Important Instructions to examiners:

1) The answers should be examined by key words and not as word-to-word as given in the model answer scheme.
2) The model answer and the answer written by candidate may vary but the examiner may try to assess the understanding level of the candidate.
3) The language errors such as grammatical, spelling errors should not be given more importance (Not applicable for subject English and Communication Skills.
4) While assessing figures, examiner may give credit for principal components indicated in the figure. The figures drawn by candidate and model answer may vary. The examiner may give credit for any equivalent figure drawn.
5) Credits may be given step wise for numerical problems. In some cases, the assumed constant values may vary and there may be some difference in the candidate's answers and model answer.
6) In case of some questions credit may be given by judgement on part of examiner of relevant answer based on candidate's understanding.
7) For programming language papers, credit may be given to any other program based on equivalent concept.

Q.1. Solve any FIVE: 20

a) Explain average and worst case analysis of algorithms.
(Average case 2M, Worst-case 2M)

Average case running Time:
The expected behavior when input is randomly drawn from a given distribution. The average-case running time of an algorithm is an estimate of the running time for an "average" input. Computation of a average-case running time entails knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequence

Worst-case Running Time:
The behavior of the algorithm with respect to the worst possible case of the input instance. The worst case running time of an algorithm is an upper bound on running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer time.

b) What is meant by efficiency of algorithm? Explain it.
(Explanation 4M)

The *complexity* of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. Usually there are natural units for the domain and range of this function. There are two main complexity measures of the efficiency of an algorithm:

- *Time complexity* is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.
- *Space complexity* is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.

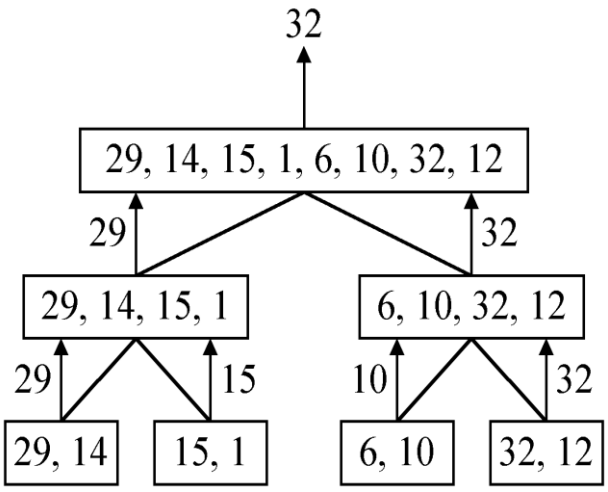
Efficiency is some time taken as the most important criterion and is not always adequately specified. Possible sub-criteria include speed of execution, amount of main or secondary storage required or amount of complexity of maintenance. The basic rule for considering efficiency is that it is not optimize an algorithm against any efficiency sub-criterion until it has first been shown to be effective**.**

c) Describe divide and conquer strategy. Explain with an example.
(Divide and conquer strategy 2M, Example 2M (any other relevant example can be considered)

Divide- and conquer is a top-down technique for designing algorithm that consists of dividing the problem into smaller sub problems hoping that the solutions of the sub problems are easier to find and then composing the partial solutions into the solution of the original problem. So, divide and conquer algorithms break the problem into several sub problems that are similar to the original problem but smaller in size, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem.

Little more formally, divide-and conquer paradigm consists of following major phases:
Breaking or divide the problem into several sub problem that are similar to the original problem but smaller size,
Conquer sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straightforward manner and then
Combine these solutions to sub problems to create a solution to the original problem.
The divide–and-conquer strategy solves problem by:

1. Breaking it into sub problems that are themselves smaller instances of the same type of problem.
2. Recursively solving these sub problems.
3. Appropriately combining their answers.



d) Explain merge sm1 algorithm. Give the time complexity of merge sort.
(Algorithm 2M, Complexity 2M)

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine' step. To perform the merging, we use an auxiliary procedure MERGE(A,p,q,r), where A is an array and p, q, and r are indices numbering elements of the array $p \le q < r$. the procedure assumes that the sub arrays $A[p .. q]$ and $A[q + 1 .. r]$ are sorted order. It merge them to form a single sorted sub array that replaces the current sub array $A[p .. q]$.

MERGE (A, p, q, r)

1. $n1 \leftarrow q - p + 1$
2. $n2 \leftarrow r - q$
3. Create arrays $L[1 . . n_1 + 1]$ and $R[1 . . n_2 + 1]$
4. for $i \leftarrow 1$ TO n_1
5. do $L[i] \leftarrow A[p + i - 1]$
6. for $j \leftarrow 1$ TO n_2
7. do $R[j] \leftarrow A[q + j]$
8. $L[n_1 + 1] \leftarrow \infty$
9. $R[n_2 + 1] \leftarrow \infty$
10. $i \leftarrow 1$
11. $j \leftarrow 1$
12. for $k \leftarrow p$ to r
13. do if $L[i] \le R[j]$
14. then $A[k] \leftarrow L[i]$
15. $i \leftarrow i + 1$
16. else $A[k] \leftarrow R[j]$
17. $j \leftarrow j + 1$

Average case analysisO(NlogN)
Best-case analysis O(NlogN)
Worst case analysis $O(N^2)$

e) Explain dynamic programming. What is meant by principle of optimality?
(Explanation 4M)

Dynamic programming is a problem solving technique that, like divide and conquer, solve problems by dividing them into subproblems. Dynamic programming is used when the subproblems are not independent, eg. When they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same subproblem multiple times.

Dynamic programming solves each subproblem once and stores the result in a table so that it can be rapidly retrieved if needed again. It is often used in Optimization Problems: a problem with many possible solutions for which we want to find an optimal(the best) solution. (there may be more than 1 optimal solution). Dynamic programming is an approach developed to solve sequential or multi stage, decision problems; hence, the name "dynamic" programming. But as we shall see, this approach is equally applicable for decision problems where sequential property is induced solely for computational convenience.

Dynamic programming is a bottom-up mechanism we solve all possible small problems and then combine them to obtain solution for bigger problem.

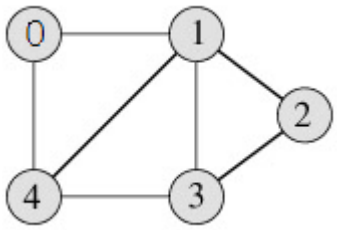
Optimal substructure:

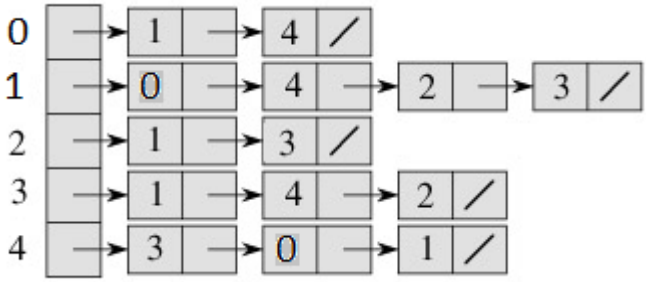
If an optimal solution contains optimal sub solutions, then a problem exhibits optimal substructure.

f) Define Graph, Cyclic graph and Directed graph. Mention how representation of graph is done in memory.
(Definition 1Mark each, graph representation 1M)
Graph: a graph is non empty set of vertices & edges denoted by G & given by G=V, E)
Directed Graph: a directed graph is also identified as digraph with an ordered pair(u,v)
Cycle: A path from a node to itself is called cycle. Thus a cycle is a path in which initial & final vertices are same. If graph contains cycle it is cyclic otherwise it is acyclic.

Sequential representation of graphs:
Two methods for sequential representation of graph.
1) Adjacency matrix.
2) Linked representation.

Linked representation of graph.





g) Prove that the Dijkstra's algorithm finds the shortest path from a single source to the other nodes of a graph.
(Explanation 4M)

Dijkstra algorithm, named after its discoverer, Dutch computer scientist Edsger Dijkstra, is a greedy algorithm that solves the signal-source shortest path problem fro a directed graph G=(V.E) with nonnegative edge weights i.e.. we assume that $w(u,v) \geq 0$ for each edge(u,v) $\in E$.
Dijkstra algorithm maintain a set of S of vertices whose final shortest path weights from the source s have already been determined. That is , for all vertices $v \in S$, we have $d[v]=\delta(s,v)$. the algorithm repeatedly selects the vertex $u \in V-S$ with the minimum shortest path estimate, inserts into S and relaxes all edges leaving u. we maintain a priority queue Q that contains all the vertices in v-s, keyed by their d values. Graph G is represented by adjacency lists.

DIJKSTRA (G, w, s)
1. INITIALIZE SINGLE-SOURCE (G, s)

2. S ← ϕ// S will ultimately contains vertices of final shortest-path weights from s
3. Initialize priority queue Q i.e., Q ← V[G]
4. while Q= ϕ priority queue Q is not empty do
5. do u ← EXTRACT_MIN(Q) // Pull out new vertex
6. S ← S ∪{u} // Perform relaxation for each vertex v adjacent to u
7. for each vertex v ∈[u]
8. do Relax (u, v, w)

Q.2. Solve any TWO: 16

a) What is asymptotic notation? Explain Big 0 notation.

(Asymptotic Notation Explanation 4M, Big-oh notation 4M)

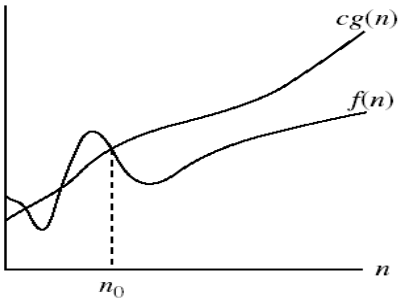
Asymptotic means a line that tends to converge to a curve, which may or may not eventually touch the curve. It is a line that stays within bounds.

Asymptotic notation is a shorthand way to write down and talk about fastest possible and slowest possible running times for an algorithm, using high and low bound on speed. These are also referred to as best case and worst case scenarios respectively.

Big-oh notation:

Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time it could possibly take for the algorithm to complete. More formally, for non-negative functions, f(n) and g(n), if there exists an integer n_0 and a constant c>0 such that for all integers $n>n_0$

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

b) Explain exponentiation as an example of divide and conquer.
(Explanation 8M any other relevant example can be considered)

- One technique for deriving a solution to a problem involves dividing the problem into sub-parts which are easier to solve, and then deriving a solution to the original problem by somehow recombining the solutions to the sub-parts.
- Commonly, a problem is broken into two nontrivial subparts and the resulting algorithm naturally involves recursion… a topic we will not explore at this time.
- But in some cases, a problem is broken into a trivial part and a more complex part, and the resulting algorithm is naturally iterative.

- Binary search can be viewed in the latter light, where the current element is the trivial case and the "in play" portion of the list constitutes the nontrivial part.

Consider that we can approximately cut the number of multiplications in half if we have an even exponent:

$$x^{2k} = \left(x^2\right)^k = x^2 \times x^2 \times x^2 \cdots \times x^2$$

That's k multiplications instead of $2k$… quite a savings.

And a similar "trick" works if we have an odd exponent:

$$x^{2k+1} = x\left(x^2\right)^k = x \times x^2 \times x^2 \times x^2 \cdots \times x^2$$

That's $k+1$ multiplications instead of $2k$.

But, it is possible to do even better; consider that:

$$x^{11} = x \times x^{10} = x \times x^2 \times x^8 = x \times x^2 \times \left(\left(x^2\right)^2\right)^2$$

That requires only 5 multiplications, and the advantage grows even larger if we have a larger exponent…

The key is to efficiently compute a factor whose exponent is a power of 2, since we can compute a factor of that form with a small number of multiplications:

$$x^{2^r} = \left(\left(\left(x^2\right)^2\right)^2 \cdots\right)^2$$

That requires only r multiplications!

But, of course, we not only have to do this, but we also have to keep track of the rest of the computation; say we want to compute x^{21} :

$$x^{21} = x \times x^4 \times x^{16}$$

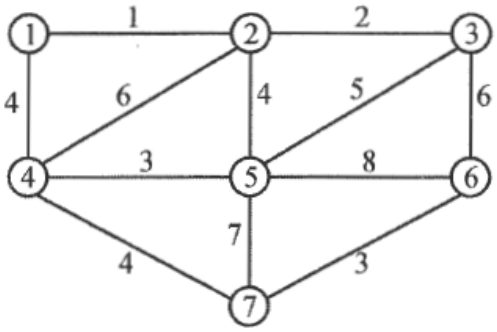
So, we notice that the exponent is odd and we remember we need to throw in x^1 ;

now we need to deal with the even exponent 20, which is the 5th power of x^4 ;

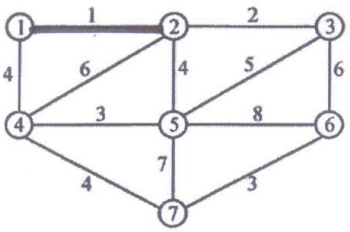
But, of course, we not only have to do this, but we also have to keep track of the rest of the computation; say we want to compute x^{21}:

Note	Current multiplier	Exp. remaining	Accumulated value
initial values	x	21 : odd	1
decrement exponent, apply multiplier to accumulated value	x	20 : even	x
divide exponent by 2, square multiplier	x^2	10 : even	x
divide exponent by 2, square multiplier	x^4	5 : odd	x
decrement exponent, apply multiplier to accumulated value	x^4	4 : even	x^5

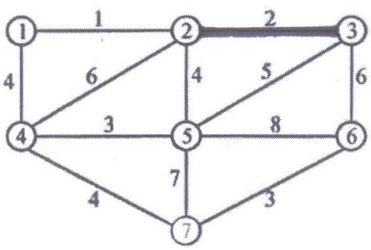
c) Write down the Prims algorithm to generate minimum cost spanning tree. Simulate the algorithm for the given graph and find MST for the given graph.
(Correct spanning Tree 8M)



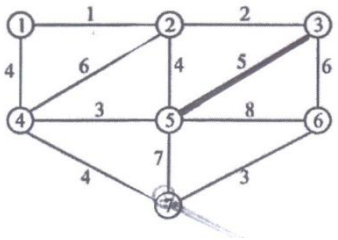
Suppose 1 vertex is the root i.e. r. by EXRACT-MIN(Q) procedure. Now u=r & Adj[u]={2.4}.
Removing u from the set Q and adds it to the set V-Q of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in the tree.
Key[2]=∞
W[1,2]=1 i.e., w(u,v)< Key[2]
So, π[2]=0 & Key[2]=1
And Key[2]=∞



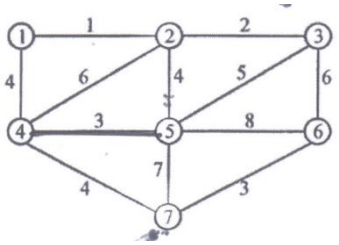
Now, by EXRACT-MIN(Q) remove 2 because Key[2]=1, which is minimum so, u=2.
Adj[2]={1, 4, 5,3}
1≠Q
Key[4]=6
Key[5]=4
Key[3]=2



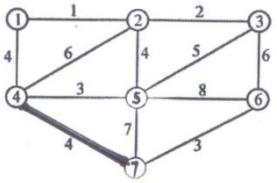
Now, by EXRACT-MIN(Q) remove 3 because Key[3]=2, which is minimum so, u=3.
Adj[3]={2,5,6}
2≠Q
Key[5]=5
Key[6]=6



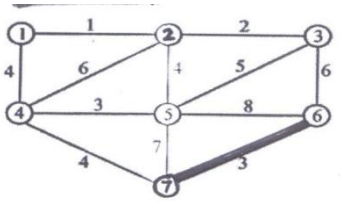
Now, by EXRACT-MIN(Q) remove 5 because Key[5]=5, which is minimum so, u=5.
Adj[5]={2,4,3,6,7}
3≠Q, 2≠Q
Key[4]=3
Key[6]=8
Key[7]=7



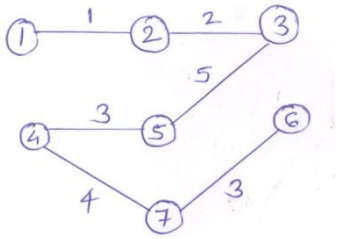
Now, by EXRACT-MIN(Q) remove 4 because Key[4]=4, which is minimum so, u=4.
Adj[4]={1,2,5,7}
2≠Q, 1≠Q, 5≠Q,
Key[7]=4



Now, by EXRACT-MIN(Q) remove 7 because Key[7]=4, which is minimum so, u=7.
Adj[7]={4,5,6}
4≠Q,
Key[5]=7
Key[6]=3



Now, by EXRACT-MIN(Q) remove 6 because Key[6]=3, which is minimum so, u=6.
Thus final spanning tree is



Q.3. Solve any TWO: 16

a) **Explain the Binary Search Algorithm using recursive function. Also explain the time complexity of Binary Search.**
(Recursive function of binary search 4M, Time complexity 4M)

BINARY WITH RECURSIVE FUNCTION

```
int binary(int a[],int n,int m,int l,int u){

    int mid,c=0;

    if(l<=u){
        mid=(l+u)/2;
        if(m==a[mid]){
            c=1;
        }
        else if(m<a[mid]){
            return binary(a,n,m,l,mid-1);
        }
        else
            return binary(a,n,m,mid+1,u);
    }
    else
      return c;
}
```

COMPEXITY OF BINARY SEARCH

Worst case performance	$O(\log n)$
Best case performance	$O(1)$
Average case performance	$O(\log n)$
Worst case space complexity	$O(1)$

b) **Explain in detail radix sort algorithm with suitable example.**
(Explanation 4M, example 4M)

Radix Sort is a sorting algorithm that is useful when there is a constant'd' such that all the keys are d digit numbers. To execute Radix-Sort, for p=1 toward 'd' sort the numbers with respect to the pth digit from the right using any linear time stable sort.

Radix sort is sometimes used to sort records of information that are keyed by multiple fields.

The following procedure assumes that each element in the n element array A has d digits where digit 1 is the lowest-order digit and digit d is the highest order digit.

RADIX-SORT(*A,d*)
1 for *i* - 1 to *d*
2 do use a stable sort to sort Array *A* on digit *i*

Since a linear time sorting algorithm is used 'd' times and d is a constant, the running time of Raix Sort is linear. When each digits is in the range 1 to *k*, and *k* is not too large, counting_sort is the obvious choice. Each pass over *n d*-digit numbers takes $\theta(n + k)$ time. There are *d* passes, so the total time for Radix sort is θ ($dn+kd$). When *d* is constant and $k = O(n)$, the Radix sort runs in linear time.

Example:
The first column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. The vertical arrows indicate the digit position sorted on to produce each list from the previous one

INPUT	1st pass	2nd pass	3rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

c) **Explain the steps of Kruskal's algorithm for finding the minimum cost spanning tree with suitable example.**
(Explanation with example 8M)

MST-KRUSKAL(*G*, *w*)
1 $A \leftarrow \emptyset$
2 **for** each vertex $v \_ V[G]$
3 **do** MAKE-SET(*v*)
4 sort the edges of *E* into nondecreasing order by weight *w*
5 **for** each edge $(u, v) \_ E$, taken in nondecreasing order by weight

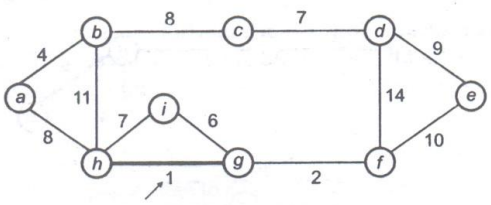
6 **do if** FIND-SET(u) ≠ FIND-SET(v)
7 **then** $A \leftarrow A \_ \{(u, v)\}$
8 UNION(u, v)
9 **return** A

First we initialize the set A to the empty set and create |v| trees, one conataing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight, i.e.

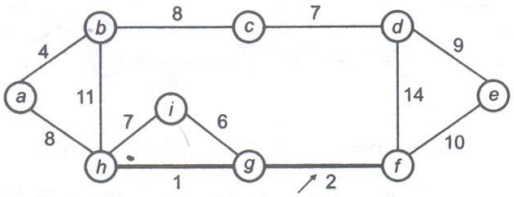
(h,g)	1
(g,f)	2
(a,b)	4
(i,g)	6
(h,i)	7
(c,d)	7
(b,c)	8
(a,h)	8
(d,e)	9
(e,f)	10
(b,h)	11
(d,f)	14

Now, check for each edge (u,v), whether the end points u and v belong to the same tree. If they do, then the edge(u,v)cannot be added. Otherwise, the two vertices belong to different trees and the edge(u,v) is added to A and the vertices in the two trees are merged in by UNION procedure.

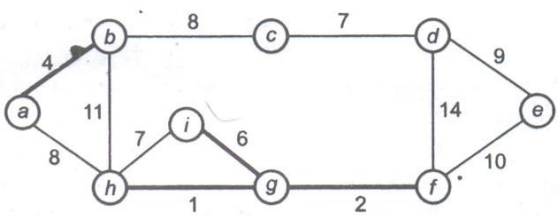
So first take (h,g edge)



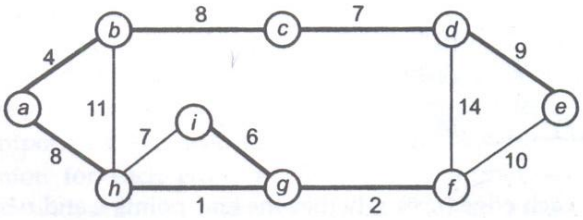
Then (g,f) edge



Then (a,b) and (i,g)edges are considered and forest becomes

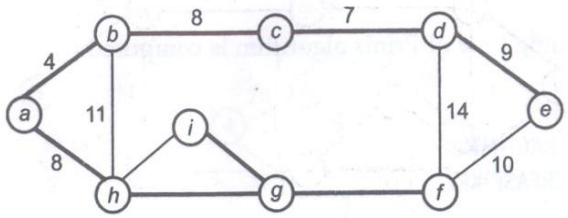


Now, edge(h,i) both h and I vertices are in same set, thus it creates a cycle. So this edge is discarded.
Then edge (c,d), (b,c), (a,h), (d,e), (e,f) are considered and forest becomes.

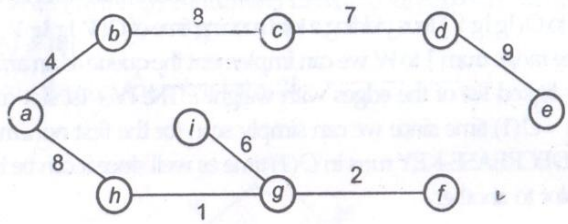


In (e,f)edge both end points e and f exist in same tree so discarded this edge then (b,h) edge, it also creates a cycle.

After that edge (d, f) and the final spanning tree is shown as in dark lines.



or



Minimum cost MST

Q.4. Solve any TWO: 16

a) Explain the following terms : (for each term 4M)

i. Problems and instances

A computational problem can be viewed as an infinite collection of *instances* together with a *solution* for every instance. The input string for a computational problem is referred to as a problem instance,
the *instance* is a particular input to the problem, and the *solution* is the output corresponding to the given input.
a problem and an instance, consider the following instance of the decision version of the traveling salesman problem: Is there a route of at most 2000 kilo metres passing through all of Germany's 15 largest cities? The quantitative answer to this particular problem instance is of little use for solving other instances of the problem, such as asking for a round trip through all sites in Milan whose total length is at most 10 km. For this reason,

complexity theory addresses computational problems and not particular problem instances.

ii. Efficiency of algorithms

The *complexity* of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. Usually there are natural units for the domain and range of this function. There are two main complexity measures of the efficiency of an algorithm:

- *Time complexity* is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.
- *Space complexity* is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.

Efficiency is some time taken as the most important criterion and is not always adequately specified. Possible sub-criteria include speed of execution, amount of main or secondary storages required or amount of complexity of maintenance. The basic rule for considering efficiency is that it is not optimize an algorithm against any efficiency sub-criterion until it has first been shown to be effective**.**

b) Explain quick sort algorithm. Also explain time complexity of quick sort algorithm. (Algorithm 4M, time complexity 4M)

Quick sort works by partitioning a given array $A[p \,..\, r]$ into two non-empty sub array $A[p \,..\, q]$ and $A[q+1 \,..\, r]$ such that every key in $A[p \,..\, q]$ is less than or equal to every key in $A[q+1 \,..\, r]$. Then the two subarrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure.

QuickSort (A, p, r)

1. If $p < r$ then
2. q Partition (A, p, r)
3. Quick Sort ($A, p, q-1$)
4. Quick Sort ($A, q + 1, r$)

As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is then partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

Worst-case: $O(N^2)$

This happens when the pivot is the smallest (or the largest) element.
Then one of the partitions is empty, and we repeat recursively the procedure for N-1 elements.

Best-case O(NlogN) The best case is when the pivot is the median of the array, and then the left and the right part will have same size.
There are **logN** partitions, and to obtain each partition we do **N** comparisons (and not more than **N/2** swaps). Hence the complexity is **O(NlogN)**

Average-case - O(NlogN)

c) Solve the following problem :
(W_1, W_2, W_3, W_4, W_5) = (1, 2, 5, 6, 7)
(V_1, V_2, V_3, V_4, V_5) = (1, 6, 18, 22, and 28)
& the capacity of knapsack (M) = 11.
(correct solved problem 8M)

Initially

items	w_i	v_i
I_1	1	1
I_2	2	6
I_3	5	18
I_4	6	22
I_5	7	28

Taking value per weight ratio i.e. $P_i = v_i/w_i$

Item	w_i	v_i	$P_i = v_i/w_i$
I_1	1	1	*1*
I_2	2	6	*3*
I_3	5	18	*3.7*
I_4	6	22	*3.6*
I_5	7	28	*4*

Now arrange the value of p_i in descending order

Item	w_i	v_i	$P_i = v_i/w_i$
I_1	1	1	*1*
I_2	2	6	*3*
I_4	6	22	*3.6*
I_3	5	18	*3.7*
I_5	7	28	*4*

Now, fill the knapsack according to decreasing value of Pi
First we choose item I_1 whose weight is 1, then choose I_2 whose weight is 2, then choose I_4 whose weight is 6. Now the total weight in knapsack is 1+2+6=9. Now the next item is I_3 and its weight is 5 but we want only 2. (The capacity of knapsack (M) =11, so we choose fractional part of it is,

2
6
2
1

=11

The value of fractional part of
I_3=(18/5*2)=7. Thus the maximum value is=1+6+22+7.2=36.2

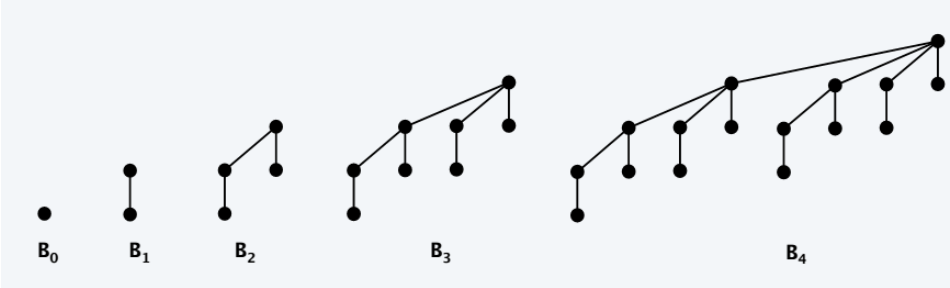
Q.5. Solve any TWO: 16

a) What is a binomial heap? What is the advantage of binomial heap over a heap? (Binomial heap 4M, advantage 4M)

A binomial heap is a set of binomial trees that satisfies the following binomial-heap properties.

1. Each binomial tree in H is heap-ordered: the key of a node is greater than or equal to the key of its parent.
2. There is at most one binomial tree in H whose root has a given degree.
3. A binomial heap is a sequence of binomial trees such that:
4. Each tree is heap-ordered.
5. There is either 0 or 1 binomial tree of order k.

The first property tells us that the root of a heap-ordered tree contains the smallest key in the tree.



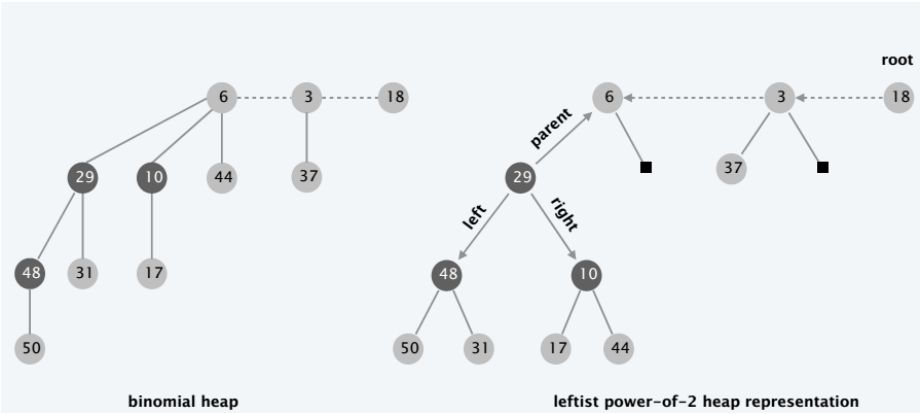
Binomial tree

Properties. Given an order k binomial tree Bk,

1. Its height is k.
2. It has 2k nodes.
3. It has nodes at depth i.
4. The degree of its root is k.
5. Deleting its root yields k binomial trees Bk–1, …, B0.

Binomial Heap Representation

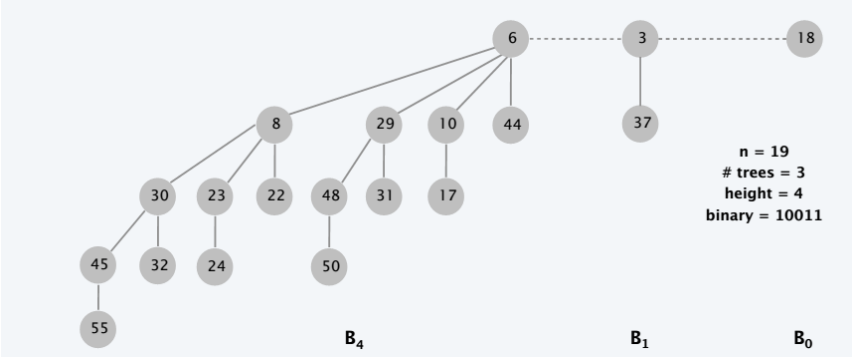
1. Binomial trees. Represent trees using left-child, right-sibling pointers.
2. Roots of trees. Connect with singly-linked list, with degrees decreasing from left to right.



Binomial heap properties

Properties. Given a binomial heap with n nodes:

1. The node containing the min element is a root of B0, B1, …, or Bk.
2. It contains the binomial tree Bi iff bi = 1, where bk· b2 b1 b0 is binary representation of n.
3. It has ≤ ⌊log2 n⌋ + 1 binomial trees.
4. Its height ≤ ⌊log2 n⌋



Operations on Binomial Heap

1. Make-Heap().
2. Insert(H, x), where x is a node to be inserted in H.
3. Minimum(H).
4. Extract-Min(H).
5. Union(H1, H2): merge H1 and H2, creating a new heap.
6. Decrease-Key(H, x, k): decrease x.key (x is a node in H) to k

Operation	Linked List	Binary Heap	Binomial Heap	Fibonacci Heap †	Relaxed Heap
make-heap	1	1	1	1	1
is-empty	1	1	1	1	1
insert	1	log *n*	log *n*	1	1
delete-min	*n*	log *n*	log *n*	log *n*	log *n*
decrease-key	*n*	log *n*	log *n*	1	1
delete	*n*	log *n*	log *n*	log *n*	log *n*
union	1	*n*	log *n*	1	1
find-min	*n*	1	log *n*	1	1

n = number of elements in priority queue † amortized

b) What is the solution generated by the job sequencing with deadlines algorithm to the following scheduling instance n = 7
(P1, P2, ... P7) = (3, 5, 20, 8 ,1, 6, 30)
and (d1, d2, ... P7) = (1, 3, 4, 3, 2, 1, 2)?

(The data provided is insufficient, any data considered and assumed by the student should be considered 8M)

1. **Deadlock** refers to a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain (see *Necessary conditions*).
2. Deadlock is a common problem in multiprocessing where many processes share a specific type of mutually exclusive resource known as a *software*, or *soft*, lock.
3. Deadlocks are particularly troubling because there is no *general* solution to avoid (soft) deadlocks.

There are four conditions that are necessary to achieve deadlock:
Mutual Exclusion - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
Hold and Wait - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
No preemption - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
Circular Wait - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)].

Deadlock Prevention

- Deadlocks can be prevented by preventing at least one of the four required conditions:

Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:

⇒ Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.

⇒ Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.

⇒ Either of the methods described above can lead to starvation if a process requires one or more popular resources.

No Preemption

- Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.

⇒ One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.

⇒ Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.

⇒ Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

Circular Wait

- One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.
- One big challenge in this scheme is determining the relative ordering of the different resources

An Example

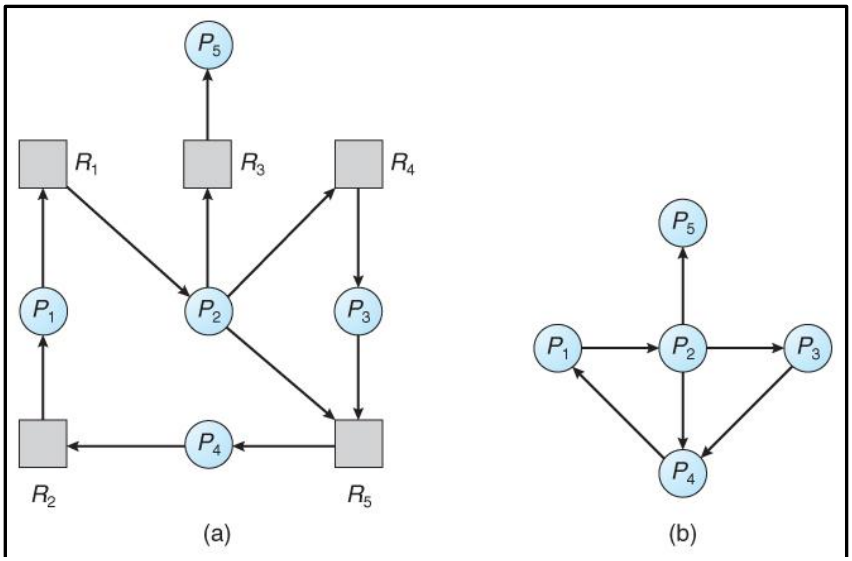
Consider the following situation:

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

- And now consider what happens if process P1 requests 1 instance of A and 2 instances of C. (Request[1] = (1, 0, 2))

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

For the above data we create resource allocation and wait graph



cycles in the wait-for graph indicate deadlocks.

c) Explain the BFS algorithm. Also using a suitable example draw BFS tree. (Explanation 4 M, Example 4M)

Breadth-first search starts at a given vertex s, which is at level 0. In the first stage, we visit all the vertices that are at the distance of one edge away. When we visit there, we paint as "visited," the vertices adjacent to the start vertex s - these vertices are placed into level 1. In the second stage, we visit all the new vertices we can reach at the distance of two edges away from the source vertex s. These new vertices, which are adjacent to level 1 vertices and not previously assigned to a level, are placed into level 2, and so on. The BFS traversal terminates when every vertex has been visited.

To keep track of progress, breadth-first-search colors each vertex. Each vertex of the graph is in one of three states:

1. Undiscovered;
2. Discovered but not fully explored; and
3. Fully explored.

The state of a vertex, u, is stored in a color variable as follows:

1. color[u] = White - for the "undiscovered" state,
2. color [u] = Gray - for the "discovered but not fully explored" state, and
3. color [u] = Black - for the "fully explored" state.

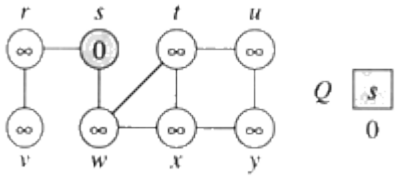
The BFS(G, s) algorithm develops a breadth-first search tree with the source vertex, s, as its root. The parent or predecessor of any other vertex in the tree is the vertex from which it was first discovered. For each vertex, v, the parent of v is placed in the variable π[v]. Another variable, d[v], computed by BFS contains the number of tree edges on the path from s to v. The breadth-first search uses a FIFO queue, Q, to store gray vertices.
BFS(V, E, s)

```
1.    for each u in V − {s}        ▷ for each vertex u in V[G] except s.
2.       do color[u] ← WHITE
3.          d[u] ← infinity
```

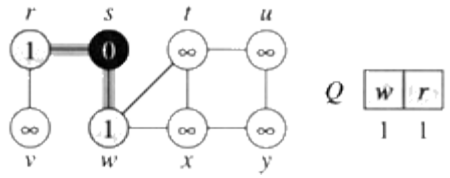
```
4.          π[u] ← NIL
5.      color[s] ← GRAY          ▷ Source vertex discovered
6.      d[s] ← 0                 ▷ initialize
7.      π[s] ← NIL               ▷ initialize
8.      Q ← {}                   ▷ Clear queue Q
9.      ENQUEUE(Q, s)
10      while Q is non-empty
11.         do u ← DEQUEUE(Q)            ▷ That is, u = head[Q]
12.            for each v adjacent to u  ▷ for loop for every node along with edge.
13.                 do if color[v] ← WHITE   ▷ if color is white you've never seen it before
14.                       then  color[v] ← GRAY
15.                             d[v] ← d[u] + 1
16.                             π[v] ← u
17.                             ENQUEUE(Q, v)
18.            DEQUEUE(Q)
19.       color[u] ← BLACK
```

The following figure (from CLRS) illustrates the progress of breadth-first search on the undirected sample graph.

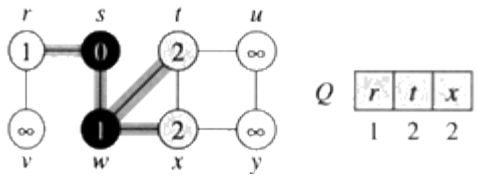
a. After initialization (paint every vertex white, set d[u] to infinity for each vertex u, and set the parent of every vertex to be NIL), the source vertex is discovered in line 5. Lines 8-9 initialize Q to contain just the source vertex s.



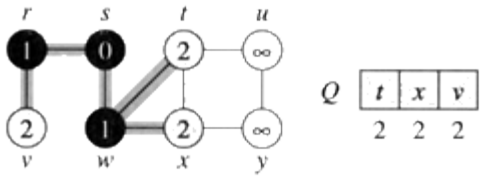
b. The algorithm discovers all vertices 1 edge from s i.e., discovered all vertices (w and r) at level 1.



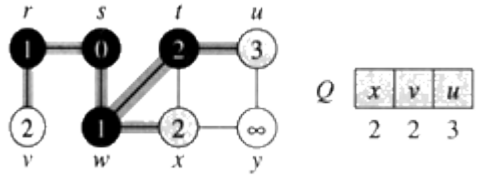
c.



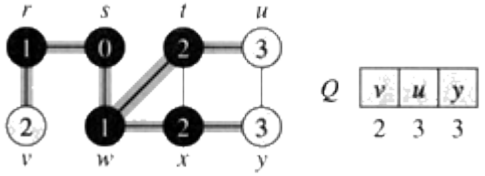
d. The algorithm discovers all vertices 2 edges from s i.e., discovered all vertices (t, x, and v) at level 2.



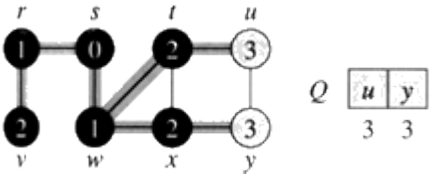
e.



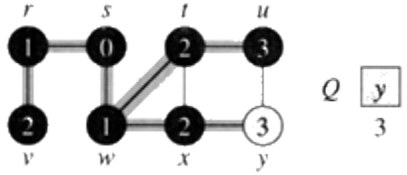
f.



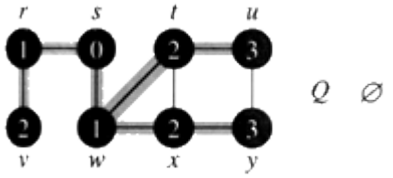
g. The algorithm discovers all vertices 3 edges from s i.e., discovered all vertices (u and y) at level 3.



h.



i. The algorithm terminates when every vertex has been fully explored.



Q.6. Solve any TWO:

a) Explain divide and conquer way of multiplication for the multiplication of981 by 1234.

(Explain and solve- 8 M)

The simple divide and conquer way of multiplication follows the following steps

1. Break A into A11, A12, A21, A22
2. Break B into B11, B12, B21, B22
3. Break C into C11, C12, C21, C22

Let C = product of 2 n/2 by n/2 arrays

$C11=A11\times B11+A12\times B21$

$C12=A11\times B12+A12\times B22$

$C21=A21\times B11+A22\times B21$

$C22=A21\times B12+A22\times B22$

$T(n)=\Theta(1)+8T(n/2)+\Theta(n2)$

- $\Theta(1)$ time to partition matrices
- 8 Multiplications of arrays of size n/2
- $\Theta(n2)$ time to add n×n matrices
- $T(n)=\Theta(n3)$ by master method

For Larger integer multiplication

- Assume integers with n digits
  - Implement with array with one digit per element [Ada bignumpkg]
- Naive divide and conquer:
  - Performance: $T(n)=4T(n/2)+\Theta(n)=\Theta(n2)$
  - More generally: $xy\times wz=xw\times 102k+(xz+yw)\times 10k+yz$
    $981 * 1234 = 09 * 12 *102 +(09*34)+(81*12)*10 + 81*34$
  - Assume xy has 2k digits
  - This requires 4 multiplications

OR

Divide and conquer Multiplication requires same number of figures (of multiplicand and multiplicator) add 0 to the left if needed and number of figures should be a power of 2.

981 x 1234 → 0981 x 1234

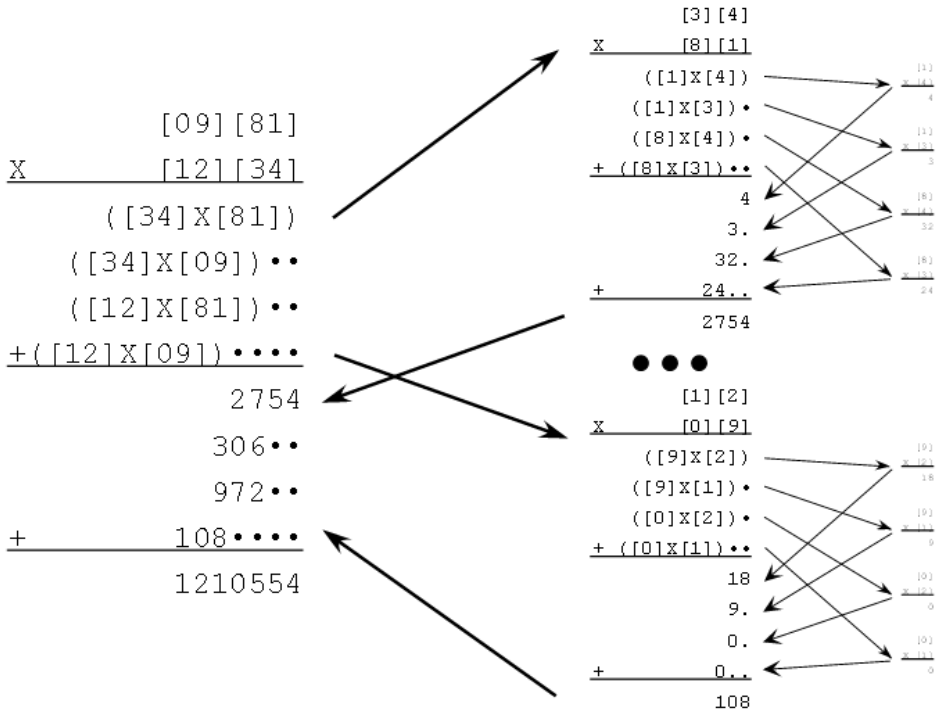
	Multiply		Shift	Result	
i.	09	12	4	108… left x left	shift by # of figures
ii.	09	34	2	302…left x right	shift by half
iii.	81	12	2	972…right x left	shift by half
iv.	81	34	0	2754…right x right	no shift
			total	1210554	

Problem reduced to 4 multiplications of 2 x 2with shift and addition, then

	Multiply		Shift	Result
i.	0	1	2	0
ii.	0	2	1	0
iii.	9	1	1	9
iv.	9	2	0	18
			total	108

OR

Divide-and-Conquer Multiplication

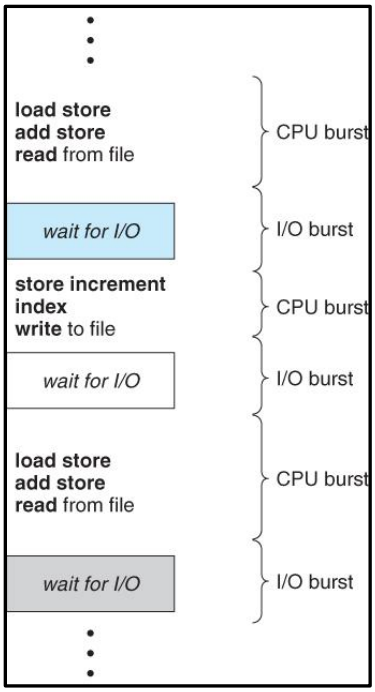


b) What do you mean by scheduling? Also explain scheduling with deadlines by taking a suitable example.
(Scheduling – 4 M, Deadlock- 4 M)

1. Whenever the CPU becomes idle, it is the job of the CPU Scheduler (the short-term scheduler) to select another process from the ready queue to run next.
2. The storage structure for the ready queue and the algorithm used to select the next process are not necessarily a FIFO queue
3. Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. (Even a simple fetch from memory takes a long time relative to CPU speeds.)
4. In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever.
5. A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.
6. The challenge is to make the overall system as "efficient" and "fair" as possible, subject to varying and often dynamic conditions, and where "efficient" and "fair" are somewhat subjective terms, often subject to shifting priority policies.

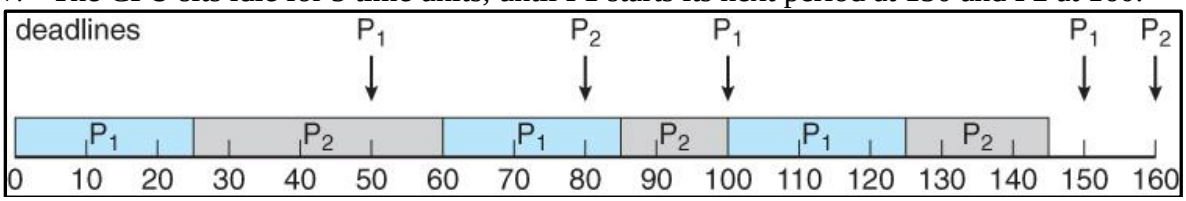
CPU-I/O Burst Cycle

- Almost all processes alternate between two states in a continuing ***cycle***, as shown in Figure below ***:***
- A CPU burst of performing calculations, and
- An I/O burst, waiting for data transfer in or out of the system.



Earliest-Deadline-First Scheduling

1. Earliest Deadline First (EDF) scheduling varies priorities dynamically, giving the highest priority to the process with the earliest deadline.
2. Figure shows our previous example repeated under EDF scheduling:
1. At time 0 P1 has the earliest deadline, highest priority, and goes first., followed by P2 at time 25 when P1 completes its first burst.
2. At time 50 process P1 begins its second period, but since P2 has a deadline of 80 and the deadline for P1 is not until 100, P2 is allowed to stay on the CPU and complete its burst, which it does at time 60.
3. P1 then starts its second burst, which it completes at time 85. P2 started its second period at time 80, but since P1 had an earlier deadline, P2 did not pre-empt P1.
4. P2 starts its second burst at time 85, and continues until time 100, at which time P1 starts its third period.
5. At this point P1 has a deadline of 150 and P2 has a deadline of 160, so P1 preempts P2.
6. P1 completes its third burst at time 125, at which time P2 starts, completing its third burst at time 145.
7. The CPU sits idle for 5 time units, until P1 starts its next period at 150 and P2 at 160.



c) Explain with suitable example the DFS for undirected graph; also explain the Depth First Search algorithm.
(DFS – 4 M, Example – 4 M)

1. Depth-first search selects a source vertex s in the graph and paint it as "visited." Now the vertex s becomes our current vertex. Then, we traverse the graph by considering an arbitrary edge (u, v) from the current vertex u. If the edge (u, v) takes us to a painted vertex v, then we back down to the vertex u. On the other hand, if edge (u, v) takes us to

an unpainted vertex, then we paint the vertex v and make it our current vertex, and repeat the above computation.

2. DFS time-stamps each vertex when its color is changed.
   1. When vertex v is changed from white to gray the time is recorded in d[v].
   2. When vertex v is changed from gray to black the time is recorded in f[v].

3. The discovery and the finish times are unique integers, where for each vertex the finish time is always after the discovery time. That is, each time-stamp is an unique integer in the range of 1 to 2|V| and for each vertex v, d[v] < f[v]. In other words, the following inequalities hold:
   $1 \leq d[v] < f[v] \leq 2|V|$

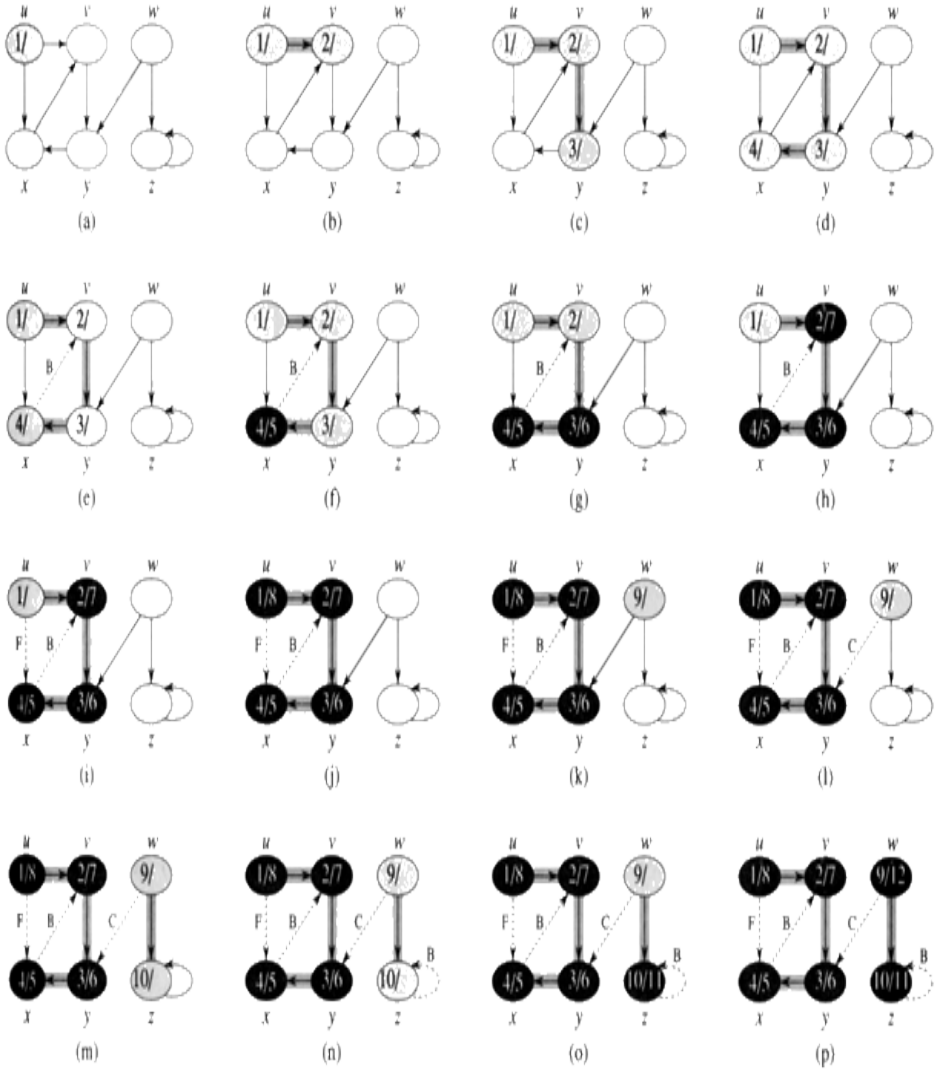
   The DFS forms a depth-first forest comprised of more than one depth-first trees. Each tree is made of edges (u, v) such that u is gray and v is white when edge (u, v) is explored. The following pseudocode for DFS uses a global timestamp time.

```
DFS (V, E)

1.  for each vertex u in V[G]
2.    do color[u] ← WHITE
3.          π[u] ← NIL
4.  time ← 0
5.  for each vertex u in V[G]
6.    do if color[u] ← WHITE
7.          then DFS-Visit(u)          ▷ build a new DFS-tree from u

DFS-Visit(u)

1.  color[u] ← GRAY                    ▷ discover u
2.  time ← time + 1
3.  d[u] ← time
4.  for each vertex v adjacent to u    ▷ explore (u, v)
5.    do if color[v] ← WHITE
6.          then π[v] ← u
7.               DFS-Visit(v)
8.  color[u] ← BLACK
9.  time ← time + 1
10. f[u] ← time                        ▷ we are done with u
```


u v w
1/
x y z
(a)
2/
(b)
3/
(c)
4/
(d)
B
(e)
4/5
(f)
3/6
(g)
2/7
(h)
F B
(i)
1/8
(j)
9/
(k)
C
(l)
10/
(m)
(n)
10/11
(o)
9/12
(p)