Algorithms

UNIT 3

PART B

This page is intentionally left blank

Chapter 1

Asymptotic Analysis

LEARNING OBJECTIVES

- *Algorithm*
- *Recursive algorithms*
- *Towers of Hanoi*
- *Time complexity*
- *Space complexity*
- *SET representation*
- *TREE representation*
- *Preorder traversal*
- *Post-order traversal*
- *In order traversal*
- *Data structure*
- *Worst-case and average-case analysis*
- *Asymptotic notations*
- *Notations and functions*
- *Floor and ceil*
- *Recurrence*
- *Recursion-tree method*
- *Master method*

Algorithm

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.

All algorithms must satisfy the following.

- Input: Zero or more quantities are externally supplied.
- Output: Atleast one quantity is produced.
- Definiteness: Each instruction should be clear and unambiguous.
- Finiteness: The algorithm should terminate after finite number of steps.
- Effectiveness: Every instruction must be very basic.

Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible inputs. This process is called algorithm validation. Analysis of algorithms refers to the task of determining how much computing time and storage an algorithm requires.

Analyzing Algorithms

The process of comparing 2 algorithms rate of growth with respect to time, space, number of registers, network, bandwidth etc is called analysis of algorithms.

This can be done in two ways

1. **Priori Analysis:** This analysis is done before the execution; the main principle behind this is frequency count of fundamental instruction.

   This analysis is independent of CPU, OS and system architecture and it provides uniform estimated values.

2. **Posterior analysis:** This analysis is done after the execution. It is dependent on system architecture, CPU, OS etc. it provides non-uniform exact values.

Recursive Algorithms

A recursive function is a function that is defined in terms of itself. An algorithm is said to be recursive if the same algorithm is invoked in the body.

Towers of Hanoi

There was a diamond tower (labeled *A*) with 64-golden disks. The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top. Besides this tower there were 2 other diamond towers (labeled *B* and *C*) we have to move the disks from tower A to tower *B* using tower *C* for intermediate storage. As the disks are very heavy, they can be moved only one at a time. No disk can be on top of a smaller disk .

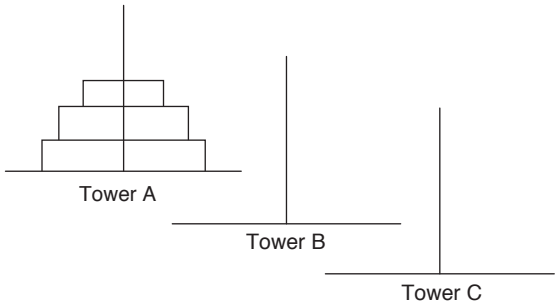
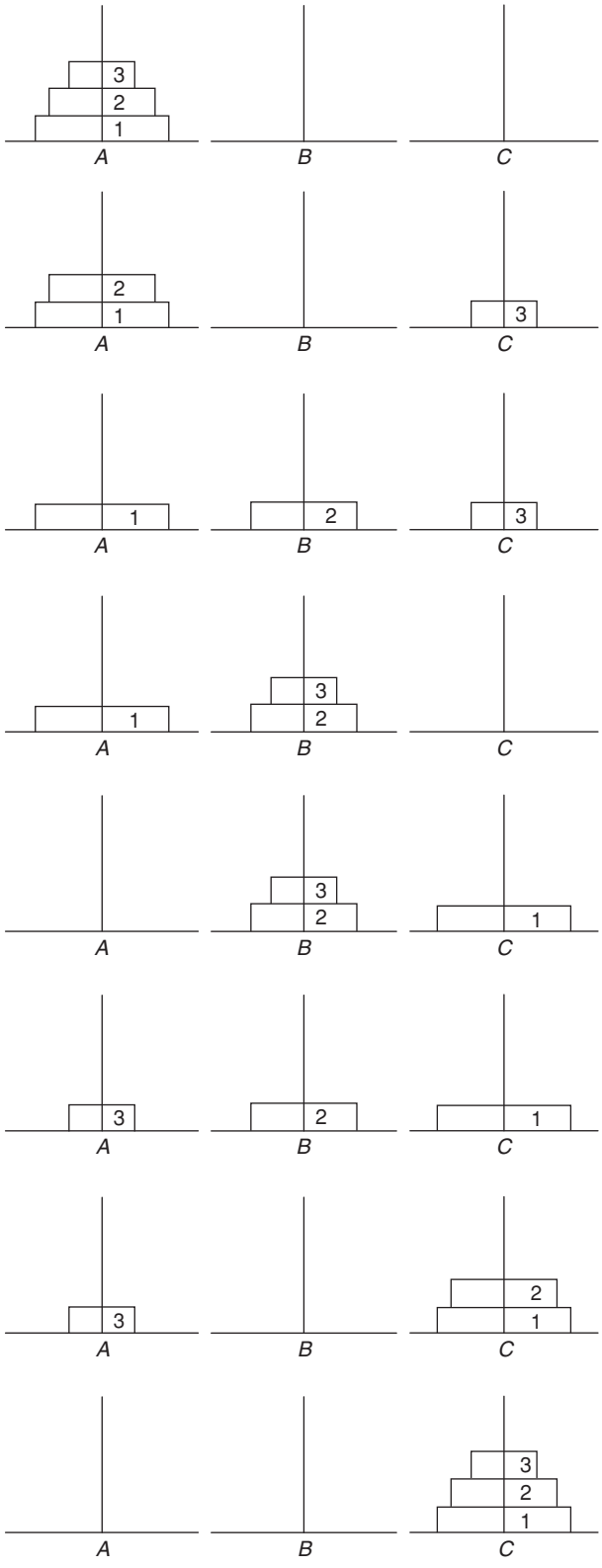


Figure 1 Towers of Hanoi

Assume that the number of disks is 'n'. To get the largest disk to the bottom of tower B, we move the remaining (n – 1) disks to tower C and then move the largest to tower B. Now move the disks from tower C to tower B.

Example:



To move '3' disks from tower A to tower 'C' requires 7 disk movements

∴ For 'n' disks, the number of disk movements required is $2^n - 1 = 2^3 - 1 = 7$

Time complexity

$$T(n) = 1 + 2T(n-1)$$
$$T(n) = 1 + 2(1 + 2\,(T(n-2)))$$
$$T(n) = 1 + 2 + 2^2\,T(n-2)$$
$$T(n) = 1 + 2 + 2^2\,(1 + 2T(n-3))$$
$$T(n) = 1 + 2 + 2^2 + 2^3 + T(n-3)$$
$$T(n) = 1 + 2 + 2^2 + \cdots + 2^{i-1} + 2^i\,\mathrm{T(n - i)}$$

$$T(n) = \sum_{i=0}^{n-1} 2^i$$

The time complexity is exponential, it grows as power of 2.

$$\therefore \quad T(n) \cong O(2^n)$$

Space complexity

The space complexity of an algorithm is the amount of memory it needs to run to completion. The measure of the quantity of input data is called the size of the problem. For example, the size of a matrix multiplication problem might be the largest dimension of the matrices to be multiplied. The size of a graph problem might be the number of edges. The limiting behavior of the complexity as size increases is called the asymptotic time complexity.

- It is the asymptotic complexity of an algorithm which ultimately determines the size of problems that can be solved by the algorithm.
- If an algorithm processes inputs of size 'n' in time cn^2 for some constant c, then we say that the time complexity of that algorithm is $O(n^2)$, more precisely a function $g(n)$ is said to be $O(f(n))$ if there exists a constant c such that $g(n) \leq c(f(n))$ for all but some finite set of non-negative values for n.
- As computers become faster and we can handle larger problems, it is the complexity of an algorithm that determines the increase in problem size that can be achieved with an increase in computer speed.
- Suppose we have 5 algorithms Algorithm 1 – Algorithm 5 with the following time complexities.

Algorithm	Time Complexity
Algorithm – 1	n
Algorithm – 2	$n \log n$
Algorithm – 3	n^2
Algorithm – 4	n^3
Algorithm – 5	$2n$

The time complexity is, the number of time units required to process an input of size 'n'. Assume that input size 'n' is 1000 and one unit of time equals to 1 millisecond.

The following figure gives the sizes of problems that can be solved in one second, one minute, and one hour by each of these five algorithms.

Algorithm	Time Complexity	Maximum Problem Size		
		1 sec	1 min	1 hour
Algorithm – 1	n	1000	6×10^4	3.6×10^6
Algorithm – 2	$n \log n$	140	4893	2.0×10^5
Algorithm – 3	n^2	31	244	1897
Algorithm – 4	n^3	10	39	153
Algorithm – 5	$2n$	9	15	21

From the above table, we can say that different algorithms will give different results depending on the input size. Algorithm – 5 would be best for problems of size $2 \le n \le 9$, Algorithm – 3 would be best for $10 \le n \le 58$, Algorithm – 2 would be best for $59 \le n \le 1025$, and Algorithm – 1 is best for problems of size greater than 1024.

Set Representation

A common use of a list is to represent a set, with this representation the amount of memory required to represent a set is proportional to the number of elements in the set. The amount of time required to perform a set operation depends on the nature of the operation.

- Suppose A and B are 2 sets. An operation such as $A \cap B$ requires time atleast proportional to the sum of the sizes of the 2 sets, since the list representing A and the list representing B must be scanned atleast once.
- The operation $A \cup B$ requires time atleast proportional to the sum of the set sizes, we need to check for the same element appearing in both sets and delete one instance of each such element.
- If A and B are disjoint, we can find $A \cup B$ in time independent of the size of A and B by simply concatenating the two lists representing A and B.

Graph Representation

A graph $G = (V, E)$ consists of a finite, non-empty set of vertices V and a set of edges E. If the edges are ordered pairs (V, W) of vertices, then the graph is said to be directed; V is called the tail and W the head of the edge (V, W). There are several common representations for a graph $G = (V, E)$. One such representation is adjacency matrix, $a|V|X|V|$ matrix M of 0's and 1's, where the ij_{th} element, $m[i, j] = 1$, if and only if there is an edge from vertex i to vertex j.

- The adjacency matrix representation is convenient for graph algorithms which frequently require knowledge of whether certain edges are present.
- The time needed to determine whether an edge is present is fixed and independent of $|V|$ and $|E|$.
- Main drawback of using adjacency matrix is that it requires $|V|^2$ storage even if the graph has only $O(|V|)$ edges.
- Another representation for a graph is by means of lists. The adjacency list for a vertex v is a list of all vertices W adjacent to V. A graph can be represented by $|V|$ adjacency lists, one for each vertex.

Example:

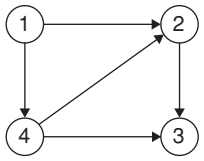


Figure 2 Directed graph

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0

Figure 3 Adjacency matrix

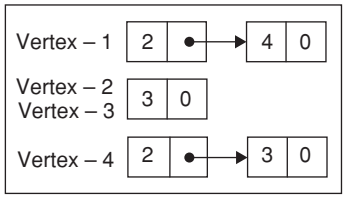


Figure 4 Adjacency lists

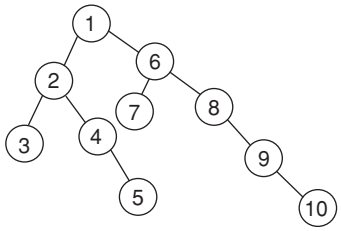
There are edges from vertex – 1 to vertex – 2 and 4, so the adjacency list for 1 has items 2 and 4 linked together in the format given above.

- The adjacency list representation of a graph requires storage proportional to $|V| + |E|$, this representation is used when $|E| << |V|^2$.

Tree Representation

A directed graph with no cycles is called a directed acyclic graph. A directed graph consisting of a collection of trees is called a forest. Suppose the vertex 'v' is root of a sub tree, then the depth of a vertex 'v' in a tree is the length of the path from the root to 'v'.

- The height of a vertex 'v' in a tree is the length of a longest path from 'v' to a leaf.
- The height of a tree is the height of the root
- The level of a vertex 'v' in a tree is the height of the tree minus the depth of 'v'.



	Left child	Right child
1	2	6
2	3	4
3	0	0
4	0	5
5	0	0
6	7	8
7	0	0
8	0	9
9	0	10
10	0	0

Figure 5 A binary tree and its representation

- Vertex 3 is of depth '2', height '0' and the level is 2 (Height of tree – depth of '3' = 4 – 2 = 2).
- A binary tree is represented by 2 arrays: left child and right child.
- A binary tree is said to be complete if for some integer k, every vertex of depth less than k has both a left child and a right child and every vertex of depth k is a leaf. A complete binary tree of height k has exactly $(2^{k+1} - 1)$ vertices.
- A complete binary tree of height k is often represented by a single array. Position 1 in the array contains the root. The left child of the vertex in position 'i' is located at position '$2i$' and the right child at position '$2i + 1$'.

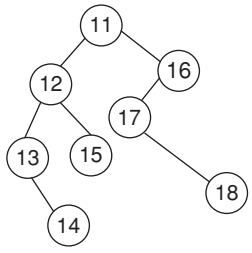
Tree Traversals

Many algorithms which make use of trees often traverse the tree in some order. Three commonly used traversals are pre-order, postorder and inorder.

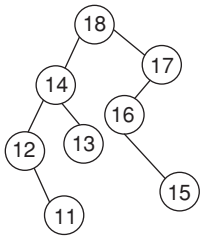
Pre-order Traversal

A pre-order traversal of T is defined recursively as follows:

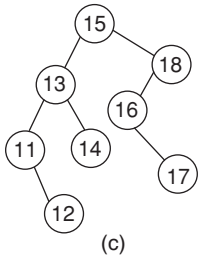
1. Visit the root.
2. Visit in pre-order the sub trees with roots $v_1, v_2 \ldots v_k$ in that order.



(a)



(b)



(c)

Figure 6 (a) Pre-order, (b) Post-order **(c)** In-order

Post-order traversal

A post-order traversal of T is defined recursively as follows:

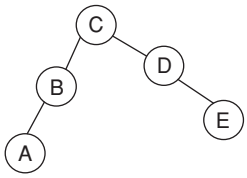
1. Visit in post-order the sub trees with roots v_1, v_2, v_3, … v_k in that order.
2. Visit the root r.

In-order Traversal

An in-order traversal is defined recursively as follows:

1. Visit in-order the left sub tree of the root 'r'.
2. Visit 'r'.
3. Visit in inorder the right sub tree of r.

Example: Consider the given tree



What are the pre-order, post-order and in-order traversals of the above tree?

Solution: Pre-order – CBADE
Post-order – ABEDC
In-order – ABCDE

Data Structure

A data structure is a way to store and organize data in-order to facilitate access and modifications. No single data structure works well for all purposes, it is important to know the strengths and limitations of several data structures.

Efficiency

Algorithms devised to solve the same problem often differ dramatically in their efficiency. Let us compare efficiencies of Insertion sort and merge sort; insertion sort, takes time equal to $C_1 n^2$ to sort 'n' elements, where C_1 is a constant that does not depend on 'n'. It takes time proportional to n^2, merge sort takes time equal to $C_2 n \log n$, C_2 is another constant that also does not depend on 'n'. Insertion sort has a smaller constant factor than merge sort ($C_1 < C_2$) constant factors are far less significant in the running time.

Merge sort has a factor of 'log n' in its running time, insertion sort has a factor of 'n', which is much larger. Insertion sort is faster than merge sort for small input sizes, once the input size 'n' becomes large enough, merge sort will perform better. No matter how much smaller C_1 is than C_2. There will always be a crossover point beyond which merge sort is faster.

Example: Consider 2 computers, computer A (faster computer), B (slower computer). Computer A runs insertion sort and computer B runs merge sort. Each computer is given 2 million numbers to sort. Suppose that computer A executes one billion instruction per second and computer B executes only 10 million instructions per second, computer A is 100 times faster than computer B ($C_1 = 4$, $C_2 = 50$). How much time is taken by both the computers?

Solution: Insertion sort takes $C_1 * n^2$ time
Merge sort takes $C_2 * n * \log n$ time
$C_1 = 4$, $C_2 = 50$
Computer A takes

$$\frac{4 \times (2 \times 10^6)^2 \text{ instructions}}{10^9 \text{ instructions/second}} \cong 4000 \text{ seconds}$$

Computer B takes

$$= \frac{50 \times 2 \times 10^6 \times \log(2 \times 10^6) \text{ insturctions}}{10^7 \text{ instructions/second}}$$
$$= 209 \text{ seconds}$$

By using an algorithm whose running time grows more slowly, even with an average compiler, computer B runs 20 times faster than computer A. The advantage of merge sort is even more pronounced when we sort ten million numbers. As the problem size increases, so does the relative advantage of merge sort.

Worst-case and average-case analysis

In the analysis of insertion sort, the best case occurs when the array is already sorted and the worst case, in which the input array is reversely sorted. We concentrate on finding the worst-case running time, that is the longest running time for any input of size 'n'.

- The worst-case running time of an algorithm is an upper bound on the running time for any input. It gives us a guarantee that the algorithm will never take any longer.
- The 'average-case' is as bad as the worst-case. Suppose that we randomly choose 'n' numbers and apply insertion sort. To insert an element $A[j]$, we need to determine where to insert in sub-array A $[1 \cdots J-1]$. On average half the elements in $A[1 \cdots J-1]$ are less than $A[j]$ and half the elements are greater. So $t_j = j/2$. The average-case running time turns out to be a quadratic function of the input size.

Asymptotic Notations

Asymptotic notations are mostly used in computer science to describe the asymptotic running time of an algorithm. As an example, an algorithm that takes an array of size n as input and runs for time proportional to n^2 is said to take $O(n^2)$ time.

5 Asymptotic Notations:

- O (Big-oh)
- θ (Theta)
- Ω (Omega)
- o (Small-oh)
- ω

How to Use Asymptotic Notation for Algorithm Analysis?

Asymptotic notation is used to determine rough estimates of relative running time of algorithms. A worst-case analysis of any algorithm will always yeild such an estimate, because it gives an upper bound on the running time $T(n)$ of the algorithm, that is $T(n)$ $g(n)$.

Example:

$a \leftarrow 0$	1 unit	1 time
for $i \leftarrow 1$ to n do{	1 unit	n times
for $j \leftarrow 1$ to i do{	1 unit	$n(n+1)/2$ times
$a \leftarrow a + 1$	1 unit	$n(n+1)/2$ times

Where the times for the inner loop have been computed as follows: For each i from 1 to n, the loop is executed i times, so the total number of times is $1+2+3+\cdots+n = \sum_{i=1}^{n} i = n(n+1)/2$

Hence in this case
$T(n) = 1 + n + 2n\ (n+1)/2 = n^2 + 2n + 1$
If we write $g(n) = n^2 + 2n + 1$, then $T(n) \in \theta(g(n))$,
That is $T(n) \in \theta(n^2 + 2n + 1)$, we actually write $T(n) \in \theta(n^2)$, as recommended by the following rule:

- Although the definitions of asymptotic notation allow one to write, for example, $T(n) \in O(3n^2 + 2)$.

  We simplify the function in between the parentheses as much as possible (in terms of rate of growth), and write instead $T(n) \in O(n^2)$
  For example: $T(n) \in \theta(4n^3 - n^2 + 3)$
  $T(n) \in \theta(n^3)$

  For instance $O\left(\sum_{i=1}^{n} i\right)$, write $O(n^2)$ after computing the sum.

- In the spirit of the simplicity rule above, when we are to compare, for instance two candidate algorithms A and B having running times ($T_A(n) = n^2 - 3n + 4$ and $T_B(n) = 5n^3 + 3$, rather than writing $T_A(n) \in O(T_B(n))$, we write $T_A(n) \in \theta(n^2)$, and $T_B(n) \in \theta(n^3)$, and then we conclude that A

is better than B, using the fact that n^2(quadratic) is better than n^3(cubic) time, since $n^2 \in O(n^3)$.

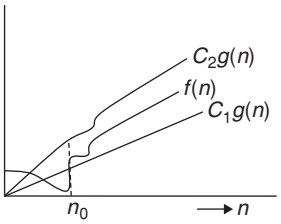
Order of Growth

In the rate of growth or order of growth, we consider only the leading term of a formula. Suppose the worst case running time of an algorithm is $an^2 + bn + c$ for some constants a, b and c. The leading term is an^2. We ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. Thus we can write, the worst-case running time is $\theta(n^2)$.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth. Due to constant factors and lower order terms, this evaluation may be in error for small inputs. But for large inputs, $\theta(n^2)$ algorithm will run more quickly in the worst-case than $\theta(n^3)$ algorithm.

θ-Notation

A function $f(n)$ belongs to the set $\theta(g(n))$ if there exists a positive constant C_1 and C_2 such that it can be "sand witched" between $C_1g(n)$ and $C_2g(n)$ for sufficiently large n. We write $f(n) \in \theta(g(n))$ to indicate that $f(n)$ is a member of $\theta(g(n))$ or we can write $f(n) = \theta(g(n))$ to express the same notation.



The above figure gives an intuitive picture of functions $f(n)$ and $g(n)$, where we have that $f(n) = \theta(g(n))$, for all the values of 'n' to the right of n_o, the value of $f(n)$ lies at or above $C_1g(n)$ and at or below $C_2g(n)$. $g(n)$ is asymptotically tight bound for $f(n)$. The definition of $\theta(g(n))$ requires that every member $f(n) \in \theta(g(n))$ be asymptotically non-negative, that is $f(n)$ must be non-negative whenever 'n' is sufficiently large.

The θ-notation is used for asymptotically bounding a function from both above and below. We would use θ(theta) notation to represent a set of functions that bounds a particular function from above and below.

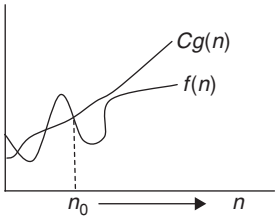
Definition: We say that a function $f(n)$ is theta of $g(n)$ written as $f(n) = \theta(g(n))$ if such exists positive constants C_1, C_2 and n_0 such that $0 \le C_1g(n) \le f(n) \le C_2\, g(n)$, $\forall\ n \ge n_0$.

Example: Let $f(n) = 5.5n^2 - 7n$, verify whether $f(n)$ is $\theta(n^2)$. Lets have constants $c_1 = 9$ and $n_0 = 2$, such that $0 \le f(n) \le C_1\, n^2$, $\forall n \ge n_0$. From example, 4 we have constants $C_2 = 3$, and $n_0 = 2.8$, such that $0 \le C_2\, n^2 \le f(n)$, $\forall n \ge n_0$. To show $f(n)$ is $\theta(n^2)$, we have got hold of two constants C_1 and C_2. We fix the n_0 for θ as maximum $\{2, 2.8\} = 2.8$.

- The lower order terms of an asymptotically positive function can be ignored in determining asymptotically tight bounds because they are insignificant for large n.
- A small fraction of the highest order term is enough to dominate the lower order term. Thus setting C_1 to a value that is slightly smaller than the coefficient of the highest order term and setting C_2 to a value that is slightly larger permits the inequalities in the definition of θ-notation to be satisfied. If we take a quadratic function $f(n) = an^2 + bn + c$, where a, b and c are constants and $a > 0$. Throwing away the lower order terms and ignoring the constant yields $f(n) = \theta(n^2)$.
- We can express any constant function as $\theta(n^0)$, or $\theta(1)$ we shall often use the notation $\theta(1)$ to mean either a constant or a constant function with respect to some variable.

O-Notation

We use O-notation to give an upper bound on a function, within a constant factor.



The above figure shows the intuition behind O-notation. For all values 'n' to the right of n_0, the value of the function $f(n)$ is on or below $g(n)$. We write $f(n) = O(g(n))$ to indicate that a function $f(n)$ is a member of the set $O(g(n))$.

$f(n) = \theta(g(n))$ implies $f(n) = O(g(n))$. Since θ notation is stronger notation than O-notation set theoretically, we have $\theta(g(n)) \subseteq O(g(n))$. Thus any quadratic function $an^2 + bn + c$, where $a > 0$, is in $\theta(n^2)$ also shows that any quadratic function is in $O(n^2)$ when we write $f(n) = O(g(n))$, we are claiming that some constant multiple of $g(n)$ is an asymptotic upper bound on $f(n)$, with no claim about how tight an upper bound it is.

The O-notation is used for asymptotically upper bounding a function. We would use O (big-oh) notation to represent a set of functions that upper bounds a particular function.

Definition We say that a function $f(n)$ is big oh of $g(n)$ written as $f(n) = O(g(n))$ if there exists positive constants C and n_0 such that

$$0 \le f(n) \le Cg(n),\ \forall\ n \ge n_o$$

Solved Examples

Example 1: let $f(n) = n^2$
Then $f(n) = O(n^2)$
$f(n) = O(n^2 \log n)$
$f(n) = O(n^{2.5})$
$f(n) = O(n^3)$
$f(n) = O(n^4)$ ⋯ so on.

Example 2: Let $f(n) = 5.5n^2 - 7n$, verity whether $f(n)$ is $O(n^2)$

Solution: Let C be a constant such that

$$5.5n^2 - 7n \le Cn^2, \text{or } n \ge \frac{7}{c - 5.5}$$

Fix $C = 9$, to get $n \ge 2$
So our $n_0 = 2$ and $C = 9$
This shows that there exists, positive constants $C = 9$ and $n_0 = 2$ such that
$0 \le f(n) \le Cn^2$, $\forall\ n \ge n_0$

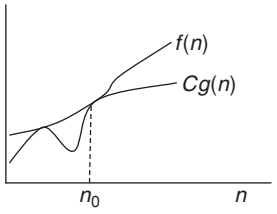
Example 3:

$$h(n) = 3n^3 + 10n + 1000 \log n \in O(n^3)$$

$$h(n) = 3n^3 + 10n + 1000 \log n \in O(n^4)$$

- Using O-notation, we can describe the running time of an algorithm by inspecting the algorithm's overall structure. For example, the doubly nested loop structure of the insertion sort algorithm yields an $O(n^2)$ upper bound on the worst-case running time. The cost of each iteration of the inner loop is bounded from above by $O(1)$ (constant), the inner loop is executed almost once for each of the n^2 pairs.
- $O(n^2)$ bound on worst-case running time of insertion sort also applies to its running time on every input.
- The $\theta(n^2)$ bound on the worst-case running time of insertion sort, however, does not imply a $\theta(n^2)$ bound on the running time of insertion sort on every input, when the input is already sorted, insertion sort runs in $\theta(n)$ time.

Ω *(omega)-notation*



The Ω-notation is used for asymptotically lower bounding a function. We would use Ω(big-omega) notation to represent a set of functions that lower bounds a particular function.

Definition We say that a function $f(n)$ is big-omega of $g(n)$ written as $f(n) = \Omega(g(n))$ if there exists positive constants C and n_0 such that

$$0 \le Cg(n) \le f(n)\ ,\ \forall\ n \ge n_0$$

The intuition behind Ω-notation is shown in the above figure. For all values 'n' to the right of n_0, the value of $f(n)$ is on or above $Cg(n)$. For any 2 functions $f(n)$ and $g(n)$ we have $f(n) = \theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. From the above statement we can say that, $an^2 + bn + c = \theta(n^2)$ for any constants a, b and c, where $a > 0$, immediately implies that

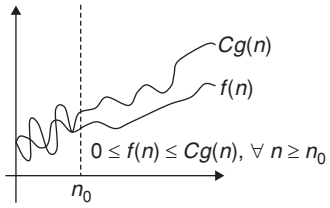
$$\therefore\ an^2 + bn + c = \Omega(n^2)$$

$$\therefore\ an^2 + bn + c = O(n^2)$$

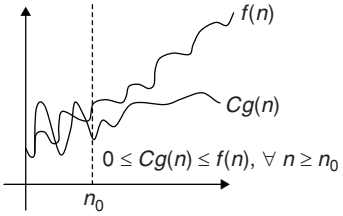
Example 4: Let $f(n) = 5.5n^2 - 7n$.
Verity whether $f(n)$ is $\Omega(n^2)$

Solution: Let C be a constant such that $5.5n^2 - 7n \ge Cn^2$ or $n \ge \frac{7}{5.5 - C}$. Fix $C = 3$, to get $n \ge 2.8$. So, our $n_0 = 2.8$ and $C = 3$

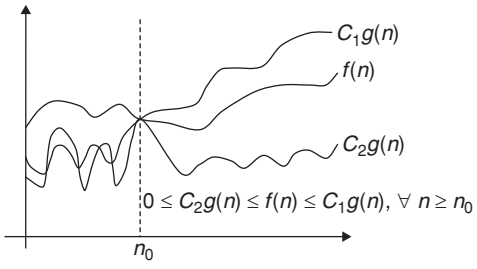
This shows that there exists positive constants $C = 3$ and $n_0 = 2.8$, such that $0 \le Cn^2 \le f(n)$, $\forall n \ge n_0$.



(a) $f(n) = O(g(n))$



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



(c) $f(n) = \theta(g(n))$

Figure 7 A diagrammatic representation of the asymptotic notations *O*, *W* and *q*

- Ω-notation describes a lower bound; it is used to bound the best-case running time of an algorithm. The best-case running time of insertion sort is $\Omega(n)$. The running time of insertion sort falls between $\Omega(n)$ and $O(n^2)$, since it falls anywhere between a linear function of 'n' and a quadratic function of 'n'.

- When we say that the running time of an algorithm is $\Omega(g(n))$, we mean that no matter what particular input of size 'n' is chosen for each value of n, the running time on that input is at least a constant times $g(n)$, for sufficiently large 'n'.

O-notation

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^3 = O(n^3)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use O-notation to denote an upper bound that is not asymptotically tight.

ω-notation

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight.

It is defined as

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$

Comparison of functions

Transitivity

1. $f(n) = \theta(g(n))$ and $g(n) = \theta(h(n))$
   $\Rightarrow f(n) = \theta(h(n))$
2. $f(n) = O(g(n))$ and $g(n) = O(h(n))$
   $\Rightarrow f(n) = O(h(n))$
3. $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$
   $\Rightarrow f(n) = \Omega(h(n))$
4. $f(n) = o(g(n))$ and $g(n) = o(h(n))$
   $\Rightarrow f(n) = o(h(n))$
5. $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$
   $\Rightarrow f(n) = \omega(h(n))$

Reflexivity

1. $f(n) = \theta(f(n))$
2. $f(n) = O(f(n))$
3. $f(n) = \Omega(f(n))$

Symmetry

$f(n) = \theta(g(n))$ if and only if $g(n) = \theta(f(n))$

Transpose symmetry

1. $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
2. $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

Notations and Functions

Floor and Ceil

For any real number 'x', we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ called as floor of x and the least integer greater than or equal to x by $\lceil x \rceil$ called as ceiling of x.

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1 \text{ for any integer } n,$$

$$\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n,$$

For any real number $n \ge 0$ and integer $a, b > 0$

$$\left\lceil \frac{\lceil n/a \rceil}{b} \right\rceil = \left\lceil \frac{n}{ab} \right\rceil$$

$$\left\lfloor \frac{\lfloor n/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{n}{ab} \right\rfloor$$

Polynomials

Given a non-negative integer k, a polynomial in n of degree 'k' is a function $p(n)$ of the form $p(n) = \sum_{i=0}^{k} a_i n^i$

Where the constants $a_0, a_1, \ldots a_k$ are the coefficients of the polynomial and $a_k \neq 0$.

For an asymptotically positive polynomial $p(n)$ of degree k, we have $p(n) = \theta(n^k)$

Exponentials

For all real $a > 0$, m and n, we have the following identities:

$$a^0 = 1$$
$$a^1 = a$$
$$a^{-1} = \frac{1}{a}$$
$$(a^m)^n = a^{mn}$$
$$(a^m)^n = (a^n)^m$$
$$a^m a^n = a^{m+n}$$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

- For all real x, we have inequality $e^x \ge 1 + x$
- If $x = 0$, we have $1 + x \le e^x \le 1 + x + x^2$

Logarithms

$\lg n = \log_2 n$ (binary logarithm)
$\ln n = \log_e n$ (natural logarithm)
$\lg^k n = (\log n)^k$ (exponentiation)
$\lg \lg n = \lg (\lg n)$ (composition)
For all real $a > 0$, $b > 0$, $c > 0$ and n,
$\log_c (ab) = \log_c a + \log_c b$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b (1/a) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b^c} = c^{\log_b^a}$$

Factorials

$n!$ is defined for integers $n \geq 0$ as

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! * n & n > 0 \end{cases}$$

A weak upper bound on the factorial function is $n! \leq n^n$ since each of the n terms in the factorial product is almost n.

$$n! = o(n^n)$$
$$n! = \omega(2^n)$$
$$\lg (n!) = \theta(n \log n)$$

Iterated Logarithm

The notation $\lg^* n$ is used to denote the iterated logarithm. Let '$\lg^{(i)}$ n' be as defined above, with $f(n) = \lg n$. The logarithm of a non-positive number is undefined, '$\lg^{(i)} n$' is defined only if $\lg^{(i-1)} n > 0$;

The iterated logarithm function is defined as $\lg^* n = \min \{i \geq 0 : \lg^{(i)} n \leq 1\}$. This function is a very slowly growing function.

$$\lg^* 2 = 1$$
$$\lg^* 4 = 2$$
$$\lg^* 16 = 3$$
$$\lg^* 65536 = 4$$
$$\lg^* (2^{65536}) = 5$$

Recurrences

When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence. A recurrence is an equation that describes a function in terms of its value on smaller inputs. For example, the worst-case running time $T(n)$ of the merge-sort can be described as

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(n/2) + \theta(n) & \text{if } n > 1 \end{cases}$$

The time complexity of merge-sort algorithm in the worst-case is $T(n) = \theta(n \log n)$

There are 3 methods to solve recurrence relations:

1. Substitution method
2. Recursion-tree method
3. Master method

Substitution Method

In this method one has to guess the form of the solution. It can be applied only in cases when it is easy to guess the form of the answer. Consider the recurrence relation

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

We guess that the solution is $T(n) = O(n \log n)$ we have to prove that

$$T(n) \leq c\, n \log n \quad (\therefore c > 0)$$

Assume that this bound holds for $\lfloor n/2 \rfloor$

$$T(n/2) \leq c(n/2).\ \log (n/2) + n$$
$$T(n) \leq 2(c(n/2 \log (n/2)) + n$$
$$\leq cn \log n - cn \log 2 + n$$
$$\leq cn \log n - cn + n$$
$$\leq cn \log \qquad (\therefore c \geq 1)$$

Recursion-tree Method

In a recursion-tree, each node represents the cost of single sub problem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion. Recursion trees are useful when the recurrence describes the running time of a divide-and-conquer algorithm.

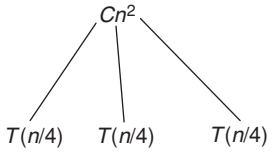
Example:

Consider the given recurrence relation

$$T(n) = 3T(n/4) + \theta(n^2)$$

We create a recursion tree for the recurrence

$$T(n) = 3T(n/4) + Cn^2$$



The Cn^2 term at the root represents the cost at the top level of recursion, and the three sub trees of the root represent the costs incurred by the sub problems of size $n/4$.

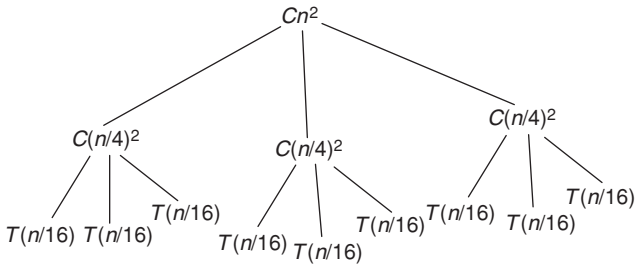


Figure 8 Recursion tree for $T(n) = 3T(n/4) + cn^2$

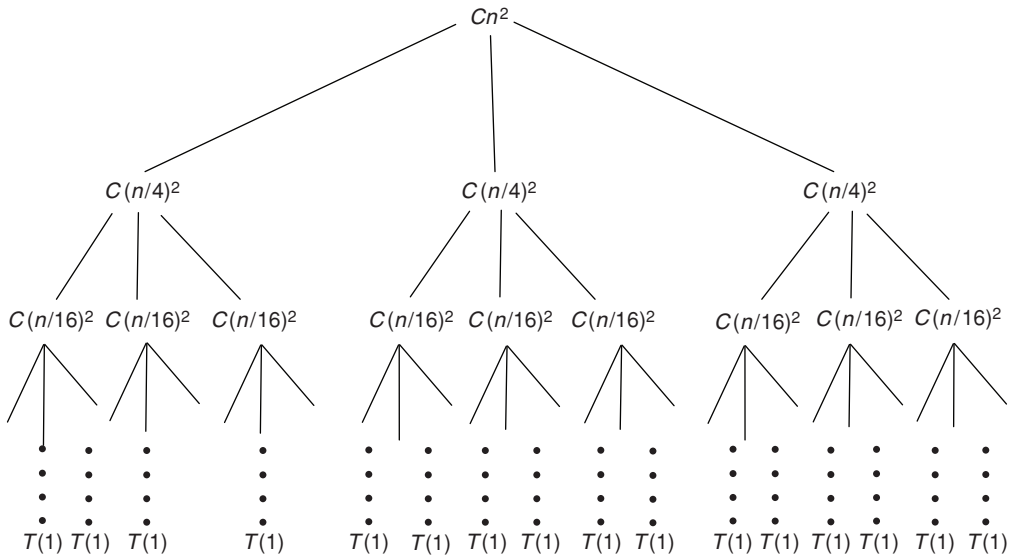


Figure 9 Expanded Recursion tree with height $\log_4 n$ ($\therefore$ levels $\log_4 n + 1$)

The sub-problem size for a node at depth 'i' is $n/4^i$, at this depth, the size of the sub-problem would be $n = 1$, when $n/4^i = 1$ or $i = \log_4 n$, the tree has $\log_4 n+1$ levels.

- We have to determine the cost at each level of the tree. Each level has 3 times more nodes than the level above, so the number of nodes at depth 'i' is 3^i.
- Sub problem sizes reduce by a factor of '4' for each level we go down from the root, each node at depth i, for $i = 0, 1, 2 \ldots \log_4 n-1$, has a cost of $c(n/4^i)^2$.

Total cost over all nodes at depth i, for $i = 0, 1, \ldots \log_4 n-1$

$$= 3^i * c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2$$

The last level, at depth $\log_4 n$ has 3^i nodes $= 3^{\log_4 n} = n^{\log_4 3}$ each contributing cost $T(1)$, for a total cost of $n^{\log_4 3}\, T(1)$, which is $\theta(n^{\log_4 3})$ cost of the entire tree is equal to sum of costs over all levels.

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots +$$

$$\left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \cdots + \theta(n^{\log_4 3})$$

$$= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \theta(n^{\log_4 3})$$

$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \theta(n^{\log_4 3})$$

$$= \frac{1}{1-\left(\frac{3}{16}\right)} cn^2 + \theta(n^{\log_4 3})$$

$$= \frac{16}{13} 2cn + \theta(n^{\log_4 3}) = O(n^2)$$

Master Method

Let $a \geq 1$ and $b > 1$ be cons-tants, let $f(n)$ be a function and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

$T(n)$ can be bounded asymptotically as follows

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \theta(n^{\log_b a})$
2. If $f(n) = \theta(n^{\log_b a})$ then $T(n) = \theta(n^{\log_b a}. \log n)$
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \theta(f(n))$.

Note: In the first case, not only must $f(n)$ be smaller than $n \log_b a$, it must be polynomially smaller. That is, $f(n)$ must be asymptotically smaller than $n^{\log_b a}$ by a factor of n^{ϵ}, for some constant $\epsilon > 0$.

In the third case, not only must $f(n)$ be larger than $n^{\log_b a}$, it must be polynomially larger and in addition satisfy the regularity condition $af(n/b) \leq Cf(n)$.

Example: Consider the given recurrence relation $T(n) = 9T(n/3) + n$.

To apply master theorem, the recurrence relation must be in the following form:

$$T(n) = aT(n/b) + f(n)$$

$$a = 9, b = 3, f(n) = n$$

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

Since $f(n) = O(n^{\log_3 9-\epsilon})$, where $\epsilon = 1$

We can apply case 1 of the master theorem and the solution is $T(n) = \theta(n^2)$.

EXERCISES

Practice Problems 1

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. What is the time complexity of the recurrence relation $T(n) = 2T\left(\frac{n}{2}\right) + n^2$?

(A) $\theta(n^2)$ (B) $\theta(n)$
(C) $\theta(n^3)$ (D) $\theta(n \log n)$

2. What is the time complexity of the recurrence relation by using masters theorem $T(n) = 2T\left(\frac{n}{2}\right) + n$?

(A) $\theta(n^2)$ (B) $\theta(n)$
(C) $\theta(n^3)$ (D) $\theta(n \log n)$

3. What is the time complexity of the recurrence relation by using master theorem, $T(n) = 2T\left(\frac{n}{4}\right) + n^{0.51}$

(A) $\theta(n^2)$ (B) $\theta(n)$
(C) $\theta(n^3)$ (D) $(n^{0.51})$

4. What is the time complexity of the recurrence relation using master theorem, $T(n) = 7T\left(\frac{n}{3}\right) + n^2$?

(A) $\theta(n^2)$ (B) $\theta(n)$
(C) $\theta(n^3)$ (D) $(\log n)$

5. Time complexity of $f(x) = 4x^2 - 5x + 3$ is

(A) $O(x)$ (B) $O(x^2)$
(B) $O(x^{3/2})$ (D) $O(x^{0.5})$

6. Time complexity of $f(x) = (x^2 + 5 \log_2 x)/(2x + 1)$ is

(A) $O(x)$ (B) $O(x^2)$
(C) $O(x^{3/2})$ (D) $O(x^{0.5})$

7. For the recurrence relation, $T(n) = 2T\left(\left\lfloor \sqrt{n} \right\rfloor\right) + \lg n$, which is tightest upper bound?

(A) $T(n) = O(n^2)$ (B) $T(n) = O(n^3)$
(C) $T(n) = O(\log n)$ (D) $T(n) = O(\lg n \lg \lg n)$

8. Consider $T(n) = 9T(n/3) + n$, which of the following is TRUE?

(A) $T(n) = \theta(n^2)$ (B) $T(n) = \theta(n^3)$
(C) $T(n) = \Omega(n^3)$ (D) $T(n) = O(n)$

9. If $f(n)$ is $100 * n$ seconds and $g(n)$ is $0.5 * n$ seconds then

(A) $f(n) = g(n)$ (B) $f(n) = \Omega(g(n))$
(C) $f(n) = w(g(n))$ (D) None of these

10. Solve the recurrence relation using master method: $T(n) = 4T(n/2) + n^2$

(A) $\theta(n \log n)$ (B) $\theta(n^2 \log n)$
(C) $\theta(n^2)$ (D) $\theta(n^3)$

11. Arrange the following functions according to their order of growth (from low to high):

(A) $\sqrt[3]{n}, 0.001n^4 + 3n^3 + 1, 3^n, 2^{2n}$

(B) $3^n, 2^{2n}, \sqrt[3]{n}, 0.001n^4 + 3n^3 + 1$

(C) $2^{2n}, \sqrt[3]{n}, 3^n, 0.001n^4 + 3n^3 + 1$

(D) $\sqrt[3]{n}, 2^{2n}, 3^n, 0.001n^4 + 3n^3 + 1$

12. The following algorithm checks whether all the elements in a given array are distinct:

Input: array $A[0 \ldots n-1]$

Output: true (or) false

For $i \leftarrow 0$ to $n-2$ do

For $j \leftarrow i+1$ to $n-1$ do

if $A[i] = A[j]$ return false

return true

The time complexity in worst case is

(A) $\theta(n^2)$ (B) $\theta(n)$
(C) $\theta(\log n)$ (D) $\theta(n \log n)$

13. The order of growth for the following recurrence relation is $T(n) = 4T(n/2) + n^3$, $T(1) = 1$

(A) $\theta(n)$ (B) $\theta(n^3)$
(C) $\theta(n^2)$ (D) $\theta(\log n)$

14. Time complexity of $T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{3}$ is

(A) $\theta(\sqrt{n} \log n)$ (B) $\theta(\sqrt{n} \log \sqrt{n})$
(C) $\theta(\sqrt{n})$ (D) $\theta(n^2)$

15. Consider the following three claims

(I) $(n+k)^m = \theta(n^m)$, where k and m are constants
(II) $2^{n+1} = O(2^n)$
(III) $2^{2n+1} = O(2^n)$

Which one of the following is correct?

(A) I and III (B) I and II
(C) II and III (D) I, II and III

Practice Problems 2

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. Arrange the order of growth in ascending order:

(A) $O(1) > O(\log n) > O(n) > O(n^2)$
(B) $O(n) > O(1) > O(\log n) > O(n^2)$
(C) $O(\log n) > O(n) > O(1) > O(n^2)$
(D) $O(n^2) > O(n) > O(\log n) > O(1)$

2. $\sqrt{n} = \Omega(\log n)$ means

(A) To the least $\sqrt{n}$ is log n
(B) $\sqrt{n}$ is log n always
(C) $\sqrt{n}$ is at most log n
(D) None of these

3. Which of the following is correct?

(i) $\theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
(ii) $\theta(g(n)) = O(g(n)) \cup \Omega(g(n))$

(A) (i) is true (ii) is false (B) Both are true
(C) Both are false (D) (ii) is true (i) is false

4. $2n^2 = x\ (n^3)$, x is which notation?
(A) Big-oh (B) Small-oh
(C) Ω – notation (D) θ – notation

5. Master method applies to recurrence of the form $T(n) = a\ T(n/b) + f(n)$ where
(A) $a \geq 1, b > 1$ (B) $a = 1, b > 1$
(C) $a > 1, b = 1$ (D) $a \geq 1, b \geq 1$

6. What is the time complexity of the recurrence relation using master method?

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

(A) $\theta(n^2)$ (B) $\theta(n)$
(C) $\theta(\log n)$ (D) $\theta(n \log n)$

7. Use the informal definitions of O, θ Ω to determine these assertions which of the following assertions are true.
(A) $n(n+1)/2 \in O(n^3)$ (B) $n(n+1)/2 \in O(n^2)$
(C) $n(n+1)/2 \in \Omega(n)$ (D) All the above

8. Match the following:

(i)	Big-oh	(A)	$\geq$
(ii)	Small-o	(B)	$\leq$
(iii)	Ω	(C)	=
(iv)	θ	(D)	<
(v)	ω	(E)	>

(A) (i) – D, (ii) – A, (iii) – C, (iv) –B , (v) – E
(B) (i) – B, (ii) – D, (iii) – A, (iv) – C, (v) – E
(C) (i) – C, (ii) – A, (iii) – B, (iv) – E, (v) – D
(D) (i) – A, (ii) – B, (iii) – C, (iv) – D, (v) – E

9. Which one of the following statements is true?
(A) Both time and space efficiencies are measured as functions of the algorithm input size.
(B) Only time efficiencies are measured as a function of the algorithm input size.
(C) Only space efficiencies are measured as a function of the algorithm input size.
(D) Neither space nor time efficiencies are measured as a function of the algorithm input size.

10. Which of the following is true?
(A) Investigation of the average case efficiency is considerably more difficult than investigation of the worst case and best case efficiencies.
(B) Investigation of best case is more complex than average case.
(C) Investigation of worst case is more complex than average case.
(D) None of these

11. Time complexity of $T(n) = T(n/3) + T(2n/3) + O(n)$ is
(A) $O(1)$
(B) $O(n \log n)$
(C) $O(\log n)$
(D) $O(n^2)$

12. Solve the recurrence relation to find $T(n)$: $T(n) = 4(n/2) + n$
(A) $\theta(n^2)$ (B) $\theta(\log_2 n)$
(C) $\theta(n^2 \log_2 n)$ (D) $\theta(n^3)$

13. What is the worst case analysis for the given code?

```
int search (int a[ ], int x, int n)
{
int i;
for (i = 0 ; i < n; i ++)
if (a [i] = = x)
return i;
return -1;
}
```

(A) $O(n)$ (B) $O(n \log n)$
(C) $O(\log n)$ (D) $O(n^2)$

14. Find the time complexity of the given code.

```
void f (int n)
{
if (n > 0)
{
f (n/2);
f (n/2);
}
}
```

(A) $\theta(n^2)$
(B) $\theta(n)$
(C) $\theta(n \log n)$
(D) $\theta(2^n)$

15. The running time of the following algorithm procedure
$A(n)$
if $n \leq 2$
return (1)
else
return $(A(\sqrt{n}))$
is described by

(A) $O(\sqrt{n} \log n)$
(B) $O(\log n)$
(C) $O(\log \log n)$
(D) $O(n)$

Previous Years' Questions

1. The median of n elements can be found in $O(n)$ time. Which one of the following is correct about the complexity of quick sort, in which median is selected as pivot? **[2006]**

(A) $\theta(n)$ (B) $\theta(n \log n)$
(C) $\theta(n^2)$ (D) $\theta(n^3)$

2. Given two arrays of numbers $a_1 \ldots a_n$ and $b_1 \ldots b_n$ where each number is 0 or 1, the fastest algorithm to find the largest span (i, j) such that $a_i + a_i + 1 + \cdots + a_j = b_i + b_i + 1 + \cdots + b_j$, or report that there is not such span, **[2006]**

(A) Takes $O(3^n)$ and $\Omega(2^n)$ time if hashing is permitted
(B) Takes $O(n^3)$ and $\Omega(n^{2.5})$ time in the key comparison model
(C) Takes $\Theta(n)$ time and space
(D) Takes $O(\sqrt{n})$ time only if the sum of the $2n$ elements is an even number

3. Consider the following segment of *C*-code:

```
int j, n;
    j = 1;
    while (j <=n)
j = j*2;
```

The number of comparisons made in the execution of the loop for any $n > 0$ is: **[2007]**

(A) $\lceil \log_2 n \rceil + 1$ (B) n
(C) $\lceil \log_2 + n \rceil$ (D) $\lfloor \log_2 n \rfloor + 1$

4. In the following *C* function, let $n \geq m$.

```
int gcd(n,m)
{
if (n%m = =0) return m;
     n = n%m;
return gcd(m,n);
}
```

How many recursive calls are made by this function? **[2007]**

(A) $\Theta(\log_2 n)$ (B) $\Omega(n)$
(C) $\Theta(\log_2 \log_2 n)$ (D) $\Theta(\sqrt{n})$

5. What is the time *complexity* of the following recursive function:

```
int DoSomething (int n) {
if (n <= 2)
    return 1;
  else
return(DoSomething (floor(sqrt(n)))+ n);}
```

[2007]

(A) $\Theta(n^2)$ (B) $\Theta(n \log_2 n)$
(C) $\Theta(\log_2 n)$ (D) $\Theta(\log_2 \log_2 n)$

6. An array of n numbers is given, where n is an even number. The maximum as well as the minimum of these n numbers needs to be determined. Which of the following is TRUE about the number of comparisons needed? **[2007]**

(A) At least $2n - c$ comparisons, for some constant c, are needed.
(B) At most $1.5n - 2$ comparisons are needed.
(C) At least $n \log_2 n$ comparisons are needed.
(D) None of the above.

7. Consider the following *C* code segment:

```
int IsPrime(n)
    {
    int i,n;
    for(i=2; i<= sqrt(n); i ++)
      if(n%i = =0)
       {printf("Not Prime\n"); return 0;}
    return 1;
}
```

Let $T(n)$ denote the number of times the *for* loop is executed by the program on input n. Which of the following is TRUE? **[2007]**

(A) $T(n) = O(\sqrt{n})$ and $T(n) = \Omega(\sqrt{n})$
(B) $T(n) = O(\sqrt{n})$ and $T(n) = \Omega(1)$
(C) $T(n) = O(n)$ and $T(n) = \Omega(\sqrt{n})$
(D) None of the above

8. The most efficient algorithm for finding the number of connected components in an undirected graph on n vertices and m edges has time complexity **[2008]**

(A) $\Theta(n)$ (B) $\Theta(m)$
(C) $\Theta(m + n)$ (D) $\Theta(mn)$

9. Consider the following functions:

$f(n) = 2^n$
$g(n) = n!$
$h(n) = n^{\log n}$

Which of the following statements about the asymptotic behavior of $f(n)$, $g(n)$, and $h(n)$ is true? **[2008]**

(A) $f(n) = O(g(n)); g(n) = O(h(n))$
(B) $f(n) = \Omega(g(n)); g(n) = O(h(n))$
(C) $g(n) = O(f(n)); h(n) = O(f(n))$
(D) $h(n) = O(f(n)); g(n) = \Omega(f(n))$

10. The minimum number of comparisons required to determine if an integer appears more than $n/2$ times in a sorted array of n integers is **[2008]**

(A) $\Theta(n)$ (B) $\Theta(\log n)$
(C) $\Theta(\log * n)$ (D) $\Theta(1)$

11. We have a binary heap on n elements and wish to insert n more elements (not necessarily one after another) into this heap. The total time required for this is **[2008]**

(A) $\Theta(\log n)$ (B) $\Theta(n)$
(C) $\Theta(n \log n)$ (D) $\Theta(n^2)$

12. The running time of an algorithm is represented by the following recurrence relation: **[2009]**

$$T(n) = \begin{cases} n & n \le 3 \\ T\left(\frac{n}{3}\right) + \text{cn otherwise} \end{cases}$$

Which one of the following represents the time complexity of the algorithm?
(A) $\theta(n)$ (B) $\theta(n \log n)$
(C) $\theta(n^2)$ (D) $\theta(n^2 \log n)$

13. Two alternative packages A and B are available for processing a database having 10^k records. Package A requires $0.0001\ n^2$ time units and package B requires $10n \log_{10}$ n time units to process n records. What is the smallest value of k for which package B will be preferred over A? **[2010]**
(A) 12 (B) 10
(C) 6 (D) 5

14. An algorithm to find the length of the longest monotonically increasing sequence of numbers in an array $A[0 : n-1]$ is given below.

Let L denote the length of the longest monotonically increasing sequence starting at index in the array.

Initialize $L_{n-1} = 1$,

For all i such that $0 \le i \le n-2$

$$Li \begin{cases} = 1 + L_{i+1}, & \text{if } A[i] < A[i+1], \\ 1 \text{ otherwise} \end{cases}$$

Finally the length of the longest monotonically increasing sequence is Max $(L_0, L_1, \ldots L_{n-1})$

Which of the following statements is TRUE? **[2011]**
(A) The algorithm uses dynamic programming paradigm.
(B) The algorithm has a linear complexity and uses branch and bound paradigm.
(C) The algorithm has a non-linear polynomial complexity and uses branch and bound paradigm.
(D) The algorithm uses divide and conquer paradigm.

15. Which of the given options provides the increasing order of asymptotic complexity of functions f_1, f_2, f_3 and f_4? **[2011]**

$f_1(n) = 2^n$
$f_2(n) = n^{3/2}$
$f_3(n) = n \log_2 n$
$f_4(n) = n^{\log_2 n}$

(A) f_3, f_2, f_4, f_1 (B) f_3, f_2, f_1, f_4
(C) f_2, f_3, f_1, f_4 (D) f_2, f_3, f_4, f_1

16. Let $W(n)$ and $A(n)$ denote respectively, the worst-case and average-case running time of an algorithm executed on input of size n. Which of the following is **ALWAYS TRUE**? **[2012]**
(A) $A(n) = \Omega(W(n))$ (B) $A(n) = \Theta(W(n))$
(C) $A(n) = O(W(n))$ (D) $A(n) = o(W(n))$

17. The recurrence relation capturing the optimal execution time of the Towers of Hanoi problem with n discs is **[2012]**
(A) $T(n) = 2T(n-2) + 2$
(B) $T(n) = 2T(n-1) + n$
(C) $T(n) = 2T(n/2) + 1$
(D) $T(n) = 2T(n-1) + 1$

18. A list of n strings, each of length n, is sorted into lexicographic order using the merge sort algorithm. The worst-case running time of this computation is **[2012]**
(A) $O(n \log n)$ (B) $O(n^2 \log n)$
(C) $O(n^2 + \log n)$ (D) $O(n^2)$

19. Consider the following function:

```
int unknown (int n) {
    int i, j, k = 0;
    for (i = n/2; i < = n; i++)
            for (j = 2; j < = n; j = j*2)
                    k = k + n/2;
return (k);
}
```

The return value of the function is **[2013]**
(A) $\Theta(n^2)$ (B) $\Theta(n^2 \log n)$
(C) $\Theta(n^3)$ (D) $\Theta(n^3 \log n)$

20. The number of elements that can be sorted in $\Theta(\log n)$ time using heap sort is **[2013]**
(A) $\Theta(1)$
(B) $\Theta(\sqrt{\log n})$
(C) $\Theta\left(\frac{\log n}{\log \log n}\right)$
(D) $\Theta(\log n)$

21. Which one of the following correctly determines the solution of the recurrence relation with $T(1) = 1$

$$T(n) = 2T\left(\frac{n}{2}\right) + \log n \text{?}$$ **[2014]**

(A) $\theta(n)$ (B) $\theta(n \log n)$
(C) $\theta(n^2)$ (D) $\theta(\log n)$

22. An algorithm performs $(\log N)^{1/2}$ find operations, N insert operations, $(\log N)^{1/2}$ delete operations, and $(\log N)^{1/2}$ decrease-key operations on a set of data items with keys drawn from a linearly ordered set. For a delete operation, a pointer is provided to the record that must be deleted For the decrease – key operation, a pointer is provided to the record that has its key decreased Which one of the following data structures is the most suited for the algorithm to use, if the

goal is to achieve the best total asymptotic complexity considering all the operations? **[2015]**

(A) Unsorted array
(B) Min-heap
(C) Sorted array
(D) Sorted doubly linked list

23. Consider the following C function.

```
int fun1(int n) {
   int i, j, k, p, q=0;
   for (i=1; i<n; ++i) {
         p=0;
         for (j=n; j>1; j=j/2)
               ++p;
         for (k=1; k<p; k=k*2)
               ++q;
         }
         return q;
   }
```

Which one of the following most closely approximates the return value of the function fun1? **[2015]**

(A) n^3 (B) $n(\log n)^2$
(C) $n \log n$ (D) $n \log(\log n)$

24. An unordered list contains n distinct elements. The number of comparisons to find an element in this list that is neither maximum nor minimum is **[2015]**

(A) $\theta(n \log n)$ (B) $\theta(n)$
(C) $\theta(\log n)$ (D) $\theta(1)$

25. Consider a complete binary tree where the left and the right subtrees of the root are max-heaps. The lower bound for the number of operations to convert the tree to a heap is **[2015]**

(A) $\Omega(\log n)$ (B) $\Omega(n)$
(C) $\Omega(n \log n)$ (D) $\Omega(n^2)$

26. Consider the equality $\sum_{i=0}^{n} i^3 =$ and the following choices for X

1. $\theta(n^4)$
2. $\theta(n^5)$
3. $O(n^5)$
4. $\Omega(n^3)$

The equality above remains correct if X is replaced by **[2015]**

(A) Only 1
(B) Only 2
(C) 1 or 3 or 4 but not 2
(D) 2 or 3 or 4 but not 1

27. Consider the following array of elements

<89, 19, 50, 17, 12, 15, 2, 5, 7, 11, 6, 9, 100>

The minimum number of interchanges needed to convert it into a max-heap is **[2015]**

(A) 4 (B) 5
(C) 2 (D) 3

28. Let $f(n) = n$ and $g(n) = n^{(1 + \sin n)}$, where n is a positive integer. Which of the following statements is/are correct? **[2015]**

I. $f(n) = O(g(n))$
II. $f(n) = \Omega(g(n))$

(A) Only I (B) Only II
(C) Both I and II (D) Neither I nor II

29. A queue is implemented using an array such that ENQUEUE and DEQUEUE operations are performed efficiently. Which one of the following statements is **CORRECT** (n refers to the number of items in the queue)? **[2016]**

(A) Both operations can be performed in O(1) time.
(B) At most one operation can be performed in O(1) time but the worst case time for the other operation will be Ω (n).
(C) The worst case time complexity for both operations will be Ω (n).
(D) Worst case time complexity for both operations will be Ω (logn).

30. Consider a carry look ahead adder for adding two n - bit integers, built using gates of fan - in at most two. The time to perform addition using this adder is **[2016]**

(A) $\Theta(1)$ (B) $\Theta(\log(n))$
(C) $\Theta\left(\sqrt{n}\right)$ (D) $\Theta(n)$

31. N items are stored in a sorted doubly linked list. For a *delete* operation, a pointer is provided to the record to be deleted. For a *decrease - key* operation, a pointer is provided to the record on which the operation is to be performed.

An algorithm performs the following operations on the list in this order: Θ (N) *delete*, O (logN) *insert*, O (log N) *find*, and Θ (N) *decrease - key*. What is the time complexity of all these operations put together? **[2016]**

(A) O ($\log^2$N) (B) O(N)
(C) O(N^2) (D) Θ (N^2logN)

32. In an adjacency list representation of an undirected simple graph G = (V,E), each edge *(u,v)* has two adjacency list entries:*[v]* in the adjacency list of *u*, and *[u]* in the adjacency list of v. These are called twins of each other. A twin pointer is a pointer from an adjacency list entry to its twin. If $|E| = m$ and $|V| = n$, and the memory size is not a constraint, what is the time complexity of the most efficient algorithm to set the twin pointer in each entry in each adjacency list? **[2016]**

(A) $\Theta(n^2)$ (B) $\Theta(n + m)$
(C) $\Theta(m^2)$ (D) $\Theta(n^4)$

33. Consider the following functions from positive integers to real numbers:

$$10,\ \sqrt{n}\ ,\ n,\ \log_2 n,\ \frac{100}{n}.$$

The CORRECT arrangement of the above functions in increasing order of asymptotic complexity is: **[2017]**

(A) $\log_2 n, \frac{100}{n}, 10, \sqrt{n}, n$

(B) $\frac{100}{n}, 10, \log_2 n, \sqrt{n}, n$

(C) $10, \frac{100}{n}, \sqrt{n}, \log_2 n, n$

(D) $\frac{100}{n}, \log_2 n, 10, \sqrt{n}, n$

34. Consider the recurrence function

$$T(n) = \begin{cases} 2T\left(\sqrt{n}\right)+1 & n > 2 \\ 2, & 0 < n \le 2 \end{cases}$$

Then $T(n)$ in terms of Θ notation is **[2017]**

(A) Θ (log log n) (B) Θ (log n)

(C) $\Theta\left(\sqrt{n}\right)$ (D) Θ (n)

35. Consider the following C function.

```
int fun (int n)  {
      int i, j;
      for(i = 1; i <= n; i++)  {
            for (j = l; j < n; j += i) {
                  printf{" %d %d",i, j);
                  }
            }
```

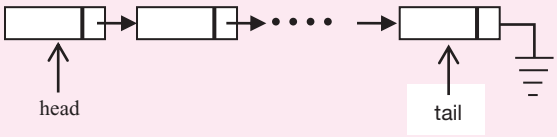
```
}
```

Time complexity of fun in terms of Θ notation is **[2017]**

(A) $\Theta\left(n\sqrt{n}\right)$ (B) $\Theta(n^2)$

(C) $\Theta(n \log n)$ (D) $\Theta(n^2 \log n)$

36. A queue is implemented using a non-circular singly linked list. The queue has a head pointer and a tail pointer, as shown in the figure. Let n denote the number of nodes in the queue. Let enqueue be implemented by inserting a new node at the head, and dequeue be implemented by deletion of a node from the tail.



Which one of the following is the time complexity of the most time-efficient implementation of enqueue and dequeue, respectively, for this data structure? **[2018]**

(A) θ(1), θ(1) (B) θ(1), θ(n)

(C) θ(n), θ(1) (D) θ(n), θ(n)

37. Consider the following C code. Assume that unsigned long int type length is 64 bits.

```
unsigned long int fun (unsigned long int n) {
unsigned long int i, j = 0, sum = 0;
for (i = n; i > 1. i = i/2) j++;
for (; j > 1; j = j/2) sum++;
return (sum);
}
```

The value returned when we call fun with the input 2^{40} is: **[2018]**

(A) 4 (B) 5

(C) 6 (D) 40

Answer Keys

Exercises

Practice Problems 1

1. A	**2.** D	**3.** D	**4.** A	**5.** B	**6.** A	**7.** D	**8.** A	**9.** A	**10.** B
11. A	**12.** A	**13.** B	**14.** A	**15.** B					

Practice Problems 2

1. A	**2.** A	**3.** A	**4.** B	**5.** A	**6.** A	**7.** D	**8.** B	**9.** A	**10.** A
11. B	**12.** A	**13.** A	**14.** B	**15.** C					

Previous Years' Questions

1. B	**2.** C	**3.** A	**4.** C	**5.**	**6.** B	**7.** B	**8.** C	**9.** D	**10.** A
11. B	**12.** A	**13.** C	**14.** A	**15.** A	**16.** C	**17.** D	**18.** B	**19.** B	**20.** C
21. A	**22.** A	**23.** D	**24.** D	**25.** A	**26.** C	**27.** D	**28.** D	**29.** A	**30.** B
31. C	**32.** B	**33.** B	**34.** B	**35.** C	**36.** B	**37.** B			

Chapter 2

Sorting Algorithms

LEARNING OBJECTIVES

- *Sorting algorithms*
- *Merge sort*
- *Bubble sort*
- *Insertion sort*
- *Selection sort*
- *Selection sort algorithm*
- *Binary search trees*
- *Heap sort*
- *Sorting–performing delete max operations*
- *Max-heap property*
- *Min-heap property*
- *Priority queues*

Sorting Algorithms

Purpose of sorting

Sorting is a technique which reduces problem complexity and search complexity.

- Insertion sort takes $\theta(n^2)$ time in the worst case. It is a fast inplace sorting algorithm for small input sizes.
- Merge sort has a better asymptotic running time $\theta(n \log n)$, but it does not operate in inplace.
- Heap sort, sorts 'n' numbers inplace in $\theta(n \log n)$ time, it uses a data structure called heap, with which we can also implement a priority queue.
- Quick sort also sorts 'n' numbers in place, but its worst – case running time is $\theta(n^2)$. Its average case is $\theta(n \log n)$. The constant factor in quick sort's running time is small, This algorithm performs better for large input arrays.
- Insertion sort, merge sort, heap sort, and quick sort are all comparison based sorts; they determine the sorted order of an inputarray by comparing elements.
- We can beat the lower bound of Ω ($n \log n$) if we can gather information about the sorted order of the input by means other than comparing elements.
- The counting sort algorithm, assumes that the input numbers are in the set $\{1, 2, \ldots. k\}$. By using array indexing as a tool for determining relative order, counting sort can sort n numbers in $\theta(k + n)$ time. Thus counting sort runs in time that is linear in size of the input array.
- Radix sort can be used to extend the range of counting sort. If there are 'n' integers to sort, each integer has 'd' digits, and each digit is in the set $\{1, 2,\ldots k\}$, then radix sort can sort the numbers in $\theta(d\,(n + k))$ time. Where 'd' is constant. Radix sort runs in linear time.
- Bucket sort, requires knowledge of the probabilistic distribution of numbers in the input array.

Merge Sort

Suppose that our division of the problem yields 'a' sub problems, each of which is $\left(\frac{1}{b}\right)$th size of the original problem. For merge sort, both a and b are 2, but sometimes $a \neq b$. If we take $D(n)$ time to divide the problem into sub problems and $C(n)$ time to combine the solutions of the sub problems into the solution to the original problem. The recurrence relation for merge sort is

$$T(n) = \begin{cases} \theta(1) \text{ if } n \leq c, \\ aT(n/b) + D(n) + C(n) \text{ otherwise} \end{cases}$$

Running time is broken down as follows:

Divide: This step computes the middle of the sub array, which takes constant time $\theta(1)$.

Conquer: We solve 2 sub problems of size (n/2) each recursively which takes $2T(n/2)$ time.

Combine: Merge sort procedure on an n-element sub array takes time $\theta(n)$.

- Worst case running time $T(n)$ of merge sort

$$T(n) = \begin{cases} 0(1) \text{ if } & n \le 1 \\ aT(n/2) + \theta(n) \text{ if } & n > 1 \end{cases}$$

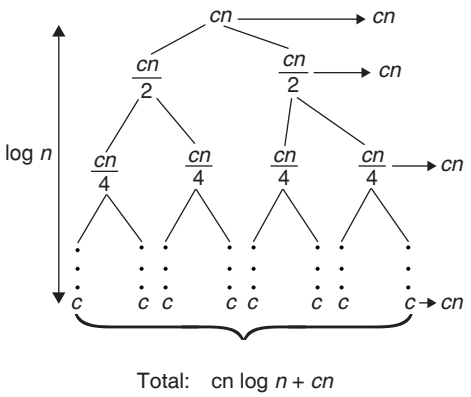


Figure 1 Recurrence tree

The top level has total cost '*cn*', the next level has total cost $c(n/2) + c(n/2) = cn$ and the next level has total cost $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$ and so on. The ith level has total cost $2^i\ c\ (n/2^i) = cn$. At the bottom level, there are 'n' nodes, each contributing a cost of c, for a total cost of 'cn'. The total number of levels of the 'recursion tree' is $\log n + 1$.

There are $\log n + 1$ levels, each costing cn, for a total cost of $cn\ (\log n + 1) = cn \log n + cn$ ignoring the low–order term and the constant c, gives the desired result of $\theta(n \log n)$.

Bubble Sort

Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements 'bubble' to the top of the list.

Example: Take the array of numbers '5 1 4 2 8'and sort the array from lowest number to greatest number using bubble sort algorithm. In each step, elements underlined are being compared.

First pass:

(5 1 4 2 8) → (1 5 4 2 8), here algorithm compares the first 2 elements and swaps them
(1 5 4 2 8) → (1 4 5 2 8), swap (5 > 4)
(1 4 5 2 8) → (1 4 2 5 8), swap (5 > 2)
(1 4 2 5 8) → (1 4 2 5 8), since these elements are already in order, algorithm does not swap them.

Second pass:

(1 4 2 5 8) → (1 4 2 5 8)
(1 4 2 5 8) → (1 2 4 5 8), swap since (4 > 2)
(1 2 4 5 8) → (1 2 4 5 8)
(1 2 4 5 8) → (1 2 4 5 8)

The array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third pass:

(1 2 4 5 8) → (1 2 4 5 8)
(1 2 4 5 8) → (1 2 4 5 8)
(1 2 4 5 8) → (1 2 4 5 8)
(1 2 4 5 8) → (1 2 4 5 8)

Finally the array is sorted, and the algorithm can terminate.

Algorithm

```
void bubblesort (int a [ ], int n)
{
  int i, j, temp;
  for (i=0; i < n-1; i++)
  {
    for (j=0; j < n - 1 - i; j++)
    if (a [j] > a [j + 1])
    {
      temp = a [j + 1];
      a [j + 1] = a [j];
      a [j] = temp;
    }
  }
}
```

Insertion Sort

Insertion sort is a comparison sort in which the sorted array is built one entry at a time. It is much less efficient on large lists than more advanced algorithms such a quick sort, heap sort, (or) merge sort. Insertion sort provides several advantages.

- Efficient for small data sets.
- Adaptive, i.e., efficient for data set that are already substantially sorted. The complexity is $O(n + d)$, where d is the number of inversions.
- More efficient in practice than most other simple quadratic, i.e., $O(n^2)$ algorithms such as selection sort (or) bubble sort, the best case is $O(n)$.
- Stable, i.e., does not change the relative order of elements with equal keys.
- In-place i.e., only requires a constant amount $O(1)$ of additional memory space.
- Online, i.e., can sort a list as it receives it.

Algorithm

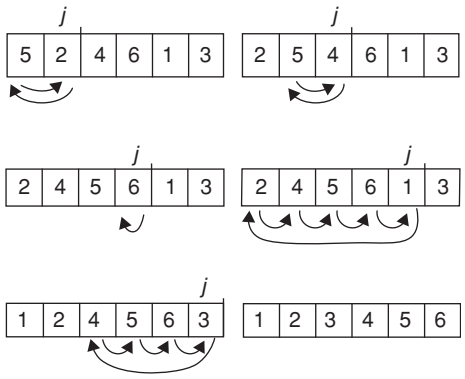
```
Insertion sort (A)
For (j ← 2) to length [A]
Do key ← A [j]
i ←j - 1;
While i > 0 and A [i] > key
{
Do A [i + 1] ← A [i]
i ← i - 1
}
A [i + 1] ← key
```

Every repetition of insertion sort removes an element from the input data, inserting it into the correct position in the already sorted list, until no input element remains. Sorting is typically done in–place. The resulting array after K iterations has the property where the first $k + 1$ entries are sorted. In each iteration the first remaining entry of the input is removed, inserted into the result at the correct position, with each element greater than X copied to the right as it is compared against. X.

Performance

- The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., $\theta(n)$).
- The worst case input is an array sorted in reverse order. In this case every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. For this case insertion sort has a quadratic running time ($O(n^2)$).
- The average case is also quadratic, which makes insertion sort impractical for sorting large arrays, however, insertion sort is one of the fastest algorithms for sorting very small arrays even faster than quick sort.

Example: Following figure shows the operation of insertion sort on the array $A = (5, 2, 4, 6, 1, 3)$. Each part shows what happens for a particular iteration with the value of j indicated. j indexes the 'Current card' being inserted.



Read the figure row by row. Elements to the left of $A[j]$ that are greater than $A[j]$ move one position to the right and $A[j]$ moves into the evacuated position.

Selection Sort

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has $O(n^2)$ complexity, making it inefficient on large lists.

The algorithm works as follows:

1. Find the minimum value in the list.
2. Swap it with the value in the first position.
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time).

Analysis

Selection sort is not difficult to analyze compared to other sorting algorithms, since none of the loops depend on the data in the array selecting the lowest element requires scanning all n elements (this takes $n - 1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining $n - 1$ elements and so on, for $(n - 1) + (n - 2) + \cdots + 2 + 1 = n(n - 1)/2 \in \theta(n^2)$ comparisons.

Each of these scans requires one swap for $n - 1$ elements (the final element is already in place).

Selection sort Algorithm

First, the minimum value in the list is found. Then, the first element (with an index of 0) is swapped with this value. Lastly, the steps mentioned are repeated for rest of the array (starting at the 2nd position).

Example 1: Here's a step by step example to illustrate the selection sort algorithm using numbers.

Original array: 6 3 5 4 9 2 7
1st pass → 2 3 5 4 9 6 7 (2 and 6 were swapped)
2nd pass → 2 3 5 4 9 6 7 (no swap)
3rd pass → 2 3 4 5 9 6 7 (4 and 5 were swapped)
4th pass → 2 3 4 5 6 9 7 (6 and 9 were swapped)
5th pass → 2 3 4 5 6 7 9 (7 and 9 were swapped)
6th pass → 2 3 4 5 6 7 9 (no swap)

Note: There are 7 keys in the list and thus 6 passes were required. However, only 4 swaps took place.

Example 2: Original array: LU, KU, HU, LO, SU, PU
1st pass → HU, KU, LU, LO, SU, PU
2nd pass → HU, KU, LU, LO, SU, PU
3rd pass → HU, KU, LO, LU, SU, PU
4th pass → HU, KU, LO, LU, SU, PU
5th pass → HU, KU, LO, LU, PU, SU

Note: There were 6 elements in the list and thus 5 passes were required. However, only 3 swaps took place.

Binary Search Trees

Search trees are data structures that support many dynamic, set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE. A search tree can be used as a dictionary and as a priority Queue. Operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with 'n' nodes, basic operations run in $\theta(\log n)$ worst-case time. If the tree is a linear chain of 'n' nodes, the basic operations take $\theta(n)$ worst-case time.

A binary search tree is organized, in a binary tree such a tree can be represented by a linked data structure in which each node is an object. In addition to key field, each node contains fields left, right and P that point to the nodes corresponding to its left child, its right child, and its parent,

respectively. If the child (or) parent is missing, the appropriate field contains the value NIL. The root node is the only node in the tree whose parent field is NIL.

Binary search tree property

The keys in a binary search tree are always stored in such a way as to satisfy the binary search tree property.

Let '*a*' be a node in a binary search tree. If '*b*' is a node in the left sub tree of '*a*', key [*b*] ≤ key [*a*]

If '*b*' is a node in the right sub tree of '*a*' then key [*a*] ≤ key [*b*].

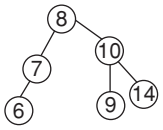


Figure 2 Binary search tree.

The binary search tree property allows us to print out all keys in a binary search tree in sorted order by a simple recursive algorithm called an inorder tree.

Algorithm

INORDER-TREE-WALK (root [*T*])

INORDER-TREE-WALK (a)

1. If $a \neq$ NIL
2. Then INORDER-TREE-WALK (left [*a*])
3. Print key [*a*]
4. INORDER-TREE-WALK (right [*a*])

It takes $\theta(n)$ time to walk an n-node binary search tree, since after the initial call, the procedure is called recursively twice for each node in the tree.

Let $T(n)$ denote the time taken by IN-ORDER-TREE-WALK, when it is called on the root of an n-node subtree.

INORDER-TREE-WALK takes a small, constant amount of time on an empty sub-tree (for the test $x \neq$ NIL).

So $T(1) = C$ for some positive constant C.

For $n > 0$, suppose that INORDER-TREE-WALK is called on a node 'a' whose left subtree has k nodes and whose right subtree has $n - k - 1$ nodes.

The time to perform in order traversal is $T(n) = T(k) + T(n - k - 1) + d$.

For some positive constant 'd' that reflects the time to execute in-order (a), exclusive of the time spent in recursive calls $T(n) = (c + d)\, n + c$.

For $n = 0$, we have $(c + d)\, 0 + c = T(0)$,

For $n > 0$,

$$T(n) = T(k) + T(n - k - 1) + d$$
$$= ((c + d)(k + c) + ((c + d)\,(n - k - 1) + c) + d$$
$$= (c + d)\, n + c - (c + d) + c + d = (c + d)n + c$$

Heap Sort

Heap sort begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the partially sorted array. After removing the largest item, it reconstructs heap, removes the largest remaining item, and places, it in the next open position from the end of the partially sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays one to hold the heap and the other to hold the sorted elements.

- Heap sort inserts the input list elements into a binary heap data structure. The largest value (in a max-heap) or the smallest value (in a min-heap) is extracted until none remain, the value having been extracted in sorted order.

Example: Given an array of 6 elements: 15, 19, 10, 7, 17, 16, sort them in ascending order using heap sort.

Steps:

1. Consider the values of the elements as priorities and build the heap tree.
2. Start delete Max operations, storing each deleted element at the end of the heap array.

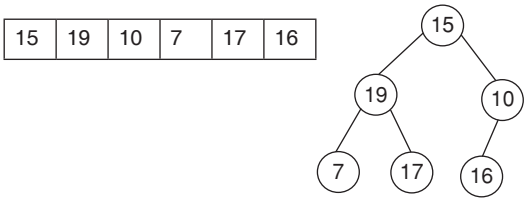
If we want the elements to be sorted in ascending order, we need to build the heap tree in descending order-the greatest element will have the highest priority.

1. Note that we use only array, treating its parts differently,
2. When building the heap-tree, part of the array will be considered as the heap, and the rest part-the original array.
3. When sorting, part of the array will be the heap and the rest part-the sorted array.

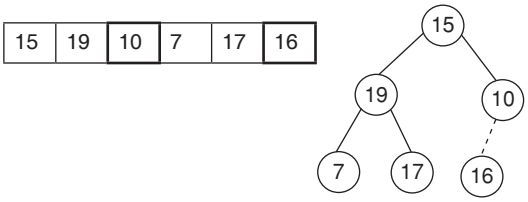
Here is the array: 15, 19, 10, 7, 17, 6.

Building the Heap Tree

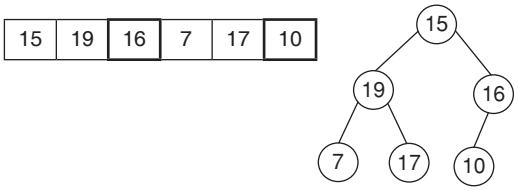
The array represented as a tree, which is complete but not ordered.



Start with the right most node at height 1 – the node at position 3 = size/2. It has one greater child and has to be percolated down.

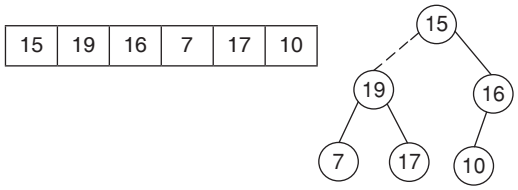


After processing array [3] the situation is:

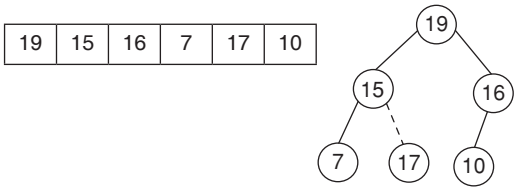


Next comes array [2]. Its children are smaller, so no percolation is needed.

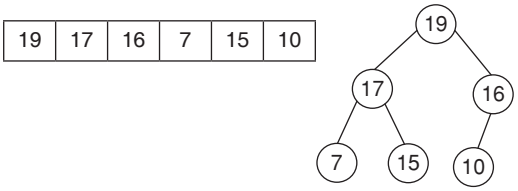
The last node to be processed is array[1]. Its left child is the greater of the children. The item at array [1] has to be percolated down to the left, swapped with array [2].



As a result:



The children of array [2] are greater and item 15 has to be moved down further, swapped with array [5].

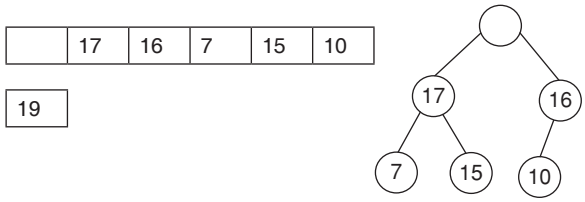


Now the tree is ordered, and the binary heap is built.

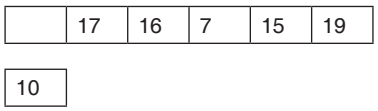
Sorting-performing Delete Max Operations

Delete the top element

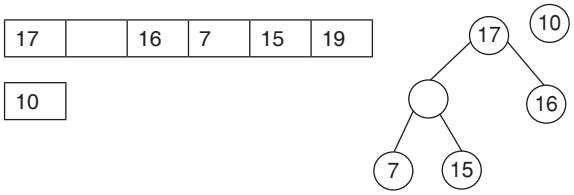
Store 19 in a temporary place, a hole is created at the top.



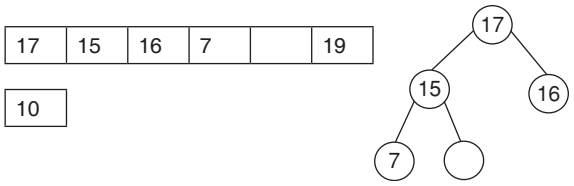
Swap 19 with the last element of the heap. As 10 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array



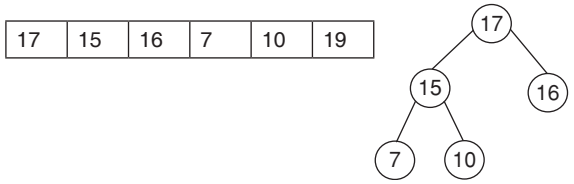
Percolate down the hole



Percolate once more (10 is less than 15, so it cannot be inserted in the previous hole)



Now 10 can be inserted in the hole



Repeat the step *B* till the array is sorted.

Heap sort analysis

Heap sort uses a data structure called (binary) heap binary, heap is viewed as a complete binary tree. An Array A that represents a heap is an object with 2 attributes: length [*A*], which is the number of elements in the array and heap size [*A*], the number of elements in the heap stored within array *A*.

No element past *A* [heap size [*A*]], where heap size [*A*] ≤ length [*A*], is an element of the heap.

There are 2 kinds of binary heaps:

1. Max-heaps
2. Min-heaps

In both kinds the values in the nodes satisfy a heap-property.

Max-heap property $A[\text{PARENT (i)}] \geq A[i]$

The value of a node is almost the value of its parent. Thus the largest element in a max-heap is stored at the root, and the sub tree rooted at a node contains values no larger than that contained at the node itself.

Min-heap property For every mode '*i*' other than the root [PARENT (i)] ≤ *A*[*i*]. The smallest element in a min-heap is at the root.

Max-heaps are used in heap sort algorithm.

Min-heaps are commonly used in priority queues.

Basic operations on heaps run in time almost proportional to the height of the tree and thus take $O(\log n)$ time

- MAX-HEAPIFY procedure, runs in $O(\log n)$ time.
- BUILD-MAX-HEAP procedure, runs in linear time.
- HEAP SORT procedure, runs in $O(n \log n)$ time, sorts an array in place.
- MAX-HEAP-INSERT
  HEAP- EXTRACT-MAX
  HEAP-INCREASE-KEY
  HEAP-MAXIMUM

  All these procedures, run in $O(\log n)$ time, allow the heap data structure to be used as a priority queue.
- Each call to MAX-HEAPIFY costs $O(\log n)$ time, and there are $O(n)$ such calls. Thus, the running time is $O(n \log n)$
- The HEAPSORT procedure takes time $O(n \log n)$, since the call to BUILD-MAX-HEAP takes time $O(n)$ and each of the $(n-1)$ calls to MAX-HEAPIFY takes time $O(\log n)$.

Priority Queues

The most popular application of a heap is its use as an efficient priority queue.

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A max-priority queue supports the following operations:

INSERT: INSERT (s, x) inserts the element x into the set S. This operation can be written as $S \leftarrow S \cup \{x\}$.

MAXIMUM: MAXIMUM (S) returns the element of S with the largest key

EXTRACT-MAX: EXTRACT-MAX(S) removes and returns the element of S with the largest key.

INCREASE-KEY: INCREASE-KEY(s, x, k) increases the value of element x's key to the new value k, which is assumed to be atleast as large as x's current key value.

One application of max–priority queue is to schedule jobs on a shared computer.

EXERCISES

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. Solve the recurrence relation $T(n) = 2T(n/2) + k.n$ where k is constant then $T(n)$ is
(A) $O(\log n)$ (B) $O(n \log n)$
(C) $O(n)$ (D) $O(n^2)$

2. What is the time complexity of the given code?

```
Void f(int n)
{
if (n > 0)
f (n/2);
}
```

(A) $\theta(\log n)$ (B) $\theta(n \log n)$
(C) $\theta(n^2)$ (D) $\theta(n)$

3. The running time of an algorithm is represented by the following recurrence relation;

$$T(n) = \begin{cases} n & n \le 3 \\ T\left\lceil \frac{n}{3} \right\rceil + cn & \text{otherwise} \end{cases}$$

What is the time complexity of the algorithm?
(A) $\theta(n)$ (B) $\theta(n \log n)$
(C) $\theta(n^2)$ (D) $\theta(n^2 \log n)$

Common data for questions 4 and 5:

4. The following pseudo code does which sorting?

*x*sort $[A, n]$
for $j \leftarrow 2$ to n
do key $\leftarrow A\ [\ i\ ]$
$i \leftarrow j - 1$
While $i > 0$ and $A\ [\ i\ ] >$ key
do $A\ [i + i] \leftarrow A\ [\ i\ ]$
$i \leftarrow i - 1$
$A\ [i + 1] \leftarrow$ key
(A) Selection sort (B) Insertion sort
(C) Quick sort (D) Merge sort

5. What is the order of elements after 2 iterations of the above-mentioned sort on given elements?

8	2	4	9	3	6

(A)

2	4	9	8	3	6

(B)

2	4	8	9	3	6

(C)

2	4	6	3	8	9

(D)

2	4	6	3	8	9

Common data for questions 6 and 7:

6. The following pseudo code does which sort?
1. If $n = 1$ done
2. Recursively sort
   $A\ [1 \ldots [n/2]\ ]$and
   $A\ [[n/2] + 1 \ldots n]$
3. Combine 2 ordered lists
(A) Insertion sort (B) Selection sort
(C) Merge sort (D) Quick sort

7. What is the complexity of the above pseudo code?
(A) $\theta(\log n)$ (B) $\theta(n^2)$
(C) $\theta(n \log n)$ (D) $\theta(2^n)$

8. Apply Quick sort on a given sequence 6 10 13 5 8 3 2 11. What is the sequence after first phase, pivot is first element?
(A) 5 3 2 6 10 8 13 11
(B) 5 2 3 6 8 13 10 11
(C) 6 5 13 10 8 3 2 11
(D) 6 5 3 2 8 13 10 11

9. Selection sort is applied on a given sequence:
89, 45, 68, 90, 29, 34, 17. What is the sequence after 2 iterations?
(A) 17, 29, 68, 90, 45, 34, 89
(B) 17, 45, 68, 90, 29, 34, 89
(C) 17, 68, 45, 90, 34, 29, 89
(D) 17, 29, 68, 90, 34, 45, 89

10. Suppose there are log n sorted lists of $\left\lfloor \frac{n}{\log n} \right\rfloor$ elements each. The time complexity of producing sorted lists of all these elements is: (hint: use a heap data structure)
(A) $\theta(n \log \log n)$ (B) $\theta(n \log n)$
(C) $\Omega(n \log n)$ (D) $\Omega(n^{3/2})$

11. If Divide and conquer methodology is applied on powering a Number X^n. Which one the following is correct?
(A) $X^n = X^{n/2} \cdot X^{n/2}$
(B) $X^n = X^{\frac{n-1}{2}} \cdot X^{\frac{n-1}{2}} \cdot X$
(C) $X^n = X^{\frac{n+1}{2}} \cdot X^{\frac{n}{2}}$
(D) Both (A) and (B)

12. The usual $\theta(n^2)$ implementation of insertion sort to sort an array uses linear search to identify the position, where an element is to be inserted into the already sorted part of the array. If binary search is used instead of linear search to identify the position, the worst case running time would be.
(A) $\theta(n \log n)$
(B) $\theta(n^2)$
(C) $\theta(n(\log n)^2)$
(D) $\theta(n)$

13. Consider the process of inserting an element into a max heap where the max heap is represented by an array, suppose we perform a binary search on the path from the new leaf to the root to find the position for the newly inserted element, the number of comparisons performed is:
(A) $\theta(\log n)$ (B) $\theta(\log \log n)$
(C) $\theta(n)$ (D) $\theta(n \log n)$

14. Consider the following algorithm for searching a given number 'X' in an unsorted array $A[1 \cdots n]$ having 'n' distinct values:
(1) Choose an 'i' uniformly at random from $1 \cdots n$
(2) If $A[i] = x$
Then stop
else
goto(1);
Assuming that X is present in A, what is the expected number of comparisons made by the algorithm before it terminates.
(A) n (B) $n - 1$
(C) $2n$ (D) $n/2$

15. The recurrence equation for the number of additions $A(n)$ made by the divide and conquer algorithm on input size $n = 2^K$ is
(A) $A(n) = 2A(n/2) + 1$ (B) $A(n) = 2A(n/2) + n^2$
(C) $A(n) = 2A(n/4) + n^2$ (D) $A(n) = 2A(n/8) + n^2$

Practice Problems 2

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1.

Input Array	Linear Search $W(n)$	Binary search $W(n)$
128 elements	128	8
1024 elements	1024	x

Find x value?
(A) 10 (B) 11
(C) 12 (D) 13

2. Choose the correct one
(i) $\log n$ (ii) n
(iii) $n \log n$ (iv) n^2
(a) A result of cutting a problem size by a constant factor on each iteration of the algorithm.
(b) Algorithm that scans a list of size 'n'.
(c) Many divide and conquer algorithms fall in this category.
(d) Typically characterizes efficiency of algorithm with two embedded loops.
(A) i – b, ii – c, iii – a, iv – d
(B) i – a, ii – b, iii – c, iv – d
(C) i – c, ii – d, iii – a, iv – b
(D) i – d, ii – a, iii – b, iv – c

3. Insertion sort analysis in worst case
(A) $\theta(n)$
(B) $\theta(n^2)$
(C) $\theta(n \log n)$
(D) $\theta(2^n)$

4. From the recurrence relation. Of merge sort
$T(n) = 2T(n/2) + \theta(n)$.
Which option is correct?
I. $n/2$ II. 2T III. $\theta(n)$
(a) Extra work (divide and conquer)
(b) Sub-problem size
(c) Number of sub-problems
(A) III – b, II – a, I – c (B) I – b, II – c, III – a
(C) I – a, II – c, III – b (D) I – c, II – a, III – b

5. What is the number of swaps required to sort 'n' elements using selection sort, in the worst case?
(A) $\theta(n)$ (B) $\theta(n^2)$
(C) $\theta(n \log n)$ (D) $\theta(n^2 \log n)$

6. In a binary max heap containing 'n' numbers, the smallest element can be found in time
(A) $O(n)$ (B) $O(\log n)$
(C) $O(\log \log n)$ (D) $O(1)$

7. What is the worst case complexity of sorting 'n' numbers using quick sort?
(A) $\theta(n)$ (B) $\theta(n \log n)$
(C) $\theta(n^2)$ (D) $\theta(n!)$

8. The best case analysis of quick sort is, if partition splits the array of size n into
(A) $n/2 : n/m$ (B) $n/2 : n/2$
(C) $n/3 : n/2$ (D) $n/4 : n/2$

9. What is the time complexity of powering a number, by using divide and conquer methodology?
(A) $\theta(n^2)$ (B) $\theta(n)$
(C) $\theta(\log n)$ (D) $\theta(n \log n)$

10. Which one of the following in-place sorting algorithm needs the minimum number of swaps?
(A) Quick sort (B) Insertion sort
(C) Selection sort (D) Heap sort

11. As the size of the array grows what is the time complexity of finding an element using binary search (array of elements are ordered)?
(A) $\theta(n \log n)$ (B) $\theta(\log n)$
(C) $\theta(n^2)$ (D) $\theta(n)$

12. The time complexity of heap sort algorithm is
(A) $n \log n$ (B) $\log n$
(C) n^2 (D) None of these.

13. As part of maintenance work, you are entrusted with the work of rearranging the library books in a shelf in a proper order, at the end of each day. The ideal choices will be_____.
(A) Heap sort (B) Quick sort
(C) Selection sort (D) Insertion sort

14. The value for which you are searching is called
(A) Binary value
(B) Search argument
(C) Key
(D) Serial value

15. To sort many large objects and structures it would be most efficient to _____.
(A) Place them in an array and sort the array
(B) Place the pointers on them in an array and sort the array
(C) Place them in a linked list and sort the linked list
(D) None of the above

Previous Years' Questions

1. What is the number of swaps required to sort n elements using selection sort, in the worst case? **[2009]**
(A) $\theta(n)$
(B) $\theta(n \log n)$
(C) $\theta(n^2)$
(D) $\theta(n^2 \log n)$

2. Which one of the following is the tightest upper bound that represents the number of swaps required to sort n numbers using selection sort? **[2013]**
(A) $O(\log n)$ (B) $O(n)$
(C) $O(n \log n)$ (D) $O(n^2)$

3. Let P be a quick sort program to sort numbers in ascending order using the first element as the pivot. Let t_1 and t_2 be the number of comparisons made by P for the inputs [1 2 3 4 5] and [4 1 5 3 2] respectively. Which one of the following holds? **[2014]**
(A) $t_1 = 5$ (B) $t_1 < t_2$
(C) $t_1 > t_2$ (D) $t_1 = t_2$

4. The minimum number of comparisons required to find the minimum and the maximum of 100 numbers is ———. **[2014]**

5. Suppose P, Q, R, S, T are sorted sequences having lengths 20, 24, 30, 35, 50 respectively. They are to be merged into a single sequence by merging together two sequences at a time. The number of comparisons that will be needed in the worst case by the optimal algorithm for doing this is ———. **[2014]**

6. You have an array of n elements. Suppose you implement quick sort by always choosing the central element of the array as the pivot. Then the tightest upper bound for the worst case performance is **[2014]**
(A) $O(n^2)$ (B) $O(n \log n)$
(C) $\theta(n \log n)$ (D) $O(n^3)$

7. What are the worst-case complexities of insertion and deletion of a key in a binary search tree? **[2015]**

(A) $\theta(\log n)$ for both insertion and deletion
(B) $\theta(n)$ for both insertion and deletion
(C) $\theta(n)$ for insertion and $\theta(\log n)$ for deletion
(D) $\theta(\log n)$ for insertion and $\theta(n)$ for deletion

8. The worst case running times of *Insertion sort, Merge sort* and *Quick sort*, respectively, are: **[2016]**
(A) $\Theta(n \log n)$, $\Theta(n \log n)$,and $\Theta(n^2)$
(B) $\Theta(n^2)$, $\Theta(n^2)$,and $\Theta(n \log n)$
(C) $\Theta(n^2)$, $\Theta(n \log n)$,and $\Theta(n \log n)$
(D) $\Theta(n^2)$, $\Theta(n \log n)$,and $\Theta(n^2)$

9. An operator delete(i) for a binary heap data structure is to be designed to delete the item in the i-th node. Assume that the heap is implemented in an array and i refers to the i-th index of the array. If the heap tree has depth d (number of edges on the path from the root to the farthest leaf), then what is the time complexity to re-fix the heap efficiently after the removal of the element? **[2016]**
(A) $O(1)$ (B) $O(d)$ but not $O(1)$
(C) $O(2^d)$ but not $O(d)$ (D) $O(d2^d)$ but not $O(2^d)$

10. Assume that the algorithms considered here sort the input sequences in ascending order. If the input is already in ascending order, which of the following are **TRUE**? **[2016]**
I. Quicksort runs in $\Theta(n^2)$ time
II. Bubblesort runs in $\Theta(n^2)$ time
III. Mergesort runs in $\Theta(n)$ time
IV. Insertion sort runs in Θ (n) time
(A) I and II only (B) I and III only
(C) II and IV only (D) I and IV only

11. A complete binary min - heap is made by including each integer in [1,1023] exactly once. The depth of a node in the heap is the length of the path from the root of the heap to that node. Thus, the root is depth 0. The maximum depth at which integer 9 can appear is _____ . **[2016]**

Answer Keys

Exercises

Practice Problems 1

1. B	**2.** A	**3.** A	**4.** B	**5.** B	**6.** C	**7.** C	**8.** B	**9.** A	**10.** B
11. D	**12.** A	**13.** A	**14.** B	**15.** A					

Practice Problems 2

1. B	**2.** B	**3.** B	**4.** B	**5.** A	**6.** A	**7.** C	**8.** B	**9.** C	**10.** C
11. B	**12.** A	**13.** D	**14.** C	**15.** B					

Previous Years' Questions

1. A	**2.** B	**3.** C	**4.** 148	**5.** 358	**6.** A	**7.** B	**8.** D	**9.** B	**10.** D
11. 8									

Chapter 3

Divide-and-conquer

LEARNING OBJECTIVES

- *Divide-and-conquer*
- *Divide-and-conquer examples*
- *Divide-and-conquer technique*
- *Merge sort*
- *Quick sort*
- *Performance of quick sort*
- *Recurrence relation*
- *Searching*
- *Linear search*
- *Binary search*

Divide-and-Conquer

Divide-and-conquer is a top down technique for designing algorithms 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.

Divide-and-conquer paradigm consists of following major phases:

- Breaking the problem into several sub-problems that are similar to the original problem but smaller in size.
- Solve the sub-problem recursively (successively and independently)
- Finally, combine these solutions to sub-problems to create a solution to the original problem.

Divide-and-Conquer Examples

- Sorting: Merge sort and quick sort
- Binary tree traversals
- Binary Search
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair and Convex-hull algorithm

Merge Sort

Merge sort is a sorting algorithm for rearranging lists (or any other data structure that can only be accessed sequentially, e.g., file streams) into a specified order.

Merge sort works as follows:

1. Divide the unsorted list into two sub lists of about half the size.
2. Sort each of the two sub lists.

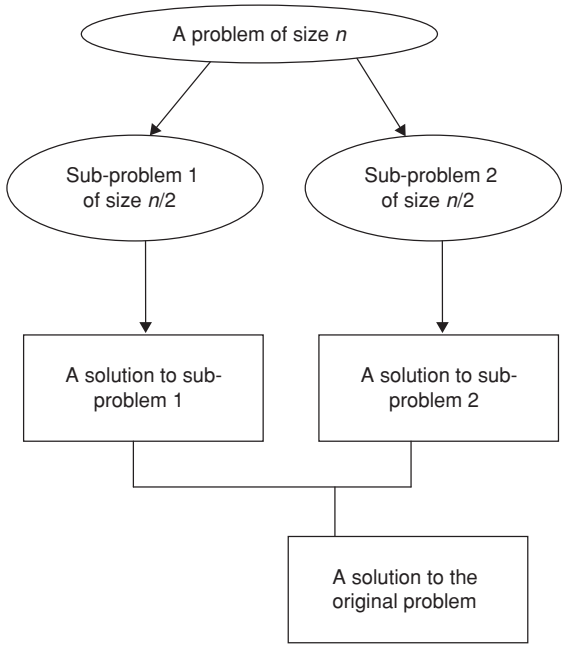


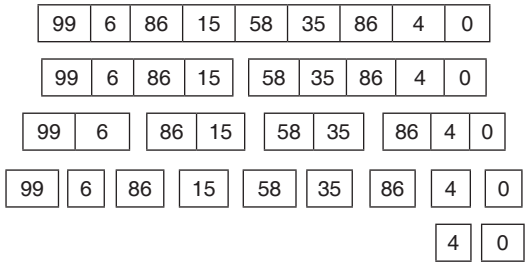
Figure 1 Divide-and-conquer technique.

3. Merge the two sorted sub lists back into one sorted list
4. The key of merge sort is merging two sorted lists into one, such that if we have 2 lists
   $X(x_1 \le x_2 \le x_3 \cdots \le x_m)$ and
   $Y(y_1 \le y_2 \le y_3 \cdots \le y_n)$ the resulting list is $z(z_1 \le z_2 \le \cdots \le z_{m+n})$

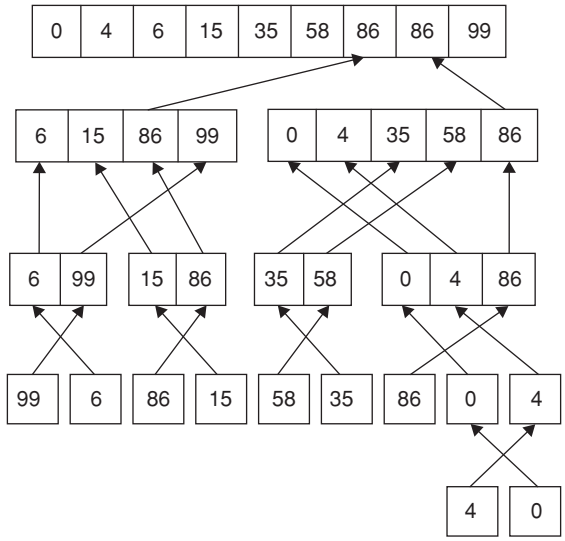
Example 1: $L_1 = \{3, 8, 9\}$, $L_2 = \{1, 5, 7\}$

Merge $(L_1, L_2) = \{1, 3, 5, 7, 8, 9\}$

Example 2:



Merge:



Implementing Merge Sort

Merging is done with a temporary array of the same size as the input array.

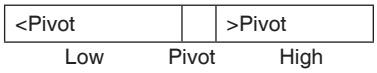
Pro: Faster than in-place since the temp array holds the resulting array until both left and right sides are merged into the temp array then the temp array is appended over the input array.

Con: The memory required is doubled. The double memory merge sort runs $O(N \log N)$ for all cases, because of its Divide-and-conquer approach.

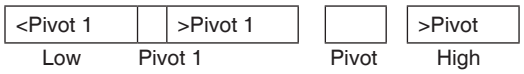
$$T(N) = 2T(N/2) + N$$
$$= O(N \log N)$$

Quick Sort

Quick sort is an example of Divide-and-conquer strategy. In Quick sort we divide the array of items to be sorted into two partitions and then call the quick sort procedure recursively to sort the two partitions, i.e., we divide the problem into two smaller ones and conquer by solving the smaller ones. The conquer part of the quick sort routine looks like this



Make bold



Divide: Partition the array A $[p - r]$ into 2 sub arrays A $[p - q - 1]$ and A $[q + 1 - r]$ such that each element of A $[p - q - 1]$ is less than or equal to A$[q]$, which is, in turn, less than or equal to each element of A $[q + 1 - r]$

Conquer: Sort the 2 sub arrays A $[p - q - 1]$ and A $[q + 1 - r]$ by recursive calls to quick sort.

Combine: Since the sub arrays are sorted inplace, no work is needed to combine them.

Sort left partition in the same way. For this strategy to be effective, the partition phase must ensure that the pivot, is greater than all the items in one part (the lower part) and less than all those in the other (upper) part. To do this, we choose a pivot element and arrange that all the items in the lower part are less than the pivot and all those in the upper part are greater than it. In the general case, the choice of pivot element is first element.

(Here $\lfloor$number of elements/2$\rfloor$ is pivot)

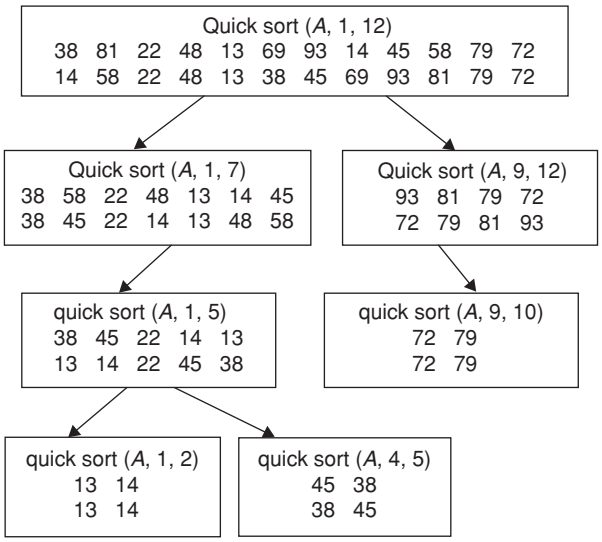


Figure 2 Tree of recursive calls to quick sort.

- Quick sort is a sorting algorithm with worst case running time $O(n^2)$ on an input array of n numbers. Inspite of this slow worst case running time, quick sort is often the best practical choice for sorting because it is efficient on the average: its expected running time is $O(n \log n)$ and the constants hidden in the O-notation are quite small
- Quick sort algorithm is fastest when the median of the array is chosen as the pivot element. This is because the resulting partitions are of very similar size. Each partition splits itself in two and thus the base case is reached very quickly.

Example: Underlined element is pivot.

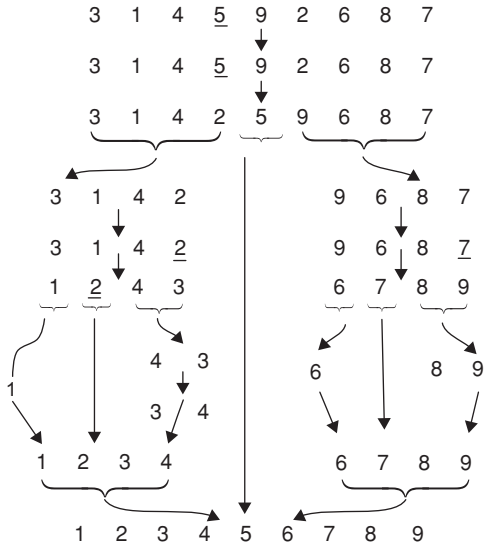


Figure 3 The ideal quick sort on a random array

Performance of Quick Sort

- Running time of quick sort depends on whether the partitioning is balanced or unbalanced, it depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, it runs as slowly as insertion sort.
- The worst case of quick sort occurs when the partitioning routine produces one sub-problem with $n - 1$ elements and one with '1' element. If this unbalanced partitioning arises in each recursive call, the partitioning costs $\theta(n)$ time.

Recurrence Relation

$$T(n) = T(n-1) + T(1) + \theta(n)$$
$$(\therefore\ T(0) = \theta(1))$$
$$= T(n-1) + \theta(n)$$

If we sum the costs incurred at each level of the recursion we get an arithmetic series, which evaluates to $\theta(n^2)$.

- Best case partitioning–PARTITION produces 2 sub problems, each of size no more than $n/2$, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$

The recurrence for the running time is then

$$T(n) \le 2T(n/2) + \theta(n)$$

The above Recurrence relation has the solution $T(n) = O(n \log n)$ by case 2 of the master theorem.

- The average–case time of quick sort is much closer to the best than to the worst case

For example, that the partitioning algorithm always produces a 8-to-2 proportional split, which at first seems unbalanced. The Recurrence relation will be

$$T(n) \le T(8n/10) + T(2n/10) + cn$$

The recursion tree for this recurrence has cost 'cn' at every level, until a boundary condition is reached at depth $\log_{10} n = \theta(\log n)$. The recursion terminates at depth $\log_{10/8} n = \theta(\log n)$. The total cost of quick sort is $O(n \log n)$

Searching

Two searching techniques are:

- Linear search
- Binary search

Linear Search

Linear search (or) sequential search is a method for finding a particular value in list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found. Linear search is a special case of brute force search. Its worst case cost is proportional to the number of elements in the list.

Implementation

boolean linear search `(int [ ] arr, int target)`

```
{
int i = 0;
while (i < arr. length) {
if (arr [i] = = target){
return true;
}
+ + i;
}
return false;
}
```

Example:

Consider the array

10	7	1	3	–4	2	20

Search for 3

10	7	1	3	–4	2	20

3?

Move to next element

10	7	1	3	–4	2	20

3?

Move to next element

10	7	1	3	–4	2	20

3?

Move to next element

10	7	1	3	–4	2	20

3?

Element found; stop the search.

Binary Search

A binary search algorithm is a technique for finding a particular value in a linear array, by ruling out half of the data at each

step; a binary search finds the median, makes comparison, to determine whether the desired value comes before or after it, and then searches the remaining half in the same manner. A binary search is an example of Divide-and-conquer algorithm.

Implementation

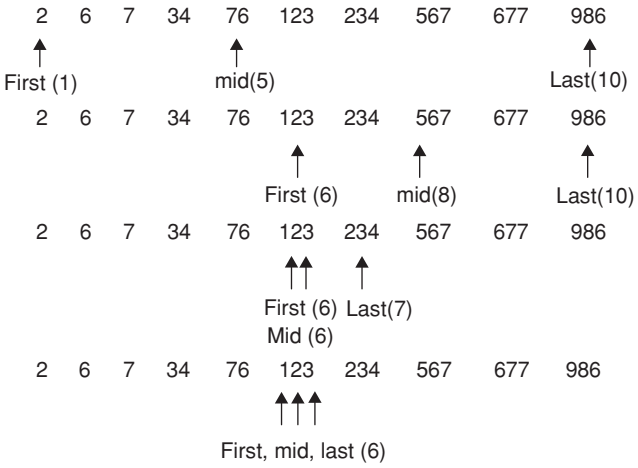
function binary search (a, value, left, right)

```
{
if right < left
       return not found
      mid: = floor ((right -left)/2) + left
      if a [mid] = value
      return mid
      if value < a[mid]
```

return binary search (a, value, left, mid –1) else return binary search (a, value, mid + 1, right)

}

Example: Value being searched 123



Exercises

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. How many comparisons are required to search an item 89 in a given list, using Binary search?

4	8	19	25	34	39	45	48	66	75	89	95
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

(A) 3 (B) 4
(C) 5 (D) 6

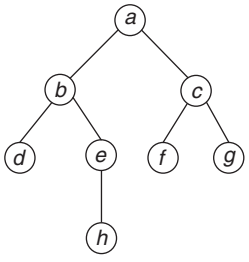
2. Construct a Binary search tree with the given list of elements:

300, 210, 400, 150, 220, 370, 450, 100, 175, 215, 250

Which of the following is a parent node of element 250?

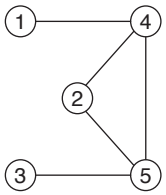
(A) 220
(B) 150
(C) 370
(D) 215

3. What is the breadth first search order of the given tree?



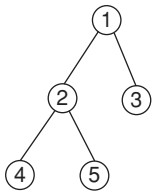
(A) acbhdefg (B) abcdefgh
(C) adbcefgh (D) aebcdfgh

4. What is the depth first search order of the given graph?



(A) 14325
(B) 12435
(C) 14253
(D) 12354

5. When pre-order traversal is applied on a given tree, what is the order of elements?

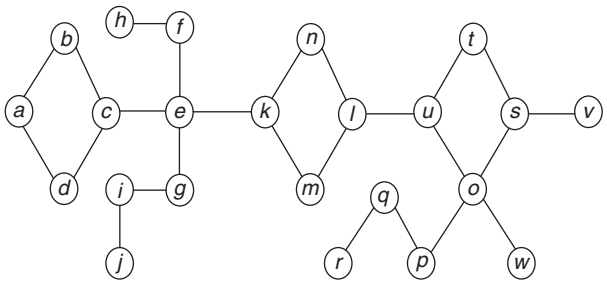


(A) 1 – 2 – 4 – 5 – 3
(B) 1 – 4 – 2 – 5 – 3
(C) 1 – 2 – 4 – 3 – 5
(D) 1 – 2 – 3 – 4 – 5

6. What is the order of post-order traversal and in-order traversals of graph given in the above question?

(A) 4 – 2 – 5 – 1 – 3 and 4 – 5 – 2 – 3 – 1
(B) 4 – 5 – 2 – 3 – 1 and 4 – 2 – 5 – 1 – 3
(C) 4 – 5 – 2 – 1 – 3 and 4 – 2 – 5 – 1 – 3
(D) 4 – 5 – 2 – 3 – 1 and 4 – 2 – 5 – 3 – 1

7. Find the number of bridges in the given graph

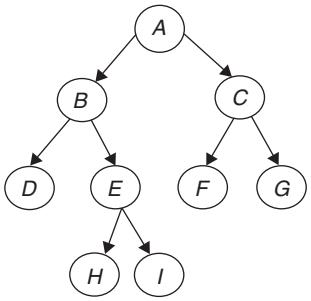


(A) 12 (B) 13
(C) 11 (D) 10

8. Match the following:

I.	In-order	1.	*ABCDEFGHI*
II.	Pre-order	2.	*DBHEIAFCG*
III.	Post-order	3.	*ABDEHICFG*
IV.	Level-order	4.	*DHIEBFGCA*

For the tree



(A) I – 2, II – 3, III – 4, IV – 1
(B) I – 3, II – 1, III – 4, IV – 2
(C) I – 1, II – 2, III – 3, IV – 4
(D) I – 4, II – 3, III – 2, IV – 1

9. A complete *n*-array tree in which each node has '*n*' children (or) no children.

Let '*I*' be the number of internal nodes and '*L*' be the number of leaves in a complete *n*-ary tree.

If $L = 51$ and $I = 10$ what is the value of '*n*'?

(A) 4 (B) 5
(C) 6 (D) Both (A) and (B)

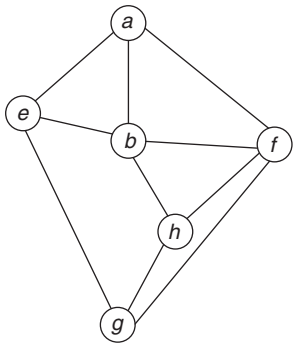
10. A complete *n*-ary tree is one in which every node has 0 (or) *n* children. If '*X*' is the number of internal nodes of a complete *n*-ary tree, the number of leaves in it is given by

(A) $X(n-1)+1$ (B) $Xn-1$
(C) $Xn+1$ (D) $X(n+1)+1$

11. The numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in the given order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers. What is the in-order traversal sequence of the resultant tree?

(A) 7 5 1 0 3 2 4 6 8 9
(B) 0 2 4 3 1 6 5 9 8 7
(C) 0 1 2 3 4 5 6 7 8 9
(D) 9 8 6 4 2 3 0 1 5 7

12. Consider the following graph:



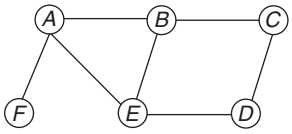
Among the following sequences

I. *a b e g h f* II. *a b f e h g*
III. *a b f h g e* IV. *a f g h b e*

Which are depth first traversals of the above graph?

(A) I, II and IV only (B) I and IV only
(C) I, III only (D) I, III and IV only

13. The breadth first search algorithm has been implemented using the queue data structure. One possible order of visiting the nodes is



(A) *A B C D E F* (B) *B E A D C F*
(C) *E A B D F C* (D) Both (A) and (B)

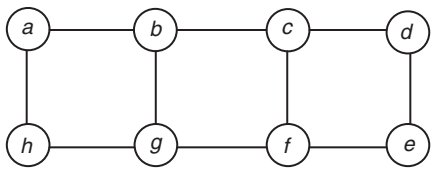
14. An undirected graph *G* has '*n*' nodes. Its adjacency matrix is given by an $n \times n$ square matrix.

(i) Diagonal elements are 0's
(ii) Non-diagonal elements are 1's

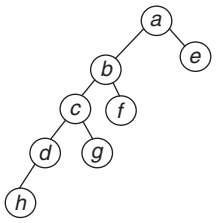
Which of the following is true?

(A) Graph *G* has no minimum spanning tree
(B) Graph *G* has a unique minimum spanning tree of cost $(n-1)$
(C) Graph *G* has multiple distinct minimum spanning trees, each of cost $(n-1)$
(D) Graph *G* has multiple spanning trees of different cost.

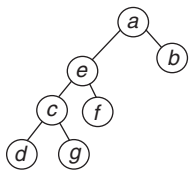
15. Which of the following is the breadth first search tree for the given graph?



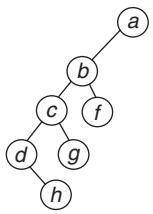
(A)



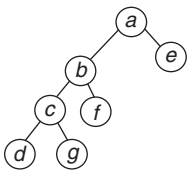
(B)



(C)



(D)



Practice Problems 2

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. Which of the following algorithm design technique is used in finding all pairs of shortest distances in a graph?
(A) Divide-and-conquer
(B) Greedy method
(C) Back tracking
(D) Dynamic programming

2. Let LASTPOST, LASTIN and LASTPRE denote the last vertex visited in a post-order, in-order and pre-order traversals respectively of a complete binary tree. Which of the following is always true?
(A) LASTIN = LASTPOST
(B) LASTIN = LASTPRE
(C) LASTPRE = LASTPOST
(D) LASTIN = LASTPOST = LASTPRE

3. Match the following:

X : Depth first search

Y : Breadth first search

Z : Sorting

a : Heap

b : Queue

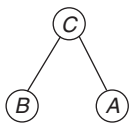
c : Stack

(A) X – a, Y – b, Z – c
(B) X – c, Y – a, Z – b
(C) X – c, Y – b, Z – a
(D) X – a, Y – c, Z – b

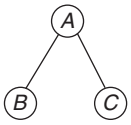
4. Let G be an undirected graph, consider a depth first traversal of G, and let T be the resulting DFS Tree. Let 'U' be a vertex in 'G' and let 'V' be the first new (unvisited) vertex visited after visiting 'U' in the traversal. Which of the following is true?
(A) $\{U, V\}$ must be an edge in G and 'U' is a descendant of V in T.
(B) $\{U, V\}$ must be an edge in 'G' and V is a descendant of 'U' in T.
(C) If $\{U, V\}$ is not an edge in 'G' then 'U' is a leaf in T.
(D) if $\{U, V\}$ is not an edge in G then U and V must have the same parent in T.

5. Identify the binary tree with 3 nodes labeled A, B and C on which preorder traversal gives the sequence C, B, A.

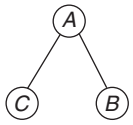
(A)



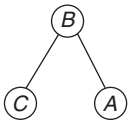
(B)



(C)



(D)

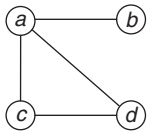


6. Consider an undirected unweighted graph G. Let a breadth first traversal of G be done starting from a node r. Let $d(r, u)$ and $d(r, v)$ be the lengths of the shortest paths from r to u and v respectively in 'G'. If u is visited before v during the breadth first travel, which of the following is correct?
(A) $d(r, u) < d(r, v)$ (B) $d(r, u) > d(r, v)$
(C) $d(r, u) \leq d(r, v)$ (D) None of these

7. In a complete 5-ary tree, every internal node has exactly 5 children. The number of leaves in such a tree with '3' internal nodes are:
(A) 15 (B) 20
(C) 13 (D) Can't predicted

8. Which of the following algorithm is single pass that is they do not traverse back up the tree for search, create, insert etc.
(A) Depth first search (B) Pre-order traversal
(C) B-tree traversal (D) Post-order traversal

9. Which of the following is the adjacency matrix of the given graph?



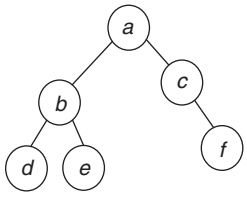
(A) $\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$

(B) $\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

(C) $\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$

(D) $\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

10. Which one of the following is the post-order traversal of the given tree?

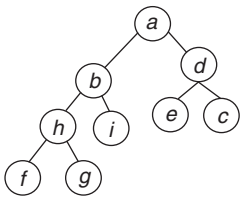


(A) *d e a f c b a* (B) *d e b f c a*
(C) *e b d f c a* (D) *a b c d e f*

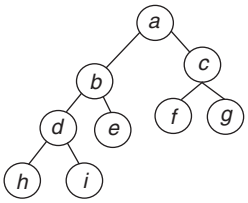
Common data for questions 11 and 12:

11. The pre-order traversal of a tree is *a b d h i e c f g*. Which of the following is the correct tree?

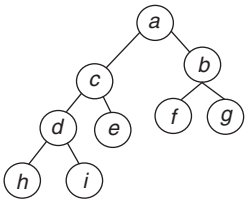
(A)



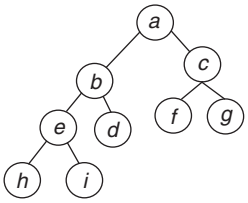
(B)



(C)



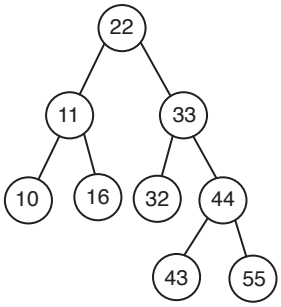
(D)



12. Which of the following is in-order traversal of the above tree?

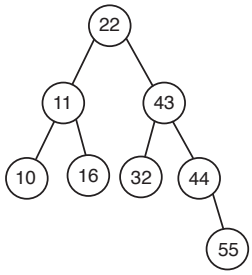
(A) *a b h d e i f g c* (B) *a b d h e i f g c*
(C) *h d i b e a f c g* (D) *i d h b e a f c g*

13. Consider the below binary search tree

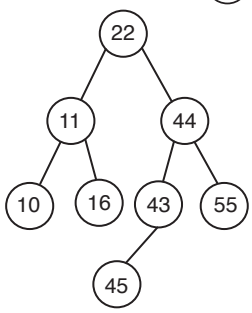


Which of the following is the resultant binary search tree after deletion of 33?

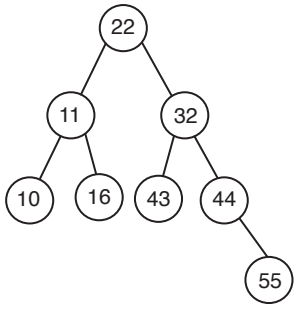
(A)



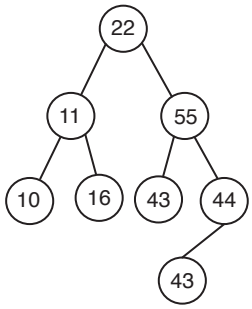
(B)



(C)



(D)



14. Match the following:

I.	Articulation Point	1.	An edge whose removal disconnects graph
II.	Bridge	2.	A vertex whose removal disconnects graph
III.	Bi connected component	3.	Maximal set of edges such that any two edges in the set lie on a common simple cycle

(A) I – 1, II – 2, III – 3 (B) I – 2, II – 1, III – 3
(C) I – 2, II – 3, III – 1 (D) I – 1, II – 2, III – 3

15. If x is the root of an n-node subtree, then the inorder-tree-walk takes

(A) $\theta(n)$ (B) $\theta(n^2)$
(C) $\theta(n^3)$ (D) $\theta(n \log n)$

Previous Years' Questions

1. Which one of the following is the tightest upper bound that represents the time complexity of inserting an object into a binary search tree of n nodes? **[2013]**

(A) $O(1)$ (B) $O(\log n)$
(C) $O(n)$ (D) $O(n \log n)$

2. Consider a rooted n node binary tree represented using pointers. The best upper bound on the time required to determine the number of sub trees having exactly 4 nodes is $O(n^a \log^b n)$. The value of $a + 10b$ is _____ **[2014]**

3. Which one of the following is the recurrence equation for the worst case time complexity of the Quicksort algorithm for sorting $n(\geq 2)$ numbers? In the recurrence equations given in the options below, c is a constant. **[2015]**

(A) $T(n) = 2T(n/2) + cn$
(B) $T(n) = T(n-1) + T(1) + cn$
(C) $T(n) = 2T(n-1) + cn$
(D) $T(n) = T(n/2) + cn$

4. Suppose you are provided with the following function declaration in the C programming language.

int partition (int a[ ], int n);

The function treats the first element of a [ ] as a pivot, and rearranges the array so that all elements less than or equal to the pivot is in the left part of the array, and all elements greater than the pivot is in the right part in addition, it moves the pivot so that the pivot is the last element of the left part. The return value is the number of elements in the left part.

The following partially given function in the C programming language is used to find the kth smallest element in an array a [ ] of size n using the partition function. We assume $k'' n$.

int $k$th_smallest (int a [ ], int n, int k) **[2015]**

```
{
    int left_end = partition(a, n);
    if (left_end+1 == k) {
      return a [left_end];
    )
    if (left_end+1 > k) {
      return kth_smallest (__________);
    } else {
      return kth_smallest (__________);
    }
}
```

The missing argument lists are respectively

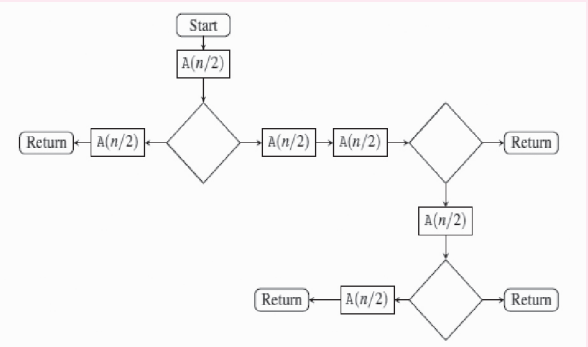
(A) (a, left_end, k) and (a+left_end+1, n-left_end-1, k-left_end-1)
(B) (a, left_end, k) and (a, n-left_end-1, k-left_end-1)
(C) (a+left_end+1, n-left_end-1, k-left_end-1) and (a, left_end, k)
(D) (a, n-left_end-1, k-left_end-1) and (a, left_end, k)

5. Assume that a mergesort algorithm in the worst case takes 30 seconds for an input of size 64. Which of the following most closely approximates the maximum input size of a problem that can be solved in 6 minutes? **[2015]**

(A) 256 (B) 512
(C) 1024 (D) 2048

6. The given diagram shows the flowchart for a recursive function A(n). Assume that all statements, except for the recursive calls, have O (1) time complexity. If the worst case time complexity of this function is O (n^α), then the least possible value (accurate up to two decimal positions) of α is ____. **[2016]**

Flowchart for Recursive Function A(n)



7. Let A be an array of 31 numbers consisting of a sequence of 0's followed by a sequence of 1's. The problem is to find the smallest index i such that $A[i]$ is 1 by probing the minimum number of locations in A. The *worst case* number of probes performed by an *optimal* algorithm is _____. **[2017]**

8. Match the algorithms with their time complexities:

Algorithm	Time complexity
(P) Towers of Hanoi with n disks	(i) $\Theta(n^2)$
(Q) Binary search given n sorted numbers	(ii) $\Theta(n \log n)$
(R) Heap sort given n numbers at the worst case	(iii) $\Theta(2^n)$
(S) Addition of two $n \times n$ matrices	(iv) $\Theta(\log n)$

[2017]

(A) P → (iii), Q → (iv), R → (i), S → (ii)
(B) P → (iv), Q → (iii), R → (i), S → (ii)
(C) P → (iii), Q → (iv), R → (ii), S → (i)
(D) P → (iv), Q → (iii), R → (ii), S → (i)

Answer Keys

Exercises

Practice Problems 1

1. A	**2.** A	**3.** B	**4.** C	**5.** A	**6.** B	**7.** B	**8.** A	**9.** C	**10.** A
11. C	**12.** D	**13.** A	**14.** C	**15.** A					

Practice Problems 2

1. B	**2.** B	**3.** C	**4.** B	**5.** A	**6.** D	**7.** C	**8.** C	**9.** A	**10.** B
11. B	**12.** C	**13.** A	**14.** B	**15.** A					

Previous Years' Questions

1. C	**2.** 1	**3.** B	**4.** A	**5.** B	**6.** 2.2 to 2.4	**7.** 5	**8.** C

Chapter 4

Greedy Approach

LEARNING OBJECTIVES

- *Greedy approach*
- *Knapsack problem*
- *Fractional knapsack problem*
- *Spanning trees*
- *Prim's algorithm*
- *Kruskal's algorithm*
- *Tree and graph traversals*
- *Back tracking*
- *Graph traversal*
- *Breadth first traversal*
- *Depth first search*
- *Huffman codes*
- *Task-scheduling problem*
- *Sorting and order statistics*
- *Simultaneous minimum and maximum*
- *Graph algorithms*

Greedy Approach

In a greedy method, we attempt to construct an optimal solution in stages.

- At each stage we make a decision that appears to be the best (under some criterion) at the time.
- A decision made at one stage is not changed in a later stage, so each decision should assure feasibility.
- Some problems that use greedy approach are:

1. Knapsack problem
2. Minimum spanning tree
3. Prims algorithm
4. Kruskals algorithm

Knapsack Problem

The knapsack problem is a problem in combinatorial optimization: given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. We have n kinds of items, 1 through n. Each kind of item i has a value V_i and a weight W_i, usually assume that all values and weights are non-negative. The maximum weight that we can carry in the bag is W.

Solved Examples

Example 1: (Making change)

Problem:	Accept n dollars, to return a collection of coins with a total value of n dollars.
Configuration:	A collection of coins with a total value of n.
Objective function:	Minimize number of coins returned.
Greedy solution:	Always return the largest coin you can.

- Coins are valued \$.30, \$.020, \$0.05, \$0.01 use a greedy choice property and make \$.40 by using 3 coins.

Solution: \$0.30 + \$0.05 + \$0.05 = \$0.40

Fractional Knapsack Problem

Given: A set S of n items, with each item i having

- b_i – a positive benefit
- w_i – a positive weight

Goal: Choose items with maximum total benefit but with weight atmost W.

If we are allowed to take fractional amounts, then this is the fractional knapsack problem.

- In this case, let x_i denote the amount we take of item i.
- Objective: Maximize $\sum_{i \in S} b_i (x_i / w_i)$
- Constraint: $\sum_{i \in S} x^i \leq W$

Example 2:

Items	1	2	3	4	5
Weight	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit	$12	$32	$40	$30	$50
Value ($ per ml)	3	4	20	5	50

Solution: 1 ml of 5, 2 ml of 3, 6 ml of 4, 1 ml of 2

- Greedy choice: Keep taking item with highest value (benefit to weight ratio).
- Correctness: suppose there is a better solution, there is an item i with higher value than a chosen item *j*. (i.e., $v_j < v_i$). If we replace some j with i, we get a better solution.

Thus there is no better solution than the greedy one.
$N = 3, m = 20$
$(P_1, P_2, P_3) = (25, 24, 15)$
$(W_1, W_2, W_3) = (18, 15, 10)$

Example 3:

	$X_1\ X_2\ X_3$	$\Sigma\ W_i X_i$	$\Sigma\ P_i X_i$
1.	1/2 1/3 1/4	9 + 5 + 2.5 = 16.5	12.5 + 8 + 3.75 = 24.25
2.	1 2/15 0	18 + 2 + 0 = 20	25 + 3.2 + 0 = 28.2
3.	0 2/3 1	0 + 10 + 10 = 20	0 + 16 + 15 = 31
4.	0 1 1/2	0 + 15 + 5 = 20	0 + 24 + 7.5 = 31.5

(1), (2), (3), (4) are feasible ones but (4) is the optimum solution.

Spanning Trees

A spanning tree of a graph is just a sub graph that contains all the vertices and is a tree. A graph may have many spanning trees.

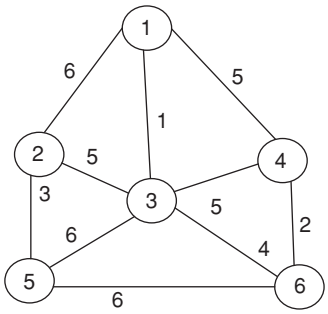
- A sub graph that spans (reaches out to) all vertices of a graph is called a spanning sub graph.
- A sub graph that is a tree and that spans all vertices of the original graph is called a spanning tree.
- Among all the spanning trees of a weighted and connected graph, the one (possibly more) with the least total weight is called a Minimum Spanning Tree (MST).

Prim's Algorithm

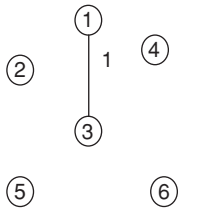
Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm continuously increases the size, of a tree, one edge at a time starting with a tree consisting of a single vertex, until it spans all vertices.

- Using a simple binary heap data structure and an adjacency list representation, prim's algorithm can be shown to run in time $O(E \log V)$ where E is the number of edges and V is the number of vertices.

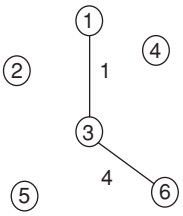
Example:



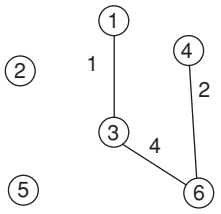
Start:



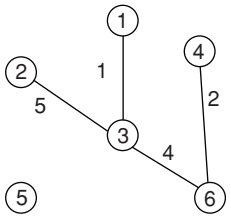
Iteration 1: U = {1, 3}



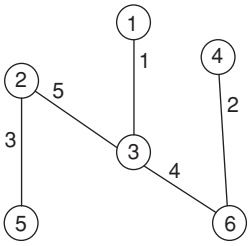
Iteration 2: U = {1, 3, 6}



Iteration 3: U = {1, 3, 6, 4}



Iteration 4: U = {1, 3, 6, 4, 2}



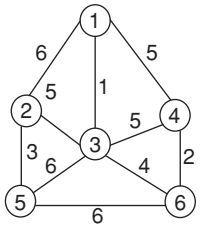
Iteration 5: U = {1, 3, 6, 4, 2, 5}

Figure 1 An example graph for illustrating prim's algorithm.

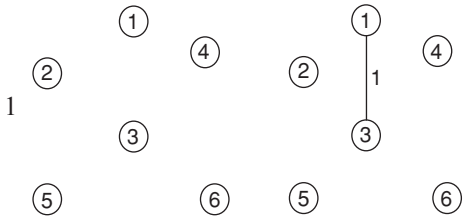
Kruskal's Algorithm

Like prim's algorithm, Kruskal's algorithm also constructs the minimum spanning tree of a graph by adding edges to the spanning tree one-by-one. At all points, during its execution the set of edges selected by prim's algorithm forms exactly one tree. On the other hand the set of edges selected by Kruskal's algorithm forms a forest of trees. Kruskals algorithm is conceptually simple. The edges are selected and added to the spanning tree in increasing order of their weights. An edge is added to the tree only if it does not create a cycle.

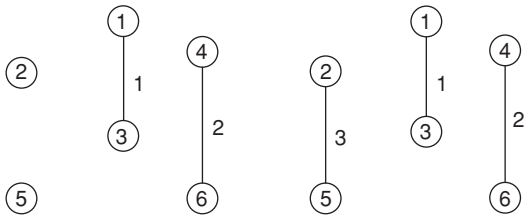
Example:



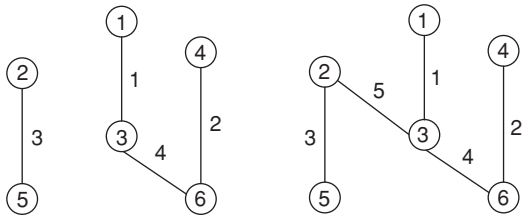
Start:



Initial configuration Setp 1: choose (1, 3)



Setep 2: choose (4, 6) Setep 3: choose (2, 5)



Setep 4: choose (3, 6) Setep 6: choose (4, 3)

Tree and Graph Traversals

Back Tracking

Backtracking is a general algorithm technique that considers searching every possible combination in order to solve an optimization problem.

Backtracking is also known as depth first search (or) branch and bound. Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, sudoku and many other puzzles. It is often the more convenient technique for parsing, for the knapsack problem and other combinational optimization problems.

- The advantage of backtracking algorithm is that they are complete, that is they are guaranteed to find every solution to every possible puzzle.

Graph Traversal

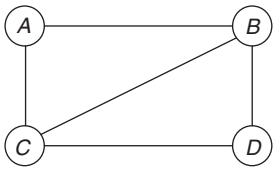
To traverse a graph is to process every node in the graph exactly once, because there are many paths leading from one node to another, the hardest part about traversing a graph is making sure that you do not process some node twice. There are general solutions to this difficulty.

1. When you first encounter a node, mark it as REACHED. When you visit a node, check if it is marked REACHED, if it is, just ignore it. This is the method our algorithms will use.
2. When you process a node, delete it from the graph. Deleting the node causes the deletion of all the arcs that lead to the node, so it will be impossible to reach it more than once.

General traversal strategy

1. Mark all nodes in the graph as NOT REACHED,
2. Pick a starting node, mark it as REACHED, and place it on the READY list.
3. Pick a node on the READY list. Process it remove it from READY. Find all its neighbors, those that are NOT REACHED should marked as REACHED and added to READY.
4. Repeat 3 until READY is empty.

Example:



Step I: *A* = *B* = *C* = *D* = NOT REACHED
Step II: READY = {A} . A = REACHED
Step III: Process A. READY = {B, C}.
B = C = REACHED
Step IV: Process C. READY = {B, D}.
D = REACHED
Step V: Process B. READY = {D}
Step VI: Process D. READY = { }

The two most common traversal patterns are

- Breadth first traversal
- Depth first traversal

Breadth First Traversal

In breadth first traversal, READY is a QUEUE, not an arbitrary list. Nodes are processed in the order they are reached (FIFO), this has the effect of processing nodes according to their distance from the initial node. First, the initial node is processed. Then all its neighbors are processed. Then all of the neighbors etc.

- Since a graph has no root, we must specify the vertex at which to start the traversal.
- Breadth first tree traversal first visits all the nodes at depth zero (i.e., the root) then all the nodes at depth 1, and so on.

Procedure

First, the starting vertex is enqueued. Then, the following steps are repeated until the queue is empty.

1. Remove the vertex at the head of the queue and call it vertex.
2. Visit vertex
3. Follow each edge emanating from vertex to find the adjacent vertex and call it 't_o'. If 't_o' has not already been put into the queue, enqueued it.

Notice, that a vertex can be put into the queue at most once. Therefore, the algorithm must some how keep track of the vertices that have been enqueued.

Procedure for BFS for undirected graph G(V, E)

To perform BFS over a graph, the data structures required are queue (Q) and the visited set (Visited), 'V' is the starting vertex.

Procedure for BFS(V)

Steps

1. Visit the vertex 'V'
2. Enqueue the vertex V
3. while (Q is not Empty)
   (i) V = dequeue ();
   (ii) for all vertices ϑ adjacent to V
      (a) if not visited (ϑ)
         - Enqueue (ϑ)
         - Visit the vertex 'ϑ'
         - end if.
         - end for
         - end while
         - Stop

Example:

Unexplored vertex

Visited vertex

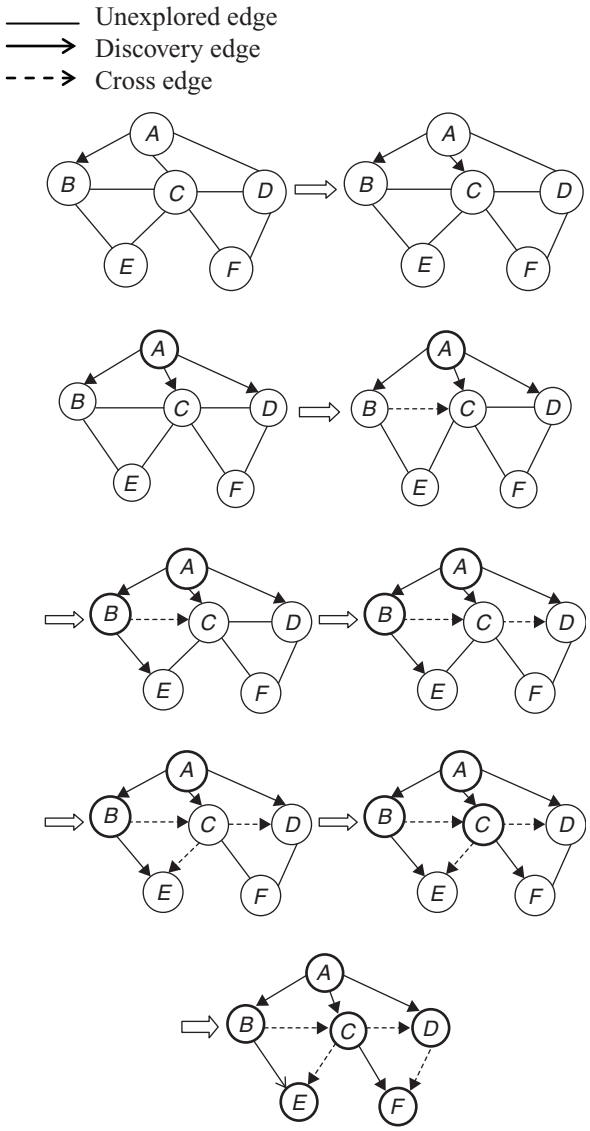


Figure 2 Breadth–first search

Depth First Search

A depth first traversal of a tree always starts at the root of the tree. Since a graph has no root, when we do a depth first traversal, we must specify the vertex at which to begin. A depth first traversal of a tree visits a node and then recursively visits the sub trees of that node similarly, depth first traversal of a graph visits a vertex and then recursively visits all the vertices adjacent to that node. A graph may contain cycles, but the traversal must visit every vertex at most once.

The solution to the problem is to keep track of the nodes that have been visited.

Procedure for DFS for undirected graph G(V, E)

To perform DFS over a graph, the data structures required are stack (S) and the list (visited), 'V' is the start vertex.

Procedure for DFS(V)

Steps

1. push the start vertex 'V' into the stack S
2. while (S is not empty)
   (i) pop a vertex V
   (ii) if 'V' is not visited
      (a) visit the vertex
      (b) Store 'V' in visited
      (c) push all the adjacent vertices of 'V' in to visited
   (iii) End if
3. End while
4. Stop.

Example:

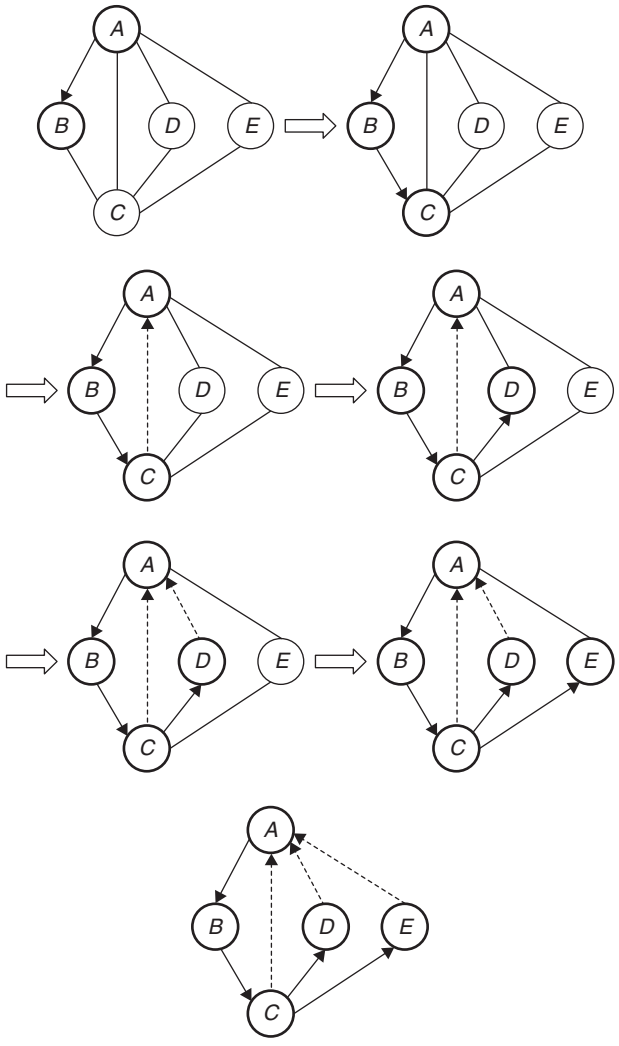
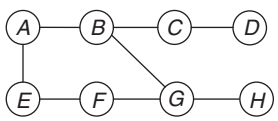


Figure 3 Depth first search

- Let us compare two traversal orders on the following graph:



Initial steps:

READY = [A]. process A. READY = [B, E]. process B.

It is at this point that two traversal strategies differ. Breadth first adds B's neighbors to the back of READY, depth first adds them to the front.

Breadth first

- READY = [E, C, G]
- Process E. READY = [C, G, F]
- Process C. READY = [G, F, D]
- Process G .READY = [F, D, H]
- Process F. READY = [D, H]
- Process D. READY = [H]
- Process H. READY = [ ]

Depth First

- READY = [C, G, E]
- Process C. READY = [D, G, E]
- Process D. READY = [G, E]
- Process G. READY = [H, F, E]
- Process H. READY = [F, E]
- Process F. READY = [E]
- Process E. READY = [ ]

Connected Components

A graph is said to be connected if every pair of vertices in the graph are connected. A connected component is a maximal connected sub graph of 'G'. Each vertex belongs to exactly one connected component as does each edge.

- A graph that is not connected is naturally and obviously decomposed into several connected components (Figure 4). Depth first search does this handily. Each restart of the algorithm marks a new connected component.
- The directed graph in (Figure 5) is "Connected" Part of it can be "Pulled apart" (so to speak, without "breaking" any edges).
- Meaningful way to define connectivity in directed graph is:

'Two nodes U and V of a directed graph $G = (V, E)$ connected if there is a path from U to V', and one from V to U. This relation between nodes is reflective, symmetric and transitive. As such, it partitions V into disjoint sets, called the strongly connected components of the graph. In the directed graph of figure 2 there are four strongly connected components.

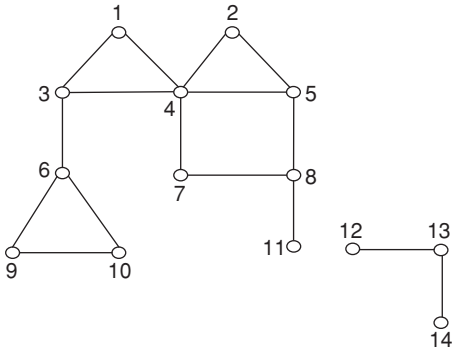


Figure 4 Undirected graph.

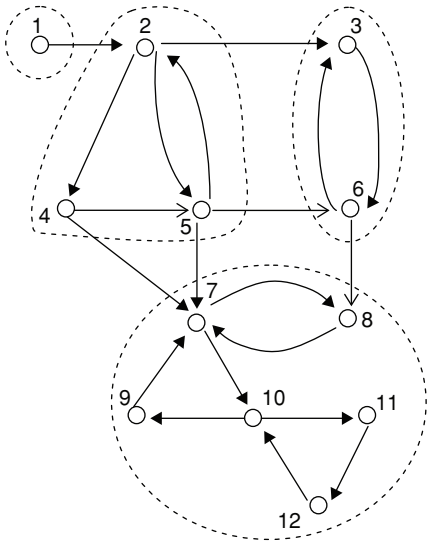
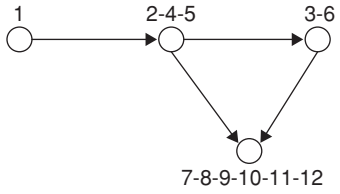


Figure 5 A directed graph and its strongly connected components

If we shrink each of these strongly connected components down to a single node and draw an edge between two of them if there is an edge from some node in the first to some node in the second, the resulting directed graph has to be a directed acyclic graph (DAG) – it has no cycles (figure 6). The reason is simple.

A cycle containing several strongly connected components would merge them all to a single strongly connected component.



Every directed graph is a DAG of its strongly connected components.

Huffman Codes

For compressing data, a very effective and widely used technique is Huffman coding. We consider the data to be a sequence of characters. Huffmans's greedy algorithm uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.

Example: Suppose we have a 1,00,000 – character data file, that we wish to store compactly. The characters in the file occur with the frequencies given below:

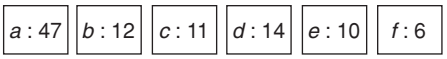
Character	a	b	c	d	e	f
Frequency	47	12	11	14	10	6

Solution: Two methods are used for compression of data are:

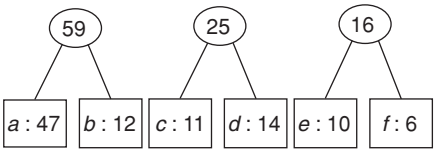
Fixed Length Coding

- Arrange all the characters in sequence (no particular order is followed)
- $a = 47, b = 12, c = 11, d = 14, e = 10, f = 6$

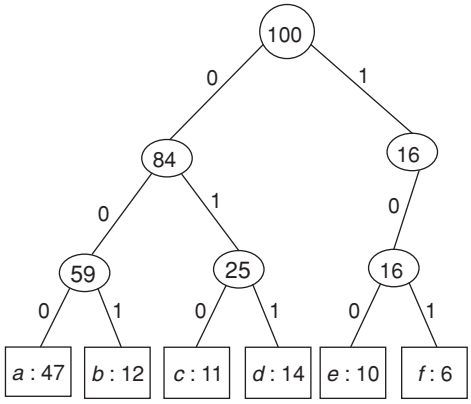
Step I:



Step II:



Step III:



We interpret the binary code word for a character as the path from the root to that character where '0' means 'go to the left child', and 1 means 'go to the right child'.

The above tree is not binary search tree, since the leaves need not appear in sorted order.

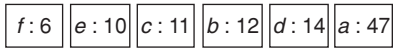
Constructing Huffman Code

This algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with set of $|C|$ leaves and performs a sequence of $|C|$-1 'merging' operations to create the final tree.

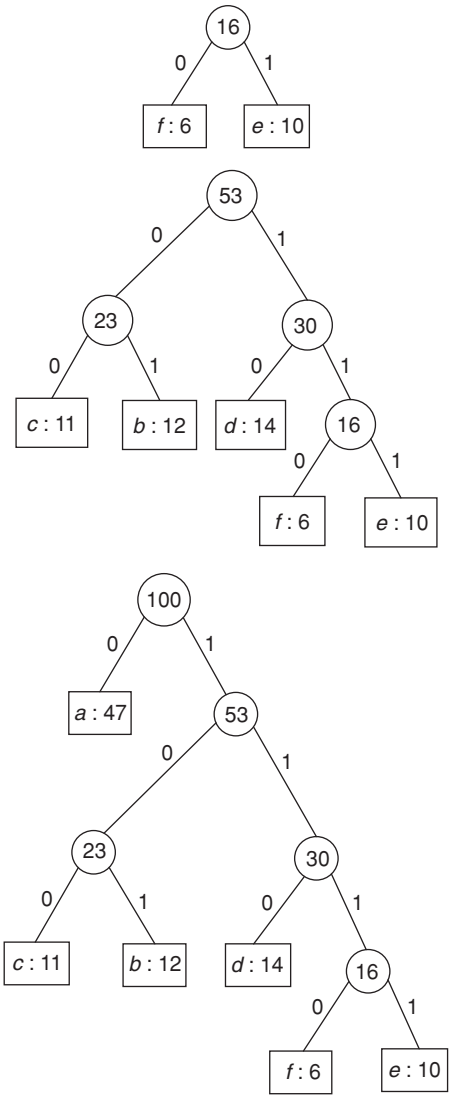
- A min-priority queue Q, keyed on f, is used to identify the 2 least – frequent objects to merge together. The result of the merger of 2 objects is a new object whose frequency is the sum of the frequencies of the 2 objects that were merged.
- In the given example, there are 6 alphabets the initial queue size is $n = 6$, and 5 merge steps are required to build the tree. The final tree represents the optimal prefix code. The code word for a letter is the sequence of edge labels on the path from the root to the letter.

$a = 47, b = 12, c = 11, d = 14, e = 10, f = 6$

Step I: Arrange the characters in non-decreasing order according to their frequencies



Let x and y be 2 characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the code words for x and y have the same length and differ only in the last bit



Analysis: The analysis of the running time of Huffman's algorithm assumes that Q is implemented as a binary min-heap for a set C of 'n' characters, the initialization of Q can be performed in $O(n)$ time using the BUILD – MIN HEAP procedure.

Each heap operation requires $O(\log n)$ time, and this will be performed exactly $(n - 1)$ times, it contributes to $O(n \log n)$ running time. Thus the total running time of HUFFMAN on a set of 'n' characters is $O(n \log n)$.

Task-Scheduling Problem

This is the problem of optimally scheduling unit – time tasks on a single processor, where each task has a deadline, along with a penalty that must be paid if the deadline is missed. The problem looks complicated, but it can be solved in simple manner using a greedy algorithm.

- A unit – time task is a job, such as a program to be run on a computer, that requires exactly one unit of time to complete.
- Given a finite set S of unit – time tasks, a schedule for S is a permutation of S specifying the order in which these tasks are to be performed.
- The first task in the schedule begins at time '0' and finishes at time 1, the second task begins at time 1 and finishes at time 2, and so on
- The problem of scheduling unit – time tasks with deadlines and penalties for a single processor has the following inputs:

  1. A set $S = \{a_1, a_2, \ldots an\}$ of n unit – time tasks:
  2. A set of n integer deadlines $d_1, d_2, \ldots d_n$ such that each d_i satisfies $1 \le d_i \le n$ and task a_i is supposed to finish by time d_i.
  3. A set of n non-negative weights or penalties w_1, $w_2, \ldots w_n$, such that we incur a penalty of w_i if task a_i is not finished by time d_i and we incur no penalty if a task finishes by its deadline.

Example: Consider the following 7 tasks, T_1, T_2, T_3, T_4, T_5 T_6, T_7. Every task is associated with profit and deadline.

Tasks	T_1	T_2	T_3	T_4	T_5	T_6	T_7
Deadline	4	2	4	3	1	4	5
Profit	75	65	55	45	40	35	30

45	65	55	75	30			
T_4	T_2	T_3	T_1	T_7			
0	1	2	3	4	5	6	7

T_1 has highest profit, so it will be executed first and the deadline of T_1, is '4' so T_1 has to be executed within 4 slots of time, same procedure is applied to other tasks also.

The tasks which are not executed by CPU are T_5 and T_6.

Profit: sum up the profits made by executing the tasks. Profit = 45 + 65 + 55 + 75 + 30 = 270

Analysis: We can use a greedy algorithm to find a maximum weight independent set of tasks. We can then create an optimal schedule having the tasks in A as its early tasks.

This method is an efficient algorithm for scheduling unit – time tasks with deadlines and penalties for a single processor. The running time is $O(n^2)$ using GREEDY METHOD, since each of the $O(n)$ independent checks made by that algorithm takes time $O(n)$.

Sorting and Order Statistics

Minimum and Maximum

This algorithm determines, how many comparisons are necessary to find minimum or maximum of a set of 'n' elements. Usually we can obtain maximum or minimum, by performing

$(n-1)$ comparisons; examine each element of the set in turn and keep track of the smallest element seen so far.

Consider the following procedure.

Assume that the set of elements reside in an array A where length $[A] = n$

```
MINIMUM (A)
Min ← A[1]
For i ←2 to length [A]
Do if min > A [i]
Then min ← A [i]
Return min.
```

Simultaneous minimum and maximum

In some applications, we must find both the minimum and the maximum of a set of 'n' elements.

We can find the minimum and maximum independently using $(n-1)$ comparisons for each, for a total of $(2n-2)$ comparisons.

- In fact, atmost $3\lfloor n/2 \rfloor$ comparisons are sufficient to find both the minimum and the maximum.
- The strategy is to maintain the minimum and maximum elements seen so far.
- Rather than processing each element of the input by comparing it against the current minimum and maximum at a cost of 2 comparisons per element, we process elements in pairs.
- Compare pairs of elements from the input with each other, and then we compare smaller to the current minimum and the larger to the current maximum, at a cost of 3 comparisons for every 2 elements.
- Setting up initial values for the current minimum and maximum depends on whether 'n' is odd or even. If 'n' is odd, we set both the minimum and maximum to the value of the first element, and then we process the rest of the elements in pairs.
- If 'n' is even, we perform 1 comparison on the first 2 elements to determine the initial values of the minimum and maximum and then process the rest of the elements in pairs.

Analysis: If 'n' is odd the total number of comparisons would be $3\lfloor n/2 \rfloor$.

If 'n' is even, we need 1 initial comparison followed by $\frac{3(n-2)}{2}$ comparisons, for a total of $\frac{3n}{2}-2$.

$\therefore$ The total number of comparisons is atmost $3\lfloor n/2 \rfloor$

Graph Algorithms

Single Source Shortest Path

In a shortest-path problem, we are given a weighted directed graph $G = (V, E)$ with weight function $W : E \to R$ mapping edges to real-valued weights. The weight of path $P = < V_O, V_1 \ldots V_K >$ is the sum of the weights of its constituent edges. Shortest-path weight from U to V is defined by

$$\delta(U,V) = \begin{cases} \{\min\{W(P): u \to v\} & \text{if there is a path from 'U' to 'V'} \\ \infty & \text{otherwise} \end{cases}$$

Edge weights can be interpreted as metrics other than distances. They are often used to represent time, cost, penalties, loss, or any other quantity.

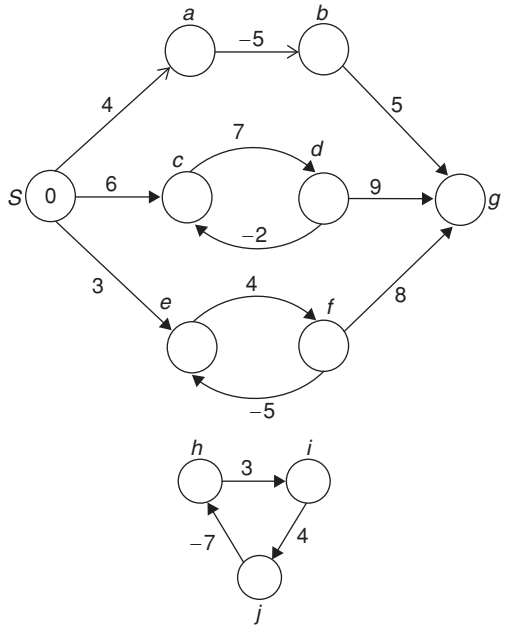
- The breadth first search algorithm is a shortest-path algorithm that works on un weighted graphs, that is, graphs in which each edge can be considered to have unit weight.

Negative-weight edges

Some of the instances of the single-source-shortest-paths problem, there may be edges whose weights are negative.

- If the graph $G = (V, E)$ contains no negative weight cycles reachable from source S, then for all $v \in V$, the shortest – path weight $d(S, V)$ remains well defined, even if it has a negative value.
- If there is a negative-weight cycle reachable from S, shortest-path weights are not well defined.
- No path from 'S' to a vertex on the cycle can be a shortest path - a lesser weight path can always be found that follows the proposed 'shortest' path and then traverses the negative-weight cycle.
- If there is a negative-weight cycle on some path form 'S' to 'V', we define $\delta(S,V) = -\infty$.

Example: Consider the following graph, calculate the shortest distance to all vertices from sources 'S'.



Solution:

- Shortest path from S to a is $\delta(S, a) = W(S, a) = 4$ (because there is only one path from 'S' to 'a')
- Similarly, there is only one path from 's' to 'b' $\delta(S, a) = W(S, a) + W(a, b) = 4 + (-5) = -1$
- Shortest-path from 's' to 'c'

There are infinitely many paths from 'S' to 'c'

1. $<S, c>$
2. $<S, c, d, c>$
3. $<S, c, d, c, d, c>$ and so on

$\delta <S, c> = 6$

$\delta (S, d, d, c) = 6 + 7 - 2 = 11$

$\delta (S, c, d, c, d, c) = 6 + 7 - 2 + 7 - 2 = 16$

$\delta (S, c, d, c, d, c, d, c)$

$= 6 + 7 - 2 + 7 - 2 + 7 - 2 = 21$

The cycle $<c, d, c>$ has weight $= 7 + (-2) = 5 > 0$

The shortest path from 'S' to 'c' is $<s, c>$ with weight $\delta (S, c) = 6$ similarly, the shortest-path from 'S' to 'd' is $<s, c, d>$, with weight $\delta (S, d) = w (S, c) + W(c, d) = 13$

May there are infinitely paths from 'S' to 'e'

1. $<s, e>$
2. $<s, e\, f, e>$
3. $<s, e, f, e, f, e>$ and so on

Since the cycle $<e, f, e>$ has weight $4 + (-5) = -1 < 0$. However, there is no shortest path from 'S' to 'e' by traversing the negative-weight cycle $<e, f, e>$ arbitrarily many times, we can find paths from 's' to 'e' with arbitrarily large negative weights,

So $\delta(S, e) = -\infty$

Similarly, $\delta(S, f) = -\infty$

- The shortest path from 'S' to 'g':
  'g' can be reachable from 'f'; we can also find paths with arbitrarily large negative weights from 's' to 'g' and $\delta(s, g) = -\infty$
- Vertices 'h', 'i' and 'j' also form a negative - weight cycle. They are not reachable from 'S' so, $\delta(S, h) = \delta(S, i) = \delta(S, j) = \infty$

Dijkstra's Algorithm

Dijkstra's algotithm solves the single-source shortest-path problem on a weighted, directed graph $G = (V, E)$, for the case in which all edge weights are non-negative.

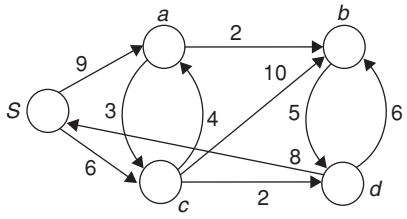
- The running time of Dijkstra's algorithm is lower than that of the Bellman–Ford algorithm.
- Dijkstra's algorithm maintains a set 's' of vertices whose final shortest-path weights from the source 'S' have already been determined.
- The algorithm repeatedly selects the vertex $u \in (V - S)$ with the minimum shortest-path estimate, adds 'u' to 'S'

DIJKSTRA (G, W, S)
INITIALIZE – SINGLE – SOURCE (G, S)
$S \leftarrow \varnothing$
$S \leftarrow V[G]$
While $Q \neq 0$
do $u \leftarrow$ EXTRACT – MIN(Q)
$S \leftarrow S\ U\{u\}$
For each vertex $v \in$ Adj[u]
do RELAX (u, v, w)

The algorithm maintains the invariant that $Q = V - S$ at the start of each iteration of the while loop. Initially the min – priority queue Q contains all the vertices in V. ($\because S = \varnothing$). Each time through the while loop, a vertex 'u' is extracted from $Q = V - S$ and added to set S.

- Each vertex is extracted from Q and added to S exactly once, so the contents of while loop will be executed exactly $|v|$ times.
- Dijkstra's algorithm always chooses the 'closest' or 'lightest' vertex in $(V - S)$ to add to set S, we say that it uses a greedy strategy.

Example: Consider the following graph, what is the shortest path?



Solution:

S	V – S
S	a b c d
S c	a b d
S c d	a b
S c d a	b
S c d a b	∅

Distance from S to all vertices of $(V - S)$

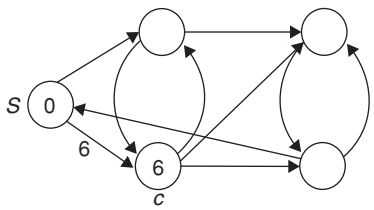
$d[a] = 9$

$d[b] = \infty$

$d\ [c] = 6$

$d[d] = \infty$

9, ∞, 6, ∞ values are given to MIN-PRIORITY Queue 'Q', '6' is returned.

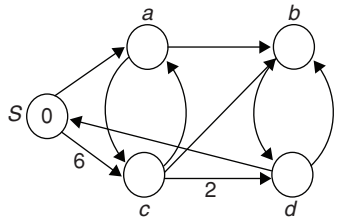


Distance from [Sc] to all vertices of $(V - S)$

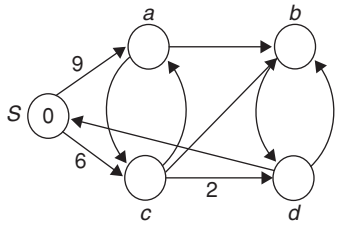
$d[b] = (S - c - b) = 6 + 10 = 16$

$d[a] = \min\{(S - a) = 9, (s - c - a) = 10\} = 9$

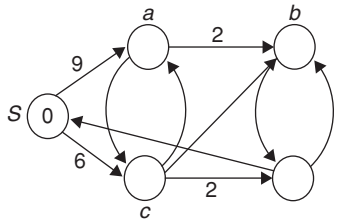
$d[d] = \min\{\infty, (S - c - d) = 6 + 2 = 8\ \} = 8$



Distance from [*s c d*] to [*ab*]
$d[a] = \min\{9, (S - c - d - S - a) = 25\} = 9$
$d[b] = \min\{16, (S - c - d - b) = 14\} = 14$



$d[a] = \min\{14, (s - a - b) = 9 + 2 = 11\} = 11$



Analysis: It maintains the min-priority queue '*Q*' by calling three priority-queue operations: INSERT, EXTRACT-MIN, and DECREASE-KEY. We maintain the min-priority queue by taking the vertices being numbered 1 to |*v*|. We store *d*[*v*] in the vth entry of an array. Each INSERT and DECREASE-KEY operation takes $O(1)$ time, and each EXTRACT- MIN operation takes $O(v)$ time ($\therefore$ we have to search through the entire array) for a total time of $O(v^2 + E) = O(v^2)$.

- If we implement the min - priority queue with a binary min-heap. Each EXTRACT-MIN operation takes time $O(\log V)$, there are $|V|$ such operations.
- The time to build the binary min-heap is $O(v)$. Each DECREASE-KEY operation takes time $O(\log V)$, and there are still atmost $|E|$ such operations. The total running time is $O((V + E) \log V)$, which is $O(E \log V)$ if all vertices are reachable form the source.
- We can achieve a running time of $O(V \log V + E)$ by implementing the min-priority queue with a Fibonacci heap.

Bellman–Ford Algorithm

Bellman–Ford algorithm solves the single-source shortest-path problems in the case in which edge weights may be negative.

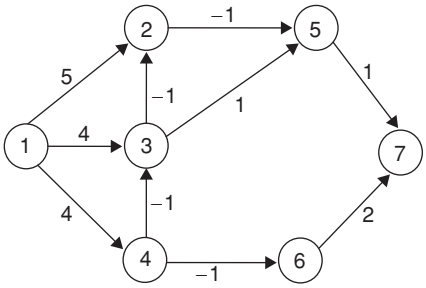
- When negative edge lengths are permitted, we require that the graph have no cycles of negative length. This is necessary to ensure that shortest-paths consist of a finite number of edges.
- When there are no cycles of negative length, there is a shortest-path between any two vertices of an *n*-vertex graph that has atmost $(n - 1)$ edges on it.
- A path that has more than $(n - 1)$ edges must repeat atleast one vertex and hence must contain a cycle
- Let $\text{dist}^x[u]$ be the length of a shortest-path from the source vertex '*v*' to vertex '*u*' under the constraint that the shortest-path contains atmost '*x*' edges. Then $\text{dist}'[u] = \text{cost}[v, u]$ $1 \le u \le n$ when there are no cycles of negative length we can limit our search for shortest-paths to paths with at most $(n - 1)$ edges. Hence, $\text{dist}^{n-1}[u]$ is the length of an unrestricted shortest-path from '*v*' to '*u*'.

The Recurrence Relation for dist is:

$$\text{Dist}^k[u] = \min\{\text{dist}^{k-1}[u], \min\{\text{dist}^{k-1}[i] + \text{cost}[i, u]\}\}$$

This recurrence can be used to compute dist^k from dist^{k-1}, for $k = 2, 3, \ldots, n - 1$.

Example: Consider the given directed graph



Find the shortest path from vertex '1' to all other vertices using Bellman–Ford algorithm?

Solution: Source vertex is '1' the distance from '1' to '1' in all '6' iterations will be zero. Since the graph has '7' vertices, the shortest-path can have atmost '6' edges. The following figure illustrates the implementation of Bellman–Ford algorithm:

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

Analysis

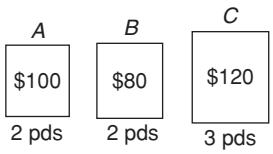
- Each iteration takes $O(n^2)$ time if adjacency matrices are used and $O(e)$ time if adjacency lists are used. Here '*e*' is the number of edges in the graph.
- The time complexity is $O(n^3)$ when adjacency matrices are used and $O(N * E)$ when adjacency lists are used.

EXERCISES

Practice Problems I

Directions for questions 1 to 14: Select the correct alternative from the given choices.

1. A thief enters a store and sees the following:



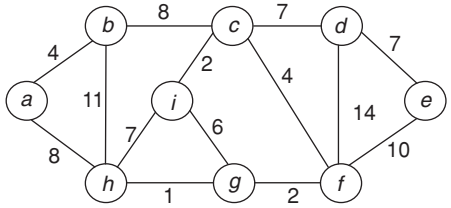
His knapsack can hold 4 pounds, what should he steal to maximize profit? (Use 0–1 Knapsack).

(A) A and B (B) A and C
(C) B and C (D) A, B and C

2. By using fractional Knapsack, calculate the maximum profit, for the data given in the above question?

(A) 180 (B) 170
(C) 160 (D) 150

3. Consider the below figure:



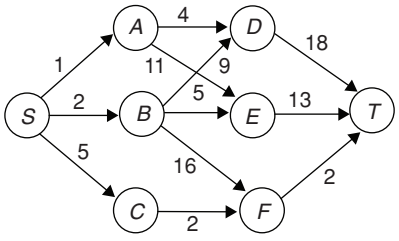
What is the weight of the minimum spanning tree using Kruskals algorithm?

(A) 34 (B) 35
(C) 36 (D) 38

4. Construct a minimum spanning tree for the figure given in the above question, using prim's algorithm. What are the first three nodes, added to the solution set respectively (consider 'a' as starting node).

(A) b, c, i (B) h, b, c
(C) c, i, b (D) h, c, b

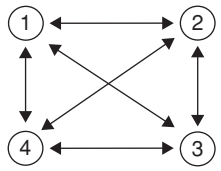
5. Consider the below graph, calculate the shortest distance from 'S' to 'T'?



(A) 23 (B) 9
(C) 20 (D) 22

6. Solve the travelling salesman problem, with the given distances in the form of matrix of graph, which of the following gives optimal solution?

C	1	2	3	4
1	0	15	20	25
2	10	0	14	15
3	11	18	0	17
4	13	13	14	0



(A) 1 – 2 – 4 – 3 – 1 (B) 2 – 3 – 4 – 1 – 2
(C) 1 – 4 – 2 – 3 – 1 (D) 2 – 4 – 3 – 1 – 2

7. Calculate the maximum profit using greedy strategy, knapsack capacity is 50. The data is given below:

$n = 3$
$(w_1, w_2, w_3) = (10, 20, 30)$
$(p_1, p_2, p_3) = (60, 100, 120)$ (dollars)? (0/1 knapsack)

(A) 180 (B) 220
(C) 240 (D) 260

Common data for questions 8 and 9: Given that

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed length code word	000	001	010	011	100	101

8. Using Huffman code, find the path length of internal nodes.

(A) 8 (B) 100
(C) 100×8 (D) 100/8

9. Using above answer, external path length will be

(A) 18 (B) 108
(C) 8 (D) None of these

Common data for questions 10 and 11:

10.

Using 0–1 knapsack select a subset of the three items shown, whose weight must not exceed 50 kg. What is the value?

(A) 2220 (B) 2100
(C) 2600 (D) 2180

11. Which of the following gives maximum profit, using fractional knapsack?

(A) $x_1 = 1, x_2 = 1, x_3 = 0$ (B) $x_1 = 1, x_2 = 1, x_3 = 2/3$
(C) $x_1 = 1, x_2 = 0, x_3 = 1$ (D) $x_1 = 1, x_2 = 1, x_3 = 1/3$

12. Using dynamic programming find the longest common subsequence (LCS) in the given 2 sub sequences:

x [1, … , m]
y [1, … , n]
x : A B C B D A B
Y : B D C A B A

Find longest sequence sets common to both.
(A) (BDAB, BCAB, BCBA)
(B) (BADB, BCAB, BCBA)
(C) (BDAB, BACB, BCBA)
(D) (BDAB, BCAB, BBCA)

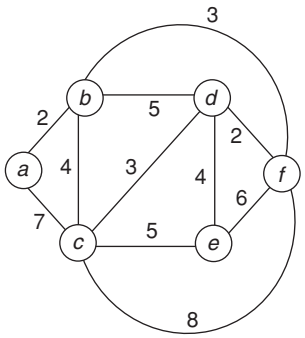
13. Let C_1, C_2, C_3, C_4 represent coins.
$C_1 = 25$ paisa
$C_2 = 10$ paisa
$C_3 = 5$ paisa
$C_4 = 1$ paisa
To represents 48 paisa, what is the minimum number of coins used, using greedy approach?
(A) 6 (B) 7
(C) 8 (D) 9

14. Worst-case analysis of hashing occurs when
(A) All the keys are distributed
(B) Every key hash to the same slot
(C) Key values with even number, hashes to slots with even number
(D) Key values with odd number hashes to slots with odd number.

Practice Problems 2

Directions for questions 1 to 15: Select the correct alternative from the given choices.

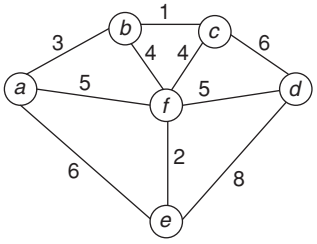
1. Consider the given graph:



Which one of the following cannot be the sequence of edges added, in that order, to a minimum spanning tree using Kruskal's algorithm?
(A) $(a-b), (d-f), (b-f), (d-c), (d-e)$
(B) $(a-b), (d-f), (d-c), (b-f), (d-e)$
(C) $(d-f), (a-b), (d-c), (b-f), (d-e)$
(D) $(d-f), (a-b), (b-f), (d-e), (d-c)$

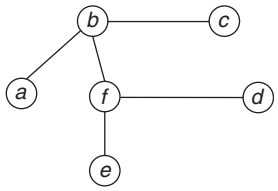
2. The worst case height analysis of B-tree is
(A) $O(n)$
(B) $O(n^2)$
(C) $O(\log n)$
(D) $O(n \log n)$

3. Consider the given graph:

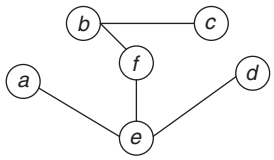


Which of the following is the minimum spanning tree. (If we apply Kruskal algorithm).

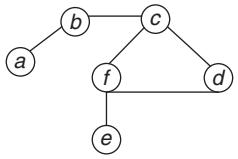
(A)



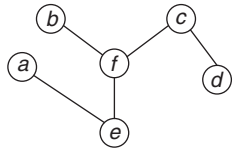
(B)



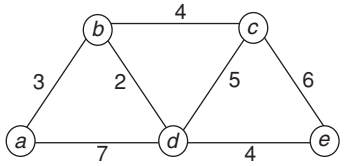
(C)



(D)



4. Consider the following graph:



Find the shortest path using Dijkstra's algorithm.
(A) a – b – d – e (B) a – b – c – d
(C) a – c – d – e (D) a – b – c – e

5. Which statement is true about Kruskal's algorithm?
(A) It is a greedy algorithm for the minimum spanning tree problem.
(B) It constructs spanning tree by selecting edges in increasing order of their weights.
(C) It does not accept creation of cycles in spanning tree.
(D) All the above

6. Dijkstra's algorithm bears similarity to which of the following for computing minimum spanning trees?
(A) Breadth first search (B) Prim's algorithm
(C) Both (A) and (B) (D) None of these

7. Which of the following algorithm always yields a correct solution for a graph with non-negative weights to compute shortest paths?
(A) Prim's algorithm (B) Kruskal's algorithm
(C) Dijkstra's algorithm (D) Huffman tree

8. Let the load factor of the hash table is number of keys is n, cells of the hash table is m then

(A) $\propto = n/m$ (B) $\propto = m/n$
(C) $\propto \frac{m+1}{n}$ (D) $\propto \frac{n+1}{m}$

9. To implement Dijkstra's shortest path algorithm on unweighted graphs so that it runs in linear time, the data structure to be used is:
(A) Queue
(B) Stack
(C) Heap
(D) B-tree

10. The development of a dynamic-programming algorithm can be broken into a sequence of four steps, which are given below randomly.
I. Construct an optimal solution from computed information.
II. Compute the value of an optimal solution in a bottom-up fashion.
III. Characterize the structure of an optimal solution.
IV. Recursively defines the value of an optimal solution.

The correct sequence of the above steps is
(A) I, II, III, IV (B) IV, III, I, II
(C) IV, II, I, III (D) III, IV, II I

11. Let V stands for vertex, E stands for edges.

For both directed and undirected graphs, the adjacency list representation has the desirable property that the amount of memory required is
(A) $\theta(V)$ (B) $\theta(E)$
(C) $\theta(V+E)$ (D) $\theta(V-E)$

12. Which of the following is false?
(A) Adjacency-matrix representation of a graph permits faster edge look up.
(B) The adjacency matrix of a graph requires $\theta(v^2)$ memory, independent of the number of edges in the graph.
(C) Adjacency-matrix representation can be used for weighted graphs.
(D) All the above

13. Dynamic programming is a technique for solving problems with
(A) Overlapped sub problems
(B) Huge size sub problems
(C) Small size sub problems
(D) None of these

14. The way a card game player arranges his cards, as he picks them up one by one is an example of _____.
(A) Bubble sort (B) Selection sort
(C) Insertion sort (D) None of the above

15. You want to check whether a given set of items is sorted. Which method will be the most efficient if it is already in sorted order?
(A) Heap sort (B) Bubble sort
(C) Merge sort (D) Insertion sort

Previous Years' Questions

Data for question 1: We are given 9 tasks T_1, T_2 … T_9. The execution of each task requires one unit of time. We can execute one task at a time. Each task T_i has a profit P_i and a deadline D_i. Profit P_i is earned if the task is completed before the end of the D_ith unit of time.

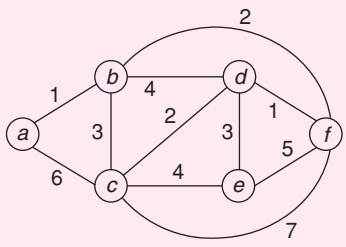
Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
Profit	15	20	30	18	18	10	23	16	25
Deadline	7	2	5	3	4	5	2	7	3

1. What is the maximum profit earned? **[2005]**
(A) 147 (B) 165
(C) 167 (D) 175

2. Consider a weighted complete graph G on the vertex set $\{v_1, v_2, \ldots, v_n\}$ such that the weight of the edge (v_i, v_j) is $2|i-j|$. The weight of the minimum spanning tree is: **[2006]**
(A) $n-1$ (B) $2n-2$
(C) $\binom{n}{2}$ (D) n^2

3. To implement Dijkstra's shortest path algorithm on unweighted graphs so that it runs in linear time, the data structure to be used is: **[2006]**
(A) Queue
(B) Stack
(C) Heap
(D) B-Tree

4. Consider the following graph: **[2006]**



Which one of the following cannot be the sequence of edges added, in that order, to a minimum spanning tree using Kruskal's algorithm?

(A) (a – b), (d – f), (b – f), (d – c), (d – e)
(B) (a – b), (d – f), (d – c), (b – f), (d – e)
(C) (d – f), (a – b), (d – c), (b – f), (d – e)
(D) (d – f), (a – b), (b – f), (d – e), (d – c)

Common data for questions 5 and 6: A 3-ary max-heap is like a binary max-heap, but instead of 2 children, nodes have 3 children. A 3-ary heap can be represented by an array as follows: The root is stored in the first location, $a[0]$, nodes in the next level, from left to right, is stored from $a[1]$ to $a[3]$. The nodes from the second level of the tree from left to right are stored from $a[4]$ location onward. An item x can be inserted into a 3-ary heap containing n items by placing x in the location $a[n]$ and pushing it up the tree to satisfy the heap property.

5. Which one of the following is a valid sequence of elements in an array representing 3-ary max-heap? **[2006]**

(A) 1, 3, 5, 6, 8, 9 (B) 9, 6, 3, 1, 8, 5
(C) 9, 3, 6, 8, 5, 1 (D) 9, 5, 6, 8, 3, 1

6. Suppose the elements 7, 2, 10 and 4 are inserted, in that order, into the valid 3-ary max-heap found in the above question, Q-76. Which one of the following is the sequence of items in the array representing the resultant heap? **[2006]**

(A) 10, 7, 9, 8, 3, 1, 5, 2, 6, 4
(B) 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
(C) 10, 9, 4, 5, 7, 6, 8, 2, 1, 3
(D) 10, 8, 6, 9, 7, 2, 3, 4, 1, 5

7. In an unweighted, undirected connected graph, the shortest path from a node S to every other node is computed most efficiently, in terms of *time complexity*, by **[2007]**

(A) Dijkstra's algorithm starting from S.
(B) Warshall's algorithm
(C) Performing a DFS starting from S.
(D) Performing a BFS starting from S.

8. A complete n-ary tree is a tree in which each node has n children or no children. Let I be the number of internal nodes and L be the number of leaves in a complete n-ary tree. If $L = 41$, and $I = 10$, what is the value of n? **[2007]**

(A) 3 (B) 4
(C) 5 (D) 6

9. Consider the following C program segment where CellNode represents a node in a binary tree: **[2007]**

```
struct CellNode {
  struct CellNode *leftChild;
     int element;
   struct CellNode *rightChild;
};
int GetValue (struct CellNode *ptr) {
   int value = 0;
   if (ptr != NULL) {
       if ((ptr->leftChild == NULL) &&
            (ptr->rightChild == NULL))
          value = 1;
   else
  value = value + GetValue(ptr->leftChild)
            + GetValue(ptr->rightChild);
     }
  return(value);
```

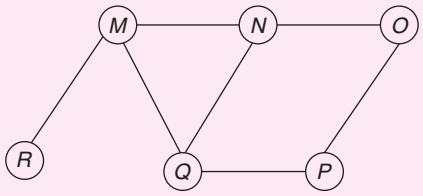
The value returned by GetValue when a pointer to the root of a binary tree is passed as its argument is:

(A) The number of nodes in the tree
(B) The number of internal nodes in the tree
(C) The number of leaf nodes in the tree
(D) The height of the tree

10. Let w be the minimum weight among all edge weights in an undirected connected graph. Let e be a specific edge of weight w. Which of the following is FALSE? **[2007]**

(A) There is a minimum spanning tree containing e.
(B) If e is not in a minimum spanning tree T, then in the cycle formed by adding e to T, all edges have the same weight.
(C) Every minimum spanning tree has an edge of weight w.
(D) e is present in every minimum spanning tree.

11. The Breadth first search algorithm has been implemented using the queue data structure. One possible order of visiting the nodes of the following graph is **[2008]**

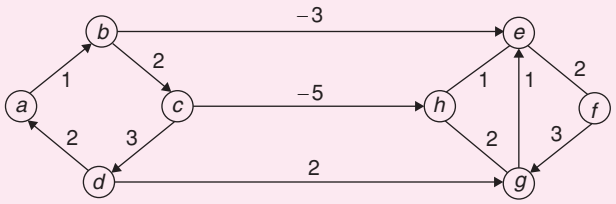


(A) MNOPQR (B) NQMPOR
(C) QMNPRO (D) QMNPOR

12. G is a graph on n vertices and $2n - 2$ edges. The edges of G can be partitioned into two edge-disjoint spanning trees. Which of the following is NOT true for G? **[2008]**

(A) For every subset of k vertices, the induced subgraph has atmost $2k - 2$ edges
(B) The minimum cut in G has atleast two edges
(C) There are two edge-disjoint paths between every pair of vertices
(D) There are two vertex-disjoint paths between every pair of vertices

13.



Dijkstra's single source shortest path algorithm when run from vertex a in the above graph, computes the correct shortest path distance to **[2008]**

(A) Only vertex *a*
(B) Only vertices *a*, *e*, *f*, *g*, *h*
(C) Only vertices *a*, *b*, *c*, *d*
(D) all the vertices

14. You are given the post-order traversal, *P*, of a binary search tree on the n elements 1, 2,…, *n*. You have to determine the unique binary search tree that has *P* as its post-order traversal. What is the time complexity of the most efficient algorithm for doing this? **[2008]**

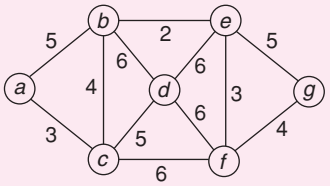
(A) $\Theta(\log n)$
(B) $\Theta(n)$
(C) $\Theta(n \log n)$
(D) None of the above, as the tree cannot be uniquely determined

15. Which of the following statement(*s*) is/are correct regarding Bellman–Ford shortest path algorithm? **[2009]**

P. Always finds a negative weighted cycle, if one exists.
Q. Finds whether any negative weighted cycle is reachable from the source.

(A) P only (B) Q only
(C) Both P and Q (D) Neither P nor Q

16. Consider the following graph: **[2009]**



Which one of the following is NOT the sequence of edges added to the minimum spanning tree using Kruskal's algorithm? **[2009]**

(A) (b, e) (e, f) (a, c) (b, c) (f, g) (c, d)
(B) (b, e) (e, f) (a, c) (f, g) (b, c) (c, d)
(C) (b, e) (a, c) (e, f) (b, c) (f, g) (c, d)
(D) (b, e) (e, f) (b, c) (a, c) (f, g) (c, d)

Common data for questions 17 and 18: Consider a binary max-heap implemented using an array.

17. Which one of the following array represents a binary max-heap? **[2009]**

(A) {25, 12, 16, 13, 10, 8, 14}
(B) {25, 14, 13, 16, 10, 8, 12}
(C) {25, 14, 16, 13, 10, 8, 12}
(D) {25, 14, 12, 13, 10, 8, 16}

18. What is the content of the array after two delete operations on the correct answer to the previous question? **[2009]**

(A) {14, 13, 12, 10, 8} (B) {14, 12, 13, 8, 10}
(C) {14, 13, 8, 12, 10} (D) {14, 13, 12, 8, 10}

Common data for questions 19 and 20: Consider a complete undirected graph with vertex set {0, 1, 2, 3, 4}. Entry W_{ij} in the matrix W below is the weight of the edge $\{i, j\}$.

$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$

19. What is the minimum possible weight of a spanning tree *T* in this graph such that vertex 0 is a leaf node in the tree *T*? **[2010]**

(A) 7 (B) 8
(C) 9 (D) 10

20. What is the minimum possible weight of a path *P* from vertex 1 to vertex 2 in this graph such that *P* contains at most 3 edges? **[2010]**

(A) 7 (B) 8
(C) 9 (D) 10

Common data for questions 21 and 22: A hash table of length 10 uses open addressing with hash function $h(k) = k \bmod 10$, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below:

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

21. Which one of the following choices gives a possible order in which the key values could have been inserted in the table? **[2010]**

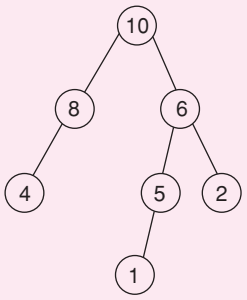
(A) 46, 42, 34, 52, 23, 33
(B) 34, 42, 23, 52, 33, 46
(C) 46, 34, 42, 23, 52, 33
(D) 42, 46, 33, 23, 34, 52

22. How many different insertion sequences of the key values using the same hash function and linear probing will result in the hash table shown above? **[2010]**

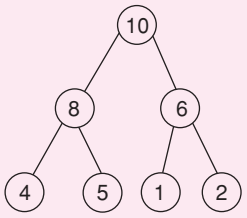
(A) 10 (B) 20
(C) 30 (D) 40

23. A max-heap is a heap where the value of each parent is greater than or equal to the value of its children. Which of the following is a max-heap? **[2011]**

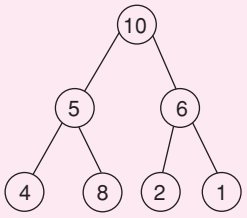
(A)



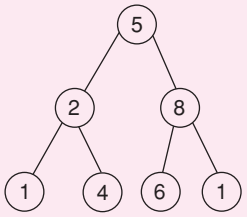
(B)



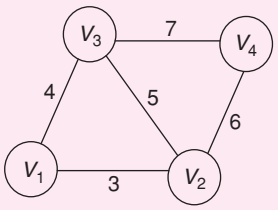
(C)



(D)



Common data for questions 24 and 25: An undirected graph $G(V, E)$ contains $n(n > 2)$ nodes named $V_1, V_2, \ldots, V_n$. Two nodes V_i, V_j are connected if and only if $0 < |i - j| \leq 2$. Each edge (V_i, V_j) is assigned a weight $i + j$. A sample graph with $n = 4$ is shown below.



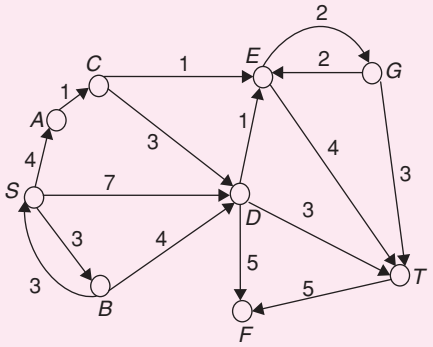
24. What will be the cost of the minimum spanning tree (MST) of such a graph with n nodes? **[2011]**

(A) $\frac{1}{12}(11n^2 - 5n)$ (B) $n^2 - n + 1$
(C) $6n - 11$ (D) $2n + 1$

25. The length of the path from V_5 to V_6 in the MST of previous question with $n = 10$ is **[2011]**

(A) 11 (B) 25
(C) 31 (D) 41

26. Consider the directed graph shown in the figure below. There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra's shortest path algorithm? Assume that, in any iteration, the shortest path to a vertex v is updated only when a strictly shorter path to v is discovered. **[2012]**



(A) SDT (B) SBDT
(C) SACDT (D) SACET

27. Let G be a weighted graph with edge weights greater than one and G^1 be the graph constructed by squaring the weights of edges in G. Let T and T^1 be the minimum spanning trees of G and G^1, respectively, with total weights t and t^1. Which of the following statements is **TRUE**? **[2012]**

(A) $T^1 = T$ with total weight $t^1 = t^2$
(B) $T^1 = T$ with total weight $t^1 < t^2$
(C) $T^1 \neq T$ but total weight $t^1 = t^2$
(D) None of the above

28. What is the time complexity of Bellman–Ford single-source shortest path algorithm on a complete graph of n vertices? **[2013]**

(A) $\Theta(n^2)$ (B) $\Theta(n^2 \log n)$
(C) $\Theta(n^3)$ (D) $\Theta(n^3 \log n)$

29. Consider the following operation along with Enqueue and Dequeue operations on queues, where k is a global parameter.

```
MultiDequeue(Q) {
m = k
```

```
while (Q is not empty) and (m > 0) {
        Dequeue (Q)
        m = m - 1
    }
}
```

What is the worst case time complexity of a sequence of n queue operations on an initially empty queue? **[2013]**

(A) $\Theta(n)$ (B) $\Theta(n + k)$
(C) $\Theta(nk)$ (D) $\Theta(n^2)$

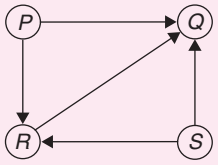
30. The preorder traversal sequence of a binary search tree is 30, 20, 10, 15, 25, 23, 39, 35, 42. Which one of the following is the post order traversal sequence of the same tree? **[2013]**

(A) 10, 20, 15, 23, 25, 35, 42, 39, 30
(B) 15, 10, 25, 23, 20, 42, 35, 39, 30
(C) 15, 20, 10, 23, 25, 42, 35, 39, 30
(D) 15, 10, 23, 25, 20, 35, 42, 39, 30

31. Let G be a graph with n vertices and m edges. What is the tightest upper bound on the running time of depth first search on G, when G is represented as an adjacency matrix? **[2014]**

(A) $\theta(n)$ (B) $\theta(n + m)$
(C) $\theta(n^2)$ (D) $\theta(m^2)$

32. Consider the directed graph given below. **[2014]**



Which one of the following is TRUE?

(A) The graph does not have any topological ordering.
(B) Both PQRS and SRQP are topological orderings.
(C) Both PSRQ and SPRQ are topological orderings.
(D) PSRQ is the only topological ordering.

33. There are 5 bags labeled 1 to 5. All the coins in a given bag have the same weight. Some bags have coins of weight 10 gm. Others have coins of weight 11 gm. I pick 1, 2, 4, 8, 16 coins respectively from bags 1 to 5. Their total weight comes out to 323 gm. Then the product of the labels of the bags having 11 gm coins is ___ **[2014]**

34. A priority queue is implemented as a max-heap. Initially it has 5 elements. The level-order traversal of the heap is : 10, 8, 5, 3, 2. Two new elements 1 and 7 are inserted into the heap in that order. The level-order traversal of the heap after the insertion of the elements is **[2014]**

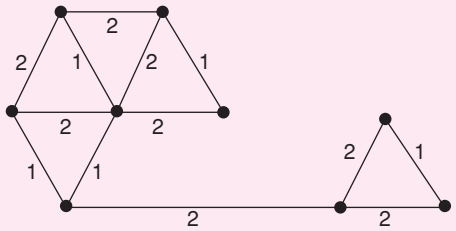
(A) 10, 8, 7, 3, 2, 1, 5 (B) 10, 8, 7, 2, 3, 1, 5
(C) 10, 8, 7, 1, 2, 3, 5 (D) 10, 8, 7, 5, 3, 2, 1

35. Consider the tree arcs of a BFS traversal from a source node W in an unweighted, connected, undirected graph. The tree T formed by the tree acrs is a data structure for computing **[2014]**

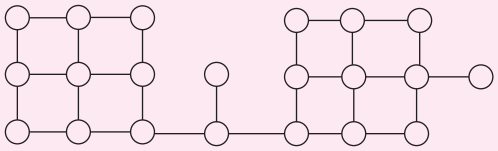
(A) The shortest path between every pair of vertices
(B) The shortest path from W to every vertex in the graph
(C) The shortest paths from W to only those nodes that are leaves of T.
(D) The longest path in the graph

36. The number of distinct minimum spanning trees for the weighted graph below is _____.

[2014]



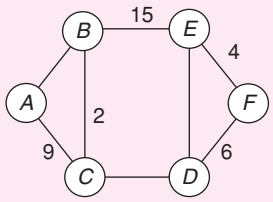
37. Suppose depth first search is executed on the graph below starting at some unknown vertex. Assume that a recursive call to visit a vertex is made only after first checking that the vertex has not been visited earlier. Then the maximum possible recursion depth (Including the initial call) is _____. **[2014]**



38. Suppose we have a balanced binary search tree T holding n numbers. We are given two numbers L and H and wish to sum up all the numbers in T that lie between L and H. suppose there are m such numbers in T. If the tightest upper bound on the time to compute the sum is $O(n^a \log^b n + m^c \log^d n)$, the value of $a + 10b + 100c + 1000d$ is ———.

[2014]

39. The graph shown below has 8 edges with distinct integer edge weights. The minimum spanning tree (MST) is of weight 36 and contains the edges: $\{(A, C), (B, C), (B, E), (E, F), (D, F)\}$. The edge weights of only those edges which are in the MST are given in the figure shown below. The minimum possible sum of weights of all 8 edges of this graph is _____ **[2015]**



40. Consider two decision problems Q_1, Q_2 such that Q_1 reduces in polynomial time to 3-SAT and 3-SAT reduces in polynomial time to Q_2. Then which one of the following is consistent with the above statement? **[2015]**

(A) Q_1 is in NP, Q_2 is NP hard.
(B) Q_2 is in NP, Q_1 is NP hard.
(C) Both Q_1 and Q_2 are in NP.
(D) Both Q_1 and Q_2 are NP hard.

41. Given below are some algorithms, and some algorithm design paradigms.

1. Dijkstra's Shortest Path	i. Divide and Conquer
2. Floyd-Warshall algorithm to compute all pairs shortest path	ii. Dynamic Programming
3. Binary search on a sorted array	iii. Greedy design
4. Backtracking search on a graph	iv. Depth-first search
	v. Breadth-first search

Match the above algorithms on the left to the corresponding design paradigm they follow.

(A) 1–i, 2–iii, 3–i, 4–v (B) 1–iii, 2–iii, 3–i, 4–v
(C) 1–iii, 2–ii, 3–i, 4–iv (D) 1–iii, 2–ii, 3–i, 4–v

42. A Young tableau is a 2D array of integers increasing from left to right and from top to bottom. Any unfilled entries are marked with ∞, and hence there cannot be any entry to the right of, or below a ∞. The following Young tableau consists of unique entries.

1	2	5	14
3	4	6	23
10	12	18	25
31	∞	∞	∞

When an element is removed from a Young tableau, other elements should be moved into its place so that the resulting table is still a Young tableau (unfilled entries maybe filled in with a ∞). The minimum number of entries (other than 1) to be shifted, to remove 1 from the given Young tableau is _____ **[2015]**

43. Which one of the following hash functions on integers will distribute keys most uniformly over 10 buckets numbered 0 to 9 for i ranging from 0 to 2020? **[2015]**

(A) $h(i) = i^2 \bmod 10$
(B) $h(i) = i^3 \bmod 10$
(C) $h(i) = (11 * i^2) \bmod 10$
(D) $h(i) = (12 * i) \bmod 10$

44. Let G be a weighted connected undirected graph with distinct positive edge weights. If every edge weight is increased by the same value, then which of the following statements is/are TRUE? **[2016]**

P : Minimum spanning tree of G does not change.

Q : Shortest path between any pair of vertices does not change.

(A) P only (B) Q only
(C) Neither P nor Q (D) Both P and Q

45. Let G be a complete undirected graph on 4 vertices, having 6 edges with weights being 1,2,3,4,5, and 6. The maximum possible weight that a minimum weight spanning tree of G can have is _____ . **[2016]**

46. G = (V,E) is an undirected simple graph in which each edge has a distinct weight, and e is a particular edge of G. Which of the following statements about the minimum spanning trees (MSTs) of G is/ are **TRUE**? **[2016]**

I. If e is the lightest edge of some cycle in G, then every MST of G includes e

II. If e is the heaviest edge of some cycle in G, then every MST of G excludes e

(A) I only (B) II only
(C) both I and II (D) neither I nor II

47. Breadth First Search (BFS) is started on a binary tree beginning from the root vertex. There is a vertex t at a distance four from the root. If t is the n-th vertex in this **BFS** traversal, then the maximum possible value of n is _____. **[2016]**

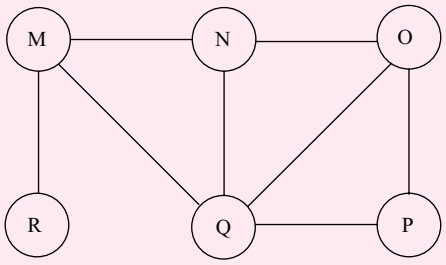
48. Let $G = (V,E)$ be *any* connected undirected edge-weighted graph. The weights of the edges in E are positive and distinct. Consider the following statements:

(I) Minimum spanning Tree of G is always unique.
(II) Shortest path between any two vertices of G is always unique.

Which of the above statements is/are necessarily true? **[2017]**

(A) (I) only
(B) (II) only
(C) both (I) and (II)
(D) neither (I) nor (II)

49. The Breadth First Search (BFS) algorithm has been implemented using the queue data structure. Which one of the following is a possible order of visiting the nodes in the graph below? **[2017]**



(A) MNOPQR
(B) NQMPOR
(C) QMNROP
(D) POQNMR

50. A message is made up entirely of characters from the set $X = \{P, Q, R, S, T\}$. The table of probabilities for each of the characters is shown below:

Character	Probability
P	0.22
Q	0.34
R	0.17
S	0.19
T	0.08
Total	1.00

If a message of 100 characters over X is encoded using Huffman coding, then the expected length of the encoded message in bits is _____. **[2017]**

51. Let G be a simple undirected graph. Let T_D be a depth first search tree of G. Let T_B be a breadth first search tree of G.

Consider the following statements.

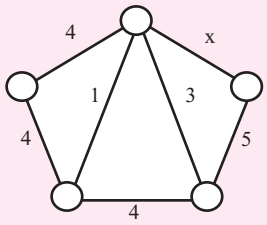
(I) No edge of G is a cross edge with respect to T_D. (A cross edge in G is between two nodes neither of which is an ancestor of the other in T_D.)

(II) For every edge (u, v) of G, if u is at depth i and v is at depth j in T_B, then $|i - j| = 1$.

Which of the statements above must necessarily be true? **[2018]**

(A) I only (B) II only
(C) Both I and II (D) Neither I nor II

52. Consider the following undirected graph G:



Choose a value for x that will maximize the number of minimum weight spanning trees (MWSTs) of G. The number of MWSTs of G for this value of x is _____. **[2018]**

Answer Keys

Exercises

Practice Problems 1

1. A **2.** A **3.** B **4.** A **5.** B **6.** A **7.** B **8.** A **9.** A **10.** C
11. B **12.** A **13.** A **14.** B

Practice Problems 2

1. D **2.** C **3.** A **4.** A **5.** D **6.** C **7.** C **8.** A **9.** A **10.** D
11. C **12.** C **13.** A **14.** C **15.** D

Previous Years' Questions

1. A **2.** B **3.** C **4.** D **5.** D **6.** A **7.** D **8.** C **9.** C **10.** B
11. C **12.** D **13.** D **14.** B **15.** B **16.** D **17.** C **18.** D **19.** D **20.** B
21. C **22.** C **23.** B **24.** B **25.** C **26.** D **27.** **28.** C **29.** A **30.** D
31. C **32.** C **33.** 12 to 12 **34.** A **35.** B **36.** 6 to 6 **37.** 19 **38.** 110 **39.** 69
40. A **41.** C **42.** 5 **43.** B **44.** A **45.** 7 **46.** B **47.** 31 **48.** A **49.** D
50. 225 **51.** A **52.** 4

Chapter 5

Dynamic Programming

LEARNING OBJECTIVES

- *Dynamic programming*
- *Multi-stage graph*
- *All pairs shortest path problem*
- *Hashing methods*
- *Mid-square method*
- *Folding method*
- *Resolving collisions*
- *Matrix chain multiplication*
- *Longest common subsequence*
- *Optimal substructure of LCS*
- *NP-hard and NP-complete*
- *P-problem*
- *NP-problem*
- *P, NP, and Co-NP*
- *Cooks theorem*
- *Non-deterministic search*

Dynamic Programming

Dynamic programming is a method for solving complex problems by breaking them down into simpler sub problems. It is applicable to problems exhibiting the properties of overlapping sub problems which are only slightly smaller, when applicable; the method takes far less time than naive method.

- The key idea behind dynamic programming is to solve a given problem, we need to solve different parts of the problem (sub problems) then combine the solutions of the sub problems to reach an overall solution. Often, many of these sub problems are the same.
- The dynamic programming approach seeks to solve each sub problem only once, thus reducing the number of computations. This is especially useful when the number of repeating sub problems is exponentially large.
- There are two key attributes that a problem must have in order for dynamic programming to be applicable 'optimal sub structure' and 'overlapping sub-problems'. However, when the overlapping problems are much smaller than the original problem, the strategy is called 'divide-and-conquer' rather than 'dynamic programming'. This is why merge sort-quick sort are not classified as dynamic programming problems.

Dynamic programming is applied for:
- Multi stage graph
- All pairs shortest path

Principle of Optimality

It states that whatever the initial state is, remaining decisions must be optimal with regard to the state following from the first decision.

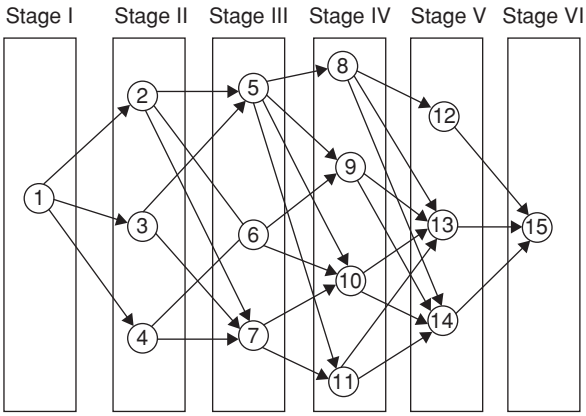
To solve a problem using dynamic programming strategy, it must observe the principle of optimality.

Multi-stage Graph

A multi-stage graph is a graph

- $G = (V, E)$ with V partitioned into $K >= 2$ disjoint subsets such that if (a, b) is in E, then a is in V_i, and b is in V_{i+1} for some sub sets in the partition;
- $|V_1| = |V_K| = 1$ the vertex S in V_1 is called the source; the vertex t is called the sink.
- G is usually assumed to be a weighted graph.
- The cost of a path from node V to node W is sum of the costs of edges in the path.
- The 'multi-stage graph problem' is to find the minimum cost path from S to t.

Example:



Costs of edges

1 – 2 → 10
1 – 3 → 20

1 – 4 → 30
2 – 5 → 10
2 – 6 → 20
2 – 7 → 30
3 – 5 → 40
3 – 7 → 50
4 – 6 → 40
4 – 7 → 30
5 – 8 → 10
5 – 9 → 20
5 – 10 → 30
5 – 11 → 40
6 – 9 → 20
6 – 10 → 30
7 – 10 → 30
7 – 11 → 20
8 – 12 → 10
8 – 13 → 20
8 – 14 → 30
9 – 13 → 20
9 – 14 → 10
10 – 13 → 10
10 – 14 → 20
11 – 13 → 10
11 – 14 → 30
12 – 15 → 20
13 – 15 → 10
14 – 15 → 30

Solution Using Backward Cost

Format: COST (Stage, node) = minimum cost of travelling to the node in stage from the source node (node 1)

Step I:

Cost (I, 1) = 0

Step II:

Cost (II, 2) = cost (I, 1) + cost (1, 2) = 0 + 10 = 10
Cost (II, 3) = cost (I, 1) + cost (1, 3) = 0 + 20 = 20
Cost (II, 4) = cost (I, 1) + cost (1, 4) = 0 + 30 = 30

Step III:

Cost (III, 5) = min {cost (II, 2) + cost (2, 5),
cost (II, 3) + cost (3, 5),
cost (II, 4) + cost (4, 5)
= min {10 + 10, 20 + 40, 30 + ∞}
= 20 → Via path 1 – 2 – 5

Cost (III, 6) = min {cost (II, 2) + cost (2, 6),
cost (II, 3) + cost (3, 6),
cost (II, 4) + cost (4, 6)}
= min {10 + 20, 20 + ∞, 30 + 40}
= 30 → via the path 1 – 2 – 6

Cost (III, 7) = min {cost(II, 2) + cost (2, 7),
Cost (II, 3) + cost (3, 7),
Cost (II, 4) + cost (4, 7)}
= min {10 + 30, 20 + 50, 30 + 30}
= 40 → Via the path 1 – 2 – 7

Step IV:

Cost (IV, 8) = min {cost (III, 5) + cost (5, 8),
Cost (III, 6) + cost (6, 8),
Cost (III, 7) + cost (7, 8)}
= min {20 + 10, 30 + ∞, 40 + ∞}
= 30 → Via path 1 – 2 – 5 – 8

Cost (IV, 9) = min {cost (III, 5) + cost (5, 9),
Cost (III, 6) + cost (6, 9),
Cost (III, 7) + cost (7, 9)}
= min {20 + 20, 30 + 20, 40 + ∞}
= 40 → Via the path 1 – 2 – 5 – 9

Cost (IV, 10) = min {cost (III, 5) + cost (5, 10),
Cost (III, 6) + cost (6, 10),
Cost (III, 7) + cost (7, 10}
= min {20 + 30, 30 + 30, 40 + 30}
= 50 → Via the path 1 – 2 – 5 – 10

Cost (IV, 11) = min {cost (III, 5) + cost (5, 11)
Cost (III, 6) + cost (6, 11),
Cost (III, 7) + cost (7, 11)}
= min {20 + 40, 30 + ∞, 40 + 20}
= 60 → Via the path 1 – 2 – 5 – 11
or Via the path 1 – 2 – 7 – 11

Step V:

Cost (V, 12) = min {cost (IV, 8) + cost (8, 12)
Cost (IV, 9) + cost (9, 12),
Cost (IV, 10) + cost (10, 12),
Cost (IV, 11) + cost (11, 12)}
= min {30 + 10, 40 + ∞, 50 + ∞, 60 + ∞}
= 40 → Via the path 1 – 2 – 5 – 8 – 12

Cost (V, 13) = min {cost (IV, 8) + cost (8, 13)
Cost (IV, 9) + cost (9, 13),
Cost (IV, 10) + cost (10, 13),
Cost (IV, 11) + cost (11, 13)}
= min {30 + 20, 40 + 20, 50 + 10, 60 + 10}
= 50 → Via the path 1 – 2 – 5 – 8 – 13

Cost (V, 14) = min {cost (IV, 8) + cost (8, 14)
Cost (IV, 9) + cost (9, 14),
Cost (IV, 10) + cost (10, 14),
Cost (IV, 11) + cost (11, 14)}
= min {30 + 30, 40 + 10, 50 + 20, 60 + 30}
50 → Via the path 1 – 2 – 5 – 9 – 14

Step VI:

$$\begin{aligned}\text{Cost (VI, 15)} &= \min \{\text{cost (V, 12)} + \text{cost (12, 15)},\\ &\quad \text{Cost (V, 13)} + \text{cost (13, 15)},\\ &\quad \text{Cost (V, 14)} + \text{cost (14, 15)}\}\\ &= \min \{40 + 20, 50 + 10, 50 + 30\}\\ &= 60 \rightarrow \text{Via the path } 1-2-5-8-13-15\\ &\quad \text{(or) } 1-2-5-8-12-15\end{aligned}$$

All Pairs Shortest Path Problem (Floyd–Warshall Algorithm)

A weighted graph is a collection of points (vertices) connected by lines (edges), where each edge has a weight (some real number) associated with it.

Example: A graph in the real world is a road map. Each location is a vertex and each road connecting locations is an edge. We can think of the distance travelled on a road from one location to another as the weight of that edge.

- The Floyd–Warshall algorithm determines the shortest path between all pairs of vertices in a graph.
- The vertices in a graph be numbered from 1 to n. Consider the subset $\{1, 2, \ldots K\}$ of these n vertices.
- Finding the shortest path from vertex i to vertex j that uses vertex in the set $\{1, 2, \ldots K\}$ only. There are two situations.

1. K is an intermediate vertex on the shortest path.
2. K is not an intermediate vertex on the shortest path.

In the first situation, we can break down our shortest path into two paths: i to K and then K to j. Note that all the vertices from i to K are from the set $\{1, 2, \ldots K-1\}$ and that all the intermediate vertices from K to j are from the set $\{1, 2, \ldots K-1\}$. Also in the second situation, we simply have that all intermediate vertices are from the set $\{1, 2, \ldots K-1\}$. Now define the function D for a weighted graph with the vertices $\{1, 2, \ldots n\}$ as follows.

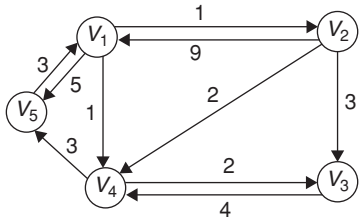
$D(i, j, K)$ = the shortest distance from vertex i to vertex j using the intermediate vertices. In the set $\{1, 2, \ldots K\}$

Using the above idea, we can recursively define the function D.

$$\begin{aligned}&D(i, j, K) = W(i, j) \text{ if } K = 0\\ &\min (D(i, j, K-1), D(i, K, K-1) + D(K, j, K-1)) \text{ if } K > 0\end{aligned}$$

- The first line says that if we do not allow intermediate vertices, then the shortest path between two vertices is the weight of the edge that connects them. If no such weightexists, we usually define this shortest path to be of length infinity.
- The second line pertains to allowing intermediate vertices. It says that the minimum path from i to j through vertices $\{1, 2, \ldots K\}$ is either the minimum path from i to j through vertices $\{1, 2, \ldots K-1\}$ OR the sum of the minimum path from vertex i to K through $\{1, 2, \ldots K-1\}$ plus the minimum path from vertex K to j through $\{1, 2, \ldots K-1\}$. Since this is the case, we compute both and choose the smaller of these.

Example:



The weight matrix will be

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

Let $D^{(K)}[i, j]$ = weight of a shortest path from v_i to v_j using only vertices from $\{v_1, v_2, \ldots v_k\}$ as intermediate vertices in the path.

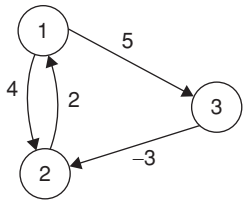
- $D^{(0)} = W$
- $D^{(n)} = D$ which is the goal matrix.

How to compute $D^{(K)}$ from $D^{(K-1)}$?

Case I: A shortest path from v_i to v_j restricted to using only vertices from $\{v_1, v_2, \ldots v_K\}$ as intermediate vertices does not use V_K.
Then $D^{(K)}[i, j] = D^{(K-1)}[i, j]$

Case II: A shortest path from v_i to v_j restricted to using only vertices from $\{v_1, v_2 \ldots v_K\}$ as intermediate vertices does use V_K. Then $D^{(K)}[i, j] = D^{(K-1)}[i, K] + D^{(K-1)}[K, j]$
Since $D^{(K)}[i, j] = D^{(K-1)}[i, j]$
or $D^{(K)}[i, j] = D^{(K-1)}[i, K] + D^{(K-1)}[K, j]$
We conclude: $D^{(K)}[i, j] = \min\{D^{(K-1)}[i, j], D^{(K-1)}[i, K] + D^{(K-1)}[K, j]\}$

Example: 1



$W = D^{\circ} =$

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	–3	0

$P =$

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

$K = 1$, vertex 1 can be intermediate node

$D^1[2, 3] = \min(D^0[2, 3], D^0[2, 1] + D^0[1, 3])$
$= \min(\infty, 7)$
$= 7$

$D^1[3, 2] = \min(D^0[3, 2], D^0[3, 1] + D^0[1, 2])$
$= \min(-3, \infty)$
$= -3$

$D^1 =$

	1	2	3
1	0	4	5
2	2	0	7
3	∞	–3	0

$P =$

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

$K = 2$, vertices 1, 2 can be intermediate nodes,

$D^2[1, 3] = \min(D'[1, 3], D'[1, 2] + D'[2, 3])$
$= \min(5, 4 + 7) = 5$

$D^2[3, 1] = \min(D'[3, 1], D'[3, 2] + D'[2, 1])$
$= \min(\infty, -3 + 2)$
$= -1$

$D^2 =$

	1	2	3
1	0	4	5
2	2	0	7
3	–1	–3	0

$P =$

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

$K = 3$ vertices 1, 2, 3 can be intermediate

$D^3[1, 2] = \min(D^2[1, 2], D^2[1, 3] + D^2[3, 2])$
$= \min(4, 5 + (-3))$
$= 2$

$D^3[2, 1] = \min(D^2[2, 1], D^2[2, 3] + D^2[3, 1])$
$= \min(2, 7 + (-1))$
$= 2$

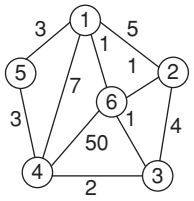
$D^3 =$

	1	2	3
1	0	2	5
2	2	0	7
3	–1	–3	0

$P =$

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

Example 2:



The final distance matrix and P

$D^6 =$

	1	2	3	4	5	6
1	0	2(6)	2(6)	4(6)	3	1
2	2(6)	0	2(6)	4(6)	5(6)	1
3	2(6)	2(6)	0	2	5(4)	1
4	4(6)	4(6)	2	0	3	3(3)
5	3	5(4)	5(4)	3	0	4(1)
6	1	1	1	3(3)	4(1)	0

The values in parenthesis are the non-zero P values.

Table 1 *Divide and conquer vs dynamic programming.*

1. This design strategy divides the problem into sub problems, conquer the each sub problem recursively, finally combine all the sub problem solutions, for the original problem.	1. This design strategy chooses an optimal solution for the problem, by recursively defining the value of optimal solution, these values are computed in bottom up fashion or top down fashion.
2. each sub problem is solved recursively, and consumes more time at each sub problem	2. Each sub problem is solved only once and is stored in table
3. Sub problems are independent of each other e.g., Binary search	3. The sub problems are dependent e.g., Traveling sales person problem

Dynamic Programming vs Greedy Method

The main difference between greedy method (*GM*) and dynamic programming (*DP*) methodology is, *DP* considers all possible solutions for the given problem and picks the optimal one. Where as greedy, considers only one set of solutions to the problem.

The other difference between *GM* and *DP* is that, *GM* considers the choice, which is best at that step, which is done at each level of the sub problem. That is, it won't reconsider its choice. The choices reflect only present, won't consider the future choices, where as *DP* tries out all the best alternatives and finds the optimal solution. It implements principle of optimality. At each stage of the problem, it decides based on the previous decision made in the previous stage.

Hashing Methods

Uniform Hash Function

If the keys, K, are integers randomly distributed in $[0, r]$ then hash function $H(K)$ is given as

$$H(K) = \left\lfloor \frac{mk}{r} \right\rfloor$$

$H(K)$ is a uniform hash function

Uniform hashing function should ensure

$$\Sigma_{K|h(K)=0} P(K) = \Sigma_{K|h(K)=1} P(K) = \cdots \Sigma_{K|h(K)=0} P(K) = \frac{1}{m}$$

$P(K)$ = probability that a key K, occurs that is the number of keys that map to each slot is equal.

Division method

Hashing an integer x is to divide x by M and then to use the remainder modulo M. This is called the division method of hashing. In this case the hash function is

$$\boxed{h(x) = x \bmod M}$$

Generally this approach is quite good for just about any value of M. However, in certain situations some extra care is needed in the selection of a suitable value for M. For example, it is often convenient to make M an even number. But this means that $h(x)$ is even if x is even, and $h(x)$ is odd of x is odd. If all possible keys are equiprobable, then this is not a problem. However, if say even keys are more likely than odd keys, the function $h(x) = x \bmod M$ will not spread the hashed values of those keys evenly.

- Let M be a power of two, i.e., $M = 2^k$ for some integer $k > 1$. In this case, the hash function $h(x) = x \bmod 2^k$ simply extracts the bottom k-bits of the binary representation of x. While this hash function is quite easy to compute, it is not a desirable function because it does not depend on all the bits in the binary representation of x.
- For these reasons M is often chosen to be a prime number. Suppose there is bias in the way the keys are created that makes it more likely for a key to be a multiple of some small constant, say two or three. Then making M a prime increases the likelihood that those keys are spread out evenly. Also if M is a prime number, the division of x by that prime number depends on all the bits of x, not just the bottom k-bits, for some small constant k.

Example: Hash table size = 10
Key value = 112
Hash function = $h(k) = k \bmod M$
$= 112 \bmod 10 = 2$

Disadvantage: A potential disadvantage of the division method is due to the property that consecutive keys map to consecutive hash values.

$h(i) = i$
$h(i + 1) = i + 1 \pmod M$
$h(i + 2) = i + 2 \pmod M$
.
.
.

While this ensures that consecutive keys do not collide, it does not mean that consecutive array locations will be occupied. We will see that in certain implementations this can lead to degradation in performance.

Multiplication method

A variation on the middle-square method that alleviates its deficiencies is called, multiplication hashing method. Instead of multiplying the key x by itself, we multiply the key by a carefully chosen constant 'a' and then extract the middle k bits from the result. In this case, the hashing function is

$$\boxed{h(x) = \left\lfloor \frac{M}{W}(ax \bmod W)) \right\rfloor}$$

if we want to avoid the problems that the middle-square method encounters with keys having a large number of leading (or) trailing zero's then we should choose an 'a' that has neither leading nor trailing zero's.

Furthermore, if we, choose an 'a' that is relatively prime to W, then there exists another number 'a' such that $aa' = 1 \pmod W$. Such a number has the nice property that if we take a key x, and multiply it by 'a' to get ax, we can recover the original key by multiplying the product again by a', since $a \times a' = aa'x = 1x$.

The multiplication method for creating a hash function operates in two steps:

Step 1: Multiply the key K by a constant A in the range $0 < A < 1$ and extract the fractional part of KA.
Step 2: Multiply this value by M and take the floor of the result.

In short the hash function is

$$h(k) = \lfloor M \cdot (KA \bmod 1) \rfloor$$

Where (KA mod 1) denotes the fractional part of KA, that is $KA \quad \lfloor KA \rfloor$

Example:
Let $m = 10000$, $K = 123456$ and $A = \dfrac{\sqrt{5}-1}{2}$

$= 0.618033$

Then $h(k) = \lfloor 10000 \cdot (123456 \cdot 0.61803 \bmod 1) \rfloor$

$= \lfloor 10000 \cdot (76300.00412 \bmod 1) \rfloor$

$= \lfloor 10000 \cdot 0.00412 \rfloor = 41$

Practical issues

- Easy to implement
  –On most machines multiplication is faster than division.
  –We can substitute one multiplication by shift operation.
  –We don't need to do floating-point operations.
- If successive keys have a large interval, $A = 0.6125423371$ can be recommended.

Mid-square method

A good hash function to use with integer key values is the mid-square method. The mid-square method squares the key value, and then takes out the middle '*r*' bits of the result, giving a value in the range 0 to $2^r - 1$. This works well because most (or) all bits of the key value contribute to the result.

Example:
Consider records whose keys are 4-digit numbers in base 10. The goal is to hash these key values to a table of size 100(i.e., *a* range of 0 to 99).

This range is equivalent to two digits in base 10.

That is $r = 2$. If the input is the number 4567, squaring yields an 8-digit number, 20857489. The middle two digits of this result are 57. All digits of the original key value (equivalently, all bits when the number is viewed in binary) contribute to the middle two digits of the squared value. Thus, the result is not dominated by the distribution of the bottom or the top digit of the original key value. Of course, if the key values all tend to be small numbers, then their squares will only affect the low order digits of the hash value.

Example: To map the key 3121 into a hash table of size 1000, we square it $(3121)^2 = 9740641$ and extract 406 as the hash value.

Folding method

The folding method breaks up a key into precise segments that are added to form a hash value, and still another technique is to apply a multiplicative hash function to each segment individually before folding.

Algorithm $H(x) = (a + b + c) \bmod m$. Where *a*, *b*, and *c* represent the preconditioned key broken down into three parts, *m* is the table size, and mod stands for modulo. In other words: The sum of three parts of the pre conditioned key is divided by the table size. The remainder is the hash key.

Example:
Fold the key 123456789 into a hash table of ten spaces (0 through 9)

We are given $x = 123456789$ and the table size (i.e., $m = 10$)

Since we can break *x* into three parts any way, we will break it up evenly.

Thus $a = 123$, $b = 456$ and $c = 789$

$H(x) = (a + b + c) \bmod M$

$H(123456789) = (123 + 456 + 789) \bmod 10$

$= 1368 \bmod 10 = 8$

123456789 are inserted into the table at address 8.

The folding method is distribution independent.

Resolving collisions In collision resolution strategy algorithms and data structures are used to handle two hash keys that hash to the same hash keys. There are a number of collision resolution techniques, but the most popular are open addressing and chaining.

- Chaining: An array of linked list, Separate chaining
- Open Addressing: Array based implementation:
  – Linear probing (Linear Search)
  – Quadratic probing (non-linear search)
  – Double hashing (use two hash functions)

Separate chaining Every linked list has each element that collides to the similar slot. Insertion need to locate the accurate slot and appending to any end of the list in that slot wherever, deletion needs searching the list and removal.

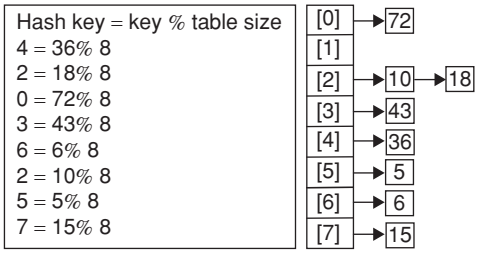


Figure 1 Separate chaining

Open addressing Open addressing hash tables are used to stock up the records straight inside the array. This approach is also known as closed hashing. This procedure is based on probing. Well known probe sequence include:

- Linear probing: In which the interval between probes is fixed often at 1.
- Quadratic probing: In which the interval between probes increases proportional to the hash value (the interval thus increasing linearly and the indices are described by a quadratic function).
- Double hashing: In which the interval between probes is computed by another hash function.

(i) Linear probing: Linear probing method is used for resolving hash collisions of values of hash functions by sequentially searching the hash table for a free location. The item will be stored in the next available slot in the table in linear probing. Also an assumption is made that the table is not already full.

This is implemented via a linear search for an empty slot, from the point of collision.

If the physical end of table is reached during the linear search, the search will again get start around to the beginning of the table and continue from there. The table is considered as full, if an empty slot is not found before reaching the point of collision.

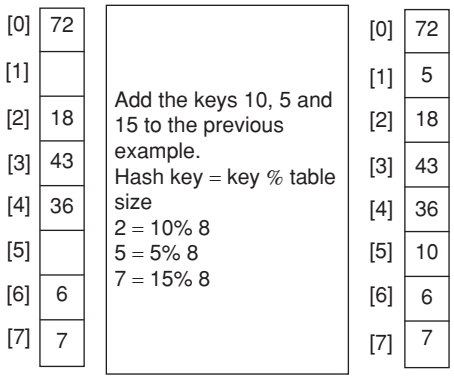
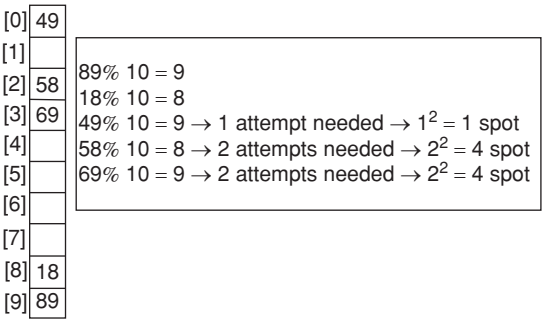


Figure 2 Linear probing

Limitation: A problem with linear probe method is primary clustering. In primary clustering blocks of data may possibly be able to form collision. Several attempts may be required by any key that hashes into the cluster to resolve the collision.

(ii) Quadratic probing: To resolve the primary clustering problem, quadratic probing can be used. With quadratic probing, rather than always moving one spot, move i^2 spots from the point of collision where i is the number of attempts needed to resolve the collision.



Limitation: Maximum half of the table can be used as substitute locations to resolve collisions. Once the table gets more than half full, its really hard to locate an unfilled spot. This new difficulty is recognized as secondary clustering because elements that hash to the same hash key will always probe the identical substitute cells.

(iii) Double hashing: Double hashing uses the idea of applying a second hash function to the key when a collision occurs, the result of the second hash function will be the number of positions from the point of collision to insert. There are some requirements for the second function:

1. It must never evaluate to zero
2. Must make sure that all cells can be probed.

A popular second hash function is:
Hash(key) = R-(Key mod R) where R is a prime number smaller than the size of the table.

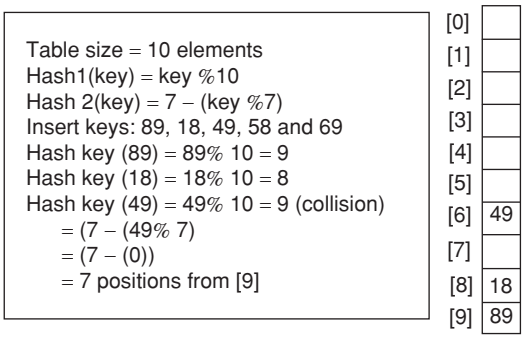


Figure 3 Double hashing

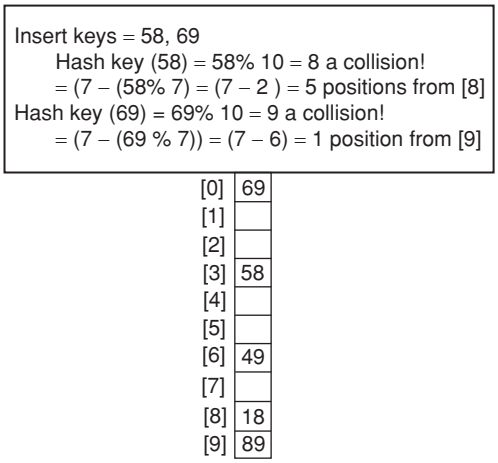


Figure 4 Double hashing

Matrix-chain Multiplication

We are given a sequence of n matrices $m_1, m_2 \ldots m_n$ to be multiplied. If the chain matrices is $< m_1, m_2, m_3, m_4>$, the product m_1, m_2, m_3, m_4 can be fully parenthesized in 5 distinct ways:

1. $(m_1 (m_2 (m_3 m_4)))$
2. $(m_1 ((m_2 m_3) m_4))$
3. $((m_1 m_2) (m_3 m_4))$
4. $((m_1 (m_2 m_3)) m_4)$
5. $(((m_1 m_2) m_3) m_4)$

The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. We can multiply 2 matrices A and B only if they are compatible i.e., the number of columns of A must equal the number of rows of B. If A is a $(p \times q)$ matrix and B is a $(q \times r)$ matrix, the resulting matrix C is a $(p \times r)$ matrix. The time to compute C is the number of scalar multiplications, which is (pqr).

Example: Consider the problem of a chain $<m_1, m_2, m_3>$ of three matrices. Suppose that the dimensions of the matrices are (6×8), (8×14), (14×20) respectively. Which parenthesization will give least number of multiplications?

Soluation:

(i) $((m_1\ m_2)\ m_3)$

$[m_1]_{6\times 8} \times [m_2]_{8\times 14} = [m_1\ m_2]_{6\times 14}$

Number of multiplications performed

$= 6 \times 8 \times 14 = 672$

$[m_1\ m_2]_{6\times 14} \times [m_3]_{14\times 20} = ((m_1\ m_2)\ m_3)_{6\times 20}$

Number of multiplications performed

$= 6 \times 14 \times 20 = 1680$

Total number of multiplications

$= 672 + 1680 = 2352$

(ii) $(m_1\ (m_2\ m_3))$

$[m_2]_{8\times 14} \times [m_3]_{14\times 20} = [m_2\ m_3]_{8\times 20}$

Number of multiplications performed

$= 8 \times 14 \times 20 = 2240$

$[m_1]_{6\times 8} \times [m_2\ m_3]_{8\times 20} = (m_1\ (m_2\ m_3))_{6\times 20}$

Number of multiplications performed

$= 6 \times 8 \times 20 = 960$

Total number of multiplications = 960 + 2240 = 3200

$\therefore$ $((m_1\ m_2)\ m_3)$ gives least number of multiplications.

We need to define the cost of an optimal solution recursively in terms of the optimal solutions to sub problems. For Matrix-chain multiplication problem, we pick as our sub problem the problems of determining the minimum cost of a parenthesization of $A_i A_{i+1} \ldots A_j$ for $1 \le i \le j \le n$ let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_i \ldots _j$; for the full problem, the cost of a cheapest way to compute $A_1 \ldots _N$ would be $m[1, n]$. We can define $m[i, j]$ recursively as follows:

$$\boxed{m\,[i, j] = m\,[i, k] + m\,[k+1, j] + P_{i-1} P_k P_j}$$

If $i = j$, the problem is trivial. The chain consists of just one matrix $A_i \ldots _i = A_i$, so that no scalar multiplications are necessary to compute the product.

Minimum cost of parenthesizing the product $A_i\ A_{i+1} \ldots A_j$ becomes

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

The $m[i, j]$ values give the costs of optimal solutions to sub problems.

At this point, to write a recursive algorithm based on recurrence to compute the minimum cost $m[1, n]$ for multiplying $A_1\ A_2 \ldots A_n$. However, this algorithm takes exponential time, which is not better than the brute force method of checking each way of parenthesizing the product. The important observation we can make at this point is that we have relatively few sub problems, one problem for each choice of i and j satisfying $1 \le i \le j \le n$ (or) $\binom{n}{2} + n = \theta(n^2)$ in all. The property of overlapping sub problems is the second hallmark of the applicability of dynamic programming.

The first hall mark being optimal substructure.

Algorithm

1. $n \leftarrow$ length $[p] - 1$
2. for $i \leftarrow 1$ to n
3. do $m[i, i] \leftarrow 0$
4. for $i \leftarrow 2$ to n
5. do for $i \leftarrow 1$ to $n - i + 1$
6. do $j \leftarrow i + i - 1$
7. $m[i, j] \leftarrow \infty$
8. for $k \leftarrow i$ to $j - 1$
9. do $q \leftarrow m\,[i, k] + m\,[k + 1, j] + P_{i-1} P_k P_j$
10. if $q < m\,[i, j]$
11. then $m\,[i, j] \leftarrow q$
12. $S[i, j] \leftarrow k$
13. return m and S

It first computes $m[i, j] \leftarrow 0$ for $i = 1, 2 \ldots n$ (the minimum costs for chains of length 1). To compute $m[i, i + 1]$ for $i = 1, 2, \ldots n - 1$(the minimum costs for chains of length $\lambda = 2$ and so on). At each step, the $m[i, j]$ cost computed depends only on table entries $m[i, k]$ and $m[k + 1, j]$ already computed. An entry $m[i, j]$ is computed using the products $P_{i-1} P_k P_j$ for $k = i, i + 1, \ldots j - 1$. A simple inspection of the nested loop structure of the above algorithm yields a running time of $O(n^3)$ for the algorithm.

Longest Common Subsequence

A sub sequence of a given sequence is just the given sequence with 0 or more elements left out. Formally, given a sequence $x = \langle x_1, x_2 \cdots x_m \rangle$, another sequence $z = \langle z_1, z_2 \cdots z_k \rangle$ is a subsequence of x if there exists a strictly increasing sequence $\langle i_1, i_2 \ldots i_k \rangle$ of indices of x such that for all $j = 1, 2 \cdots k$, we have $x_{ij} = z_j$

Example: $z = < B, C, D, B >$ is a subsequence of $x = <A, B, C, B, D, A, B>$ with corresponding index sequence <2, 3, 5, 7>

Example: Given 2 sequences x and y, we say that a sequence z is a common sub sequence of x and y if z is a sub sequence of both x and y.

If $x = <A, B, C, B, D, A, B>$

$y = <B, D, C, A, B, A>$

The sequence $<B, C, A>$ is a common subsequence of both x and y.

The sequence $<B, C, A>$ is not a longest common subsequence (LCS) of x and y since it has length '3' and the sequence $<B, C, B, A>$, which is also common to both x and y, has length 4. The sequence $<B, C, B, A>$ is an LCS of x and y, as is the sequence $<B, D, A, B>$, since there is no common subsequence of length 5 or greater.

- In the longest-common-sub sequence problem, we are given 2 sequences $x = <x_1, x_2, x_3 \ldots x_m>$ and $y = <y_1, y_2 \ldots y_n>$ and wish to find a maximum length common subsequence of x and y.
- LCS problem can be solved efficiently using dynamic programming.
- A brute force approach to solve the LCS problem is to enumerate all subsequences of x and check each subsequence to see if it is also a subsequence of y, keeping track of the longest subsequence found. Each subsequence of x corresponds to a subset of the indices $\{1, 2 \ldots m\}$ of x. There are 2^m subsequences of x, so this approach requires exponential time, making it impractical for long sequences.
- The classes of sub problems correspond to pairs of 'pre fixes' of 2 input sequences:

Given a sequence $x = <x_1, x_2 \cdots x_m>$, we define the ith prefix of x, for $i = 0, 1, \ldots m$, as

$$x_i = <x_1 x_2 \ldots x_i>$$

Example: If $x = <A, B, C, B, D, A, D>$, then $x_4 = <A, B, C, B>$ and x_0 is the empty sequence. LCS problem has an optimal sub-structure property.

Optimal Substructure of LCS

Let $x = <x_1, x_2 \ldots x_m>$ and $y = <y_1, y_2 \ldots y_n>$ be sequences and let $z = <z_1, z_2 \ldots z_k>$ be any LCS of x and y then

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and z_{k-1} is an LCS of x_{m-1} and y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that z is an LCS of x_{m-1} and y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that z is an LCS of x and y_{n-1}.

NP-HARD AND NP-COMPLETE

A mathematical problem for which, even in theory, no shortcut or smart algorithm is possible that would lead to a simple or rapid solution. Instead the only way to find an optimal solution is a computationally intensive, exhaustive analysis in which all possible outcomes are tested. Examples of NP-hard problems include the travelling salesman problem.

P-problem

A problem is assigned to the P(polynomial time) class if there exists at least one algorithm to solve that problem, such that number of steps of the algorithm is bounded by a polynomial in n, where n is the length of the input.

NP-problem

A problem is assigned to the *NP* (non-deterministic polynomial time) class if it is solvable in polynomial time by a non-deterministic turing machine.

A *P*-problem (whose solution time is bounded by a polynomial) is always also *NP*. If a problem is known to be *NP*, and a solution to the problem is somehow known, then demonstrating the correctness of the solution can always be reduced to a single *P* (polynomial time) verification. If *P* and *NP* are not equivalent then the solution of *NP*-problems requires (in the worst case) an exhaustive search.

A problem is said to be *NP*-hard, if an algorithm for solving it can be translated into one for solving any other *NP*-problem. It is much easier to show that a problem is *NP* than to show that it is *NP*-hard. A problem which is both *NP* and *NP*-hard is called an *NP*-complete problem.

P versus NP-problems

The *P* versus *NP* problem is the determination of whether all *NP*-problems are actually *P*-problems, if *P* and *NP* are not equivalent then the solution of *NP*-problem requires an exhaustive search, while if they are, then asymptotically faster algorithms may exist.

NP-complete problem

A problem which is both *NP* (verifiable in non-deterministic polynomial time) and *NP*-hard (any *NP*-problem can be translated into this problem). Examples of *NP*-hard problems include the Hamiltonian cycle and travelling sales man problems.

Example:

Circuit satisfiability is a good example of problem that we don't know how to solve in polynomial time. In this problem, the input is a Boolean circuit. A collection of and, or and not gates connected by wires. The input to the circuit is a set of m Boolean (true/false) values $x_1 \ldots x_m$. The output is a single Boolean value. The circuit satisfiability problem asks, given a circuit, whether there is an input that makes the circuit output TRUE, or conversely, whether the circuit always outputs FLASE. Nobody knows how to solve this problem faster than just trying all 2^m possible inputs to the circuit but this requires exponential time.

P, NP, and Co-NP

- *P* is a set of yes/no problems that can be solved in polynomial time. Intuitively *P* is the set of problems that can be solved quickly.
- *NP* is the set of yes/no problems with the following property: If the answer is yes, then there is a proof of this fact that can be checked in polynomial time. Intuitively *NP* is the set of problems where we can verify a YES answer quickly if we have the solution in front of us.

Example: The circuit satisfiability problem is in *NP*.

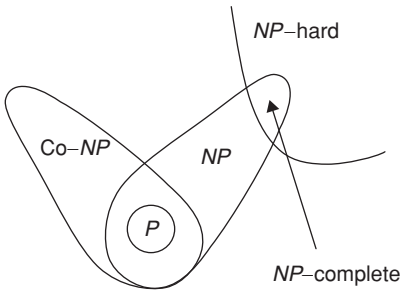
If the answer is yes, then any set of m input values that produces TRUE output is a proof of this fact, we can check the proof by evaluating the circuit in polynomial time.

- Co-*NP* is the exact opposite of *NP*. If the answer to a problem in co-*NP* is no, then there is a proof of this fact that can be checked in polynomial time.

- π is *NP*-hard $\Rightarrow$ if π can be solved in polynomial time, then $P = NP$.

This is like saying that if we could solve one particular *NP*-hard problem quickly, then we could solve any problem whose solution is easy to understand, using the solution to that one special problem as a subroutine. *NP*-hard problems are atleast as hard as any problem in *NP*.

- Saying that a problem is *NP*-hard is like saying 'If I own a dog, then it can speak fluent English'. You probably don't know whether or not I own a dog, but you're probably pretty sure that I don't own a talking dog. Nobody has a mathematical proof that dogs can't speak English. The fact that no one has ever heard a dog speak English is evidence as per the hundreds of examinations of dogs that lacked the proper mouth shape and brain power, but mere evidence is not a proof nevertheless, no sane person would believe me if I said I owned a dog that spoke fluent English. So the statement 'If I own a dog then it can speak fluent English' has a natural corollary: No one in their right mind should believe that I own a dog ! Likewise if a problem is *NP*-hard no one in their right mind should believe it can be solved in polynomial time.



Cooks Theorem

Cook's theorem states that CNFSAT is *NP*-Complete

It means, if the problem is in *NP*, then the deterministic Turing machine can reduce the problem in polynomial time.

The inference that can be taken from these theorems is, if deterministic polynomial time algorithm exists for solving satisfiability, then to all problems present in *NP* can be solved in polynomial time.

Non-deterministic Search

Non-deterministic algorithms are faster, compared to deterministic ones. The computations are fast as it always chooses right step

The following functions are used to specify these algorithms

1. Choice (A), which chooses a random element from set A
2. Failure (A), specifies failure
3. Success ( ), Specifies success

The non-deterministic search is done as follows.

Let us consider an array $S[1 \ldots n]$, $n \geq 1$ we need to get the indice of 'i' such that $S[i] = t$ (or) $i = 0$. The algorithm is given below.

Steps:

1. i = Choice (1, n);
2. if $S[i] = t$, then
   (i) Print (i);
   (ii) Success ( );
3. Print (0)
   failure
4. Stop.

If the search is successful it returns the indice of array 'S', otherwise it returns '0', the time complexity is $\Omega(n)$.

Exercises

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. Hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20 using the hash function with chaining $(2k + 5) \bmod 11$, which of the following slots are empty?

(A) 0, 1, 2, 3, 4 (B) 0, 2, 3, 4, 8, 10
(C) 0, 1, 2, 4, 8, 10 (D) 0, 1, 2, 4, 8

2. Using linear probing on the list given in the above question with the same hash function, which slots are not occupied?

(A) 3, 4 (B) 4, 5
(C) 3, 6 (D) 4, 6

3. In hashing, key value 123456 is hashed to which address using multiplication method ($m = 10^4$)?

(A) 40 (B) 41
(C) 42 (D) 44

4. Insert element 14 into the given hash table with double hashing? $h_1(k) = k \bmod 13$, $h_2(k) = 1 + (k \bmod 11)$. The element will occupy, which slot?

0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	
10	
11	50
12	

(A) 7th (B) 8th
(C) 2nd (D) 9th

5. Consider the below given keys:

257145368, 25842354, 12487654, 248645452. Find the hash values of keys using shift folding method?

(A) 770, 221, 153, 345 (B) 221, 770, 153, 345
(C) 760, 770, 153, 345 (D) 815, 770, 153, 345

6. Consider the following two problems on unidirected graphs.

β : Given $G(V, E)$, does G have an independent set of size $|V|-4$?

α : Given $G(V, E)$, does G have an independent set of size 5?

Which of the following is true?

(A) β is in P and α is in NP-Complete
(B) β is in NP-Complete and α is in P
(C) Both α and β are NP-Complete
(D) Both α and β are in P

7. Let S be an NP-complete problem and Q and R be two other problems not known to be in NP. Q is polynomial-time reducible to S and S is polynomial-time reducible to R. Which one of the following statements is true?

(A) R is NP-Complete (B) R is NP-Hard
(C) Q is NP-Complete (D) Q is NP-Hard

8. Let $FHAM_3$ be the problem of finding a Hamiltonian cycle in a graph $G = (V, E)$ with $|V|$ divisible by 3 and $DHAM_3$ be the problem of determining if a Hamiltonian cycle exists in such graphs. Which of the following is true?

(A) Both $FHAM_3$ and $DHAM_3$ are NP-hard
(B) $FHAM_3$ is NP-hard but $DHAM_3$ is not
(C) $DHAM_3$ is NP-hard but $FHAM_3$ is not
(D) Neither $FHAM_3$ nor $DHAM_3$ is NP-hard

9. Consider a hash table of size 7, with starting index '0' and a hash function $(3x + 4)$ mod 7. Initially hash table is empty. The sequence 1, 3, 8, 10 is inserted into the table using closed hashing then what is the position of element 10?

(A) 1st (B) 2nd
(C) 6th (D) 0th

10. Place the given keys in the hash table of size 13, index from '0' by using open hashing, hash function is $h(k)$ mod 13.

Keys: A, FOOL, HIS, AND

(hint : Add the positions of a word's letters in the alphabet, take $A \rightarrow 1$, $B \rightarrow 2$, $C \rightarrow 3$. $D \rightarrow 4$ … $Z \rightarrow 26$).

Which of the following shows the correct hash addresses of keys?

(A) A – 1, FOOL – 10, HIS – 9, AND – 6
(B) A – 1, FOOL – 9, HIS – 10, AND – 6
(C) A – 0, FOOL – 6, HIS – 10, AND – 9
(D) A – 0, FOOL – 9, HIS – 9, AND – 6

11. Consider the following input (322, 334, 471, 679, 989, 171, 173, 199) and the hash function is x mod 10 which statement is true?

I. 679, 989, 199 hash to the same value
II. 471, 171, hash to the same value
III. Each element hashes to a different value
IV. All the elements hash to the same value

(A) I Only (B) II Only
(C) I and II (D) III

12. For the input 30, 20, 56, 75, 31, 19 and hash function $h(k) = k$ mod 11, what is the largest number of key comparisons in a successful search in the open hash table.

(A) 4 (B) 3
(C) 5 (D) 2

13. The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an empty hash table of length 10 using open addressing with hash function, $h(k) = k$ mod 10 and linear probing.

Which is the resultant hash table?

(A)

	0
	1
2	2
23	3
13	4
15	5
	6
	7
	8
	9

(B)

	0
3	1
12	2
13	3
	4
15	5
	6
	7
	8
	9

(C)

	0
	1
12	2
13	3
2	4
3	5
23	6
5	7
18	8
15	9

(D)

	0
	1
2	2
3	3
12	4
13	5
23	6
5	7
18	8
15	9

14. Which one of the following is correct?

(A) Finding shortest path in a graph is solvable in polynomial time.
(B) Finding longest path from a graph is solvable in poly-nomial time.
(C) Finding longest path from a graph is solvable in polynomial time, if edge weights are very small values.
(D) Both (A) and (B) are correct

15. In the following pair of problems

$$\frac{\text{2 CNF Satisfiability}}{\text{I}} \text{ Vs } \frac{\text{3 CNF Satisfiability}}{\text{II}}.$$

(A) I is solvable in polynomial time, II is NP complete problem.
(B) II is solvable in polynomial time, I is NP complete problem.
(C) Both are solvable in polynomial time
(D) None can be solved in polynomial time.

Practice Problems 2

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. For *NP*-complete problems
(A) Several polynomial time algorithms are available
(B) No polynomial time algorithm is discovered yet
(C) Polynomial time algorithms exist but not discovered
(D) Polynomial time algorithms will not exist, hence cannot be discovered

2. In the division method for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is
(A) $h(k) = m \bmod k$ (B) $h(k) = m \bmod m/k$
(C) $h(k) = k \bmod m$ (D) $h(k) = mk \bmod k$

3. In the division method for creating hash function, which of the following hash table size is most appropriate?
(A) 2 (B) 7
(C) 4 (D) 8

4. Which of the following techniques are commonly used to compute the probe sequence required for open addressing?
(A) Linear probing (B) Quadratic probing
(C) Double hashing (D) All the above

5. Which of the following problems is not *NP*-hard?
(A) Hamiltonian circuit problem
(B) The 0/1 knapsack problem
(C) The graph coloring problem
(D) None of these

6. For problems x and y, y is *NP*-complete and x reduces to y in polynomial time. Which of the following is true?
(A) If x can be solved in polynomial time, then so can y
(B) x is *NP*-hard
(C) x is *NP*-complete
(D) x is in *NP*, but not necessarily *NP*-complete

7. If P_1 is *NP*-complete and there is a polynomial time reduction of P_1 to P_2, then P_2 is
(A) NP-complete
(B) Not necessarily NP-complete
(C) Cannot be *NP*-complete
(D) None of these

8. A problem is in *NP*, and as hard as any problem in *NP*. The given problem is
(A) *NP* hard
(B) *NP* complete
(C) *NP*
(D) *NP*-hard $\cap$ *NP*-complete

9. Which of the following is TRUE?
(A) All *NP*-complete problems are *NP*-hard.
(B) If an *NP*-hard problem can be solved in polynomial time, then all *NP*-complete problems can be solved in polynomial time.
(C) *NP*-hard problems are not known to be *NP*-complete.
(D) All the above

10. If a polynomial time algorithm makes polynomial number of calls to polynomial time subroutines, then the resulting algorithm runs in
(A) Polynomial time (B) No-polynomial time
(C) Exponential time (D) None of these

11. If a polynomial time algorithm makes atmost constant number of calls to polynomial time subroutines, then the resulting algorithm runs in
(A) Polynomial time (B) No-polynomial time
(C) Exponential time (D) None of these

12. When a record to be inserted maps to an already occupied slot is called
(A) Hazard
(B) Collision
(C) Hashing
(D) Chaining

13. Worst-case analysis of hashing occurs when
(A) All the keys are distributed
(B) Every key hash to the same slot
(C) Key values with even number, hashes to slots with even number
(D) Key values with odd number hashes to slots with odd number

14. Main difference between open hashing and closed hashing is
(A) Closed hashing uses linked lists and open hashing does not.
(B) Open hashing uses linked list and closed hashing does not
(C) Open hashing uses tree data structure and closed uses linked list
(D) None of the above

15. The worst case scenario in hashing occurs when
(A) All keys are hashed to the same cell of the hash table
(B) The size of hash table is bigger than the number of keys
(C) The size of hash table is smaller than the number of keys
(D) None of the above

Previous Years' Questions

1. Consider a hash table of size seven, with starting index zero, and a hash function $(3x + 4)$ mod7. Assuming the hash table is initially empty, which of the following is the contents of the table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing? Note that − denotes an empty location in the table. **[2007]**

(A) 8, −, −, −, −, −, 10 (B) 1, 8, 10, −, −, −, 3
(C) 1, −, −, −, −, −, 3 (D) 1, 10, 8, −, −, −, 3

Common data for questions 2 and 3: Suppose the letters a, b, c, d, e, f have probabilities $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}$, respectively.

2. Which of the following is the Huffman code for the letter a, b, c, d, e, f? **[2007]**

(A) 0, 10, 110, 1110, 11110, 11111
(B) 11, 10, 011, 010, 001, 000
(C) 11, 10, 01, 001, 0001, 0000
(D) 110, 100, 010, 000, 001, 111

3. What is the average length of the correct answer to above question? **[2007]**

(A) 3 (B) 2.1875
(C) 2.25 (D) 1.9375

4. The subset-sum problem is defined as follows: Given a set S of n positive integers and a positive integer W, determine whether there is a subset of S whose elements sum to W.

An algorithm Q solves this problem in $O(nW)$ time. Which of the following statements is false? **[2008]**

(A) Q solves the subset-sum problem in polynomial time when the input is encoded in unary
(B) Q solves the subset-sum problem in polynomial time when the input is encoded in binary
(C) The subset sum problem belongs to the class NP
(D) The subset sum problem is NP-hard

5. Let π_A be a problem that belongs to the class NP. Then which one of the following is TRUE? **[2009]**

(A) There is no polynomial time algorithm for π_A.
(B) If π_A can be solved deterministically in polynomial time, then $P = NP$.
(C) If π_A is NP-hard, then it is NP-complete.
(D) π_A may be undecidable.

6. The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function $h(k) = k$ mod 10 and linear probing. What is the resultant hash table? **[2009]**

(A)

0	
1	
2	12
3	23
4	
5	15
6	
7	
8	18
9	

(B)

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(C)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

(D)

0	
1	
2	12,2
3	13,3,23
4	
5	5,15
6	
7	
8	18
9	

Common data for questions 7 and 8: A sub-sequence of a given sequence is just the given sequence with some elements (possibly none or all) left out. We are given two sequences $X[m]$ and $Y[n]$ of lengths m and n, respectively, with indexes of X and Y starting from 0.

7. We wish to find the length of the longest common subsequence (LCS) of $X[m]$ and $Y[n]$ as $l(m, n)$, where an incomplete recursive definition for the function $l(i, j)$ to compute the length of the LCS of $X[m]$ and $Y[n]$ is given below:

$l(i, j) = 0$, if either $i = 0$ or $j = 0$
$=$ expr1, if $i, j > 0$ and $X[i - 1] = Y[j - 1]$
$=$ expr2, if $i, j > 0$ and $X[i - 1] \neq Y[j - 1]$

Which one of the following options is correct? **[2009]**

(A) expr1 $\equiv l(i - 1, j) + 1$
(B) expr1 $\equiv l(i, j - 1)$
(C) expr2 $\equiv \max(l(i - 1, j), l(i, j - 1))$
(D) expr2 $\equiv \max(l(i - 1, j - 1), l(i, j))$

8. The values of $l(i, j)$ could be obtained by dynamic programming based on the correct recursive definition of $l(i, j)$ of the form given above, using an array $L[M, N]$, where $M = m + 1$ and $N = n + 1$, such that $L[i, j] = l(i, j)$.

Which one of the following statements would be TRUE regarding the dynamic programming solution for the recursive definition of $l(i, j)$? **[2009]**

(A) All elements of L should be initialized to 0 for the values of $l(i, j)$ to be properly computed.

(B) The values of $l(i, j)$ may be computed in a row major order or column major order of $L(M, N)$.
(C) The values of $l(i, j)$ cannot be computed in either row major order or column major order of $L(M, N)$.
(D) $L[p, q]$ needs to be computed before $L[r, s]$ if either $p < r$ or $q < s$.

9. The weight of a sequence $a_0, a_1, \cdots, a_{n-1}$ of real numbers is defined as $a_0 + a_1/2 + \cdots + a_{n-1}/2^{n-1}$. A subsequence of a sequence is obtained by deleting some elements from the sequence, keeping the order of the remaining elements the same. Let X denote the maximum possible weight of a subsequence of $a_0, a_1, \ldots, a_{n-1}$. Then X is equal to **[2010]**
(A) max $(Y, a_0 + Y)$ (B) max $(Y, a_0 + Y/2)$
(C) max $(Y, a_0 + 2Y)$ (D) $a_0 + Y/2$

10. Four matrices M_1, M_2, M_3 and M_4 of dimensions $p \times q$, $q \times r$, $r \times s$ and $s \times t$ respectively, can be multiplied in several ways with different number of total scalar multiplications. For example when multiplied as $((M_1 \times M_2) \times (M_3 \times M_4))$, the total number of scalar multiplications is $pqr + rst + prt$. When multiplied $(((M_1 \times M_2) \times M_3) \times M_4)$ the total number of scalar multiplications is $pqr + prs + pst$.

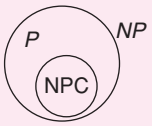
If $p = 10$, $q = 100$, $r = 20$, $s = 5$ and $t = 80$, then the minimum number of scalar multiplications needed is **[2011]**

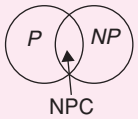
(A) 248000
(B) 44000
(C) 19000
(D) 25000

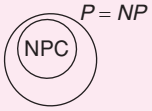
11. Assuming $P \neq NP$, which of the following is **TRUE**? **[2012]**
(A) NP-complete = NP
(B) NP-complete $\cap$ $P = \varnothing$
(C) NP-hard = NP
(D) $P = NP$-complete

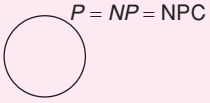
12. Which of the following statements are TRUE?
(i) The problem of determining whether there exists a cycle in an undirected graph is in P.
(ii) The problem of determining whether there exists a cycle in an undirected graph is in NP.
(iii) If a problem A is NP-Complete, there exists a non-deterministic polynomial time algorithm to solve A. **[2013]**
(A) 1, 2 and 3 (B) 1 and 2 only
(C) 2 and 3 only (D) 1 and 3 only

13. Suppose a polynomial time algorithm is discovered that correctly computes the largest clique in a given graph. In this scenario, which one of the following represents the correct Venn diagram of the complexity classes P, NP and NP-complete (NPC)? **[2014]**

(A) 

(B) 

(C) 

(D) 

14. Consider a hash, table with 9 slots. The hash function is $h(K) = K$ mod 9. The collisions are resolved by chaining. The following 9 keys are inserted in the order: 5, 28, 19, 15, 20, 33, 12, 17, 10. The maximum, minimum, and average chain lengths in the hash table, respectively, are **[2014]**
(A) 3, 0 and 1 (B) 3, 3 and 3
(C) 4, 0 and 1 (D) 3, 0 and 2

15. Consider two strings A = 'qpqrr' and B = 'pqprqrp'. Let x be the length of the longest common subsequence (not necessarily contiguous between A and B and let y be the number of such longest common subsequences between A and B. then $x + 10y =$ _____ **[2014]**

16. Suppose you want to move from 0 to 100 on the number line. In each step, you either move right by a unit distance or you take a *shortcut.* A shortcut is simply a pre-specified pair of integers i, j with $i < j$. Given a shortcut i, j if you are at position i on the number line, you may directly move to j. Suppose $T(k)$ denotes the smallest number of steps needed to move from k to 100. Suppose further that there is at most 1 shortcut involving any number, and in particular from 9 there is a shortcut to 15. Let y and z be such that $T(9) = 1 + \min(T(y), T(z))$. Then the value of the product yz is _____ **[2014]**

17. Consider the decision problem 2CNFSAT defined as follows: **[2014]**

$\{\phi \mid \phi$ is a satisfiable propositional formula in CNF with at most two literals per clause$\}$

For example, $\phi = (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_4)$ is a Boolean formula and it is in 2CNFSAT.

The decision problem 2CNFSAT is

(A) NP-complete
(B) Solvable in polynomial time by reduction to directed graph reach ability.
(C) Solvable in constant time since any input instance is satisfiable.
(D) NP-hard, but not NP-complete

18. Consider a hash table with 100 slots. Collisions are resolved using chaining. Assuming simple uniform

hashing, what is the probability that the first 3 slots are unfilled after the first 3 insertions? **[2014]**

(A) $(97 \times 97 \times 97)/100^3$
(B) $(99 \times 98 \times 97)/100^3$
(C) $(97 \times 96 \times 95)/100^3$
(D) $(97 \times 96 \times 95)/(3! \times 100^3)$

19. Match the following **[2014]**

(P) prim's algorithm for minimum spanning tree	(i) Backtracking
(Q) Floyd-Warshall algorithm for all pairs shortest paths	(ii) Greedy method
(R) Mergesort	(iii) Dynamic programming
(S) Hamiltonian circuit	(iv) Divide and conquer

(A) P–iii, Q–ii, R–iv, S–i
(B) P–i, Q–ii, R–iv, S–iii
(C) P–ii, Q–iii, R–iv, S–i
(D) P–ii, Q–i, R–iii, S–iv

20. Given a hash table T with 25 slots that stores 2000 elements, the load factor $\propto$ for T is _____ **[2015]**

21. Language L_1 is polynomial time reducible to language L_2. Language L_3 is polynomial time reducible to L_2, which in turn is polynomial time reducible to language L_4. Which of the following is/are true? **[2015]**

(1) if $L_4 \in P$, then $L_2 \in P$
(2) if $L_1 \in P$ or $L_3 \in P$, then $L_2 \in P$
(3) $L_1 \in P$, if and only if $L_3 \in P$
(4) if $L_4 \in P$, then $L_1 \in P$ and $L_3 \in P$

22. The Floyd - Warshall algorithm for all -pair shortest paths computation is based on **[2016]**

(A) Greedy paradigm
(B) Divide-and-Conquer paradigm
(C) Dynamic Programming paradigm
(D) Neither Greedy nor Divide-and-Conquer nor Dynamic Programming paradigm.

23. Let A_1, A_2, A_3, and A_4 be four matrices of dimensions 10×5, 5×20, 20×10, and 10×5, respectively. The minimum number of scalar multiplications required to find the product $A_1 A_2 A_3 A_4$ using the basic matrix multiplication method is _____ . **[2016]**

24. Consider the following table:

Algorithms	Design Paradigms
(P) Kruskal	(i) Divide and Conquer
(Q) Quicksort	(ii) Greedy
(R) Floyd-Warshall	(iii) Dynamic Programming

Match the algorithms to the design paradigms they are based on. **[2017]**

(A) (P) $\leftrightarrow$ (ii), (Q) $\leftrightarrow$ (iii), (R) $\leftrightarrow$ (i)
(B) (P) $\leftrightarrow$ (iii), (Q) $\leftrightarrow$ (i), (R) $\leftrightarrow$ (ii)
(C) (P) $\leftrightarrow$ (ii), (Q) $\leftrightarrow$ (i), (R) $\leftrightarrow$ (iii)
(D) (P) $\leftrightarrow$ (i), (Q) $\leftrightarrow$ (ii), (R) $\leftrightarrow$ (iii)

25. Assume that multiplying a matrix G_1 of dimension $p \times q$ with another matrix G_2 of dimension $q \times r$ requires pqr scalar multiplications. Computing the product of n matrices $G_1 G_2 G_3, ..., G_n$ can be done by parenthesizing in different ways. Define $G_i\ G_{i+1}$ as an explicitly computed pair for a given paranthesization if they are directly multiplied. For example, in the matrix multiplication chain $G_1 G_2 G_3 G_4 G_5 G_6$ using parenthesization $(G_1(G_2 G_3))(G_4(G_5 G_6))$, $G_2 G_3$ and $G_5 G_6$ are the only explicitly computed pairs.

Consider a matrix multiplication chain $F_1 F_2 F_3 F_4 F_5$, where matrices F_1, F_2, F_3, F_4, and F_5 are of dimensions 2×25, 25×3, 3×16, 16×1 and 1×1000, respectively. In the parenthesization of $F_1 F_2 F_3 F_4 F_5$ that minimizes the total number of scalar multiplications, the explicitly computed pairs is/are: **[2018]**

(A) $F_1 F_2$ and $F_3 F_4$ only
(B) $F_2 F_3$ only
(C) $F_3 F_4$ only
(D) $F_1 F_2$ and $F_4 F_5$ only

26. Consider the weights and values of items listed below. Note that there is only one unit of each item.

Item no.	Weight (in Kgs)	Value (in Rupees)
1	10	60
2	7	28
3	4	20
4	2	24

The task is to pick a subset of these items such that their total weight is no more than 11 kgs and their total value is maximized. Moreover, no item may be split. The total value of items picked by an optimal algorithm is denoted by V_{opt}. A greedy algorithm sorts the items by their value-to-weight ratios in descending order and packs them greedily, starting from the first item in the ordered list. The total value of items picked by the greedy algorithm is denoted by V_{greedy}. The value of $V_{opt} - V_{greedy}$ is _____. **[2018]**

Answer Keys

Exercises

Practice Problems 1

1. B	**2.** A	**3.** B	**4.** D	**5.** A	**6.** C	**7.** C	**8.** A	**9.** B	**10.** B
11. C	**12.** B	**13.** C	**14.** A	**15.** A					

Practice Problems 2

1. B	**2.** C	**3.** B	**4.** D	**5.** B	**6.** C	**7.** A	**8.** B	**9.** D	**10.** C
11. A	**12.** B	**13.** B	**14.** B	**15.** A					

Previous Years' Questions

1. B	**2.** A	**3.** D	**4.** B	**5.** C	**6.** C	**7.** C	**8.** B	**9.** C	**10.** B
11. B	**12.** A	**13.** D	**14.** A	**15.** 34	**16.** 150	**17.** B	**18.** A	**19.** C	**20.** 80
21. C	**22.** C	**23.** 1500	**24.** C	**25.** C	**26.** 16				

TEST

ALGORITHMS (PART 2)

Time: 45 min.

Directions for questions 1 to 30: Select the correct alternative from the given choices.

1. The worst case running time of an algorithm means
(A) The algorithm will never take any longer.
(B) The algorithm will take less time than running time
(C) The algorithm will run in a finite time
(D) None of the above

2. Analyzing an algorithm involves
(A) Evaluating the complexity
(B) Validating the Algorithm
(C) Both A and B
(D) None of the above

3. $f(n) = \theta(g(n))$ is
(A) $g(n)$ is asymptotic lower bound for $f(n)$
(B) $g(n)$ is asymptotic tight bound for $f(n)$
(C) $g(n)$ is asymptotic upper bound for $f(n)$
(D) None of the above

4. Which case yields the necessary information about an algorithm's behaviour on a random input?
(A) Best-case (B) Worst-case
(C) Average-case (D) Both A and C

5. Algorithms that require an exponential number of operations are practical for solving.
(A) Only problems of very small size
(B) Problems of large size
(C) Problems of any size
(D) None of these

6. Problems that can be solved in polynomial time are called
(A) Tractable (B) Decidable
(C) Solvable (D) Computable

7. Problems that cannot be solved at all by any algorithm are known as
(A) Tractable (B) Undecidable
(C) Untractable (D) Unsolvable

8. Which of the following problems is decidable but intractable?
(A) Hamiltonian circuit (B) Traveling sales man
(C) Knapsack problem (D) All the above

9. Which method is used to solve recurrences?
(A) Substitution method
(B) Recursion-tree method
(C) Master method
(D) All the above

10. Consider the following
(i) Input
(ii) Output
(iii) Finiteness
(iv) Definiteness means clear and unambiguous
(v) Effectiveness

Which of the following is not a property of an algorithm?
(A) (iv) only (B) (iv) and (v) only
(C) (iii) and (iv) only (D) None of the above

11. Finiteness of an algorithm means
(A) The steps of the algorithm should be finite
(B) The algorithm should terminate after finite time
(C) Algorithm must terminate after a finite number of steps
(D) Algorithm should consume very less space

12. Asymptotic analysis on efficiency of algorithm means
(A) The efficiency of the algorithm on a particular machine
(B) How the running time of an algorithm increases as the size increases without bound
(C) How efficiently the algorithm is applied to solve a problem without thinking of input size.
(D) None of the above

13. What is the input size of a problem?
(A) Number of variables used to solve the problem
(B) Number of constants used to solve the problem
(C) it is problem specific that is in case of graph it is number of edges and vertices and so on.
(D) None of these

14. (i) An algorithm must take input
(ii) An algorithm must give out put
Which is true in the following options?
(A) (i) Only (B) (ii) Only
(C) (i) and (ii) Only (D) None of the above

15. As $n\ \theta°$
Which of the following is efficient?
(A) $\theta(n^3)$ (B) $\theta(n^2)$
(C) $\theta(2^n)$ (D) $\theta(n^4)$

16. Suppose
$T_1(n) = O(f(n))$
$T_2(n) = O(f(n))$
which of the following is true,.
(A) $T_1(n) + T_2(n) = O(f(n))$ (B) $\frac{T_1(n)}{T_2(n)} = 0(1)$
(C) $T_1(n) = 0(T_2(n))$ (D) None of these

17. The following program computes $n!$

Find the complexity?

Input: A non-negative integer

Output: Value of $n!$

If $n = 0$ return 1

Else return $F(n-1)\ \theta n$

(A) $\theta(n)$ (B) $\theta(n \log n)$

(C) $\theta(n^2)$ (D) $\theta(n^3)$

18. Which of the following functions are often referred as 'exponential growth function'?

(A) 2^n, $\log n$ (B) 2^n, $n!$

(C) $n!$, $n \log n$ (D) $n!$, $\log n$

19. Consider the following code

```
sort (a, n)
{
    for i = 1 to n do
    {
            j = i;
    for k = i + 1 to n do
            if (a[k] < a [j]) then j = k;
            t = a[i];
            a[i] = a[j];
            a[j] = t;
            }
    }
```

The above code implements which sorting?

(A) Merge sort

(B) selection sort

(C) Insertion sort

(D) Radix sort

20. Assume that the number of disks in a 'Towers of Hanoi problem' is 'n', with '3' towers, Initially all disks are placed on tower 1, to get the largest disk are placed on tower 1, to get the largest disk to the bottom of 2nd tower, How many moves are required? ($\theta n = 3$)

(A) n

(B) $(n-1)$

(C) $(n+1)$

(D) $2n$

21. Each new term in Fibonacci sequence is obtained by taking the sum of the two previous terms. The first term of the sequence is $f_0 = 0$, and the second term $f_1 = 1$. Which of the following gives Fibonacci sequence?

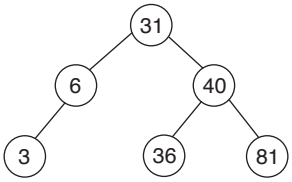
(A) $f_n = f_{n+1} + f_{n-2},\ n \geq 2$

(B) $f_n = f_{n-1} + f_{n-2},\ n \geq 2$

(C) $f_n = f_{n-1} + f_{n+1},\ n \geq 2$

(D) All the above

22. Consider the binary search tree



Delete node '31', what would be the parent node in the new binary search tree?

(A) 36

(B) 40

(C) 81

(D) 6

23. Consider the given array [4, 6, 7, 8, 21, 9, 3, 10, 13, 16, 31] after performing '1' delete max operation, on the max heap. What would be the sequence of elements in the array?

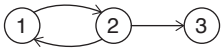
(A) 9, 21, 13, 16, 3, 7, 10, 8, 4, 6

(B) 21, 9, 13, 16, 7, 3, 10, 8, 4, 6

(C) 21, 9, 13, 16, 3, 7, 10, 8, 4, 6

(D) 21, 9, 13, 16, 7, 3, 10, 4, 8, 6

24. Consider the given Di-graph

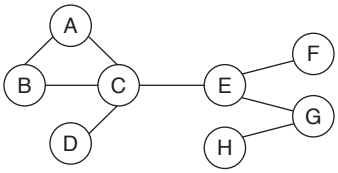


How many strongly connected components does the above graph contain?

(A) 1 (B) 2

(C) 3 (D) many

25. Consider the given graph



Which of the following shows the adjacency matrix of the above graph?

(A)

	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	0	0
B	1	0	1	0	0	0	0	0
C	1	1	0	1	1	0	0	0
D	0	0	1	0	0	0	0	0
E	0	0	1	0	0	1	1	0
F	0	0	0	0	1	0	0	0
G	0	0	0	0	1	0	0	1
H	0	0	0	0	0	0	1	0

(B)

	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	0	0
B	1	0	1	0	0	0	0	0
C	1	1	0	1	1	0	0	0
D	0	0	1	0	0	0	0	0
E	0	0	1	0	0	1	1	0
F	0	0	**0**	0	1	0	0	0
G	0	0	0	0	1	0	0	1
H	0	0	0	0	1	0	1	0

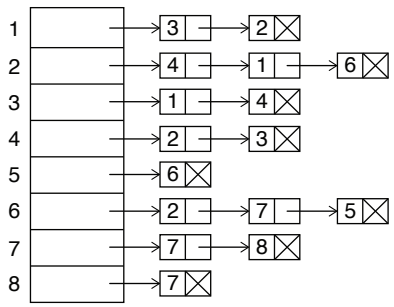
(C)

	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	0	0
B	1	0	1	0	0	0	0	0
C	1	1	0	1	1	0	0	0
D	0	0	1	0	0	0	0	0
E	0	0	1	0	0	1	1	0
F	0	0	**0**	0	1	0	0	0
G	0	0	0	0	0	1	0	1
H	0	0	0	0	0	0	1	0

(D)

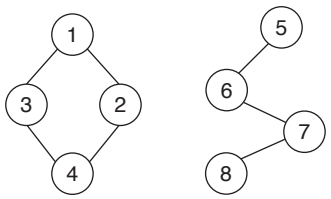
	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	0	0
B	1	0	1	0	0	0	0	0
C	1	1	0	1	1	0	0	0
D	0	0	1	0	0	0	0	0
E	0	0	1	0	0	1	1	0
F	0	0	**0**	0	1	0	0	0
G	0	0	0	0	1	0	0	1
H	0	0	0	0	1	1	0	1

26. Consider the given adjacency list

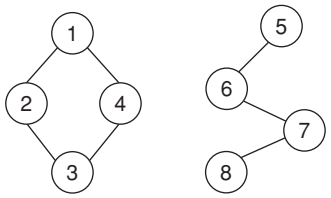


The above list is representation of which of the following graph?

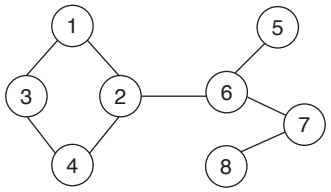
(A)



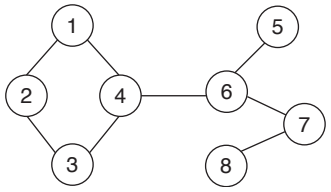
(B)



(C)



(D)



27. Which of the following is FALSE?

(A) In dynamic programming an optimal sequence of decisions is obtained by making explicit appeal to the principle of optimality

(B) In greedy method only one decision sequence is generated.

(C) In dynamic programming, many decision sequences may be generated.

(D) In greedy method many decision sequences are generated.

28. Consider an array $a[n]$ of 'n' numbers that has '$n/2$' distinct elements and '$n/2$' copies of another element, to identify that repeated element, how many steps are required in the worst case?

(A) $n/2$ (B) $n/2 + 1$

(C) $n/2 + 2$ (D) n

29. Match the following, for a very large value of 'n'

I. $36n^3 + 2n^2$

II. $5n^2 - 6n$

III. $n^{1.001} + n \log n$

P. $\theta(n^2)$

Q. $\mid (n^3)$

R. $\theta(n^{1.001})$

(A) I – P, II – Q, III – R (B) I – Q, II – P, III – R

(C) I – R, II – Q, III – P (D) I – R, II – P, III – R

30. Consider the following code

```
        T(a, n)
{
   for i = 1 to n - 1 do
      for j = i + 1 to n do
            {
                  t = a[i, j];
                  a[i, j] = a[j, i];
                  a[j, i] = t;
            }
   }
```

The above code performs
(A) Matrix multiplication
(B) Matrix addition
(C) Matrix transpose
(D) Matrix chain multiplication

Answers Keys

1. A	**2.** C	**3.** B	**4.** C	**5.** A	**6.** A	**7.** B	**8.** D	**9.** D	**10.** D
11. C	**12.** D	**13.** C	**14.** B	**15.** B	**16.** A	**17.** A	**18.** B	**19.** A	**20.** C
21. B	**22.** A	**23.** B	**24.** B	**25.** A	**26.** C	**27.** D	**28.** C	**29.** B	**30.** C