C \cup \{S_4\} = \{S_1, S_2, S_4\}$

$U = U - S_4 = \phi$

So stop.

Hence the set-cover is $\{S_1, S_2, S_4\}$.

Q.14 ***Assuming 3-CNF satisfiability problem to be NP-complete, prove clique problem is also NP-complete.*** *[R.T.U. 2017, 2011, 2010]*

Ans. A special case of SAT that is particularly useful in proving NP-hardness results is called 3-SAT.

$T_{CSAT}(n) \le O(n) + T_{3SAT}(O(n)) \Rightarrow T_{3SAT}(n) \ge T_{CSAT}(\Omega(n) - O(n))$

As 3SAT is NP-hard and because 3SAT is a special case of SAT, it is also in NP. Therefore, 3SAT is NP-complete.

Clique is NP-complete.

Proof : It is easy to verify that a graph has clique of size k if we guess the vertices forming the clique. We merely examine the edges. This can be done in polynomial time. We shall now reduce 3-SAT to Clique. We are given a set of k clauses and must build a graph which has a clique if and only if the clauses are satisfiable. The literals from the clauses become the graph's vertices. And collection of true literals shall make up the clique in the graph we build. Then a truth assignment which makes at least one literal true per clause will force a clique of size k to appear in the graph. And, if no truth assignment satisfies all of the clauses, there will not be a clique of size k in the graph. To do this, let every literal in every clause be a vertex of the graph we are building. We wish to be able to connect true literals but not two from the same clause. And two which are complements cannot both be true at once. So, connect all of the literals which are not in the same clause and are not complements of each other. We are building the graph G = (V, E) where:

$V = \{\langle x, i \rangle | x \text{ is in the } i^{th} \text{ clause}\}$

$E = \{\langle x, i \rangle, \langle y, j \rangle | x \neq y \text{ i } j\}$

Now we shall claim that if there were k clauses and there is some truth assignment to the variables which satisfies them, then there is a clique of size k in our graph. If the clauses are satisfiable then one literal from each clause is true. That is the clique. Because a collection of literals (one from each clause) which are all true cannot contain a literal and its complement. And they are all connected by edges because we connected literals not in the same clause (except for complements). On the other hand, suppose that there is a

clique of size k in the graph. These k vertices must have come from different clauses since no two literals from the same clause are connected. And, as literal and its complement are in the clique, so setting the truth assignments to make the literals in the clique true provides satisfaction. A small inspection reveals that the above transformation can indeed be carried out in polynomial time. Thus the Clique problem has been shown to be NP-hard.

Theorem : SAT reduces to the k-clique problem.

Proof

- Let F be a Boolean expression.
- F can be given into a conjunctive normal form $A = f_1 f_2 \ldots F$ where every factor F_i is a sum of literals (a literal is a Boolean variable or its complement).
- Let x and $G = (V, E)$ defined as follows:
  $V = \{(x, F_i) : x$ is a variable in $F_i\}$
  $E = \{((x, F_i), (y, F_j)) : F_i \neq F_j$ and $y \neq \bar{x}\}$
  where $\bar{x}$ is the complement of x.
- We prove first that if F is satisfiable, then there is a k-clique.
- Assume F is satisfiable.
- This mean that there is an assignment that makes F equal to 1.
- Thus implies that $F = F_1 F_2 \ldots F_k = 1$.
- Therefore, in every factor F_i there is (at least) one variable assigned '1' all that variable x.
- As a result $(x, F_1), (y, F_2), \ldots, (z, F_k)$ is a k-clique in G because they are k distinct nodes and each pair $((x, F_i), (y, F_j))$ forms an edge since the end points come from different factors and $y \neq \bar{x}$ due to the fact that they are both assigned 1.
- We finally prove that if G has k-clique, then F is a satisfiable.
- Assume G has a k-clique $(x_1, F_1), (x_2, F_2), \ldots, (x_k, F_k)$ which are pair-wise adjacent.
- These k nodes come from a different factors, one per factor because no two nodes from the same factor can be adjacent.
- Furthermore, no two x_i and x_j are complements because the two nodes (x_i, F_i) and (x_j, F_j) are adjacent and adjacent nodes have non-complement literal components.
- As a result, we can consistently assign each x_i a value 1.
- This assignment makes each F_i equal to 1 because x_i is one of the additive literals in F_i.
- Consequently, F is equal to 1.

Q.16 Write short note on NP-completeness. *(R.T.U. 2013)*

OR

Explain NP and Hard NP Complete with example. *(R.T.U. 2016)*

OR

Explain the terms P, NP, NP-Hard, NP-complete with suitable example. Also give relationship between them. *(R.T.U. 2014)*

OR

Define the term P, NP, NP-complete. Give suitable examples of each. *(R.T.U. 2015)*

OR

Define the terms P, NP, NP complete and NP hard problems. *(R.T.U. 2012)*

OR

Explain the terms P, NP, NP-complete.

(R.T.U. 2009, Raj. Univ. 2008, 2007, 2005, 2004, 2003)

Ans.(i) P : P is the set of decision problems with a yes-no answer that is polynomial bound.

A problem is said to be polynomial–bound if there exists a polynomial bound algorithm for it. It is also to be noted that not for all the problems the class P has "acceptably efficient" algorithm. Also, if a problem does not belong to class P then it is **intractable**.

Note : An algorithm is said to be polynomial bounded if its worst–case complexity is bound by a polynomial function of input size n. That means, for each input of size n the algorithm terminates after atmost $P(n)$ steps. For instance, $n^2 + 2n + 6$.

Decision Problems : The problems under this class have the single bit output which shows 'yes' or 'no'. The answer for the problem is either zero or one.

For instance, some decision problems are

- Given two sets of strings S_1 and S_2, does S_1 a subset of S_2 ?
- Given two sets of elements S_1 and S_2, does both the sets contain same number of elements ?

Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as optimization problem. For solving optimization problem, an optimization algorithm is used. For instance, the optimization problem is as follows :

- Given a weighted graph G_1 and an integer i, does G have a minimum spanning tree of weight at most i.
- Given S_1, does there exist a subset of elements that fits in the knapsack, and has total profit of at least S_2 ?

We can say that a given algorithm A accepts the string S_1 only when A produces the output 'yes' on input S_1. As a set of string is referred to a language, a decision problem can

also be viewed as a set L of strings where L is a language. Thus, an algorithm A accepts language L if A produces the output 'yes' on input 'i' which belong to L otherwise it produces output 'no'.

It is to be noted that the class P problems include all the decision problems (or languages) L that can be accepted in the worst–case running time. Thus, for algorithm A, it accepts $\forall$ L, in polynomial time $p(n)$, where 'n' is the input size of 'i' and produces output 'yes'. But it is noticeable that the class P definition does not say anything about output 'no'. We refer to this situation as a complement of the algorithm A for output 'yes' for a given set of binary strings that are not present in L.

We can also create an algorithm C that accepts the complement of L, if given an algorithm A that accepts a language L in polynomial time, $p(n)$. Therefore, if a language L, showing such decision problem, is in class P.

(ii) NP : The complexity class NP includes the complexity class P but allows for the languages that are not present in P. But, in the case of NP problems we perform an additional operation.

Select : This problem selects a bit (0 or 1) in a non-deterministic way and assigns it to b. When an algorithm A takes the advantage of Select primitive operation then we say that A is non–deterministic. In this approach out of several calls to Select operation, those calls are chosen which lead to acceptance if there exists a set of outcomes. It is noticeable that this operation is working is not same as of using random choices.

The complexity class NP includes all decision problems (or languages L) that can be accepted non-deterministically in the polynomial time. Thus for a given algorithm A, if $S \in L$, an input S, there exists a set of outcomes to the Select calls in A so that it produces output 'yes' in polynomial time, $p(n)$, where n is the input size of S.

The definition of complexity class NP does not say anything about rejection of the string. Algorithm A running in polynomial time $p(n)$ can take more than $p(n)$ steps when A produces outputs 'no'. Also, a polynomial number of calls to select is involved in the non–deterministic acceptance, the complement of L is not necessarily in NP, where L is the language in NP.

There exists a special class, called co–NP which includes all the languages whose complement is in NP, and many researchers and scientists believe that co–NP ≠ NP.

Is P = NP ? : Most of the researchers and scientists believe that class P problems are different from NP and co – NP or their intersection.

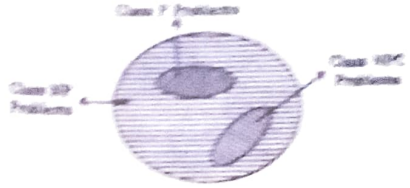


Fig.

Problem : Given a Graph G, does the graph G contain a Hamiltonian cycle. That means whether there exists a simple cycle such that each vertex is visited only once other than the starting vertex or not.

Lemma : Hamiltonian cycle is in NP.

Proof : Here we will define a non–deterministic algorithm A that takes an input graph G encoded as an adjacency list or binary notation, with the vertices labelled from '1' to 'n'.

Then we make A to call the Select method recursively for obtaining the sequence of vertices, and perform the check so that all the vertices appear only once except the start vertex (which comes twice). This can be done by sorting the sequence. Also, we verify that the sequence defines a cycle of vertices and edges in G.

If there exists a cycle in G where all vertices are visited only once except the first and the last which are the same then there also exists the sequence for which A produces output 'yes'. Similarly, we can say that if A outputs 'yes' then a graph G has cycle in such a manner that all vertices are visited once except first and last which are the same. That means A non–deterministically accepts the language Hamiltonian – Cycle. Thus, we can say that Hamiltonian–Cycle is in NP.

(iii) NP–Complete : A decision problem L is NP–complete if it is in class NP for every other problem L' in NP, $L' \propto L$. That means L is polynomially reducible to L. (that means NP–complete problems are the hardest in NP).

Theorem : If any NP–complete problem belongs to class P, then P = NP.

Proof : Let any decision problem $L \in$ NP and also $L \in$ P. Then by the rules of NP–complete problems

$$\forall L' \in NP, L' \propto L, L \in P$$

$$\forall L \in NP, L' \propto L : L \in P$$

Then according to the polynomial reducibility (if $L_1 \propto L_2$ and $L_2 \in P$ then $L_1 \in P$)

$$\forall L \in NP, L \in P$$

That means, P = NP.

All NP problems are NP- hard, but some NP-hard problems are not known to be NP complete.

Since deterministic algorithm are just a special case of non-deterministic ones, we conclude that $P \subseteq NP$

It is easy to see that there are NP- hard problems that are not NP complete. Only a decision problem can be NP complete. However, an optimization problem may be NP- hard. Furthermore if L_1 is a decision problem and L_2 is an optimization problem it is quite possible that $L_1 \propto L_2$. One can trivially show that knapsack decision problem reduces to knapsack optimization problem.

For clique problem one can easily show that the clique decision problem reduces to clique optimization problem. In fact one can also show that these optimization problem reduces to their corresponding decision problem. Yet optimization problem cannot be NP complete whereas decision problem can. There also exist NP- hard decision problem that are not NP complete.

Q.16 ***Write short notes on the following :***

(a) Complexity classes of decision problems.

(b) Approximation algorithms.

[R.T.U. 2015]

Ans.(a) The purposes of complexity theory are to ascertain the amount of computational resources required to solve important computational problems, and to classify problems according to their difficulty. The resource most often discussed is computational time, although memory (space) and circuitry (orhardware) have also been studied. The main challenge of the theory is to prove lower bounds, i.e.,that certain problems cannot be solved without expending large amounts of resources. Although it is easy to prove that inherently difficult problems exist, it has turned out to be much more difficult to prove that any interesting problems are hard to solve. There has been much more success in providing strong evidence of intractability, based on plausible, widely-held conjectures.

In both cases, the mathematical arguments of intractability rely on the notions of reducibility and completeness. Before one can understand reducibility and completeness, one must grasp the notion of a complexity class.

First, however, we want to demonstrate that complexity theory really can prove to even the most skeptical practitioner that it is hopeless to try to build programs or circuits that solve certain problems. As our example, we consider the manufacture and testing of logic circuits and communication protocols. Many problems in these domains are solved by building a logical formula over a certain vocabulary, and then determining whether the formula is logically valid, or whether counter examples (that is, bugs) exist. The choice of vocabulary for the logic is important here.

Typically, a complexity class is defined by

(1) A model of computation

(2) A resource (or collectionof resources)

(3) A function known as the complexity bound for each resource

The models used to define complexity classes fall into two main categories:

(a) Machine based models

(b) Circuit-based models.

Turing Machines (TMs) and Random-Access Machines (RAMs) are the two principal families of machine models. Other kinds of (Turing) machines were also introduced including deterministic, nondeterministic, alternating, and oracle machines. When we wish to model real computations, deterministic machines and circuits are our closest links to reality. Then why consider the other kinds of machines? There are two main reasons. The most potent reason comes from the computational problems whose complexity we are trying to understand. The most notorious examples are the hundreds of natural NP-complete problems. To the extent that we understand anything about thecomplexity of these problems, it is because of the model of non-deterministic Turing machines. Non-deterministic machines do not model physical computation devices, but they do model real computational problems. There are many other examples where a particular model of computation has been introduced in order to capture some well-known computational problem in a complexity.

The second reason is related to the first. Our desire to understand real computational problems has forced upon us a repertoire of models of computation and resource bounds. In order to understand the relationships between these models and bounds, we combine and mix them and attempt to discover their relative power. Consider, for example, nondeterminism. By consideringthe complements of languages accepted by non-deterministic machines, researchers were naturally led to the notion of alternating machines. When alternating machines and deterministic machines were compared, a surprising virtual identity of deterministic space and alternating time emerged.

Subsequently, alternation was found to be a useful way to model efficient parallel computation.

This phenomenon, whereby models of computation are generalized and modified in order to clarify their relative complexity, has occurred often through the brief history of complexity theory, and has generated some of the most important new insights.Other underlying principles in complexity theory emerge from the major theorems showing relationships between complexity classes. These theorems fall into two broad categories. Simulation theorems show that computations in one class can be simulated by computations that meet the defining resource bounds of another class. The containment of Nondeterministic Logarithmic space(NL) in

polynomial time (P), and the equality of the class P with alternating logarithmic space,are simulation theorems. Separation theorems show that certain complexity classes are distinct Complexity theory currently has precious few of these. The main tool used in those separation theorems we have is called diagonalization. This ties in to the general feeling in computer science that lower bounds are hard to prove. Our current inability to separate many complexity classes from each other is perhaps the greatest challenge to our intellect posed by complexity theory.

Time and Space Complexity Classes:

We begin by emphasizing the fundamental resources of time and space for deterministic and nondeterministicTuring machines. We concentrate on resource bounds between logarithmic and exponential,because those bounds have proved to be the most useful for understanding problems that arise in practice.Time complexity and space complexity were defined . Fundamental time classes and fundamental space classes, given functions t(n) and s(n).

- DTIME [t(n)] is the class of languages decided by deterministic Turing machines of time complexity t(n).
- NTIME [t(n)] is the class of languages decided by nondeterministic Turing machines of time complexity t(n).
- DSPACE [s(n)] is the class of languages decided by deterministic Turing machines of space complexity t(n).
- NSPACE [s(n)] is the class of languages decided by nondeterministic Turing machines of space complexity t(n).

We some times abbreviate DTIME [t(n)] to DTIME[t] (and so on) when t is understood to be a function, and when no reference is made to the input length n.

Canonical Complexity Classes

The following are the canonical complexity classes:

- L = DSPACE [log n] (determine log space)
- NL = NSPACE [log n] (nondeterministic log space)
- $P = DTIME [n^{O(1)}] = U_{k\geq 1} DTIME [n^k]$ (nondeterministic polynomial time)
- $PSPACE = DSPACE [n^{O(1)}] = U_{k\geq 1} DSPACE [n^k]$ (polynomial time)
- $E = DTIME [2^{O(n)}] = U_{k\geq 1} DTIME [k^n]$
- $NE = NTIME [2^{O(n)}] = U_{k\geq 1} NTIME [k^n]$
- $EXP = DTIME [2^{n^{O(n)}}] = U_{k\geq 1} DTIME [2n^k]$ (deterministic exponential time)
- $NEXP = NTIME [2^{n^{O(1)}}] = U_{k\geq 1} NTIME [2n^k]$ (nondeterministic exponential time)
- $EXPSPACE = DSPACE [2^{n^{O(1)}}] = U_{k\geq 1} DSPACE [2n^k]$ (exponential time)

The space classes PSPACE and EXPSPACE are de_ned in terms of the DSPACE complexity measure.By Savitch's Theorem.

Ans.(b) An approximation algorithm returns a solution to a combinatorial optimization problem that is probably close to optimal (as opposed to a heuristic that may or may not find a good solution). Approximation algorithms are typically used when finding an optimal solution is intractable, but can also be used in some situations where a near-optimal solution can be found quickly and an exact solution is not needed.

Many problems that are NP-hard are also non-approximable assuming P?NP. There is an elaborate theory that analyzes hardness of approximation based on reductions from core non-approximable problems that is similar to the theory of NP-completeness based on reductions from NP-complete problems; we will not discuss this theory in class but a sketch of some of the main results can be found , which is also a good general reference for approximation. Instead, we will concentrate on some simple examples of algorithms for which good approximations are known, to give a feel for what approximation algorithms look like.

- 2-approximation for vertex cover via greedy matchings.
- 2-approximation for vertex cover via LP rounding.
- Greedy O(log n) approximation for set-cover
- Approximation algorithms for MAX-SAT.

Suppose we are given an NP-complete problem to solve. Even though (assuming P 6= NP) we can't hope for a polynomial-time algorithm that always gets the best solution, can we develop polynomial-time algorithms that always produce a "pretty good" solution? We consider such approximation algorithms, for several important problems.

Suppose we are given a problem for which (perhaps because it is NP-complete) we can't hope for a fast algorithm that always gets the best solution. Can we hope for a fast algorithm that guarantees to get at least a "pretty good" solution? E.g., can we guarantee to find a solution that's within 10% of optimal? If not that, then how about within a factor of 2 of optimal? Or, anything non-trivial? The class of NP-complete problems are all equivalent in the sense that a polynomial-time algorithm to solve any one of them would imply a polynomial-time algorithm to solve all of them (and, moreover, to solve any problem in NP). However, the difficulty of getting a good approximation to these problems varies quite a bit. In this lecture we will examine several important NP-complete problems and look at to what extent we can guarantee to get approximately optimal solutions, and by what algorithms.

Approximates Strategies :

We will define optimization problems in a traditional way. Each optimization problem has three defining features: the structure of the input instance, the criterion of a feasible solution to the problem, and the measure function used to determine which feasible solutions are considered to be optimal. It will be evident from the problem name whether we desire a feasible solution with a minimum or maximum measure. To illustrate, the minimum vertex cover problem may be defined in the following way.

Instance : An undirected graph $G = (V, E)$

Solution: A subset $S \subseteq V$ such that for every $\{u, v\} \in E$, either $u \in S$ or $v \in S$.

Measure : $|S|$

We use the following notation for items related to an instance I.

- Sol(I) is the set of feasible solutions to I.
- m_I : Sol(I) $\rightarrow$ R is the measure function associated with I, and

Opt(I) $\subseteq$ Sol(I) is the feasible solutions with optimal measure (be it minimum or maximum).

Hence, we may completely specify an optimization problem Π by giving a set of tuples {(I, Sol(I), m_I, Opt(I))} over all possible instances I. It is important to keep in mind that Sol(I) and I may be over completely different domains. In the above example the set of I is all undirected graphs, while Sol(I) is all possible subsets of vertices in a graph.

Approximation and Performance : Roughly speaking, an algorithm approximately solves an optimization problem if it always returns a feasible solution whose measure is close to optimal. This intuition is made precise below.

Let Π be an optimization problem. We say that an algorithm A feasibly solves Π if given an instance $I \in \Pi$, $A(I) \in$ Sol(I), that is, A returns a feasible solution to I.

Let A feasibly solve Π. Then we define the approximation ratio $\alpha(A)$ of A to be the minimum possible ratio between the measure of A(I) and the measure of an optimal solution. Formally,

$$\alpha(A) = \min_{I \in \Pi} \frac{m_I(A(I))}{m_I(Opt(I))}$$

For minimization problems, this ratio is always at least 1. Respectively, for maximization problems, it is always at most 1.

Complexity Background : We define a decision problem as an optimization problem in which the measure is 0 – 1 valued. That is, solving an instance I of a decision problem corresponds to answering a yes/no question about I (where yes corresponds to a measure of 1, and no corresponds to a measure of 0). We may therefore represent a decision problem as a subset S of the set of all possible instances. The members of S represent instances with measure 1.

Informally, P (polynomial time) is defined as the class of decision problems Π for which there exists a corresponding algorithm A_Π such that every instance $I \in \Pi$ is solved by A_Π within a polynomial ($|I|^k$ for some constant k) number of steps on any "reasonable" model of computation. Reasonable models include single-tape and multi-tape Turing machines, random access machines, pointer machines, etc.

❑❑❑

WIRELESS COMMUNICATION (ELECTIVE)

V C.S.

1 WIRELESS CHANNELS

IMPORTANT QUESTIONS

PART-A

Q.1 ***For a channel with Doppler spread $B_d = 80Hz$, what time separation is required in samples of the received signal such that the samples are approximately independent.***

Ans. The coherence time of the channel is $T_c \approx 1/B_d = 1/80$, so samples spaced 12.5 ms apart are approximately uncorrelated and thus, given the Gaussian properties of the underlying random process, these samples are approximately independent.

Q.2 ***Consider an indoor wireless LAN with $f_c = 900MHz$, cells of radius 100 m, and non-directional antennas. Under the free-space path loss model, what transmit power is required at the access point such that all terminals within the cell receive a minimum power of 10μW. How does this change if the system frequency is 5 GHz?***

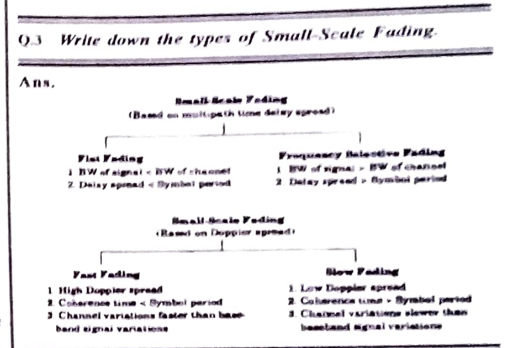
Ans. We must find the transmit power such that the terminals at the cell boundary receive the minimum required power.

$$P_t = P_t \left[\frac{4\pi d}{\sqrt{G_t}\lambda} \right]^2$$

Substituting in $G_t = 1$ (non-directional antennas), $\lambda = c/f_c = 0.33$ m, $d = 10$ m, and $P_t = 10\mu W$ yields $P_t = 1.45W = 1.61$ dBW (recall that P Watts equals $10 \log_{10}[P]$ dBW, dB relative to one Watt, and $10\log_{10}[P/.001]$ dBm, dB relative to one milliwatt). At 5 GHz only $\lambda = .06$ changes, so $P_t = 43.9$ KW = 16.42 dBW.

Q.3 ***Write down the types of Small-Scale Fading.***

Ans.



Q.4 ***Define coherence bandwidth.***

Ans. Coherence Bandwidth : The coherence bandwidth is related to the specific multipath structure of the channel. The coherence bandwidth is a measure of the maximum frequency difference for which signals are still strongly correlated in amplitude.

Q.5 ***Distinguish coherence time and coherence bandwidth.***

Ans. Corehence Bandwith: Refer to Q. 4.

Coherence bandwidth is inversely proportional to the rms value of time delay spread. The coherence timer is defined as the required time interval to obtain an envelope correlation of 0.9 or less.