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:***

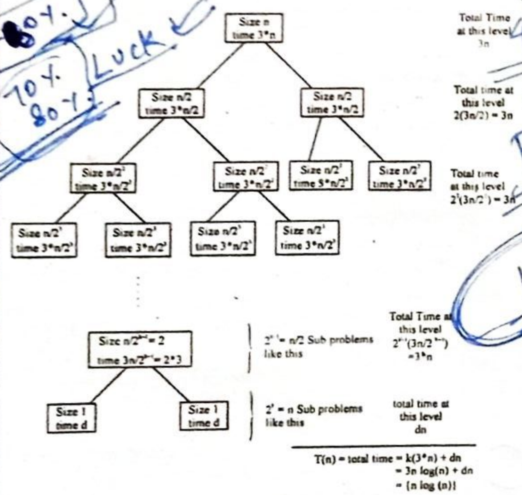
$T(n) = 2 \quad if\ n = 2$

$T(n) = 2T(n/2)+3n \quad if\ n > 2$ **[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 $2(n = 2^k, k = \log(n))$

10-80% 70% 80% LUCK



Substitution method

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

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

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

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

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

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

$= n\log n$

AOA 40Lec RTU 100%

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

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

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

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

then

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \quad .....(3)$$

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.

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned} \quad .....(4)$$

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned} \quad .....(5)$$

The resulting recurrence relation for T(n) is

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

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 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 $n^{\log_b a} = n^{\log_3 4}$

Since $f(n) O(n^{\log_3 4-\epsilon})$ 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

$T_{w\ seq\ search}(n) = n$
$= \Theta(n)$

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 $0(n)$.

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

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

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

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

$$= \Theta(n)$$

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

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

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

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

$P_1 = a(f - n)$
$= 3(5 - 6)$
$= -3$

$P_2 = (a + b)n$
$= (3 + 2) \times 6$
$= 30$

$P_3 = (c + d)e$
$= (4 + 8) \times 1$
$= 12$

$P_4 = d(g - e)$
$= 8(9 - 1)$
$= 64$

$P_5 = (a+d)(e + n)$
$= (3 + 8)(1 + 6)$
$= 77$

$P_6 = (b - d)(g + h)$
$= (2 - 8)(9 + 6)$
$= -90$

$P_7 = (a - c)(e + f)$
$= (3 - 4)(1 + 5)$
$= -6$

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

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

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

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

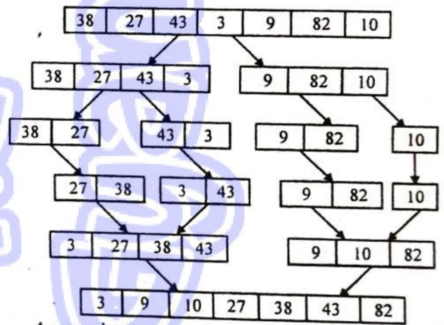
Ans. Analysis of Merge Sort

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

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,

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

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

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]**

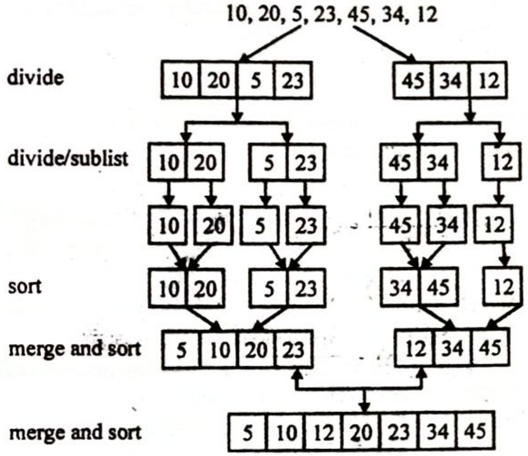
OR

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. $p \leftarrow 1$
2. $r \leftarrow n$
3. $q \leftarrow (p + r)/2$
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. $L_1[1.... n_1 + 1]$, $L_2[1.... n_2 + 1]$ 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_o,

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

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, $n \geq n_o$

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

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

$$0 \leq 100n + 6 \leq c\, n$$

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

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

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_o, $f(n) \geq cg(n)$ for all n, $n \geq n_o$.

For all values of $n < n_o$ 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' ($i.e., n \geq n_o$).

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 $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_o$. 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,

$$c_1 g(n) \leq c_2 g(n)$$

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

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

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

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

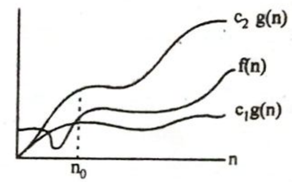
Whereas, for $3n + 5 \neq \omega(1)$,

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

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.

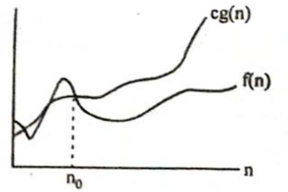
(a) θ-notation bounds a function to within constant factors. We write $f(n) = \theta(g(n))$ if there exist positive constants n_0,c_1, and c_2 such that to the right of n_0, the value of f(n) always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive.



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).

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

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

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

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

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

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]*

Master

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

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

$\log_2^3 = 0.7298$

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 ≤ 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.

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

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

$$\begin{aligned} T(n) &= n\theta(1) + n\sum_{j=1}^{\log_5 n-1} (n/(\log(n/S^{j-1}))) \\ &= \theta(n) + n\sum_{j=1}^{\log_5 n-1} (1/\log n - (j-1)\log_2 5) \\ &= \theta(n) + n(1/\log_2 5)\sum_{j=1}^{\log_5 n-1} (1/(\log_5 n - (j-1))) \\ &= \theta(n) + n\log_5 2\sum_{i=2}^{\log_5 n}\left(\frac{1}{i}\right) \end{aligned}$$

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

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

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

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

$$\begin{aligned} T(n) &\le cn\left(\frac{n}{3}+5\right)\log\left(\frac{n}{3}+5\right)+\frac{n}{2} \\ &= cn \log\left(\frac{n}{3}\right)+10+\frac{n}{2} \\ &= cn\log n - cn\log 3 + 10 + \frac{n}{2} \\ &= cn\log n - cn + 10 + \frac{n}{2} \\ &= cn\log n - (c - 1/2)n + 10 \\ &= cn \log n - b \le cn \log n \end{aligned}$$

if $c \ge 1$, b is constant

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

(v) We solve this problem by algebraic subsitution

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

□□□

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.

EARS QUESTIONS

PART-B

Algo

mcm

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]*

Theory

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			

mcm

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 $(m_1\ m_{23})]$

$= \min[160 + (4 \times 4 \times 40), 1600 + (4 \times 10 \times 40)]$

$= [800, 3200] = 800$

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

$= \min[1600 + (10 \times 40 \times 5), 800 + (10 \times 4 \times 5)]$

$= \min[3600, 1000]$

$= 1000$

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

$= \min[800 + (4 \times 40 \times 5), 1000 + (4 \times 10 \times 5)]$

$= \min[1600, 1200]$

$= 1200$

	4	3	2	1
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 X_i == Y_j
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_i : D Y_j = D	Step : X_i = 'D' equal to Y_j='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
		y	B	A	C	C	D	C	A	B	B	D
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	↑1	↖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	[illegible]	↑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	[illegible]	[illegible]	←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

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

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

$$\Rightarrow \quad m_{12} = 0 + 0 + 15 \times 10 \times 20$$
$$= 3000$$

When i = 1

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

When i = 2

$$= 0 + 0 + 10 \times 20 \times 25$$
$$= 5000$$

Similarly, we find the value for whole table of M

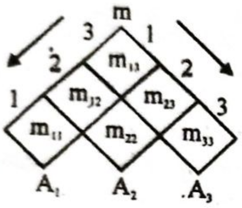


Fig. General structure of table

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 v_e calculate the minimum scalar multiplication and is added to obtain the find matrix value.

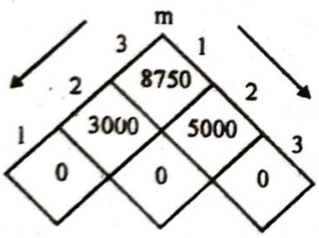


Fig. Table after filling values of m.

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

0/1 → knapsack

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

= {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

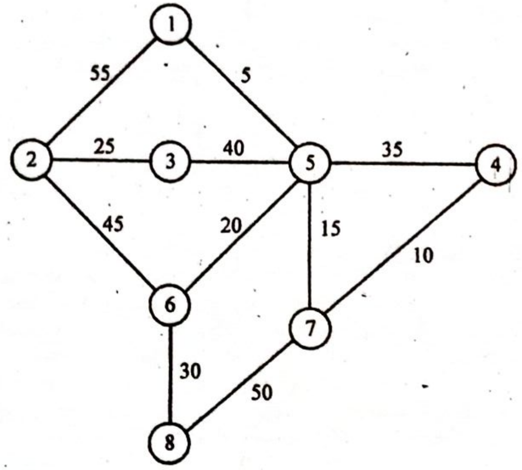
What is the difference between divide and conquer and dynamic programming method? Explain with example. ***[R.T.U. 2014]***

(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.

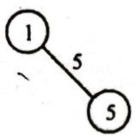
Q.17 *Trace the Kruskal's algorithm to obtain minimum spanning tree from the graph.*



[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.

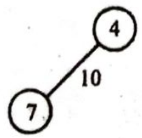


So, A = A ∪ {1, 5}

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}

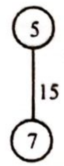


So, A = A ∪ {4, 7}

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 ∪ {5, 7}

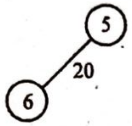
Thus, at this step updated sets are :

{(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}

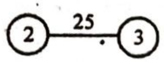


So, A = A ∪ {5, 6}

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}

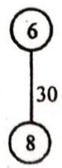


So, A = A ∪ {2, 3}

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}

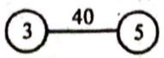


So, A = A ∪ {6, 8}

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.



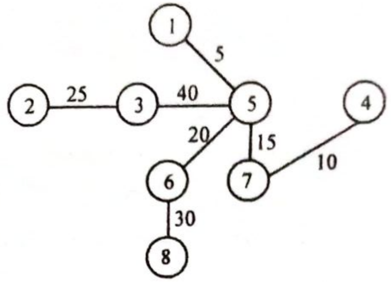
So, A = A ∪ {5, 3}

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.



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

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

$$= 145$$

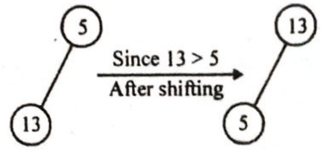
Q.18 *Illustrate the operation of heap on following array:*

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

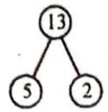
Ans. Creation of a Maximum heap

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

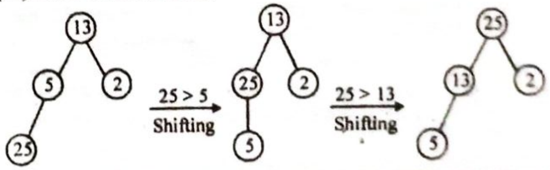
(ii) Next element is 13. Make it as left child of root node and compare that whether the element inserted is smaller than the parent node or not, then this element should be shifted with the parent node.



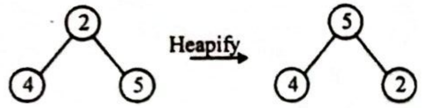
(iii) Next element is 2



(iv) Next element is 25.

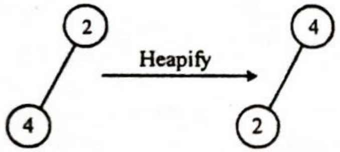


Next iteration, exchange node 1 with node 4.



5	4	2	7	8	13	17	20	25

Next iteration, exchange node 1 with node 3



4	2	5	7	8	13	17	20	25

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 $x_i = y_i$

- 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

i	j	0	1	2	3	4
		y_i	b	a	b	b
0	x_i	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] "↖"

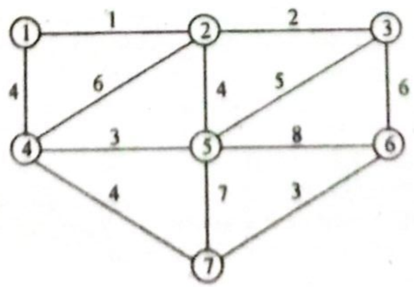
Similarly we fill all values of c[i, j] and finally we get,

		0	1	2	3	4
		y_i	b	a	b	b
0	x_i	a	[illegible]	0	0	0
1	a	0	↑0	↖1	←1	←1
2	a	0	←0	↖1	←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

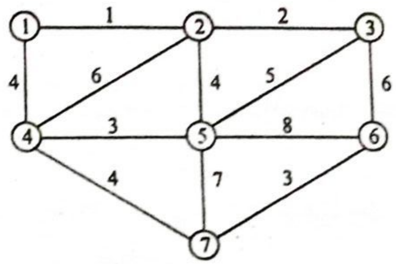
Z = < a ,b, b>

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

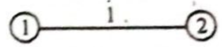


[R.T.U. 2010, Raj. Univ. 2006,2004]

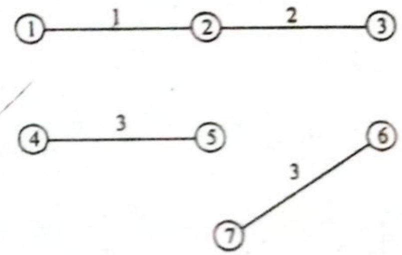
Ans. Kruskal's Method : Minimum spanning tree using Kruskal's method :

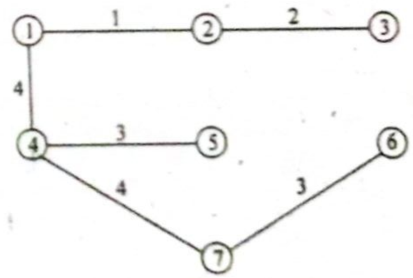


Edge is selected in such a manner that it contains a minimum weight and adding to 'M' does not includes any cycle.

(1)


(2)


(3)


(4)


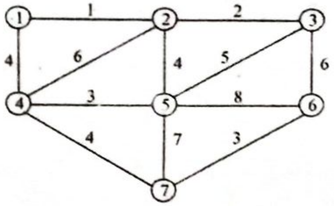
(5) We will not add next 4 weight edge 2 to 5 to 'M' because by adding this will get the closed paths.

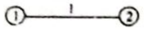
(6) We will not add 5 weight edge 5 to 3 to 'M'.

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

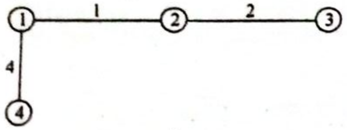
(8) So as above we will not add 6, 7 and 8 weight edges to 'M'.

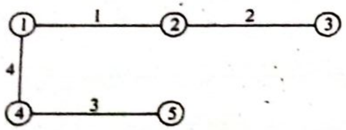
Prim's Method : We start from source node and add the edge to 'M' which is having the least weight among the edges connected to that node. And that will not make a closed path.

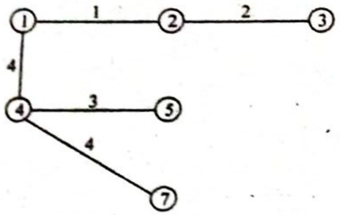


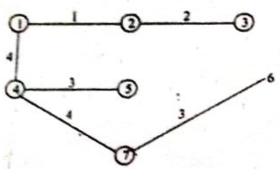
(1)


(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	0	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	4	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.

AA.30

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

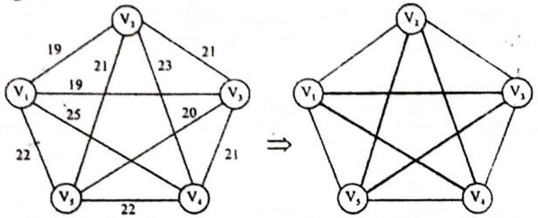
w = (30 + 20), value = (100 + 120) = 220

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.

- The method is repeated until all the cities are visit and at every step we have to select the city with the minimu 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.

S YEARS QUESTIONS

Q.5 ***What do you mean by bad character heuristic.***

Ans. As the name suggests it concentrates on the "b character" in the text where the mismatch has occurred. that character is not contained in P, then the pattern is shift after "bad character" in the string for further matching. B if the "bad character" is somewhere in the pattern then v search for the right most appearance of the "bad characte in the pattern and match it against the text.

PART-B

Q.6 ***Using Rabin Karp algorithm to solve the fe T = 2359023141526739921 and P = 31415 an modulo q = 13, m = 5.*** *[R.T.U. 2013, 20]*

OR

Given the text T = {2, 3, 5, 9, 0, 2, 3, 1, 4, 1, 5, 2, 7, 3, 9, 9, 2, 1} and P = {3, 1, 4, 1, 5} and modu q = 13, m = 5. Choose the pattern matching wi average case complexity and explain the sear process. Justify the answer for choosing su algorithm. *[R.T.U. 20]*

OR

Explain Rabin Karp method with suitable exampl Also give the algorithm for the same. *[R.T.U. 20]*

Ans. The Rabin Karp algorithm involves both the steps pattern matching, *i.e.*, preprocessing and matching. For bett explanation, we assume that string character conta

$\Sigma = \{0, 1, \ldots 9\}$ 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,

$$P[1....m] = T[S + 1 \ldots S + m]$$

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:

$p = P[m] + 10(P[m-1] + 10(P[m-2] + \ldots 10(P[2] + 10\,P[1])\ldots))$

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

T	2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

S = 0

P	3	1	4	1	5

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	2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

S = 1

P	3	1	4	1	5

$t_{s+1} = (d(t_s - T[S + 1]h) + T[S + m + 1])$ mod 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, $t_{s+1} = t_2 = (10(23590 - 2 \times 10000) + 2)$ mod 13

= 9 ≠ 7

So, pattern unmatched and shift is applied.

Step 3

T	2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

S = 2

P	3	1	4	1	5

For S = 2, $t_{s+1} = t_3 = (10(35902 - 3 \times 10000) + 3)$ mod 13

= 59023 mod 13

= 3 ≠ 7

Pattern unmatched and shift is applied.

Step 4

T	2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

S = 3

P	3	1	4	1	5

For S =3, $t_{s+1} = t_4 = (10(59023 - 5 \times 10000) + 1)$ mod 13

= 90231 mod 13

= 11 ≠ 7

Pattern unmatched and shift is applied.

Step 5

T	2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

S = 4

P	3	1	4	1	5

For S = 4, $t_{s+1} = t_5 = (10(90231 - 9 \times 10000) + 4)$ mod 13

= 2314 mod 13

= 0 ≠ 7

Pattern unmatched and shift is applied.

Step 6

T	2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

S = 5

P	3	1	4	1	5

For S = 5, $t_{s+1} = t_6 = (10(2314 - 0 \times 10000) + 1)$ mod 13

= 23141 mod 13

= 1 ≠ 7

Pattern unmatched and shift is applied.

Step 7

T	2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

S = 6

P	3	1	4	1	5

For S = 6, $t_{s+1} = t_7 = (10(23141 - 2 \times 10000) + 5)$ mod 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 pr[ocessed]. Thus, a position can be obtained from where t[o] restart the matching when a mismatch occurs immediatel[y] after jth position. It can be accomplished in the followin[g] manner.

- From the original string consider a substring 0..j.
- Starting at the end of the substring, "slide" it along th[e] original string.
- We have a potential restart position for a mismatch aft[er] jth position, where all overlapping character matches.
- Take an auxiliary array - "next", and store that large[st] 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] (true)
Step 5. then k ← 1
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 = lrrrlrrlllrrrlrrlrrllrlrrlrrlr 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 = lrrrlrrlllrrrlrrlrrllrlrrlrrlr

(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)

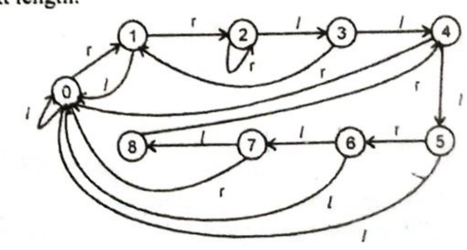


Fig.

Q.9 … Ans. ≤ t(a … satisf… whic… sales… a tou… spann… Appl… 1. 2. 3. 4. propo… Fig. : illustr… spann… like … tour, … verti… from … mini… mini… of an…

PART-C

Q.15 *Solve the TSP problem having the following cost matrix using branch and bound technique.*

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & X & 5 & 2 & 3 \\ B & 4 & X & 2 & 3 \\ C & 4 & 2 & X & 3 \\ D & 7 & 6 & 8 & X \end{array}$$

[R.T.U. 2018, 2011]

Ans. According to cost matrix

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & X & 5 & 2 & 3 \\ B & 4 & X & 2 & 3 \\ C & 4 & 2 & X & 3 \\ D & 7 & 6 & 8 & X \end{array}$$

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

$$RC = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 3 & 0 & 1 \\ B & 2 & \infty & 0 & 1 \\ C & 2 & 0 & \infty & 1 \\ D & 1 & 0 & 2 & \infty \end{array}$$

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

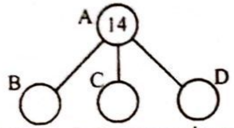
Hence, the reduced matrix is

$$RC = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 3 & 0 & 0 \\ B & 1 & \infty & 0 & 0 \\ C & 1 & 0 & \infty & 0 \\ D & 0 & 0 & 2 & \infty \end{array}$$

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.

$$FC = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & 0 & 0 \\ C & 1 & \infty & \infty & 0 \\ D & 0 & \infty & 2 & \infty \end{array}$$

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.

$$FC = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & 1 & \infty & \infty & 0 \\ C & \infty & 0 & \infty & 0 \\ D & 0 & 0 & \infty & \infty \end{array}$$

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] = ∞.

$$FC = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & 1 & \infty & 0 & \infty \\ C & 1 & 0 & \infty & \infty \\ D & \infty & 0 & 2 & \infty \end{array}$$

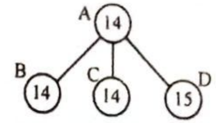
Subtract 1 from column A.

$$FC = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & 0 & \infty & 0 & \infty \\ C & 0 & 0 & \infty & \infty \\ D & \infty & 0 & 2 & \infty \end{array}$$

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,

$$FC = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & \infty & \infty \\ C & 1 & 0 & \infty & 0 \\ D & 0 & 0 & \infty & \infty \end{array}$$

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 ∞.

$$FC = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & \infty & \infty \\ C & 1 & 0 & \infty & \infty \\ D & \infty & \infty & 2 & \infty \end{array}$$

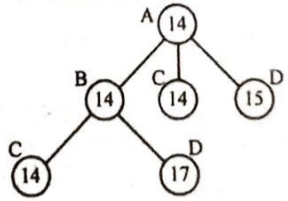
Subtract 1 from column A and 2 from column C

$$FC = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & \infty & \infty \\ C & 0 & 0 & \infty & \infty \\ D & \infty & \infty & 0 & \infty \end{array}$$

So, r = 1 + 2 = 3

Node value, $I_D = I_B + r + RC\,[C, D]$
$= 14 + 3 + 0 = 17$

The tree now looks like,



A, D]

D

and, we select node
to following paths.
the matrix FC by
∞. Also the entries

D

C, B]
4
late the matrix FC
∞. Also the entries

D

m column C

D]

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]**

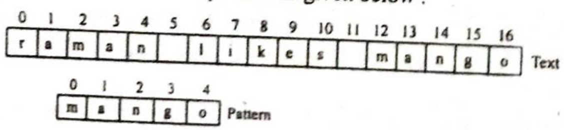
Ans. String matching algorithms are normally used in text processing. This text processing is often done during compilation of source program. In software design or in system design, text processing is an important activity. String matching means finding one or more of all the occurrences of a string 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*.

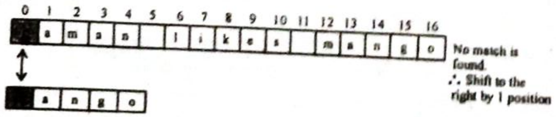
Naive Method : This is the simplest method which works using *Brute Force* approach. The *Brute Force* is a straight forward approach of solving the problem. This method has "Just do it" approach. This algorithm performs checking at all positions in the text between 0 to n–m, whether an occurrence of 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.

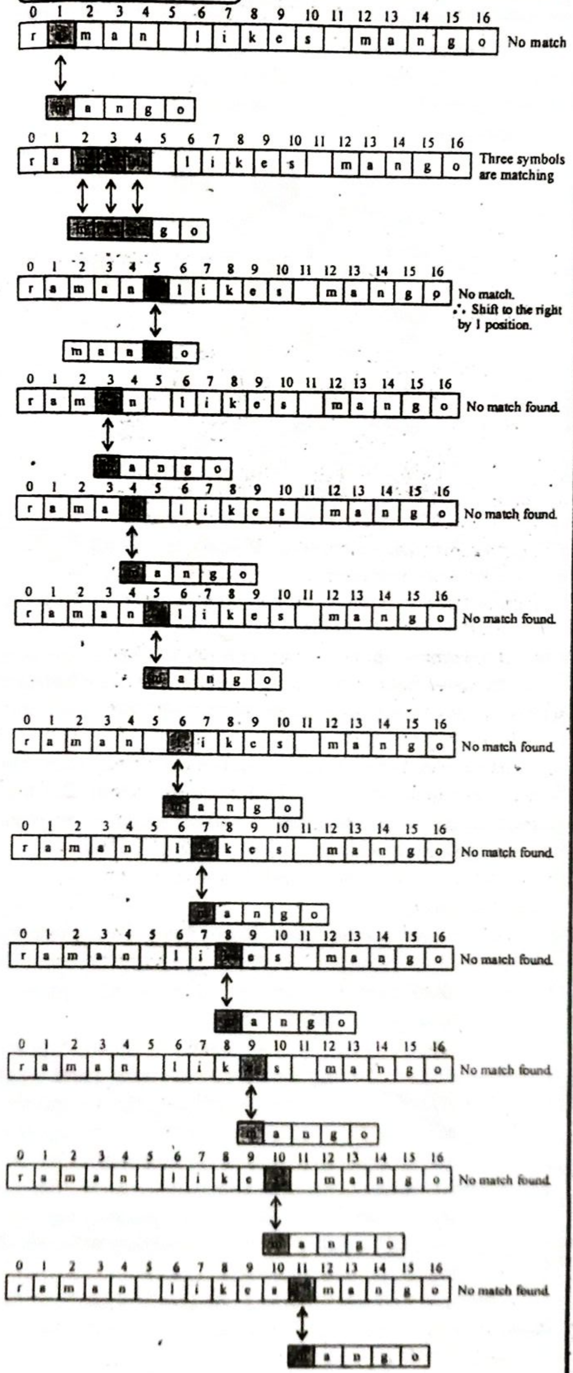
Example

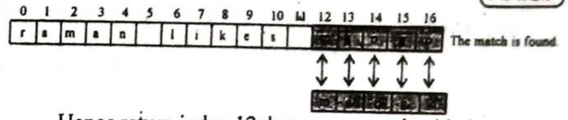
Consider the text and pattern as given below :



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

Σ^* : It is pronounced as 'sigma star'. This notation denotes the set of all finite length strings formed using input set Σ

e.g. :

$\Sigma = \{a\}$ then $\Sigma^* = \{\varepsilon, a, aa, aaa, ...\}$

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

$$\pi : \{1, 2, \ldots\ldots, m\} \rightarrow \{0, 1 \ldots\ldots, m-1\}$$

Such that

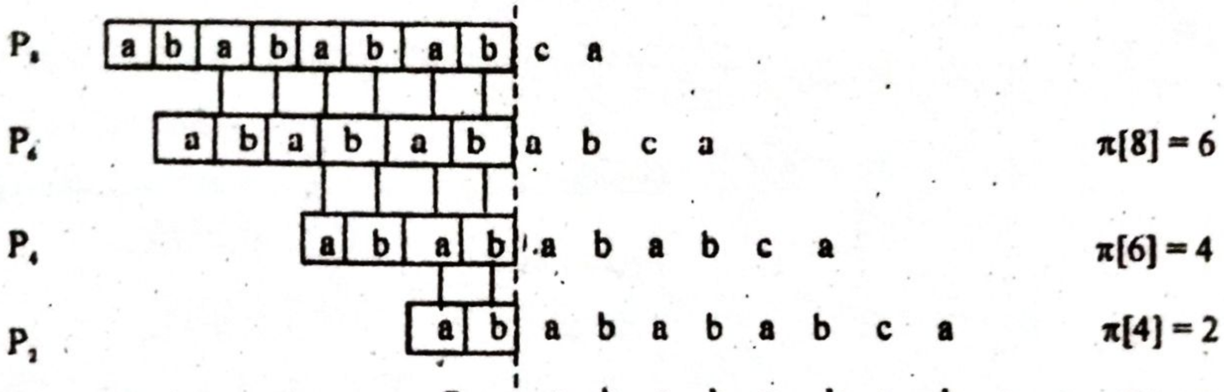
$$\pi[q] = \max\left\{k + k < q \text{ and } P_k] P_q\right\}$$

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= ababababca 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.*

Algorithm TSP Backtrack(A, l, lengthSoFar, minCost)
1. n ← length[A] // number of elements in the array A
2. if l = n
3. then minCost ← min(minCost, lengthSoFar distance[A[n], A[l]])
4. else for i ← l + 1 to n
5. do Swap A[l + 1] and A[i] // select A[i] as the next city
6. newLength ← lengthSoFar + distance [A[l], A[l + 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, l + 1, newLength minCost))
11. Swap A[l + 1] and A[i] // undo the selection
12. return minCost

The worst case complexity of Branch and Bound remain 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[i] = 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]])
```

15K+view

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 Σj – λs 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].

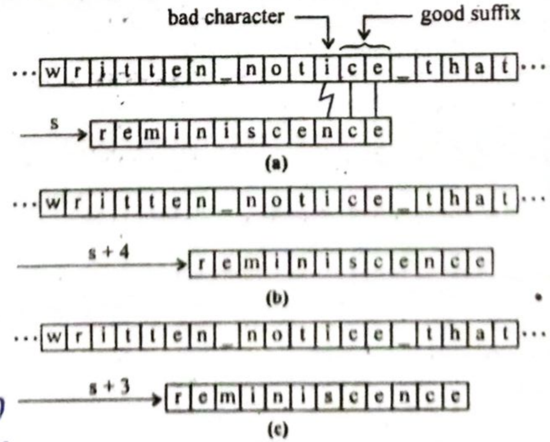


Fig.

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].

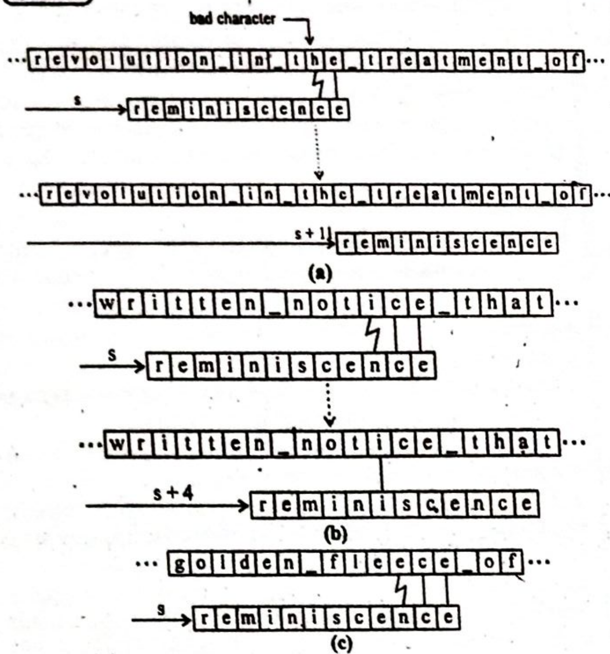


Fig.

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)

```
for each character α ∈ Σ
    do λ[α] = 0
for j ← 1 to m
    do λ[P[j]] ← j
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.

$$Q \sqsupset R \text{ and } S \sqsupset R \text{ imply } Q \sim S. \quad \text{....(1)}$$

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] ~ I

That is, Y[j] is the least amount we can advance s a not cause any characters in the "good suffix" T[s + j + 1 s + m] to be mismatched against the new alignment of pattern. The function y is well defined : for all j, sir P[j + 1..m] ~ P_0 for all j : the empty string is similar to strings. We call y the good–suffix function for the patterr

We first observe that y[j] ≤ m – π[m] for all j, as follov If w = π[m], then $P_\omega \sqsupset P$ by the definition of π. Furthermo since P[j + 1 .. m] ⊐ P for any j, we have P_ω ~ P[j + 1 .. r Therefore, y[j] ≤ m – π[m] for all j.

Definition of *y*

$$y[j] = m - \max \{k : \pi[m] \text{ and } P[j + 1..m] \sim P_k\}.$$

To simplify the expression for Y further, we define P' the reverse of the pattern P and π' as the corresponding pre function. That is P'[i] = P[m – i + 1] for i = 1, 2, ..., m, a π'(t) is the largest u such that u < t and $P'_u \sqsupset P'_t$.

If k is the largest possible value such that P[j + 1.. m] ⊐ I then we claim that π'[*l*] = m – j.

where P = (m – k) + (m – j). To see that this claim is w defined, note that P(j + 1 ... m) ⊐ P_k implies that m – j ≤ and thus *l* ≤ m. Also j < m and k ≤ m, so that *l* ≥ 1. We pro this claim as follows. Since P[j + 1..m] ⊐ P_k, we ha $P'_{m-j} \sqsupset P'_l$. Therefore, π'[*l*] ≥ m – j. Suppose now tl p > m – j, where p = π'[*l*]. Then, by the definition of π', v have P''p' ⊐ P'_l or equivalently, P'[1..p] = P'[1 – p + 1..*l*].

Rewriting this equation in terms of P rather than P', v have P(m – p + 1 ...... m) = ... P[m – *l* + 1..m – *l* + 1 Substituting for *l* = 2m – k – j, we obtain P(m – p + 1..r = P[k – m + j + 1 ..k – m + j + p], which implies P[m – p + 1 ..r ⊐ $P_{k-m+j+p}$. since p > m – j, we have j + 1 > m – p + 1, and 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, v have k' > k, where k' = k – m + j + p, contradicting our choi of k as the largest possible value such that P[j + 1..m] ⊐ I This contradiction means that we can't have p > m – j, ar thus p = m – j, which proves the claim. π'[*l*] = m – j impli that j = m – π'[*l*] and k = m – *l* + π'[*l*], we can rewrite o definition of *y* still further.

$$y[j] = m - \max(\{\pi[m]\}$$
$$\cup\{m - l + \pi'[l] ; 1 \le l \le M \text{ m and } j = m - \pi'[l]\})$$
$$= \min(\{m - \pi[m]\}$$
$$\cup \{l - \pi'[l] : 1 \le l \le m \text{ and } j = m - \pi'[l]\}).$$

Again, the second set may be empty.

Compute–Good–Suffix–Function (P, m)

```
π ← Compute–Prefix–Function (P)
P' ← reverse (P)
π' ← Compute–Prefix–Function (P')
for j ← 0 to m
    do y[j] ← m – π [m]
for l ← 1 to m
```



```
    do j ← m – π'[l]
    if y[j] > l – π'[l]
        then y[j] ← l – π'[l]
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.

In g
ν = (a_1, a
finite set
general a
empty so
we are gc
Step
Step
Step
Step
To g
Instead o
solution
solution
Typ
versions
1.
2.
St
constra
optimi
represe
space

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,

$\pi[1] = 0$

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, $\pi[q] = \pi[2] = k = 1$

For q = 3, k > 0,

P[k + 1] = P[2] = a

P[q] = P[3] = b

⇒ P[k + 1] ≠ P[q]

So, $k = \pi[k] = \pi[1] = 0$

Now, P[k + 1] = P[1] = a

P[q] = P[3] = b

P[k + 1] ≠ P[q]

So, $\pi[q] = \pi[3] = k = 0$

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 $\pi[q] = \pi[4] = k = 1$

For q = 5, k > 0,

P[k + 1] = P[2] = a

P[q] = P[5] = b

⇒ P[k + 1] ≠ P[q]

So, $k = \pi[k] = \pi[1] = 0$

Now, P[k + 1] = P[1] = a

P[q] = P[5] = b

P[k + 1] ≠ P[q]

So, $\pi[5] = k = 0$

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, $q = \pi[q] = \pi[2] = 1$

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 $q = \pi[q] = \pi[5] = 0$

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, $q \leftarrow \pi[q] = \pi[4] = 1$

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 $q = \pi[q] = \pi[5] = 0$

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. It 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 Pf[1] ← 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 ← Pf[k]
if (P[k + 1] = P[q]) then
set k ← k + 1
set Pf[q] ← k

Step 3 : Return value at the point of call : return (Pf).

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 the given pattern P is present in the text string T or not. It uses auxiliary function 'compute-prefix' to computer Pf.

Step 1 : Initialization

set n ← length [T] ; set m ← length [P]
: Calling Compute-Prefix
Set Pf ← 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 ← Pf [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 ← Pf[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'.

T	z	x	y	x	y	x	y	x	y	z	x	y	x
		↑	↑	↑	↑	#							
P		x	y	x	y	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.

T	z	x	y	x	y	x	y	x	y	z	x	y	x
				↑	↑	↑	↑	#					
P				x	y	x	y	z					

Again a mismatch occurs, and we shift ahead by two places.

T	z	x	y	x	y	x	y	x	y	z	x	y	x
						↑	↑	↑	↑	↑			
P						x	y	x	y	z			

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.

❑❑❑

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). 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 particula input elicits its worst - case behaviour.

Q.4 *Define Literals.*

Ans. 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.

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 proble* *[R.T.U. 2018, 20*

OR

Explain the quadratic assignment problem wi suitable example. *[R.T.U. 2016, 20*

Ans. Quadratic Assignment Problem (QAP)

The Quadratic Assignment Problem (QAP) is one fundamental combinatorial optimization problems in the bran of optimization or operations research in mathematics, fro 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 locatior For each pair of locations, a distance is specified and f each pair of facilities a weight or flow is specified (*e.g.*, tl amount of supplies transported between the two facilitie: The problem is to assign all facilities to different locatio with the goal of minimizing the sum of the distances multipli by the corresponding flows.

Intuitively, the cost function encourages factories wi high flows between each other to be placed close togethe

The problem statement resembles that of the assignme problem, only the cost function is expressed in terms quadratic inequalities, hence the name.

It is stated as:

"If there are n locations and n facilities and each facili is assigned to only one location at a time then the quadrat assignment problem is to obtain minimum cost. This cost c be computed using distance between two locations and flo between two facilities".*e.g.*

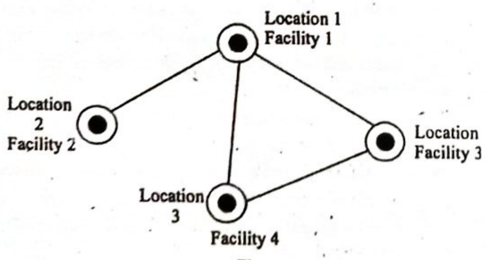


Fig.

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 those two facilities. Suppose

dist (1, 2) = 10 dist (1, 4) = 23

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) = 3

flow (3, 4) = 2

Then total cost $Z = \sum \text{dist}(i, j) * \text{flow}(i, j)$

$$Z = 10\times1 + 40\times4 + 23\times3 + 15\times2$$

$$= 10 + 160 + 69 + 30 = 269$$

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 $|v| \geq 2$ 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 mode. 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 points 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

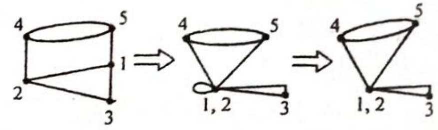


Fig.

Q.8 *Explain Las Vegas algorithm with example.* *[R.T.U. 2018, 2012]*

2

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 Algorithms : 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 seen 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_GEN is a random_number generator then

RANDOM_GEN (3, 7) returns either 3, 4, 5, 6, or 7 each with probability 1/5.

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 result it only gambles with the resources used.

In Monte Carlo there is a boundation on running ti and we are not sure to get a 100% correct result. It gamb with the result but it have to keep in mind the time allotec

Randomized Quick Sort (S)

(1) We choose an element y from S. In this, e element in S have equal probability of being chos

(2) Now we divide the S sequence in two parts containing elements smaller than y (S_1) and ot (S_2) containing element greater than y.

(3) We recursively call random sort (S) for seque S_1 and S_2.

(4) We place the elements after sorting like eleme of S_1, y and then elements of S_2.

(5) Exit.

Randomized algorithm deals or gambles with the va choosen, it could be time as well as resources.

Q.9 ***Solve*** $f = (x_1 v \bar{x}_2)(x_3 v \bar{x}_4)(\bar{x}_1 v x_3)(x_4 v x_6)$ ***usi randomized algorithm.*** *[R.T.U. 20*

Ans. Using Randomized Algorithm

$f = (x_1 v \bar{x}_2)(x_3 v \bar{x}_4)(\bar{x}_1 v x_3)(x_4 v x_6)$

Pick x_1 at random, so $T = \{x_1\}$

Put x_1 = True

Remove clauses centering x_1, which is $(x_1 v \bar{x}_2)$

Now, $f = (x_3 v \bar{x}_4)(\bar{x}_1 v x_3)(x_4 v x_6)$

due to $\bar{x}_1$, force variables are x_3

$T = \{x_3\}$

Put x_3 = True

Remove clauses centering x_3, which $(\bar{x}_1 v x_3)$ a $(x_3 v \bar{x}_4)$

Now, $f = (x_4 \text{ v } x_6)$

There are no forced variable due to $\bar{x}_3$

Pick x_4 at random

$T = \{x_4\}$

Remove clauses centering x_4, which $(x_4 \text{ v } x_6)$

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.*

$$\text{Cost} = \begin{matrix} 4 & 7 & 5 \\ 2 & 6 & 1 \\ 3 & 9 & 8 \end{matrix}$$

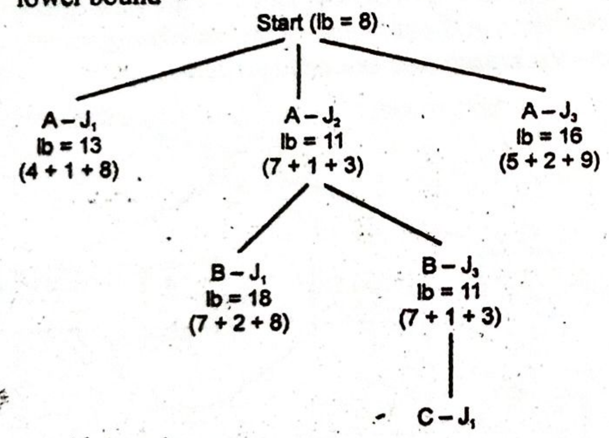
[R.T.U. 2017]

Ans.

$$\begin{array}{c|ccc|} & J_1 & J_2 & J_3 \\ \hline A & 4 & 7 & 5 \\ = B & \overline{2} & 6 & 1 \\ C & 3 & 9 & \overline{8} \\ \hline \end{array}$$

$lb = 4 + 1 + 3 = 8$

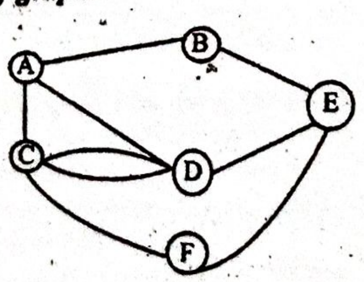
lower bound



Final job assignment

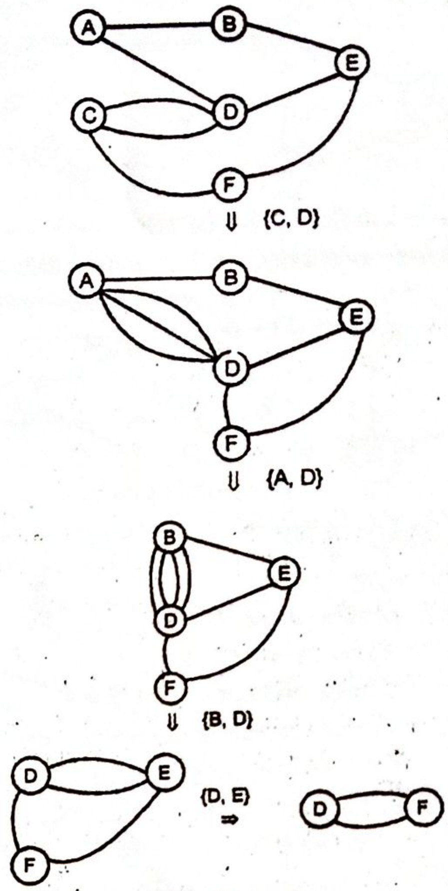
$$\begin{array}{cccc} & J_1 & J_2 & J_3 \\ A & 4 & \boxed{7} & 5 \\ B & 2 & 6 & \boxed{1} \\ C & \boxed{3} & 9 & 8 \end{array}$$

Q.11 *Give randomized algorithm for min cut of the following graph.*



[R.T.U. 2017]

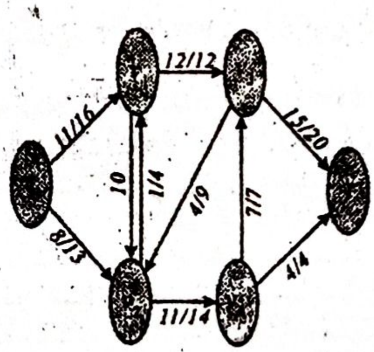
Ans. Randomized algorithm for min cut



Min Cut = 2

$V = \{F\} \cup \{A, B, C, D, E\}$

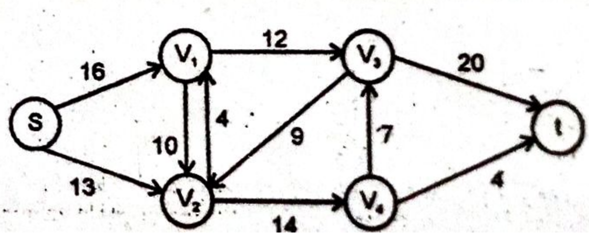
Q.12



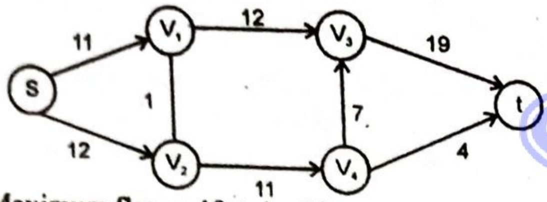
(a) Find Maximum flow in above network.

(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.

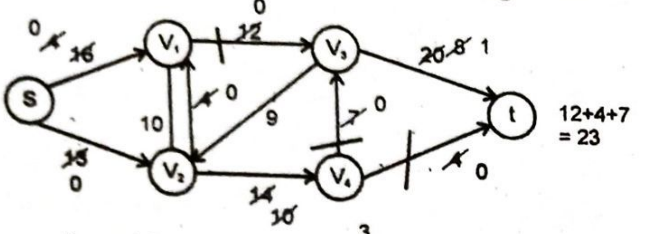


(a) Maximum Flow



Maximum flow = 19 + 4 = 23

(b) Minimum cut using ford fulkerson's algorithm



In residual graph, $v_1 \to v_3$, $v_3 \to v_4$ and $v_4 \to t$ has residual capacity of 0.

Hence minimum cut

$= v_1 \to v_3, v_3 \to v_4, v_4 \to t$

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 :	6	4	3	7
Person 1	5	8	1	8
Person 1	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-

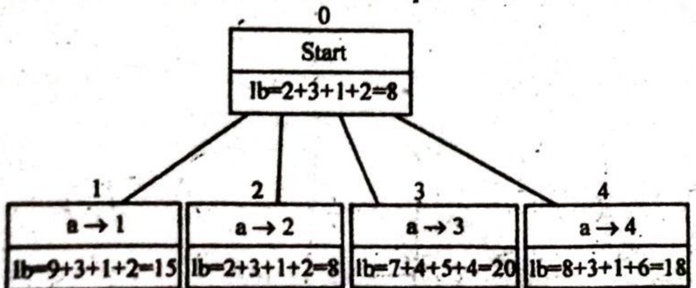


Fig.1

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.

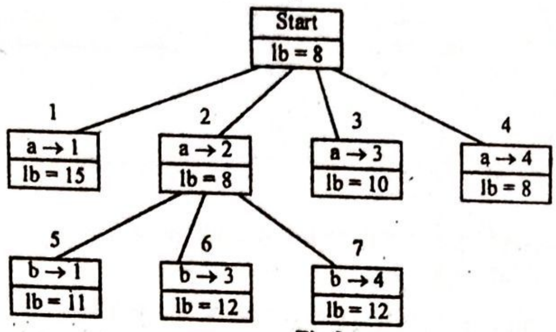


Fig.2

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.

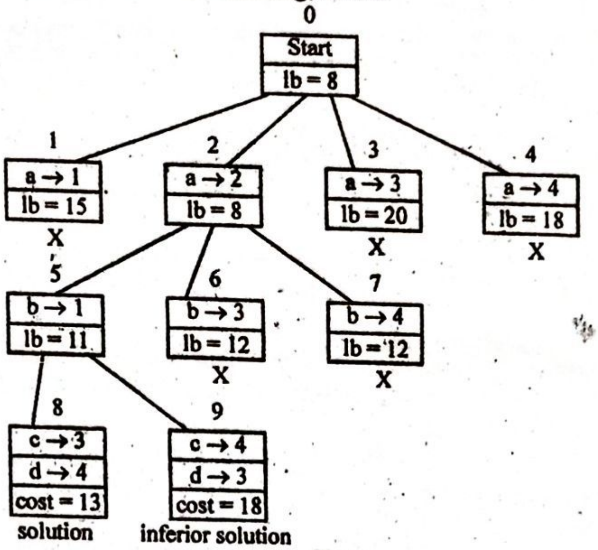


Fig.3

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.***

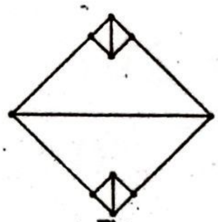
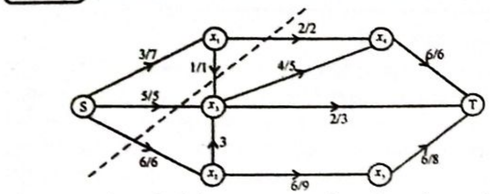


Fig.

[R.T.U. 2015]



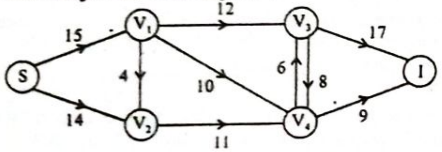
Also, capacity of minimum cut = value of maximum flow

$$6 + 5 + 1 + 2 = 6 + 5 + 3$$

(min-cut capacity) (max-flow value)

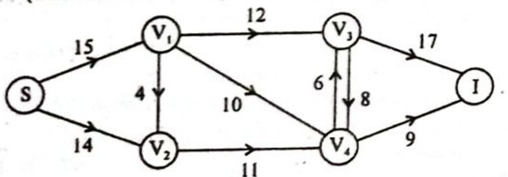
$$14 = 14$$

Q.21 *Define flow networks and solve the following flow network for maximum flow :*



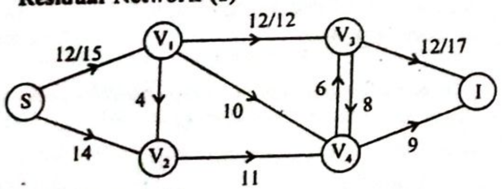
[R.T.U. 2011, Raj. Univ. 2006]

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).

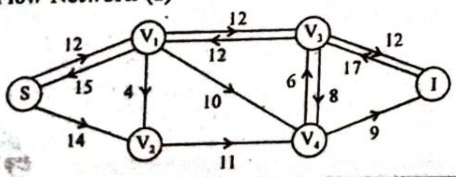


To make flow network for maximum flow first we make residual network.

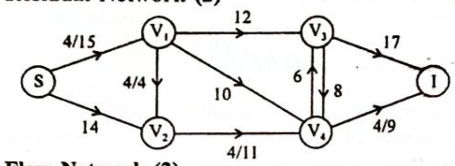
Residual Network (1)



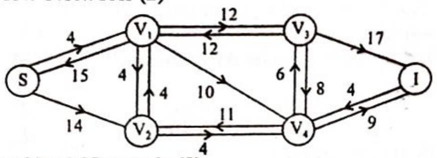
Flow Network (1)



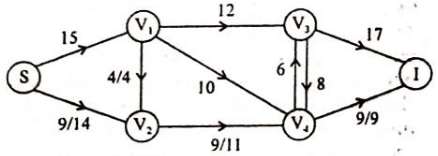
Residual Network (2)



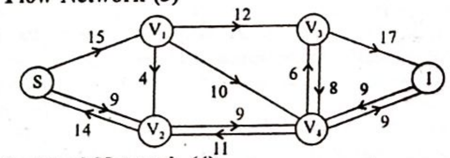
Flow Network (2)



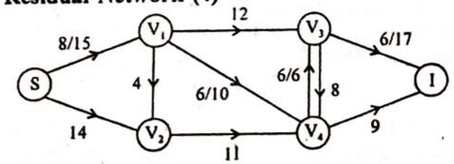
Residual Network (3)



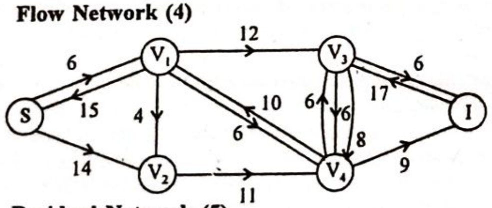
Flow Network (3)



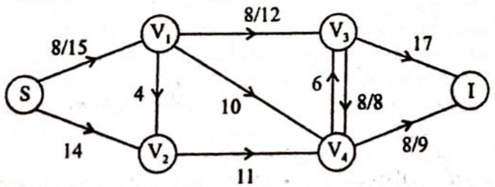
Residual Network (4)



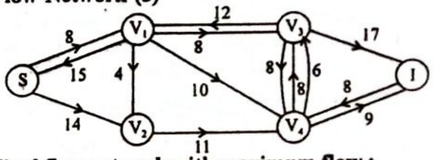
Flow Network (4)



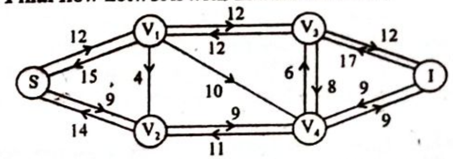
Residual Network (5)



Flow Network (5)



Final flow network with maximum flow :



Q.22 *Write short note on Bi-quadratic Assignment Problem.* *[R.T.U. 2011]*

Ans. Biquadratio 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, S_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 suggest

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.

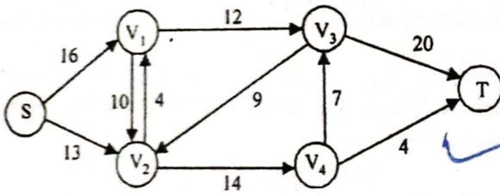


Fig. *[R.T.U. 2018, 2014]*

OR

Describe problem definition of Multicommodity flow in the network. State and prove the Ford Fulkerson's theorem. *[R.T.U. 2016]*

OR

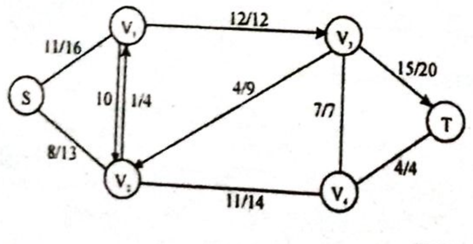
State multicommodity flow problem. *[R.T.U. 2015]*

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 i

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 agreegate 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. $\sum_{i=1}^{k} f_i(u, v) \leq c(u, v)$ for each $u, v \in V$

$f_i(u, v) = f_i(v, u)$ for $i = 1, 2, \ldots k$

Example

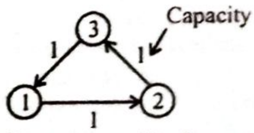


Fig. : Three commodity flow problem

As shown in above figure, there are three commodities flowing through the network.

4. While there exists a path p from s to t in the residual network G_f

5. do $C_f(p) \leftarrow \min\left[C_f(u,v):(u,v) \text{ is in } p\right]$

6. for each edge $(u,v) \in p$

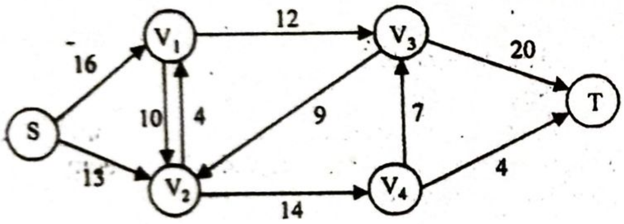
7. do $f[u,v] \leftarrow f[u,v] + C_f(p)$

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 fulKerson 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 $f(u, v) = 0, f(v, u) = 0$

$$f_0(s,V_1) = f_0(V_1,V_3) = f_0(V_3,t) = f_0(V_1,V_2) = f_0(s,V_2) = 0$$

Also, $f_0(V_2,V_1) = f_0(V_3,V_2) = f_0(V_2,V_4) = f_0(V_4,V_3) = 0$

$$f_0(V_4,t) = 0$$

Step 2: Consider path $s \rightarrow V_1 \rightarrow V_3 \rightarrow V_2 \rightarrow V_4 \rightarrow t$ from s to t. Calculate FAP (Flow augmented path)

$$\omega(s,V) - f_0(s,V) = 16 - 0 = 16 > 0$$

$$\omega(V_1,V_3) - f_0(V_1,V_3) = 12 - 0 = 12 > 0$$

$$\omega(V_3,V_2) - f_0(V_3,V_2) = 9 - 0 = 9 > 0$$

$$\omega(V_2,V_4) - f_0(V_2,V_4) = 14 - 0 = 14 > 0$$

$$\omega(V_4,t) - f_0(V_4,t) = 4 - 0 = 4 > 0$$

$$\Rightarrow \quad C_f(p) = \min\{C_f(u,v):(u,v) \in p\}$$

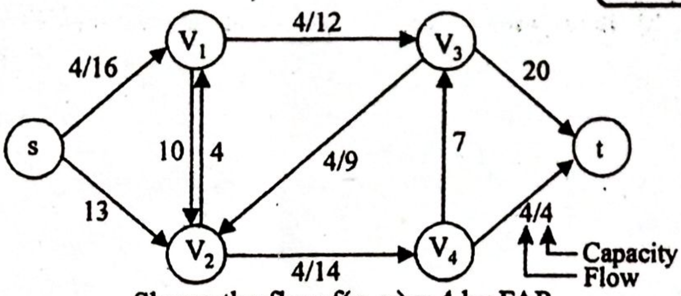
$$= \min\{16,12,9,14,4\}$$

$$= 4$$

Step 3: For each edge (u, v) $\in$ p (flow augmented path)

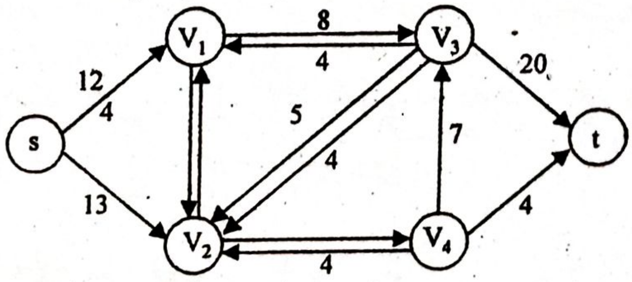
Set $\quad f[u,v] \leftarrow f[u,v] + C_f(p)$

$\Rightarrow \quad f[u,v] = 0 + 4 = 4$



Shows the flow f(u, v) = 4 by FAP

Step 4: *Set* $f[v,u] \leftarrow -f[u,v]$



Since path $s \rightarrow V_1 \rightarrow V_3 \rightarrow V_2 \rightarrow V_4 \rightarrow t$ 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 $s \rightarrow V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow V_3 \rightarrow t$ from s to t. Calculate, flow augmented path, with new flow.

$$\omega(s,V) - f_1(S,V) = 16 - 4 = 12 > 0$$

$$\omega(V_1,V_2) - f_0(V_1,V_2) = 10 - 0 = 10 > 0$$

$$\omega(V_2,V_3) - f_1(V_2,V_3) = 14 - 4 = 10 > 0$$

$$\omega(V_4,V_3) - f_0(V_4,V_3) = 7 - 0 = 7 > 0$$

$$\omega(V_3,t) - f_0(V_3,t) = 20 - 0 = 20 > 0$$

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:

$$f(u,v) \leftarrow f(u,v) + C_f(p)$$

$$f_2(s,V_1) = f_1(s,V) + C_f(p) = 4 + 7 = 11$$

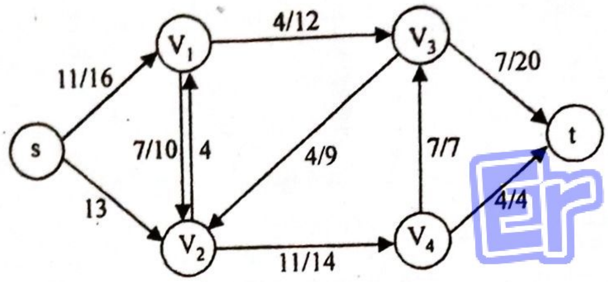
$$f_1(V_1,V_2) = f_0(V_1,V_2) + C_f(p) = 0 + 7 = 7$$

$$f_2(V_2,V_3) = f_1(V_2,V_3) + C_f(p) = 4 + 7 + 11$$

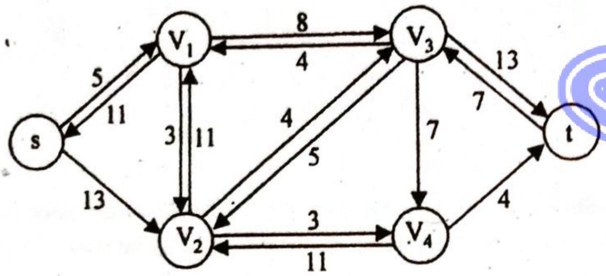
$$f_1(V_4,V_3) = f_0(V_4,V_3) + C_f(p) = 0 + 7 = 7$$

$$f_1(V_3,t) = f_0(V_3,t) + C_f(p) = 0 + 7 = 7$$

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 $s \to V_2 \to V_1 \to V_3 \to t$ 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:

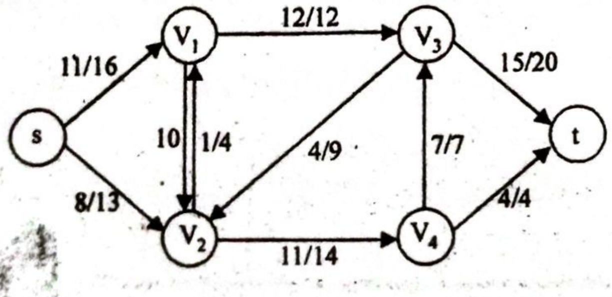
$$C_f = \min\left[C_f(u,v)\right]$$

$$= \min[13,8,8,13] = 8$$

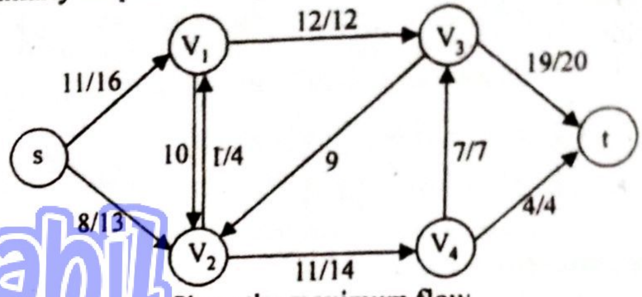
That is, minimum capacity of flow C_f is 8.

Step 11: For each edge $(u,v) \in p(FAP)$ we find f(u, v) as:

$$f(u,v) \leftarrow f(u,v) + C_f(p)$$

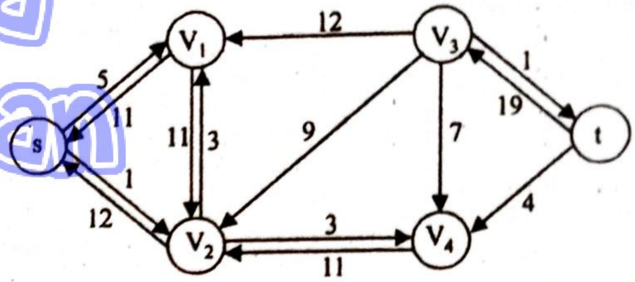


Similarly for path $s \to V_2 \to V_3 \to t$



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. 201***

OR

Briefly describe flow shop scheduling and network capacity assignment problem. ***[R.T.U. 201***

OR

Write Flow shop scheduling algorithm. ***[R.T.U. 201***

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 associate machine loading 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 operatio packet length measured in bits, (c) Maximu allowable delayfor each packet class measured i seconds, (d) Priority of each packet class, (e) Lin lengths measured in kilometers, and (f) Candidat capacities and their associated cost facto measured in bps and dollars respectively.

5. A non-preemptive FIFO queuing system is used t calculate the average link delay and the averag network delay for each class of packet.
6. Propagation and nodal processing delays a assumed to be zero.

Based on the standard network delay expressions, a the researchers in the field have used the following formul for the network delay cost:

$$T_{jk} = \frac{\eta_j\left(\sum_i \frac{\lambda_{ji,ml}}{\eta_k c_j}\right)}{(1-U_{r-1})(1-U_r)} + \frac{m_k}{c_j}$$

$$U_r = \sum_{l \in Vr} \frac{\lambda_{jl,ml}}{c_j}$$

$$Z_k = \frac{\sum_j T_{jk}\lambda_{jk}}{\gamma_k}$$

In the above, T_{jk} is the Average Link Delay for pack class k on link j, U_r is the Utilization due to the packets priority 1 through r (inclusive), V_r is the set of classes who priority level is in between 1 and r (inclusive), z_k is the Avera Delay for packet class, $\eta_j = \sum_l \lambda_{jl}$ is the Total Packet Ra on link j. $\gamma_k = \sum_j \lambda_{jk}$.

Total Rate of packet class k entering the network, ljk the Average Packet Rate for class k on link j, mkis the Avera Bit Length of class k packets, and Cjis the capacity of link As a result of the above, it can be shown that the proble reduces to an integer programming problem.

Q.26 ***Solve the assignment problem using Hungaria 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. 201

Ans. First of all we subtract the minimum of each row fon their row to have at least one zero in every row of the matri

$$\begin{array}{c} \\ J_1 \\ J_2 \\ J_3 \\ J_4 \end{array}\begin{array}{c} \begin{array}{cccc} M_1 & M_2 & M_3 & M_4 \end{array} \\ \begin{bmatrix} 15 & 5 & 9 & 7 \\ 2 & 13 & 6 & 5 \\ 7 & 8 & 3 & 11 \\ 2 & 4 & 6 & 10 \end{bmatrix} \end{array}\begin{array}{c} \\ -5 \\ -2 \\ -3 \\ -2 \end{array}$$

$$\downarrow$$

$$\begin{array}{c} \\ J_1 \\ J_2 \\ J_3 \\ J_4 \end{array}\begin{array}{c} \begin{array}{cccc} M_1 & M_2 & M_3 & M_4 \end{array} \\ \begin{bmatrix} 10 & 0 & 4 & 2 \\ 0 & 11 & 4 & 3 \\ 4 & 5 & 0 & 8 \\ 0 & 2 & 4 & 8 \end{bmatrix} \\ \qquad\qquad\qquad -2 \end{array}$$

M_4 does not contain any zero. Now we have to subtract the minimum number from each element of M_4.

$$\begin{array}{c} \\ J_1 \\ J_2 \\ J_3 \\ J_4 \end{array}\begin{array}{c} \begin{array}{cccc} M_1 & M_2 & M_3 & M_4 \end{array} \\ \begin{bmatrix} 10 & 0 & 4 & 0 \\ 0 & 11 & 4 & 1 \\ 4 & 5 & 0 & 6 \\ 0 & 2 & 4 & 6 \end{bmatrix} \end{array}$$

Now we try to cover all the zero with minimum number of horizontal (or vertical) lines.

Junction elements

$$\begin{array}{c} \\ J_1 \\ J_2 \\ J_3 \\ J_4 \end{array}\begin{array}{c} \begin{array}{cccc} M_1 & M_2 & M_3 & M_4 \end{array} \\ \begin{bmatrix} 10 & 0 & 4 & 0 \\ 0 & 11 & 4 & 1 \\ 4 & 5 & 0 & 6 \\ 0 & 2 & 4 & 6 \end{bmatrix} \end{array}$$

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.

$$\begin{array}{c} \\ J_1 \\ J_2 \\ J_3 \\ J_4 \end{array}\begin{array}{c} \begin{array}{cccc} M_1 & M_2 & M_3 & M_4 \end{array} \\ \begin{bmatrix} 11 & 0 & 5 & 0 \\ 0 & 10 & 4 & 0 \\ 4 & 4 & 0 & 5 \\ 0 & 1 & 4 & 5 \end{bmatrix} \end{array} \quad \text{or} \quad \begin{array}{c} \\ J_1 \\ J_2 \\ J_3 \\ J_4 \end{array}\begin{array}{c} \begin{array}{cccc} M_1 & M_2 & M_3 & M_4 \end{array} \\ \begin{bmatrix} 11 & 0 & 5 & 0 \\ 0 & 10 & 4 & 0 \\ 4 & 4 & 0 & 5 \\ 0 & 1 & 4 & 5 \end{bmatrix} \end{array}\begin{array}{c} \\ R_0 \\ R_1 \\ R_2 \\ R_3 \end{array}$$

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 $R_2 \to J_3$ and $R_3 \to J_4$ 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 :

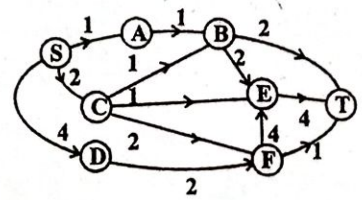
$$\begin{array}{c} \\ J_4 \to M_1 \\ J_1 \to M_2 \\ J_3 \to M_3 \\ J_2 \to M_4 \end{array} \quad \begin{array}{c} \\ J_1 \\ J_2 \\ J_3 \\ J_4 \end{array}\begin{array}{c} \begin{array}{cccc} M_1 & M_2 & M_3 & M_4 \end{array} \\ \begin{bmatrix} 11 & \boxed{0} & 5 & 0 \\ 0 & 10 & 4 & \boxed{0} \\ 4 & 4 & \boxed{0} & 5 \\ \boxed{0} & 1 & 4 & 5 \end{bmatrix} \end{array}$$

Now we calculate the total cost using cost matrix given initially.

$$\text{Cost} = (J_4 \to M_1) + (J_1 \to M_2) + (J_3 \to M_3) + (J_2 \to M_4)$$
$$\Rightarrow \quad = 2 + 5 + 3 + 5$$
$$= 15$$

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, ∈ 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.

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.

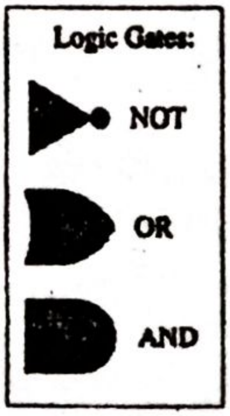
Q.5 ***What do you mean by interactable problems.***

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.

Ans. Let us defir
as input, a grapl
notation, with th
first iteratively ca
S of N + 1 numb
each number fro
example, by sort
in S, which sho
sequence S defi
binary encoding
where n is the siz
made on the sequ

Observe th
vertex of G exact
there is a sequenc
if A outputs "yes
each vertex of G
That is, A non-
HAMILTONIAN
Cycle is in NP.

Our next ex
testing. A Boolea
called a logic gat
AND, OR, or N
correspond to inp
edges correspond
of course, for tha
edges are input n
an output node.



Fig

Circuit-Sat
circuit with a sin
assignment of va

ch that it visits all the
st I.
e use as a certificate the
e verification algorithm
ch vertex exactly once,
ether the sum is at most
n polynomial time.
nan Problem (TSP) is
n cycle $\leq_p$ TSP. Let
nian. We construct an

h G'=(V,E), where
e cost function c by

c, 0), which is easily

Hamiltonian cycle if
t at most 0. Suppose
e h. Each edge in h
Thus, h is a tour in G'
graph G' has a tour h'
ges in E' are 0 and 1,
n graph G .

problem belongs to
[R.T.U. 2017, 2014]

gorithm for accepting
e the choose method
as well as the output
ply visit each logic
least one incoming
value for the output
olean function, be it

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.

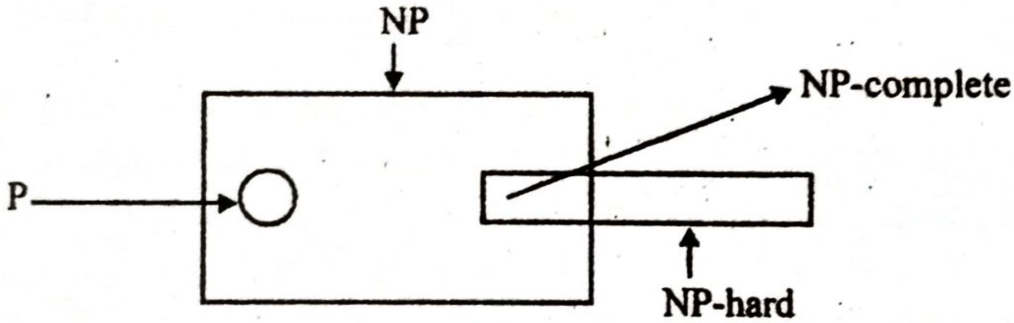


Fig.

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 $\in$ NP.

Significance of Cook's Theorem : If one can find a poly-algorithm for SAJ, 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.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_i in the Boolean formula 'B_{fg}', we add two vertices in G, one of which is labelled as I_i and other as $\bar{I}_i$. After this we add the edge $(I_i, \bar{I}_i)$.
- For each clause C_i = (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.

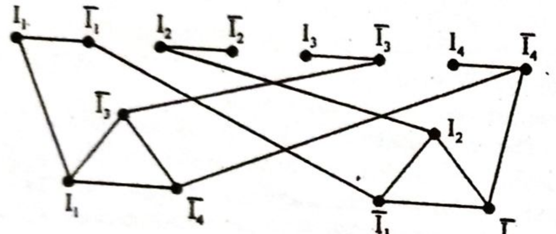


Fig.

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.

$C \leftarrow \phi$

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.

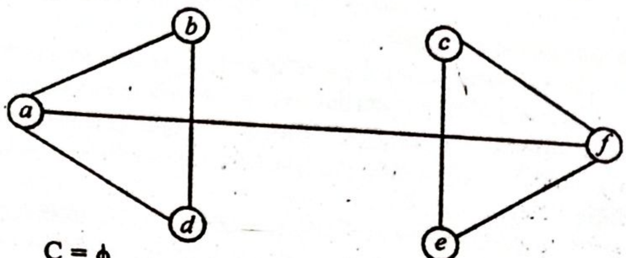
Step 5. Add the endpoints u, v to vertex cover C.

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

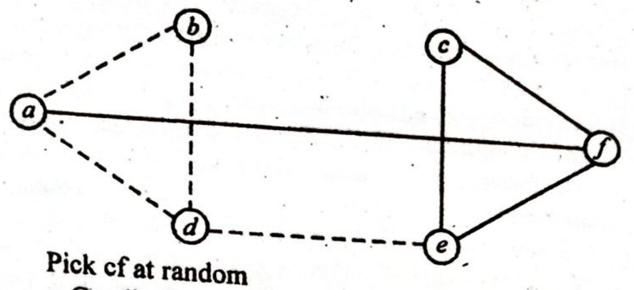
E = {ab, ad, bd, de, af, cf, ef, ce}

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

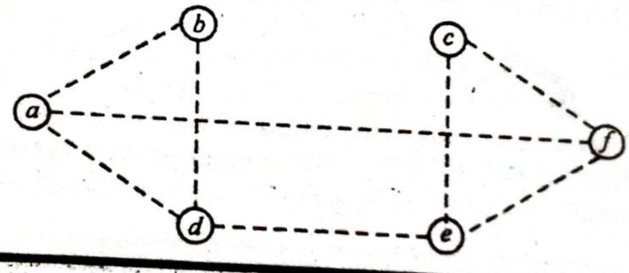
C = {b, d, c, f}

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 algorigm 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 : $C = \phi$.

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,

$C = \phi$; $U = S = \{a, b, c, d, e, f, g, h\}$

Pick S_i with max $|S_i \cap U| = S_2$

$\therefore \quad C = C \cup \{S_2\} = \{S_2\}$

$U = U - S_2 = \{a, c, e, h\}$

Pick S_i 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:

$V = \{\langle x, i\rangle \mid x \text{ is in the } i^{th} \text{ clause}\}$

$E = \{\langle x, i\rangle, \langle y, j\rangle \mid x \,\overline{y}\, i\, j\}$

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

Q.15 ***Write short note on NP-completeness.*** ***[R.T.U. 2017]***

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. 2013]***

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, 2000]

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 Phas "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 P of input size n. That means, for each input of size n the algorithm terminates after atmost P(n) steps; For instance, $n^7 + 24n^2 + 65$

Decision 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_2 a substring of S_1 ?
- 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, and an integer i, does G have a minimum spanning tree of weight at most i ?
- Given S, does there exist a subset of elements that fits in the knapsack, and has total profit of at least S ?

We can say that a given algorithm A accepts the string 'S' only when A produces the output 'yes' on input 'S'. 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 languages L if A produces the output 'yes' on input 'S' 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 $S \in L$, in polynomial time p(w), where 'n' is the input size of S 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's 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.

L is a language.
A produces the
L otherwise it

s include all the
be accepted in
hm A, it accepts
e input size of S
that the class P
'no'. We refer
hm A for output
not present in

at accepts the
that accepts a
, if a language
P.
the complexity
ot present in P.
an additional

1) in a non-
algorithm A
ition then we
out of several
which lead to
is noticeable
sing random

ion problems
rministically
m A, if S ∈ L,
elect calls in
al time, p(n),

oes not say
A running in
eps when A
er of calls to
ptance, the
re L is the

ch includes
and many
NP.
l scientists
m NP and

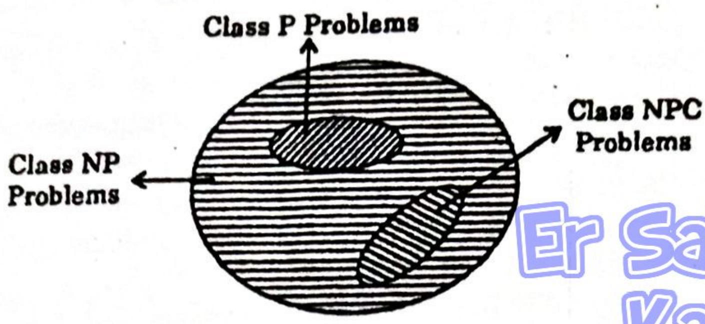


Fig.

Problem : Given a Graph G, does this 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 in binary notation, with the vertices labelled from 1 to n.

Then we make A to call the Select method iteratively 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 Hamiltonain–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 problem 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

$$\forall L' \in NP, L \propto L\ L \in P.$$

$$\forall L' \in NP, L' \propto L \wedge L \in P$$

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.

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

the con
to unde
of natu
underst
it is be
machin
comput
problen
model c
some w
Th
underst
a repert
In order
and bou
their rel
By con
non-det
to the
machin
surprisi
time em
Su
to mode
Th

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.