NEERAJ®

M.C.S.-211

Design and Analysis of Algorithm

Chapter Wise Reference Book
Including Many Solved Sample Papers

Based on

I.G.N.O.U.

& Various Central, State & Other Open Universities

By: *Anand Prakash Srivastava*

NEERAJ PUBLICATIONS
(Publishers of Educational Books)

Mob.: 8510009872, 8510009878 E-mail: info@neerajbooks.com
Website: www.neerajbooks.com

MRP ₹ 350/-

Content

DESIGN AND ANALYSIS OF ALGORITHM

S.No.	Chapterwise Reference Book	Page

■■

Sample Preview of the Solved Sample Question Papers

Published by:

NEERAJ PUBLICATIONS

www.neerajbooks.com

QUESTION PAPER

June – 2023

(Solved)

DESIGN AND ANALYSIS OF ALGORITHM M.C.S.-211

Time: 3 Hours] *[Maximum Marks: 100*
(Weightage : 70%)

Note **:** *Question No.* ***1*** *is compulsory. Attempt any* ***three*** *questions from the rest.*

Q. 1. ***(a)*** **What is an algorithm? What are its desirable characteristics?**

Ans. Ref.: See Chapter-1, Page No. 1, 'Introduction'.

(b) **What are asymptotic notations? Explain any *two* asymptotic notations with suitable example for each.**

Ans. Ref.: See Chapter-2, Page No. 26, 'Asymptotic notation'.

(c) **Solve the following recurrence relation using substitution method:**

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

Ans. Ref.: See Chapter-4, Page No. 53, 'Example 4'.

(d) **Write and explain binary search algorithm with an suitable example.**

Ans. Ref.: See Chapter-6, Page No. 84-85, Q. No. 6.

(e) **Explain Depth First Search (DFS) algorithm with an suitable example.**

Ans. Ref.: See Chapter-7, Page No. 94-95, 'Depth First Search'.

(f) **What is Dynamic Programming approach of problem solving ? Write the steps involved in dynamic programming.**

Ans. Ref.: See Chapter-9, Page No. 138, 'Dynamic Programming Approach'.

See Also: Here are the general steps involved in solving a problem using dynamic programming:

1. Identify the problem and define the objective: Clearly understand the problem and what you are trying to achieve. Identify the parameters or variables involved.

2. Formulate a recursive solution: Express the problem in terms of smaller subproblems. Determine how the solution to the main problem can be broken down into solutions to smaller subproblems. This step involves defining a recursive relationship or recurrence relation.

3. Define the base cases: Specify the base cases or smallest subproblems that can be solved directly without further decomposition. These base cases serve as the termination condition for the recursion.

4. Design the memorization table: Create a data structure (often a table or array) to store the solutions to subproblems. Initialize the table with default values or values from the base cases.

5. Fill in the memorization table: Use a bottom-up or top-down approach to populate the memorization table. In a bottom-up approach, solve subproblems in a systematic order, starting from the smallest subproblems and working upwards towards the main problem. In a top-down approach (also known as memorization), recursively solve subproblems and store their solutions in the memorization table for future use.

6. Compute the solution to the main problem: Once the memorization table is filled, extract the solution to the main problem from the table. This solution will be available in the last cell or specific cells of the memorization table, depending on the problem.

QUESTION PAPER

December – 2022

(Solved)

DESIGN AND ANALYSIS OF ALGORITHM **M.C.S.-211**

Time: 3 Hours] ***[Maximum Marks: 100***
(Weightage : 70%)

Note **:** *Question No.* ***1*** *is compulsory. Attempt any* ***three*** *questions from the rest.*

Q. 1. (*a*) Write a mathematical definition of big omega (Ω). For the functions defined by $f(n) = 3n^3 + 2n^2 + 1$ and $g(n) = 2n^2 + 3$, verify that $f(n) = \Omega(g(n))$.

Ans. Ref.: See Chapter-2, Page No. 26, 'Big-Omega(Ω) Notation;,

Also Add: To verify that $f(n) = \Omega(g(n))$ for the functions defined by $f(n)=3n^3+2n^2+1$ and $g(n)=2n^2+3$, we need to find constants C and n^0 that satisfy the Ω condition.

First, let's simplify the functions:

$f(n) = 3n^3 + 2n^2 + 1$

$g(n) = 2n^2 + 3$

Now, let's find constants C and n^0 such that $f(n) \geq Cg(n)$ for all $n > n^0$.

We can rewrite the inequality as: $3n^3 + 2n^2 + 1 \geq C(2n^2 + 3)$

Simplify the right side: $3n^3 + 2n^2 + 1 \geq 2Cn^2 + 3C$

Now, let's choose C = 1 and $n^0 = 1$. We can verify that for all n > 1:

$3n^3 + 2n^2 + 1 \geq 2n^2 + 3$

Therefore, we have shown that $f(n) = \Omega(g(n))$ for the given functions *f(n)* and *g(n)*. This means that the function *f(n)* grows at least as fast as g(n) for sufficiently large n, establishing a lower bound relationship between the two functions.

(b) **Explain the principle of optimality in dynamic programming, with the help of an example.**

Ans. Ref.: See Chapter-9, Page No. 134, 'Principle of Optimality'.

(c) **Apply a master method to give the tight asymptotic bounds of the following recurrences:**

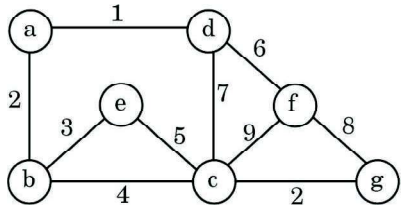
(i) **$T(n) = 4T(n/2) + n^2$.**

Ans. Ref.: See Chapter-4, Page No. 58, Q. No. 4 *(b).*

(ii) **$T(n) = 9T(n/3) + n$.**

Ans. Ref.: See Chapter-4, Page No. 56, 'Example-1'.

(d) **Run the Prim's algorithm on the following graph. Assume that the root vertex is ⓐ.**



Derive the complexity of the algorithm.

Ans. Ref.: See Chapter-8, Page No. 115-116, 'Prim's Algorithm Example'.

Also Add: Prim's algorithm is used to find the minimum spanning tree (MST) of a weighted, undirected graph. The complexity of Prim's algorithm depends on the data structures and implementations used. Prim's algorithm can be implemented using various data structures such as priority queues, adjacency matrices, or adjacency lists. Here, we'll consider the most common implementation using a priority queue.

Let's define some terms before deriving the complexity:

- *V* is the number of vertices in the graph.
- *E* is the number of edges in the graph.

Sample Preview of The Chapter

Published by:

NEERAJ PUBLICATIONS

www.neerajbooks.com

DESIGN AND ANALYSIS OF ALGORITHM

Basics of an Algorithm and its Properties

1

INTRODUCTION

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. In addition, every algorithm must satisfy the following characteristics or properties:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases, algorithm will terminate after a finite number of steps.

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

CHAPTER AT A GLANCE

EXAMPLE OF AN ALGORITHM

The Greatest Common Divisor (GCD) of two non-negative integers *m* and *n* (not-both-zero), denoted by GCD (*m*, *n*), is defined as the largest integer that divides both *m* and *n* evenly, i.e., with a remainder of zero:

Euclid's Algorithm: It is based on applying repeatedly the equality GCD (*m*, *n*) = GCD (*n*, *m* mod *n*), where *m* mod *n* is the remainder of the division of *m* by *n*, until m mod *n* is equal to 0. Since, GCD (*m*, 0) = *m*, the last value of *m* is also the greatest common divisor of the initial *m* and *n*. GCD (60, 24) can be computed as follows: GCD (60, 24) = GCD (24, 12) = GCD (12, 0) = 12.

Euclid's algorithm for computing GCD (*m*, *n*) in simple steps:

Step 1: If *n* = 0, return the value of *m* as the answer and stop; otherwise, proceed to Step 2.

Step 2: Divide *m* by *n* and assign the value of the remainder to *r*.

Step 3: Assign the value of *n* to *m* and the value of *r* to *n*. Go to Step 1.

Euclid's Algorithm: For computing GCD (*m*, *n*) expressed in pseudo-code

ALGORITHM Euclid_GCD(*m*, *n*)

//Computes GCD(*m*, *n*) by Euclid's algorithm

//Input: Two non-negative, not-both-zero integers *m* and *n*

//Output: Greatest common divisor of *m* and *n*

while $n \neq 0$ do

$r \leftarrow m$ mod n

$m \leftarrow n$

$n \leftarrow r$

return *m.*

BASICS BUILDING BLOCKS OF ALGORITHMS

An algorithm follows a procedure to write the solution of a problem. It is designed with five basic building blocks, namely: Sequencing, selection, iteration and recursion.

Sr. N.	Building Block	Action
1.	Sequencing	Step by step actions
2.	Selection	Decision
3.	Iteration	Repetition or Loop
4.	Procedure	Set of instructions
5.	Recursion	Function calling itself

Sequencing, Selection and Iteration

Sequencing: An algorithm is a step-by-step process, and the order of those steps are crucial to ensure the correctness of an algorithm.

Selection: Algorithms can use selection to determine a different set of steps to execute, based on a Boolean expression.

Iteration: Algorithms often use repetition to execute steps a certain number of times or until a certain condition is met.

Procedure and Recursion

***(i)* Procedure:** An algorithm defines the specific steps required to solve a problem. A procedure is a recipe or method for accomplishing a result such as solving a problem or performing a task. A procedure is usually considered to have the following characteristics. It consists of a finite sequence of discrete steps.

For example: We can define GCD (*a*,*b*) as a procedure/function only once and can call it a number of times in a main function with different values of *a* and *b*.

The general format for defining a procedure might look like this:

Procedure <Name>(<parameter-list>)() [: < type>]

<declarations>

<sequence of instruction expected to be occurred repeatedly>

end;

***(ii)* Recursion:** Generally speaking, recursion is the concept of *well-defined* self-reference.

We recall from Mathematics, one of the ways in which the factorial of a natural number *n* is defined:

factorial (1) = 1

factorial (*n*) = *n** factorial (*n* – 1)

By definition

factorial (4) = 4 * factorial (3).

Again by the definition

factorial (3) = 3 ∗ factorial (2)

Similarly

factorial (2) = 2* factorial (1)

And by definition

factorial (1) = 1

Substituting back values of factorial (1), factorial (2) etc., we get factorial (4) = 4.3.2.1 = 24, as desired.

This definition suggests the following procedure/algorithm for computing the factorial of a natural number *n*:

```
Procedure factorial (n)
    fact: integer;
begin
    fact ← 1
    if n equals 1 then return fact
    else begin
    fact ← n ∗ factorial (n – 1)
    return (fact)
    end;
end;
```

Definition: A procedure, which can call itself, is said to be recursive procedure/algorithm. For successful implementation of the concept of recursive procedure, the following conditions should be satisfied:

(i) There must be in-built mechanism in the computer system that supports the calling of a procedure by itself, e.g, there may be in-built stack operations on a set of stack registers.

(ii) There must be conditions within the definition of a recursive procedure under which, after finite number of calls, the procedure is terminated.

(iii) The arguments in successive calls should be simpler in the sense that each succeeding argument takes us towards the conditions mentioned in *(ii).*

A SURVEY OF COMMON RUNNING TIMES

When solving a problem we are faced with a choice among algorithms. The basis for this can be any one of the following:

(i) We would like an algorithm that is easy to understand, code and debug.

(ii) We would like an algorithm that makes efficient use of the computer's resources, especially, one that runs as fast as possible.

Measuring the running time of a program

The running time of a program depends on factors such as:

1. The input to the program.
2. The quality of code generated by the compiler used to create the object program.

3. The nature and speed of the instructions on the machine used to execute the program, and
4. The time complexity of the algorithm underlying the program.

Following are the generalized form of running time for the algorithms:

1. Constant Time (O*(k)*): If the running time does not depend on the input size (n), then it is known as constant running time. It can be represented as $T(n) = O(k)$, where k is a constant.

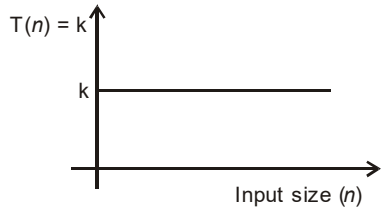


Fig. T(n) = O*(k)*

2. Linear Time O(kn): If the time complexity is at most a constant factor times the size of the input, then it is known as linear time complexity and is presented as $T(n) = O(kn)$, where k is a constant or $O(n)$.

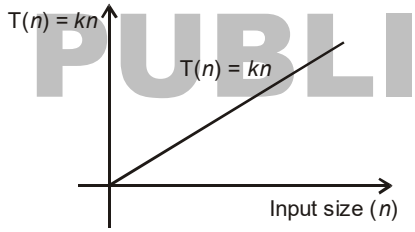


Fig. T(*n*) = O(*n*)

3. Logarithmic Time (log(*n*)): If the time complexity of an algorithm is proportional to the logarithm of the input size, i.e., every time the size of input remains half of that of previous iteration, then it is known as logarithmic time complexity and depicted as O(log n) time. For example, running time of binary search algorithm is O(log n).

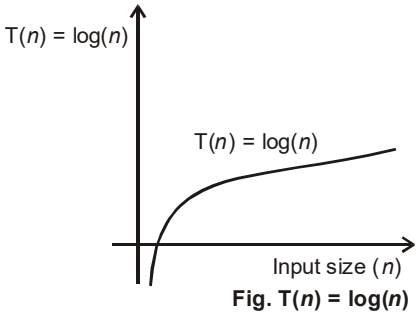


Fig. T(*n*) = log(*n*)

Quadratic Time: (T(*n*)= O(*n*)²): It occurs when the algorithm is having a pair of nested loops. The outer loop iterates $O(n)$ time and for each iteration the inner-loop takes $O(n)$ time. So, we get $O(n^2)$ by multiplying these two factors of n.

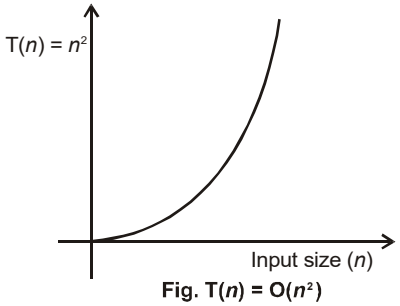


Fig. T(*n*) = O(*n*²)

4. Cubic Time: (T(*n*)= O(*n*³)): It often occurs when the algorithm is having there nested loops, and each loop has a maximum n iterations. Let us have one interesting example which requires cubic time complexity. Suppose, we are given n sets: $S_1, S_2,...S_n$. Size of each set is n (i.e., each set is having n elements).

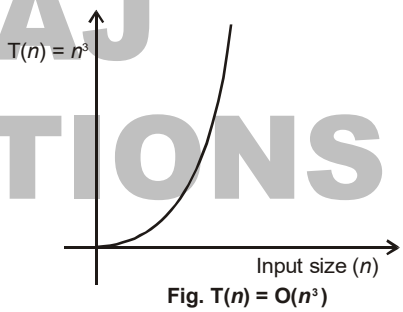


Fig. T(*n*) = O(*n*³)

Pseudo-code for finding common elements in pair of sets:

```
for each set S_i of n elements
   for each other set S_j of n elements
     for each element x of S_i
      check whether x also belongs to S_j
      endfor
      if x belongs to both S_i and S_j
      print "S_j and S_j are not disjoint"
      endif
      endfor
      endfor
```

Time Complexity: The innermost loop takes $O(n)$ time because of n elements. The second inner loop over S_j also takes $O(n)$ iterations around the innermost loop, and finally $O(n)$ over S_i around S_j iterations. Multiplying all the three iterations we obtain $O(n^3)$ time complexity.

5. Polynomial Time: (O (n^k)): This running time is obtained when the search over all subsets of a set of a size *k* is performed. To understand the complexity of running time, we have to find how many distinct subsets of size *k* of n elements of a set can be chosen. For that, we have to take a combination of *n* elements taken *k* at a time.

The brute force method to solve this problem would require searching for all subsets of *k* nodes and for each subset it would examine whether there is an edge connecting any two nodes for each subset *s* of a size *k*. Below is a pseudo-code for finding an independent set.

Pseudo-code

```
for each subset s of a size k in a graph G
check whether s is an independent set
if yes, print " s is an independent set
else stop
```

In this case the outer loop will iterate $O(n^k)$ times and it selects all *k* node subsets of *n* node of the graph. In the inner loop within each subset it loops for each pair of nodes to find out whether there is an edge between the pair which will require O(2 out of *k*) pairs of search i.e.$O(k^2)$ search. Therefore, the total time now is $O(k^2 n^k)$. Since k is a constant, it can be dropped, finally it is $O(n^k)$.

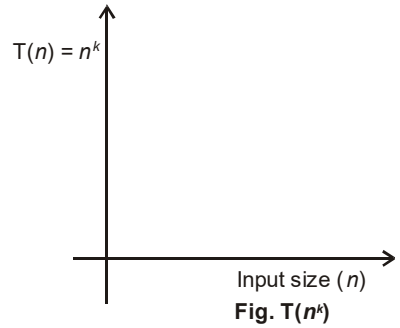


Fig. T(n^k)

6. Exponential Time (O(k^n)): The polynomial time complexity there are other two types of bounds: Exponential time $O(2^N)$ and factorial time O(*n*!):

The modified version of the pseudo-code is presented below.

Pseudo-code:

```
Input G(V,E)
{
    for each subset s of n number of nodes
    verify whether s is an independent set
    if s is the largest among all the subsets
    examined so for
    print "s is the largest independent set "
 endif
endfor
}
end of code fragment
```

Factorial Time (O(*n*!)): In comparison to the growth of exponential running time, the growth of factorial time (*n*!) is more rapid. The running time of this type of complexity arises in two types of problems:

(i) Matching Type of Problem, for example, bipartite matching problem. Suppose, there are n number of boys and *n* number of girls. To find perfect matching between *n* number of boys & *n* number of girls, the first boy will be compared with *n* numbers of girls .The second boy will be left with (*n*-1) choices among girls for comparison. There will be only (*n*-2) options for matching for the third boy, and so forth. After array girls Multiplying all these options for n boys we obtain *n*! ie. *n*(*n*–1) (*n*-2) .....(2) (1)

(ii) O(*n*!) also occurs where the problem requires arranging *n* elements into a particular order (i.e., a permutation of *n* numbers). A classic example is travelling salesman problem.

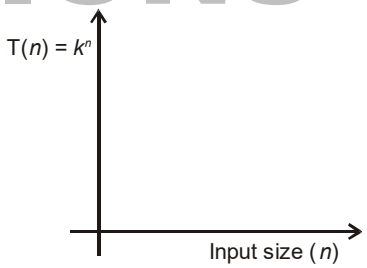


Fig. T(n) = O(k^n)

ANALYSIS & COMPLEXITY OF ALGORITHM

Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

The complexity function *f*(*n*) for certain cases are:

1. **Best Case:** The minimum possible value of *f*(*n*) is called the best case.
2. **Average Case:** The expected value of *f*(*n*).
3. **Worst Case:** The maximum value of *f*(*n*) for any key possible input.

Average Case: In average case analysis, we take all possible inputs and calculate computing