COMPUTER SCIENCE & INFORMATION TECHNOLOGY

Algorithms

Comprehensive Theory
with Solved Examples *and* Practice Questions

MADE EASY PUBLICATIONS

www.madeeasypublications.org

MADE EASY Publications Pvt. Ltd.

Corporate Office: 44-A/4, Kalu Sarai (Near Hauz Khas Metro Station), New Delhi-110016 | **Ph. :** 9021300500

Email : infomep@madeeasy.in | **Web :** www.madeeasypublications.org

Algorithms

EDITIONS

First Edition : 2015
Second Edition : 2016
Third Edition : 2017
Fourth Edition : 2018
Fifth Edition : 2019
Sixth Edition : 2020
Seventh Edition : 2021
Eighth Edition : 2022
Ninth Edition : 2023

Tenth Edition : 2024

CONTENTS

Algorithms

Algorithms

Goal of the Subject

This course provides an introduction to mathematical modeling of computational problems. It covers the common algorithms, algorithmic paradigms, and data structures used to solve these problems. The course emphasizes the relationship between algorithms and programming, and introduces basic performance measures and analysis techniques for these problems.

Algorithms

INTRODUCTION

In this book we tried to present the algorithms in a most simplified way. Each topic required for GATE is crisply covered with illustrative examples and each chapter is provided with Student Assignment at the end of each chapter so that the students get the thorough revision of the topics that he/she had studied. This subject is carefully divided into six chapters as described below:

Chapter 1: Asymptotic Analysis of Algorithms: In this chapter we discuss the asymptotic notations to represent the average, worst and best cases and we also learn how to identify time complexity of given algorithm.

Chapter 2: Recurrence Relations: In this chapter we study the three methods to study solve recurrence relations. The three methods are substitution method, master theorem and recursion tree method.

Chapter 3: Divide Conquer: In this chapter we discuss the various algorithms that can be implemented using divide and conquer paradigm namely merge and quick sort, Strassen's matrix multiplications. We also discuss the other sorting techniques.

Chapter 4: Greedy Algorithms: In this chapter we discuss the graph representations, algorithms' that require local optimizations at each step. Those algorithms include job scheduling, fractional Knapsack, Prim's and Kruskal's algorithms for MST's and other algorithms for shortest paths.

Chapter 5: Graph based Algorithms: In this chapter we discuss the BFS and DFS traversals. We also discuss the topological sorting algorithm.

Chapter 6: Dynamic Programming: In this chapter we discuss the algorithms for which greedy fails to give correct solution. We will discuss matrix chain multiplication, Floyd Warshall's algorithm, 0/1 Knapsack, longest common sequences and other algorithms.

■■■■

CHAPTER

1

Asymptotic Analysis of Algorithms

1.1 NEED FOR PERFORMANCE ANALYSIS

There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc. Why to worry about performance?

The answer to this is simple — we can have all the above things only if we have performance.

Goal: To write an algorithm which takes minimum time irrespective of the machine whether it is supercomputer or normal desktop.

Choosing the Best Algorithm

One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.

1. It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
2. It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

ASYMPTOTIC ANALYSIS is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.

For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic). To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer and Binary Search on a slow computer. For small values of input array size n, the fast computer may take less time. But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine. The reason is the **order of growth** of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after certain values of input size.

Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take 1000 *n* log *n* and 2 *n* log *n* time respectively on a machine. Both of these algorithms are asymptotically same (order of growth is *n* log *n*). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis. Also, in asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower always performs better for your particular situation. So, you may end up choosing an algorithm that is asymptotically slower but faster for your software.

Types of Asymptotic analysis:

1. Apriori analysis of algorithm. 2. Apostiari analysis of algorithms.

Apriori analysis	Apostiari analysis
1. It means we do analysis (space and time) of an algorithm prior to running it on specific system. 2. It does not depends on system. 3. It provides approximate answer.	1. It means we do analysis of algorithm only after running it on system. 2. It directly depends on system and changes from system to system 3. It gives exact answer.

1.2 WORST, AVERAGE AND BEST CASES

We can have three cases to analyze an algorithm:

1. Worst Case 2. Average Case 3. Best Case

Let us consider the following implementation of Linear Search.

```
#include <stdio.h>
// Linearly search x in arr []. If x is present then return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
      int i;
      for (i=0; i<n; i++)
      {
            if (arr[i] == x)
            return i;
      }
      return -1;
}
/* Driver program to test above functions*/
int main()
{
      int arr[] = {1, 10, 30, 15};
      int x = 30;
      int n = sizeof(arr)/sizeof(arr[0]);
      printf("%d is present at index %d", x, search(arr, n, x));
      getchar();
      return 0;
}
```

1.2.1 Worst Case Analysis

- In worst case analysis, we shall usually concentrate on finding only the **worst-case running time**, that is, the longest running time for any input of size n.
- The worst case analysis is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer.

Example: In the program given in section 1.2. That is for **linear search**, the worst case happens when the element to be searched is not present in the array (or) it is the last element, the search function compares it with all the elements of arr[ ] one by one. Therefore the worst case time complexity of linear search would be $\theta(n)$.

1.2.2 Average Case Analysis

- In average case analysis, we take all possible inputs and calculate computing time for all of the inputs.
- Considering all the cases, calculate the sum, and divide the sum by total number of inputs.

Example: For the linear search problem, let us assume that all cases are uniformly distributed (including the case of X not being present in array).

Considering all possible inputs:

i.e., when element found at 1st position — $\theta(1)$
when element found at 2nd position — $\theta(2)$
⋮
⋮
⋮
when element found at n^{th} position — $\theta(n)$

$$\text{Sum} = \sum_{i=1}^{n} \theta(i)$$

$$\text{Average case time} = \frac{\sum_{i=1}^{n} \theta(i)}{n} = \frac{\theta(n)(n+1)}{2} = \frac{\theta\left(\frac{n^2+n}{2}\right)}{n} = \theta(n)$$

1.2.3 Best Case Analysis

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Theta(1)$.

- Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.
- The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
- The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

1.3 ASYMPTOTIC NOTATIONS

Let f be a non negative function. Then we can define the three most common asymptotic bounds as follows.

1.3.1 Big-Oh(O)

We say that $f(n)$ is Big-O of $g(n)$, written as $f(n) = O(g(n))$, iff there are positive constants c and n_0 such that

$$0 \le f(n) \le c \cdot g(n) \text{ for all } n \ge n_0$$

If $f(n) = O(g(n))$, we say that $g(n)$ is an **upper bound** on $f(n)$.

Example 1.1 Let us consider a given function:

$f(n) = 4 \cdot n^3 + 10n^2 + 5n + 8$

$g(n) = n^3$

Checking whether $f(n) = O(g(n))$ or not?

Solution:

For above condition to be true

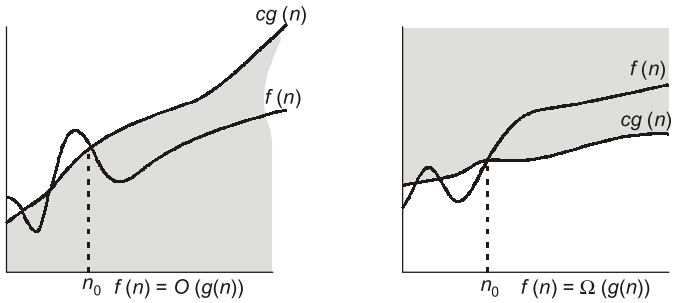
$0 \le f(n) \le c \cdot g(n)$

$4n^3 + 10n^2 + 5n + 8 \le c \cdot n^3$

when $c = 5$ and $n \ge 4$

$f(n)$ is always lesser than $g(n)$

Hence above statement is true.



1.3.2 Big-Omega (Ω)

We say that $f(n)$ is Big-Omega of $g(n)$, written as $f(n) = \Omega(g(n))$, iff there are positive constants c and n_0 such that

$$0 \le c \cdot g(n) \le f(n) \text{ for all } n \ge n_0$$

If $f(n) = \Omega(g(n))$, we say that $g(n)$ is a **lower bound** on $f(n)$.

Example 1.2 Let us consider a given function:

$f(n) = 3n + 2$

$g(n) = n$

Checking whether $f(n) = \Omega(g(n))$ or not?

Solution:

For above condition to be true

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

$c \cdot n \le 3n + 2$

Student's Assignments 1

Q.1 Which one of the following is true?

1. $an = o(n^2)\ a \geq 0$
2. $an^2 = O(n^2)\ a > 0$
3. $an^2 \neq o(n^2)\ a > 0$

(a) Only 1 and 2 are correct
(b) Only 1 is correct
(c) 1 and 3 are correct only
(d) All are correct

Q.2 Which of the following statements is true?

S1: $\frac{1}{2}n^2 = \omega(n)$ S2: $\frac{1}{2}n^2 = \omega(n^2)$

(a) S1 is correct
(b) S2 is correct
(c) S1 and S2 both are correct
(d) None of the above

Q.3 $f(n) = 3n^2 + 4n + 2$. Which will be the exact value for $f(n)$

(a) $\Theta(n^2)$ (b) $o(n^2)$
(c) $O(n^2)$ (d) $\Omega(n^2)$

Q.4 $f(n) = O(g(n))$ if and only if

(a) $g(n) = O(f(n))$ (b) $g(n) = \omega(f(n))$
(c) $g(n) = \Omega(f(n))$ (d) None of these

Q.5 $f(n) = o(g(n))$ if and only if

(a) $g(n) = \Omega(f(n))$
(b) $g(n) = \omega(f(n))$
(c) $g(n) = \Omega(f(n))$ and $g(n) = \omega(f(n))$
(d) None of these

Q.6 Which of the fallowing is not correct?

(a) $f(n) = O(f(n))$
(b) $c^* f(n) = O(f(n))$ for a constant C
(c) $(f(n) + g(n)) = o(g(n) + f(n))$
(d) None of the above

Q.7 $f(n) = \Theta(g(n))$ is

(a) $0 \leq C_1 g(n) \leq f(n) \leq C_2 g(n)\ \forall n \geq n_0$ where C_1, C_2, n_0 are + integer
(b) $0 \leq C_1 g(n) \leq f(n)\ \forall n \geq n_0$ where C_1, C_2, n_0 are + integer
(c) $0 \leq f(n) \leq C\ g(n)\ \forall n \geq n_0$ where C_1, n_0 are + integer
(d) None of the above

Q.8 $f(n) = O(g(n))$ implies

(a) $0 \leq C_1 g(n) \leq f(n)\ \forall n \geq n_0$ where C_1, n_0 are + integer
(b) $0 \leq C_1 f(n) \leq C_2 g(n)\ \forall n \geq n_0$ where C_1, C_2, n_0 are + integer
(c) $O \leq f(n) \leq C\ g(n)\ \forall n \geq n_0$ where C, n_0 are + integer
(d) $0 \leq C_1 g(n) \leq C_2 f(n)\ \forall n \geq n_0$ where C_1, C_2, n_0 are + integer

Q.9 $f(n) = \Theta(g(n))$ implies

(a) $f(n) = O(g(n))$ only
(b) $f(n) = \Omega(g(n))$ only
(c) $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
(d) None of the above

Q.10 $f(n) = o(g(n))$ implies

(a) $0 \leq f(n) \leq Cg(n)$ such that there exists some positive constant C and $n_0 > \forall n \geq n_0$
(b) $0 \leq f(n) < Cg(n)$ for every +ve constant $C >$ 0 there exists $n_0 > 0$, $\forall n \geq n_0$
(c) $0 \leq C_1 f(n) \leq C_2 g(n)\ \forall n \geq n_0$ such that C_1, C_2 and n_0 are +ve constants
(d) $0 \leq f(n) < C\ g(n)$ for some +*ve* constant $C >$ $0\ \exists\ n_0 > 0$, $\forall n \geq n_0$

Q.11

```
void x (int A[ ]), int n)
{   int i, j;
    for (i = 0; i < n; i++)
    {   j = n – 1;
        while (j > i)
        {
            swap (A[j], A[j – 1]);
            j --;
        }
    }
}
```

What will be time complexity of the above algorithm if swap function takes constant time?

(a) $O(n)$ (b) $O(n^2)$
(c) $O(n \log_2 n)$ (d) $O(n^3)$

Q.12 Which of the following statements are True?

(a) $100\, n\log n = O(n\log n/100)$

(b) $\sqrt{\log n} = O(\log\log n)$

(c) If $0 < x < y$ then $n^x = O(x^y)$

(d) $2^n \neq O(n^c)$ where c is a constant and $c > 0$

Q.13 Which of the following statements (k, m are constants) are True?

(a) $(n + k)^m = O(n^m)$ (b) $2^{n+1} = O(2^n)$

(c) $2^{2^n} = O(2^n)$ (d) $f(n) = O(f(n)^2)$

Q.14 Which of the following statements are True?

(a) $n^2 \cdot 2^{3\log_2 n} = \Theta(n^5)$

(b) $\frac{4^n}{2^n} = \Theta(2^n)$

(c) $2^{\log_2 n} = \Theta(n^2)$

(d) if $f(n) = O(g(n))$ then $2^{f(n)} = O(2^{g(n)})$

Q.15 Consider $f(n)$, $g(n)$ and $h(n)$ be function defined as follows:

$$f(n) = \Omega(n^3)$$
$$g(n) = O(n^2)$$
$$h(n) = \Theta(n^2)$$

Which of the following represents correct asymptotic solution for $f(n) + [g(n) \times h(n)]$?

(a) $\Omega(n^3)$ (b) $O(n^4)$

(c) $\Theta(n^4)$ (d) $O(n^3)$

Q.16 Consider the following C-function:

```
int Rec (int n)
{
    int i, j, k, p, q = 0;
    for (i = n; i > 1; i/2) {
        p = 0;
        for (j = 1; j < n; j++)
            p = p +1;
        for (k = 1; k < p; k = k × 3)
            q++;
    } return q
}
```

Then time complexity of Rec in term of Θ notation is

(a) $\Theta(n)$ (b) $\Theta(n^2)$

(c) $\Theta(n \log n)$ (d) $\Theta(n \log\log n)$

Q.17 Consider following functions:

$$f(n) = (\log n)^{n-1}$$
$$g(n) = 2^n$$
$$h(n) = e^n / n$$

Which of the following is true?

(a) $f(n) = O(h(n))$ (b) $f(n) \neq O(n(n))$

(c) $g(n) = \Omega(f(n))$ (d) None of these

Q.18 Consider the following C function:

```
void dosomething (int n)
{
    int m, j, k;
    for ( j = 0; j < 200; j++)
    {
        for (k = 0; k < n; k++)
        {
            for (m = 0; m < j, m++)
            printf("%d", i + m);
        }
    }
}
```

What is the time complexity of the above function?

(a) $O(n^2)$ (b) $O(n\log n)$

(c) $O(n)$ (d) $O(n^2 \log n)$

Q.19 Let $g(n) = \Omega(n)$, $f(n) = O(n)$ and $h(n) = \theta(n)$ then what is the time complexity of $[g(n)\ f(n) + h(n)]$

(a) $O(n)$ (b) $\theta(n)$

(c) $\Omega(n)$ (d) $\theta(n^2)$

Q.20 Consider the following functions:

$$f_1 = n^4,\ f_2 = 4^n,\ f_3 = n^{110/37},\ f_4 = \left(\frac{119}{37}\right)^n$$

Which of the following is correct order of increasing growth rate?

(a) f_1, f_3, f_2, f_4 (b) f_3, f_1, f_4, f_2

(c) f_3, f_1, f_2, f_4 (d) f_1, f_3, f_4, f_2

Q.21 Consider the following program segments:

```
main ( )
    {
        int i = 0, m = 0;
    for (i = 1; i < n; i++) {
            for ( j = 1; j < i * i; j++) {
                if (( j % i) = = 0)
            for (k = 1; k < j; k++) {
```

```
                    m = m + 1; }
                }
            }
        }
```

What is the time complexity of above program?

(a) $O(n^2 \log n)$ (b) $O(n^3)$

(c) $O(n^4)$ (d) $O(n \log n)$

Answer Key:

1. (d)	**2.** (a)	**3.** (a)	**4.** (c)	**5.** (c)
6. (c)	**7.** (a)	**8.** (c)	**9.** (c)	**10.** (b)
11. (b)	**12.** (a, c, d)	**13.** (a, b)	**14.** (a, b)	**15.** (a)
16. (c)	**17.** (b)	**18.** (c)	**19.** (c)	**20.** (b)
21. (c)				

Student's Assignments Explanations

1. (d)

1. $an = o(n^2)$, $a \geq 0$
   $o \leq an < c \cdot n^2$ for all value of c this condition is true. Hence 1 is correct.
2. $an^2 = O(n^2)$
   $O \leq an^2 \leq c \cdot n^2$ when $c \geq a$ this condition is true. Hence, 2 is correct.
3. $an^2 \neq o(n^2)$
   for this statement to be false $o \leq an^2 < c \cdot n^2$ must be true

Since above condition needs to be true for all c thus if we could prove for some c that violates above condition. Whole option becomes true.

$O \leq an^2 < c \cdot n^2$ is not true when $c < a$ thus $an^2 \neq o(n^2)$ is true.

2. (a)

S1: $c \cdot \frac{1}{2} n^2 > n$ for every $c > 0$ which can be seen easily.

S2: False since $c \cdot \frac{1}{2} n^2 > n^2$ is false when $c = 2$ hence for every c it is not true thus, false.

3. (a)

$$f(n) = 3n^2 + 4n + 2$$

maximum degree = 2

$\Rightarrow \theta(n^2)$

Note: option (c) and (d) are also possible but they are not exact.

4. (c)

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

Take some arbitrary values in order to solve these type of questions.

As given, $f(n) = O(g(n))$

$\Rightarrow f(n) \leq g(n)$

Let $f(n) = n$ and $g(n) = n^2$

(a) $g(n) = O(f(n)) \rightarrow n^2 = O(n)$ or $n^2 \leq n$ not true

(b) $g(n) = \omega(f(n)) \rightarrow g(n) > f(n) \rightarrow n^2 > n$
True (only for assumed value).
But it become false when both f and g are n.

(c) $g(n) = \Omega(f(n)) \rightarrow g(n) \geq f(n)$
Same thing written as given in question.

5. (c)

$$f(n) = o(g(n))$$

$\Rightarrow f(n)$ is strictly lesser than $g(n)$

Thus, $f(n) < g(n)$

For (c), $g(n) = \Omega(f(n))$

$\Rightarrow g(n) \geq f(n)$

Satisfied by given equation.

$\Rightarrow g(n) = \omega(f(n))$

$\Rightarrow g(n) > f(n)$ this is also satisfied

6. (c)

(a) $f(n) = O(f(n))$ by reflexive property.

(b) $c * f(n) = O(f(n))$ since constant does not matter in case of asymptotic analysis.

(c) $(f(n) + g(n)) = o(g(n) + f(n))$
let $f(n) = n^2$
$g(n) = n$
$(n^2 + n) = o(n^2 + n)$
$n^2 = o(n^2)$ not true since tightest bound not allowed.

9. (c)

Go Through the Topic θ notation.

$F(n) = \theta(g(n))$ if and only if $f(n) = O(g(n))$ and $F(n) = \Omega(g(n)$ simultaneously.

CHAPTER

2

Recurrence Relations

2.1 INTRODUCTION

A recurrence is an equation or inequality that describes a function in term of its value on smaller inputs. For instance we describe the worst-case running time $T(n)$ of the **Merge-Sort** procedure by the recurrence whose solution is $\Theta(n \log n)$. The recurrence relation for Merge-Sort is shown below.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases} \quad \ldots(2.1)$$

Recurrences can take many forms. For example, a recursive algorithm might divide subproblems into unequal sizes, such as a 2/3 to 1/3 split. If the divide and combine steps linear time, such an algorithm would give rise to the recurrence

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

This chapter offers three methods for solving recurrences–that is, for obtaining asymptotic "Θ" or "O" bounds on the solution.

Substitution Method: This method consists of guessing an asymptotic (upper or lower) bound on the solution, and trying to prove it by induction.

Recurrence Tree Method:

- Each node represents the cost of a single subproblem.
- We sum the cost within each level of the tree to obtain a set of per level costs.
- At last we sum all the per level costs to determine the total cost of all levels of the recursion.

Master Method: This is a cookbook method for determining asymptotic solutions to recurrences of a specific form.

Technicalities in Recurrences

We neglect certain technical details when we state and solve recurrences. For example, if we call **Merge-Sort** on n elements when n is odd, we end up with subproblems of size $\left\lfloor \frac{n}{2} \right\rfloor$ and $\left\lfloor \frac{n}{2} \right\rfloor$. Neither size is actually $\frac{n}{2}$ because $\frac{n}{2}$ is not an integer when n is odd. The actual recurrence describing the worst-case of **Merge-Sort** is given by

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 \end{cases}$$

The running time of an algorithm on a constant-sized input is a constant, the recurrences that arise from the running times of algorithms generally have $T(n) = \Theta(1)$ for sufficiently small n. We shall generally omit statements of the boundary conditions of recurrences and assume that $T(n)$ is constant of small n. For example, we normally state recurrence (2.1) as $T(n) = 2T(n/2) + \Theta(n)$, without explicitly giving values for small n. The reason is that although changing the value of $T(1)$ changes the exact solution to the recurrence, the solution typically doesn't change by more than a constant factor, and so the order of growth is unchanged.

Making a Good Guess

If $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ is given

- Clearly it looks difficult because of the added 17 in the argument to T on the RHS.
- This term not substantially affect the solution to the recurrence.
- When n is large, constant factor does not affect.
  So the recurrence relation becomes
  $$T(n) = 2T(\lfloor n/2 \rfloor) + n$$
- Also floors and ceilings does not affect the time complexity.
  Thus, $T(n) = 2T(n/2) + n$
  This can be solved easily.

2.2 SUBSTITUTION METHOD

Example 2.1 What is the time complexity of the following recurrence relation

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

Solution:

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &\vdots \\ &= T(n-(n-1)) + (n-(n-2)) + (n-(n-3)) + \ldots + n-2 + n-1 + n \\ &= T(1) + 2 + 3 + 4 + \ldots + n \\ &= 1 + 2 + 3 + \ldots + n \\ &= \frac{n(n+1)}{2} \Rightarrow O(n^2) \end{aligned}$$

Example 2.2 Find the complexity of the recurrence:

$$T(n) = \begin{cases} 2T(n-1) - 1 & \text{If } n > 0 \\ 1 & \text{otherwise} \end{cases}$$

Solution:

$$T(n) = 2T(n-1) - 1$$

Solution:

$$\begin{aligned} T(n) &= T(n-1)n \\ &= [T(n-2)\times(n-1)]n \qquad \text{by putting } n = n-1 \\ &= T(n-3)\times(n-2)\times(n-1)\times n \\ &\vdots \\ &= T(n-(n-1))\times(n-(n-2))\ldots(n-2)\times(n-1)\times n \\ &= 1\times 2\times 3\times\ldots\times n = n! \\ &= O(n^n) \qquad [\because\ n! = O(n^n)] \end{aligned}$$

Example 2.5 What is the time complexity of the following recurrence relation (use substitution method)?

$$T(N) = 3T(n/4) + n \text{ for } n > 1$$

Solution:

Assume n to be a power of 4, i.e., $n = 4^k$ and $k = \log 4n$

$$\begin{aligned} T(n) &= 3T(n/4) + n \\ &= 3(3T(n/16) + 3(n/4) + n \\ &= 9T(n/16) + 3(n/4) + n \\ &= 27T(n/64) + 9(n/16) + n + 3(n/4) + n \\ &= \ldots \\ &= 27T(n/64) + 9(n/16) + n + 3(n/4) + n \end{aligned}$$

With $n = 4^k$ and $T(1) = 1$

$$T(n) = 3^k T\left(\frac{n}{4^k}\right) + \sum_{i=0}^{k-1}\frac{3i}{4^i}n = 3^{\log_4 n}T(1) + \sum_{i=0}^{(\log_4 n)-1}\frac{3i}{4^i}n$$

$$= n^{\log_4 3} + \sum_{i=0}^{(\log_4 n)}\frac{3i}{4^i}n$$

We use the formula $a^{\log_b n} = n^{\log_b a}$, n remains constant throughout the sum.

$$\sum_{i=0}^{m} X^i = \frac{X^{m+1}-1}{X-1}$$

In this case X = 3/4 and $m \log_4 n - 1$. We get

$$T(n) = n^{\log_4 3} + n\frac{(3/4)^{\log_4 n+1}-1}{(3/4)-1}$$

$$\left(\frac{3}{4}\right)^{\log_4 n} = n^{\log_4(3/4)} = n^{\log_4 3-\log_4 4} = n^{\log_4 3-1} = \frac{n^{\log_4 3}}{n}$$

If we plug this back, we get

$$T(n) = n^{\log_4 3} + n\frac{\frac{n^{\log_4 3}}{n}-1}{(3/4)-1} = n^{\log_4 3} + \frac{n^{\log_4 3}-n}{-1/4}$$

$$= n^{\log_4 3} + 4(n - n^{\log_4 3}) = 4n - 3n^{\log_4 3}$$

$$= \Theta(n)$$

Since, $a = 1$, $b = 2$,

$$S^{\log_2 1} = S^0 = 1 < \log S$$

Thus, $T(S) = \log S$

or $T(n) = O(\log n)$

Summary

- Three methods for solving recurrences–that is, for obtaining asymptotic "Θ" or "O" bounds on the solution.
  1. Substitution Method:
  2. Recursion Tree Method-Iteration Method:
  3. Master Method:
- Recurrences characterize the running times of algorithms. The substitution method for solving recurrences comprises two steps:
  1. Guess the form of the solution.
  2. Use mathematical induction to find the constants and show that the solution works.
- A recursion tree models the costs (time) of a recursive execution of an algorithm. The recursion tree method is good for generating guesses for the substitution method.
- The Master Theorem applies to recurrences of the following form: $T(n) = aT(n/b) + f(n)$

Student's Assignments 1

Q.1 Find the time complexity of the following recurrence relation using Master Theorem

(a) $T(n) = 3T(n/2) + n^2$

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

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

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

(e) $T(n) = 16T(n/4) + n$

(f) $T(n) = 2T(n/2) + n/\log n$

(g) $T(n) = 2T(n/4) + n^{0.51}$

(h) $T(n) = 0.5T(n/2) + 1/n$

(i) $T(n) = 16T(n/4) + n!$

(j) $T(n) = \sqrt{2}\,T(n/2) + \log n$

(k) $T(n) = 3T(n/2) + n$

(l) $T(n) = 3T(n/3) + \sqrt{n}$

(m) $T(n) = 4T(n/2) + cn$

(n) $T(n) = 3T(n/4) + n\log n$

(o) $T(n) = 3T(n/3) + n/2$

(p) $T(n) = 6T(n/3) + n^2 \log n$

(q) $T(n) = 4T(n/2) + n/\log n$

(r) $T(n) = 64T(n/8) - n^2 \log n$

(s) $T(n) = 7T(n/3) + n^2$

(t) $T(n) = 4T(n/2) + \log n$

Q.2 Find the time complexity?

(a) $T(n) = 4T(n-1)$ if $n > 0$ and 1 otherwise

(b) $T(n) = T(n-1) + \log n \quad n > 1$

(c) $T(n) = T(n-1) + 1/n$ if $n > 1$ and 1 if $n = 1$

(d) $T(n) = T(n-2) + 2\log n$ if $n > 1$ and 1 if $n = 1$

(e) $T(n) = T(n-2) + n^2$ if $n > 1$ and 1 if $n = 1$

Q.3 Consider the recurrance function:

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

Then $T(n)$ in terms of Θ notation is

(a) $\Theta(n \log n)$ (b) $\Theta(\sqrt{n}\log n)$

(c) $\Theta(\sqrt{n})$ (d) $\Theta(n \log \log n)$

Q.4 Which of the following represents most appropriate asymptotic solution for given reccurance:

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

(a) $O(n)$ (b) $O(\log n)$

(c) $O(\log \log n)$ (d) $O(\log n)^2$

Q.5 Consider the following function:

```
function (int n)
{
    if(n <= 1) return;
    for (int i = 1; i < n^2; i++)
    {
        printf("GATE");
    }
    function (0.6n);
}
```

Which of the following recursion formula depicts the running time of function, as a function of 'n'?

(a) $T(n) = T\left(\frac{3}{5}n\right) + O(1)$

(b) $T(n) = T\left(\frac{3}{5}n\right) + T\left(\frac{2}{5}n\right) + O(1)$

(c) $T(n) = T\left(\frac{3}{5}n\right) + O(n^2)$

(d) $T(n) = T\left(\frac{3}{5}n\right) + T\left(\frac{2}{5}n\right) + O(n^2)$

Q.6 Consider the recurrence relation:

$$T(n) = \begin{cases} T(\sqrt{n}) + \log n & \text{if } n \geq 2 \\ O(1) & \text{else} \end{cases}$$

Then $T(n)$ in terms of Θ notation is

(a) $\Theta(\log n)$ (b) $\Theta(n)$
(c) $\Theta(\log \log n)$ (d) $\Theta(n \log n)$

Q.7 Which of the following is correct?

(a) $\frac{e^{n\log n}}{n} < 2^{n\log n} < n^{\sqrt{n}}$

(b) $2^{n\log n} < \frac{e^{n\log n}}{n} < n^{\sqrt{n}}$

(c) $n^{\sqrt{n}} < \frac{e^{n\log n}}{n} < 2^{n\log n}$

(d) $2^{n\log n} < n^{\sqrt{n}} < \frac{e^{n\log n}}{n}$

Q.8 Consider the following recursive function find:

```
int find (int A[ ], int n)
{
    int sum = 0;
    if (n = = 0) return 0;
    sum = find (A, n – 1)
    if (A [n – 1] < 0) sum = sum + 1;
    return sum;
}
```

What is the worst case running time of above function find (A [ ], n) when array A has 0 to $n-1$ elements?

(a) $O(1)$ (b) $O(\log n)$
(c) $O(n)$ (d) $O(n^2)$

Q.9 Match the following recurrence relations with their time complexities:

List-I		List-II	
A.	$T(n) \geq 2T\left(\frac{n}{2}\right) + \theta(n)$	1.	$\theta(n \log n)$
B.	$T(n) \leq 2T\left(\frac{n}{2}\right) + \theta(n)$	2.	$O(n \log n)$
C.	$T(n) \leq T\left(\frac{n}{2}\right) + c$	3.	$\theta(n)$
		4.	$O(n)$
		5.	$O(\log_2 n)$
		6.	$O(\log_n 2)$
		7.	$\Omega(n \log_2 n)$

Codes:

	A	B	C
(a)	2	3	6
(b)	1	3	5
(c)	7	2	5
(d)	2	3	5

Q.10 Which of the following is true?

I. $T(n) = 2T\left(\frac{n}{2}\right) + n \log n \quad n \geq 2$

$T(1) = 0$
$T(n) = O(n \log n)$

II. $T(n) = T(n-1) + \frac{1}{n}$

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

(a) I only (b) II only
(c) Both I and II (d) Neither I nor II

Answer Key:

1. (a) $\Theta(n^2)$
(b) $\Theta(n^2 \log n)$
(c) $\Theta(2^n)$
(d) (Master theorem does not apply)
(e) $\Theta(n^2)$
(f) (Master theorem does not apply)
(g) $n^{0.51}$
(h) Master theorem does not apply
(i) $\Theta(n!)$
(j) $\Theta(\sqrt{n})$
(k) $\Theta(n^{\log 3})$
(l) $\Theta(n)$
(m) $\Theta(n^2)$
(n) $\Theta(n \log n)$
(o) $\Theta(n \log n)$
(p) $\Theta(n^2 \log n)$
(q) $\Theta(n^2)$
(r) (Master theorem does not apply)
(s) $\Theta(n^2)$
(t) $\Theta(n^2)$

2. (a) $O(4^n)$ (b) $O(n \log n)$
(c) $O(\log n)$ (d) $O(n \log n)$
(e) $O(n^3)$

3. (d) **4.** (b) **5.** (c) **6.** (a) **7.** (c)
8. (c) **9.** (c) **10.** (b)

Student's Assignments Explanations

1. (Sol.)

(a) $T(n) = 3T(n/2) + n^2$

Here $a = 3,\ b = 2$

$n^{\log_2 3} = n^{1.732} < n^2 (f(n))$

Hence, $T(n) = \Theta(n^2)$

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

Here $a = 4,\ b = 2$

$n^{\log_2 4} = n^2 \approx \Theta(f(n))$

Case 2 applied thus

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

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

$n^{\log_2 1} = n^0 = 1$

$f(n)$ is larger also polynomially larger because

$$\frac{f(n)}{n^{\log_b a}} = n^e \text{ i.e. } 2^n > n^e \quad \text{(True)}$$

Hence, TC $= O(2^n)$

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

Master theorem cannot be applied here because $a = 2^n$ that is not constant which should not been Master theorem.

(e) $T(n) = 16T(n/4) + n$

$n^{\log_4 16} = n^{\log_4 4^2} = n^2$

$n^2 = n$

Hence, $\Theta(n^2)$

(f) $T(n) = 2T(n/2) + \dfrac{n}{\log n}$

$n^{\log_2 2} = n$

$n > \dfrac{n}{\log n}$ but not polynomially

Since, $\dfrac{n^{\log_b a}}{f(n)} = \dfrac{n}{\frac{n}{\log n}} = \log n > n^E$

is not true.

Hence Master Theorem cannot be applied.

(g) $T(n) = 2T(n/4) + n^{0.51}$

$n^{\log_2 4} = n^{0.50} < n^{0.51}$

Hence, $\Theta(n^{0.51})$

(h) Master Theorem cannot be applied
Since $a \geq 1$ is not satisfied here.

(i) $T(n) = 16T(n/4) + n!$

$n^{\log_4 16} = n^2 < n!$

Also polynomially greater since

$$\frac{n!}{n^2} > n^E$$

Hence, $\Theta(n!)$