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 .

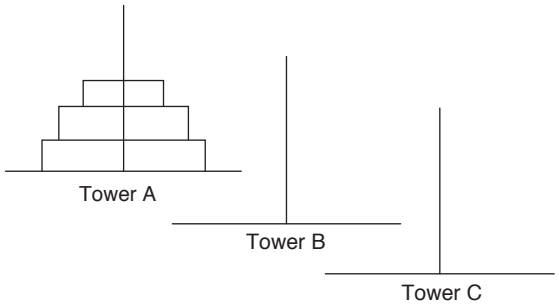
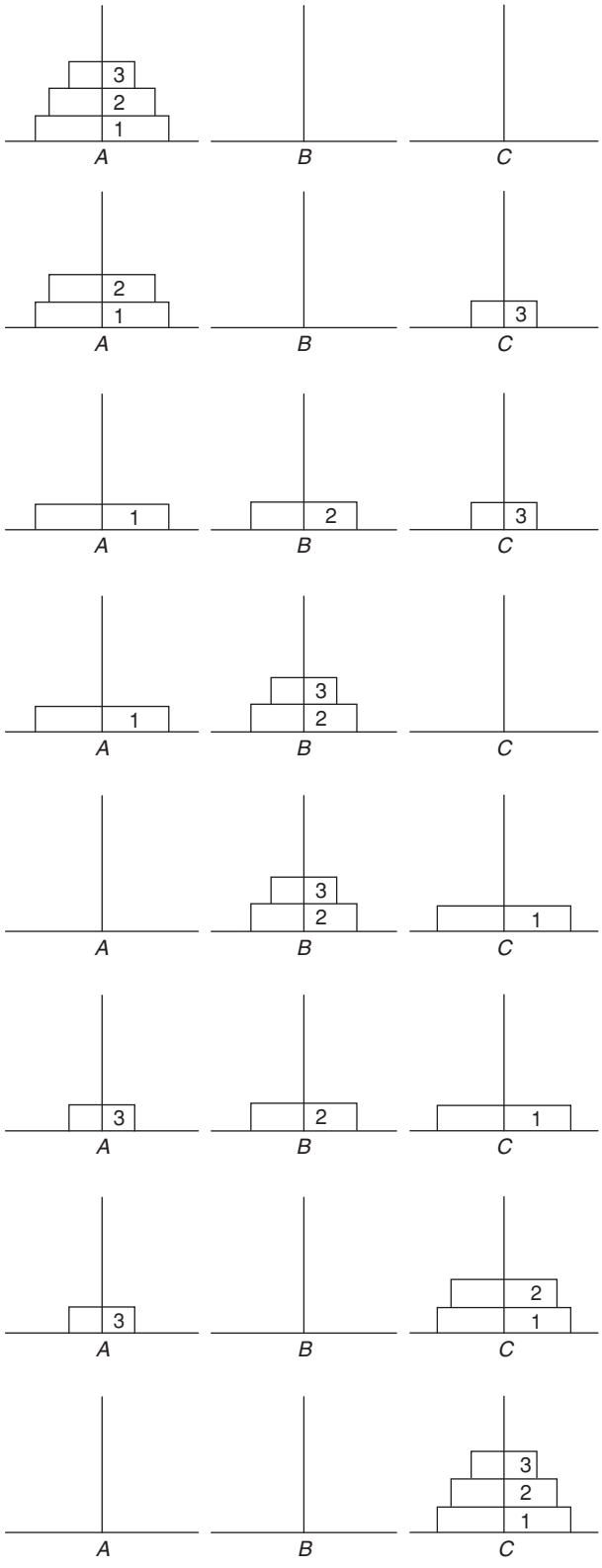


Figure 1 Towers of Hanoi

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:

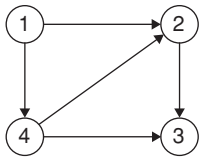


Figure 2 Directed graph

	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

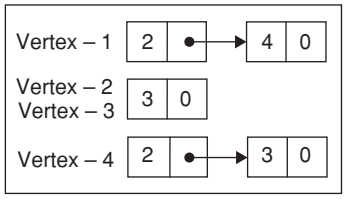


Figure 4 Adjacency lists

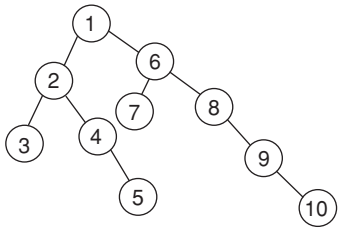
There are edges from vertex – 1 to vertex – 2 and 4, so the adjacency list for 1 has items 2 and 4 linked together in the format given above.

- 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 '$2i + 1$'.

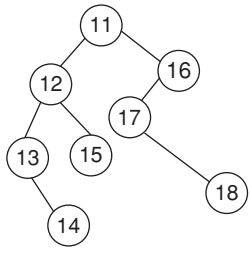
Tree Traversals

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

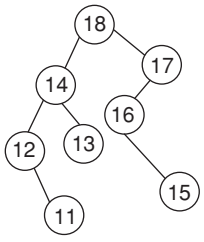
Pre-order Traversal

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

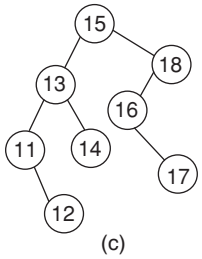
1. Visit the root.
2. Visit in pre-order the sub trees with roots $v_1, v_2 \ldots v_k$ in that order.



(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:

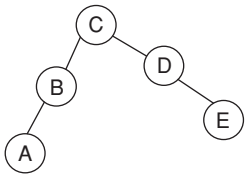
1. Visit in post-order the sub trees with roots v_1, v_2, v_3, … v_k in that order.
2. Visit the root r.

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 $n^2 \in O(n^3)$.

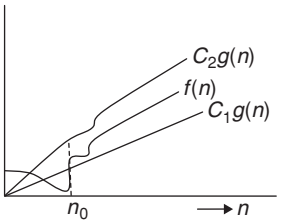
Order of Growth

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.

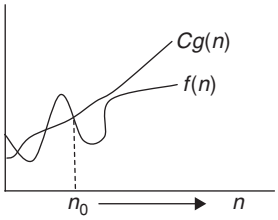
Definition: We say that a function $f(n)$ is theta of $g(n)$ written as $f(n) = \theta(g(n))$ if such exists positive constants C_1, C_2 and n_0 such that $0 \le C_1g(n) \le f(n) \le C_2\, g(n)$, $\forall\ n \ge n_0$.

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
$0 \le f(n) \le Cn^2$, $\forall\ n \ge n_0$

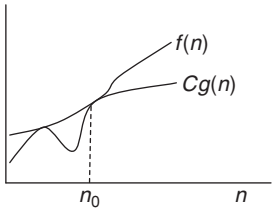
Example 3:

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

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

- 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

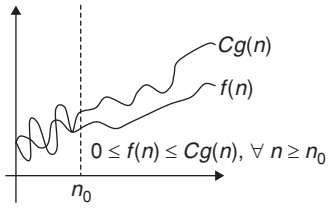
$$\therefore\ an^2 + bn + c = \Omega(n^2)$$

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

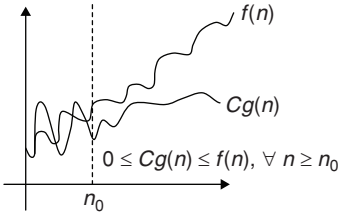
Example 4: Let $f(n) = 5.5n^2 - 7n$.
Verity whether $f(n)$ is $\Omega(n^2)$

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 $C = 3$

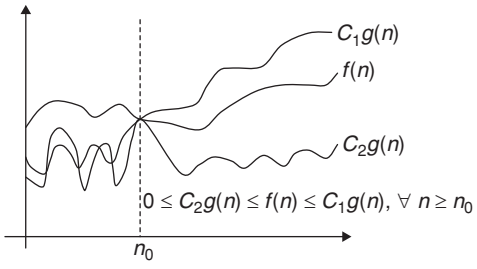
This shows that there exists positive constants $C = 3$ and $n_0 = 2.8$, such that $0 \le Cn^2 \le f(n)$, $\forall n \ge n_0$.



(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.

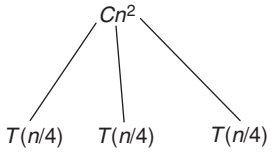
Example:

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 $n/4$.

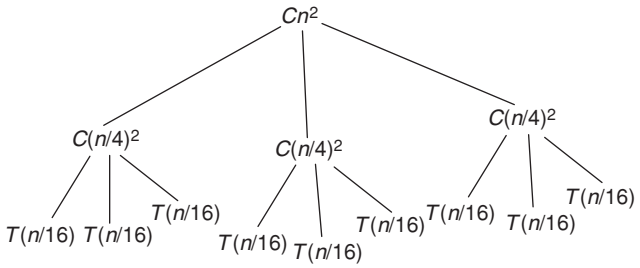


Figure 8 Recursion tree for $T(n) = 3T(n/4) + cn^2$

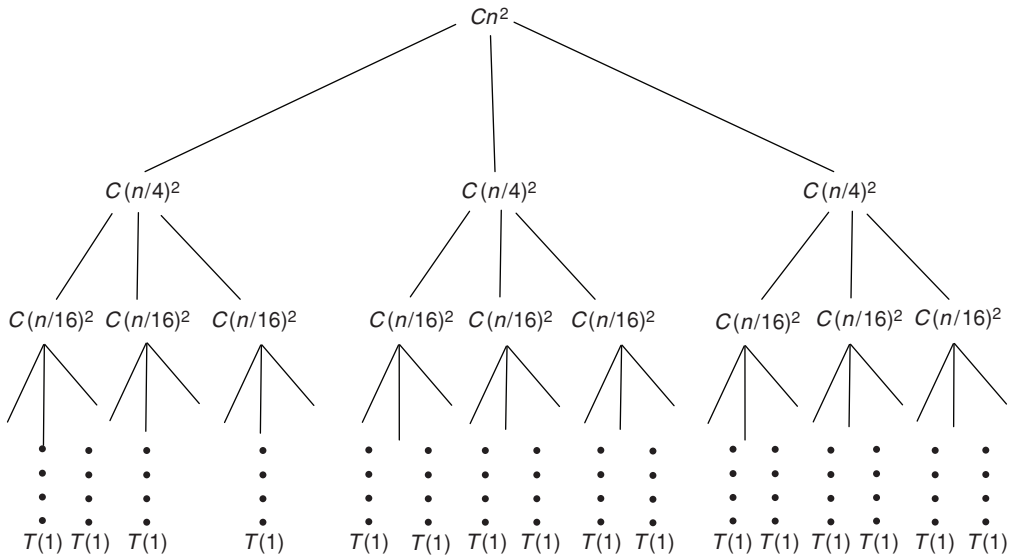


Figure 9 Expanded Recursion tree with height $\log_4 n$ ($\therefore$ levels $\log_4 n + 1$)

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);
                  }
            }
```

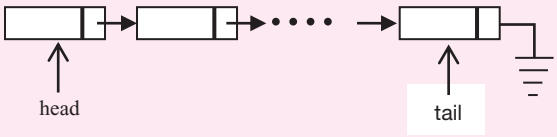
```
}
```

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			