INTRODUCTION

Algorithm Analysis

Analysis of algorithms

- How good is the algorithm?
  - time efficiency
  - space efficiency

- Does there exist a better algorithm?
  - lower bounds
  - optimality

Analysis of algorithms

- Issues:
  - time efficiency
  - space efficiency

- Approaches:
  - theoretical analysis
  - empirical analysis

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*

- *Basic operation*: the operation that contributes the most towards the running time of the algorithm

$$T(n) \approx c_{op}C(n)$$

running time

execution time for basic operation or cost

Number of times basic operation is executed

Empirical analysis of time efficiency

- Select a specific (typical) sample of inputs

- Use physical unit of time (e.g., milliseconds)

  or

  Count actual number of basic operation's executions

- Analyze the empirical data

Best-case, average-case, worst-case

For some algorithms, efficiency depends on form of input:

- Worst case: $C_{worst}(n)$ – maximum over inputs of size n
- Best case: $C_{best}(n)$ – minimum over inputs of size n
- Average case: $C_{avg}(n)$ – “average” over inputs of size n

Example: Sequential search

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

//Searches for a given value in a given array by sequential search
//Input: An array $A[0..n-1]$ and a search key K
//Output: The index of the first element of A that matches K
// or -1 if there are no matching elements

$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
    $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

- Worst case — n key comparisons
- Best case — 1 comparisons
- Average case — (n+1)/2, assuming K is in A

Order of growth

- Most important: Order of growth within a constant multiple as $n \rightarrow \infty$

- Example:
  - How much faster will algorithm run on computer that is twice as fast?

  - How much longer does it take to solve problem of double input size?

Values of some important functions as $n \to \infty$

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3{\cdot}10^1$	10^2	10^3	10^3	$3.6{\cdot}10^6$
10^2	6.6	10^2	$6.6{\cdot}10^2$	10^4	10^6	$1.3{\cdot}10^{30}$	$9.3{\cdot}10^{157}$
10^3	10	10^3	$1.0{\cdot}10^4$	10^6	10^9		
10^4	13	10^4	$1.3{\cdot}10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7{\cdot}10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0{\cdot}10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes (because?)

- $O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$

Big-oh

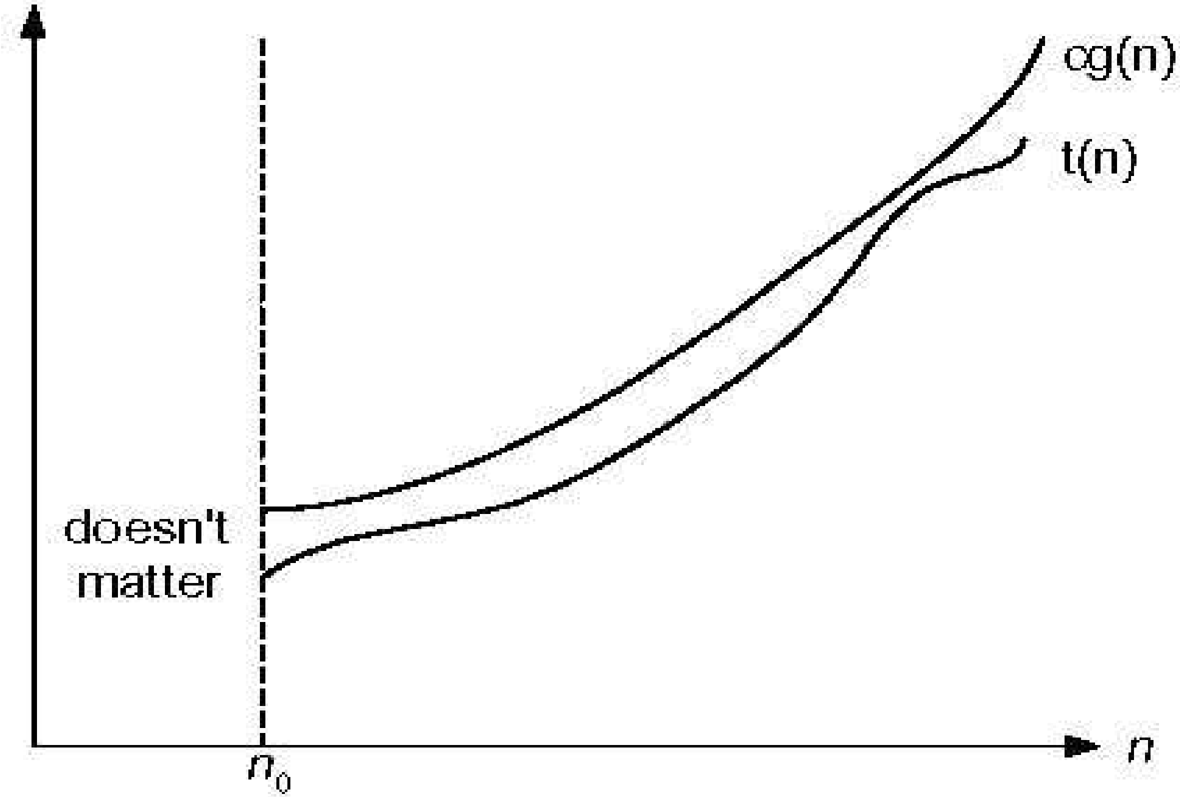


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

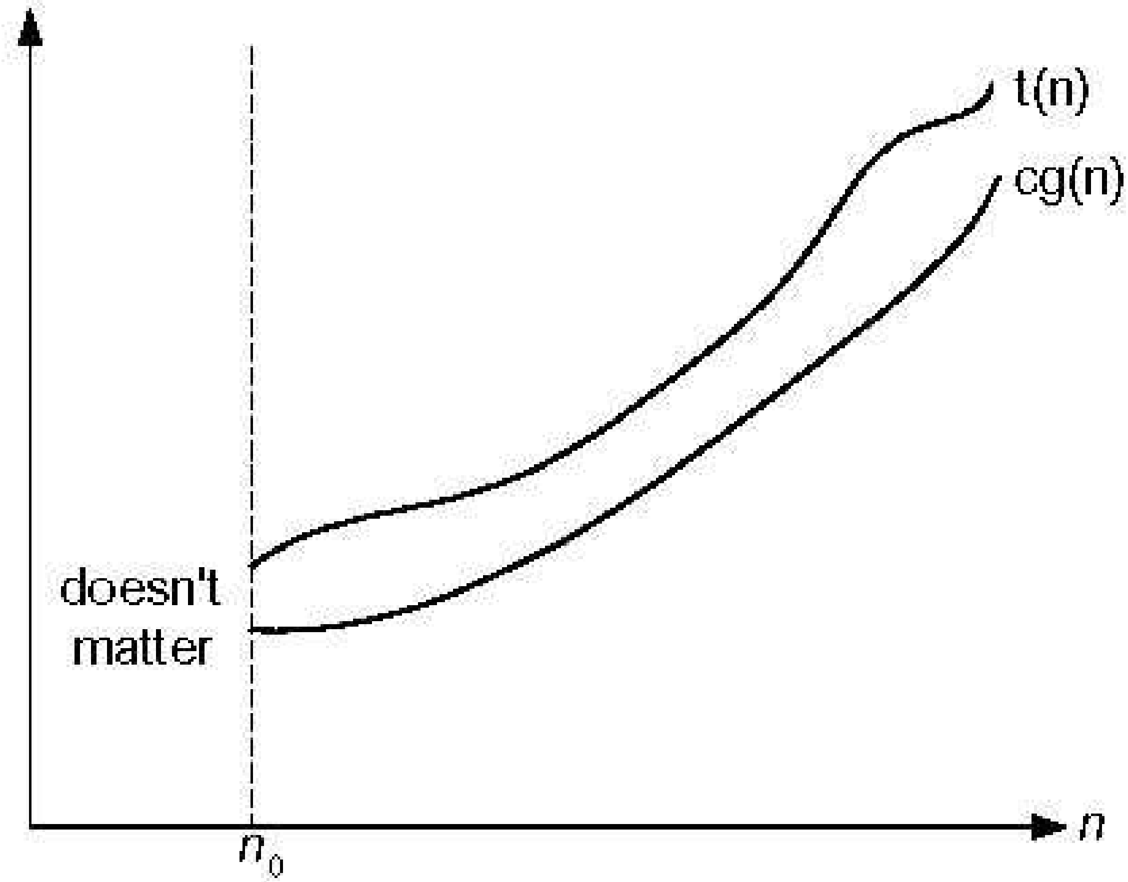


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

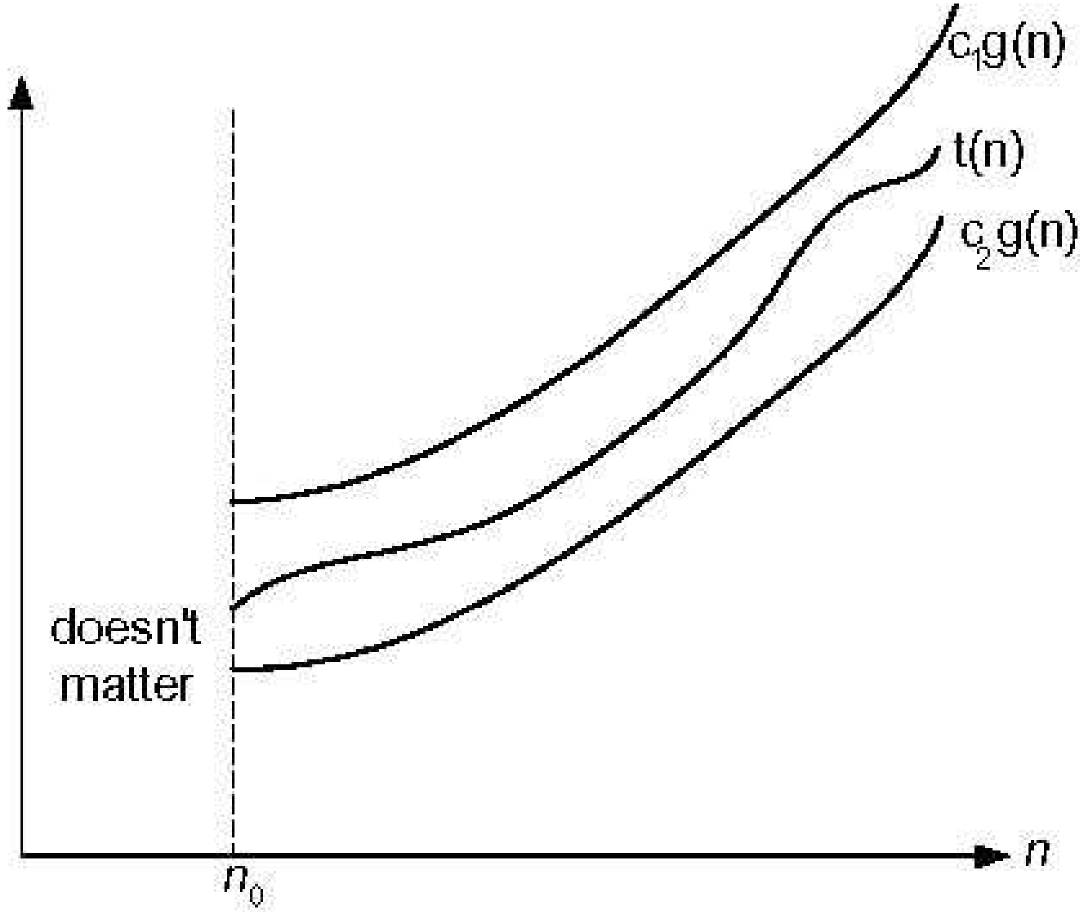


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Establishing order of growth using the definition

Definition: $f(n)$ is in $O(g(n))$, denoted $f(n) \in O(g(n))$, if order of growth of $f(n) \leq$ order of growth of $g(n)$ (within constant multiple), i.e., there exist positive constant c and non-negative integer n_0 such that

$$f(n) \leq c\, g(n) \text{ for every } n \geq n_0$$

Examples:

- $10n$ is in $O(n^2)$

- $5n+20$ is in $O(n)$

Ω-notation

- Formal definition
  - A function *t(n)* is said to be in *Ω(g(n)),* denoted *t(n)* $\in$ *Ω(g(n)),* if *t(n)* is bounded below by some constant multiple of *g(n)* for all large *n*, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

    $t(n) \geq cg(n)$ for all $n \geq n_0$

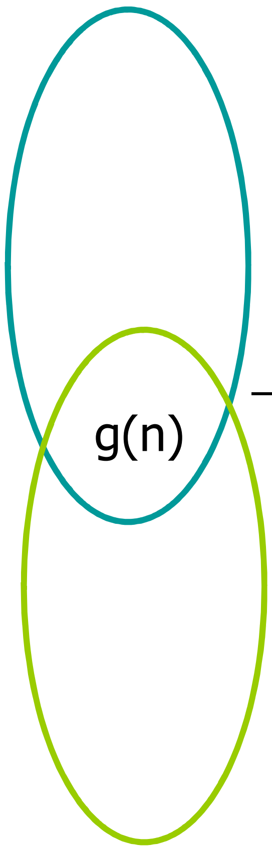
- Exercises: prove the following using the above definition
  - $10n^2 \in \Omega(n^2)$
  - $0.3n^2 - 2n \in \Omega(n^2)$
  - $0.1n^3 \in \Omega(n^2)$

Θ-notation

- Formal definition
  - A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., <u>if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that</u>

    $c_2\, g(n) \leq t(n) \leq c_1\, g(n)$ for all $n \geq n_0$

- Exercises: prove the following using the above definition
  - $10n^2 \in \Theta(n^2)$
  - $0.3n^2 - 2n \in \Theta(n^2)$
  - $(1/2)n(n+1) \in \Theta(n^2)$



>=

Ω(g(n)), functions that grow at least as fast as g(n)

=

Θ(g(n)), functions that grow at the same rate as g(n)

<=

O(g(n)), functions that grow no faster than g(n)

Theorem

- If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(max\{g_1(n), g_2(n)\})$.
  - The analogous assertions are true for the Ω-notation and Θ-notation.

- Implication: The algorithm's overall efficiency will be determined by the part with a larger order of growth, i.e., its least efficient part.
  - For example, $5n^2 + 3nlogn \in O(n^2)$

Proof. There exist constants *c1, c2, n1, n2* such that

$$t1(n) \le c1 * g1(n), \quad \text{for all } n \ge n1$$

$$t2(n) \le c2 * g2(n), \quad \text{for all } n \ge n2$$

Define $c3 = c1 + c2$ **and** $n3 = max\{n1, n2\}$. ***Then***

$$t1(n) + t2(n) \le c3 * max\{g1(n), g2(n)\}, \quad \text{for all } n \ge n3$$

Some properties of asymptotic order of growth

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

  Note similarity with $a \leq b$
- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then

  $$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

Also, $\Sigma_{1\leq i\leq n} \Theta(f(i)) = \Theta(\Sigma_{1\leq i\leq n} f(i))$

Establishing order of growth using limits

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{ order of growth of } g(n) \\ c > 0 & \text{order of growth of } T(n) = \text{ order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{ order of growth of } g(n) \end{cases}$$

Examples:

- $10n$ vs. n^2
- $n(n+1)/2$ vs. n^2

L'Hôpital's rule and Stirling's formula

L'Hôpital's rule: If $lim_{n\to\infty} f(n) = lim_{n\to\infty} g(n) = \infty$ and the derivatives f', g' exist, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Example: log n vs. n

Stirling's formula: $n! \approx (2\pi n)^{1/2} (n/e)^n$

Basic asymptotic efficiency classes

1	**constant**
log n	**logarithmic**
n	**linear**
$n \log n$	n**-log-**n
n^2	**quadratic**
n^3	**cubic**
2^n	**exponential**
$n!$	**factorial**

Time efficiency of nonrecursive algorithms

General Plan for Analysis

- Decide on parameter *n* indicating ***input size***
- Identify algorithm's ***basic operation***
- Determine ***worst***, ***average***, and ***best*** cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules

Useful summation formulas and rules

$\Sigma_{l \le i \le n} 1 = 1+1+\ldots+1 = n - l + 1$

In particular, $\Sigma_{l \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n}\, a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \ne 1$

In particular, $\Sigma_{0 \le i \le n}\, 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i$ $\quad \Sigma c a_i = c \Sigma a_i$ $\quad \Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

Example 1: Maximum element

```
ALGORITHM  MaxElement(A[0..n − 1])
    //Determines the value of the largest element in a given array
    //Input: An array A[0..n − 1] of real numbers
    //Output: The value of the largest element in A
    maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > maxval
            maxval ← A[i]
    return maxval
```

T(n) = $\Sigma_{1 \le i \le n-1} 1 = n-1 = \Theta(n)$ comparisons

Example 2: Element uniqueness problem

ALGORITHM *UniqueElements*$(A[0..n-1])$

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise

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

T(n) = $\Sigma_{0 \le i \le n-2} (\Sigma_{i+1 \le j \le n-1} 1)$

$= \Sigma_{0 \le i \le n-2}\, n-i-1 = (n-1+1)(n-1)/2$

$= \Theta(n^2)$ comparisons

Example 3: Matrix multiplication

```
ALGORITHM  MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
    //Multiplies two n-by-n matrices by the definition-based algorithm
    //Input: Two n-by-n matrices A and B
    //Output: Matrix C = AB
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            C[i, j] ← 0.0
            for k ← 0 to n − 1 do
                C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
    return C
```

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

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

$= \Theta(n^3)$ multiplications

Example 5: Counting binary digits

ALGORITHM $Binary(n)$

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
return $count$

Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Example 1: Recursive evaluation of *n*!

Definition: $n! = 1 * 2 * \ldots * (n-1) * n$ for $n \geq 1$ and $0! = 1$

Recursive definition of $n!$: $F(n) = F(n-1) * n$ for $n \geq 1$

$F(0) = 1$

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

Size: n

Basic operation: multiplication

Recurrence relation: $M(n) = M(n-1) + 1$
$M(0) = 0$

Solving the recurrence for M(n)

$M(n) = M(n\text{-}1) + 1, \;\; M(0) = 0$

$$
\begin{aligned}
M(n) &= M(n-1) + 1 \\
&= (M(n-2) + 1) + 1 \;=\; M(n-2) + 2 \\
&= (M(n-3) + 1) + 2 \;=\; M(n-3) + 3 \\
&\ldots \\
&= M(n-i) + i \\
&= M(0) + n \\
&= n
\end{aligned}
$$

The method is called backward substitution.

Example 2: The Tower of Hanoi Puzzle

Recurrence for number of moves:

$$M(n) = 2M(n-1) + 1$$

Solving recurrence for number of moves

$M(n) = 2M(n\text{-}1) + 1, \quad M(1) = 1$

$$
\begin{aligned}
M(n) &= 2M(n-1) + 1 \\
&= 2(2M(n-2) + 1) + 1 = 2^2 \cdot M(n-2) + 2^1 + 2^0 \\
&= 2^2 \cdot (2M(n-3) + 1) + 2^1 + 2^0 \\
&= 2^3 \cdot M(n-3) + 2^2 + 2^1 + 2^0 \\
&= \ldots \\
&= 2^{(n-1)} \cdot M(1) + 2^{(n-2)} + \ldots + 2^1 + 2^0 \\
&= 2^{(n-1)} + 2^{(n-2)} + \ldots + 2^1 + 2^0 \\
&= 2^n - 1
\end{aligned}
$$

References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, Third Edition, Prentice-Hall, 2012 (unit – I, II, III)
2. Jeff Edmonds, How to Think about Algorithms, Cambridge University Press, 2008 (unit – IV, V)
3. Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Data Structures and Algorithms, Pearson Education, Reprint 2006.
4. Robert Sedgewick and Kevin Wayne, ALGORITHMS, Fourth Edition, Pearson Education.
5. S.Sridhar, Design and Analysis of Algorithms, First Edition, OxfordUniversity Press. 2014