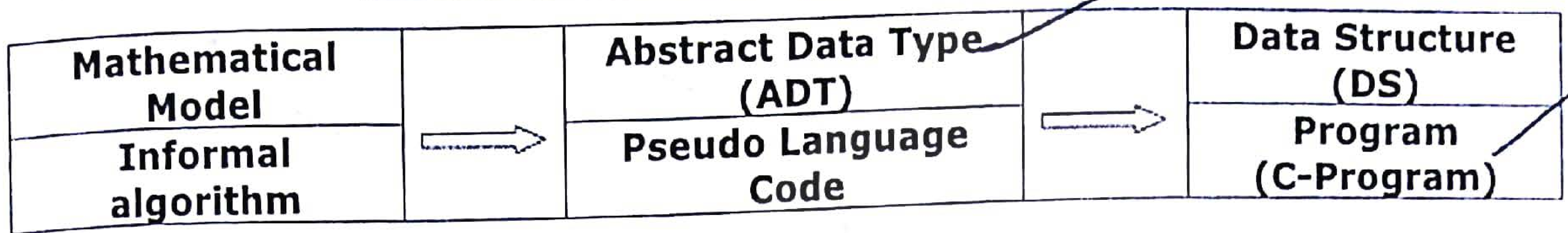
SET - I

Topic	Page No
Problem to Program	1
Introduction	1 - 2
Performance Evaluation	2 – 7
Sorting Problem	7 – 18
Mathematical Preliminaries	18 – 20
Asymptotic Notation	21 – 27
Key	28

GradeSetter

WWW.GRADESETTER.COM

1. PROBLEM TO PROGRAM



1) Model the problem using an appropriate mathematic model (Informal algorithm)

2) The informal algorithm is written in pseudo language

3) The stepwise refinement of pseudo language gives various types of data used and Operations to be performed on data. (i.e., data type)

4) We create ADT for each data type.

5) We choose an appropriate Data Strucure to implement each ADT

6) Finally replace informal statements in pseudo language code by C-code

An algorithm is a finite sequence of computational steps that transform the input into the output in finite number of steps

A **data type** is a collection of objects and a set of operations that act on those objects

An **abstract data type (ADT)** is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operation. **ADT is mathematical model of data type**

A **data structure(DS)** is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them. We use DS to implement ADT.

Pseudo code: Mixture of natural language and high level programming language constructs that describes algorithm

A **program** is an expression of an algorithm in a programming language

2. INTRODUCTION

1. Non-Computational problem

A problem that has no precise and simple specification
Example: Convince your boss for salary hike; convince your faculty for marks.

2. Computational problem

Specification of input	Specification output as function of input

Example: The sorting problem:
Input: a sequence <a1, a2,......,an> of n numbers.
Output: a permutation <a1', a2',...., an'> of the input with a1' <= a2' <=....<= an'.

WWW.GRADESETTER.COM

3. Algorithm Definition

An Algorithm is well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output.

Instance: A particular input called an **instance** of a computational problem.
Example: The input sequence <31, 41, 59, 26, 41, 58> is an instance of the sorting problem.

4. Algorithm Characteristics

All algorithms should satisfy the following criteria.
Input: Zero or more quantities are externally supplied.
Output: At least one quantity is produced.
Definiteness: Each instruction is clear and unambiguous.
Finiteness: For all cases, the algorithm terminates after a finite number of steps.
Effectiveness: instruction is basic enough to be carried out.

Definition: Algorithms that are definite and effective are called **computational procedures**
Example: Digital computer.

Definition: An algorithm is said to be **correct** if, for every input instance, it halts with the correct output.

5. The study of Algorithm includes the following important areas

- **Design an algorithm:** Creating and algorithm is an art which may never be fully automated.
  Different design strategies: Divide and Conquer, Greedy, Dynamic programming....etc.
- **Express an algorithm:** Algorithm specification using Pseudo code.
- **Validate an algorithm (correctness):** To show that algorithm computes the correct answer for all possible legal inputs.
- **Analysis an algorithm:** Find the time and space complexity. Prove that we cannot solve the problem any faster using asymptotic analysis
- **Implementation :** Implementing algorithm in a programming language
- **Verification :-** Test the program (debugging and profiling)

3. PERFORMANCE EVALUATION

1. Performance evaluation

As an algorithm is executed, it uses the computers CPU to perform operations and its memory to hold the program and data.
An efficient algorithm

- needs less running time
- uses less space

1.1. **Space Complexity:** The space complexity of an algorithm is the amount of memory if needs to run to completion.
1.2. **Time Complexity:** The time complexity of an algorithm is the amount of computer time it needs to run to completion.
1.3. **Performance evaluation** of an algorithm refers to the task of computing space and time complexity of an algorithm
1.4 **Performance evaluation** can be loosely divided into two major phases:

1) **a priority estimates (Performance Analysis)**
   - Uses analytical methods
   - Machine independent

2) **a posterior testing (Performance Measurement or profiling) :**
- It is the process of executing a correct program on data sets and measuring time and space it takes to compute results
- Machine dependent

Performance Analysis is general methodology because
- It uses high level description of an algorithm(Pseudo-code)
- All possible input instances are taken into account
- Machine independent

2. Space complexity

2.1 Components of space complexity

Instruction space:
Space needed for code

Data Space:
i. Space needed for constants and simple variables
ii. Space needed for dynamically allocated objects (Such as arrays, structures, etc)

Environment stack space:
i. It is used to save information needed to resume execution of partially executed function.
ii. Each time a function is invoked the following data are saved on the environment stack
The return address
The values of local variables and formal parameters

Recursion stack space: Amount of stack space needed by recursive functions is called recursion stack space. It depends on
- Space needed by local variables and formal parameters
- Maximum depth of recursion (i.e., maximum number of nested recursive calls)
- Compiler being used

The total space needed by a program is divided into two parts:
1. Fixed Part
2. Variable Part

A **fixed part** independent of instance characteristics (e.g., size, number, value)
1. Instruction space
2. Data space (space needed for constants and simple variables and some dynamically allocated objects)
Note: The space needed by some of the dynamically allocated memory may also be independent of problem size
3. Environment stack space for non-recursive functions

A **variable part** dependent on instance characteristics
1. Dynamically allocated space
2. Recursion stack space

2.2 Definition

The space complexity S (P) of any algorithm P can be written as

$$S(P) = C + S_P(I)$$

C constant that denotes fixed part
S_P Variable part that depends on instance characteristics (I) (e.g., size, number, value)

GradeSetter

2.3 Examples

1.
```
Algorithm abc(a, b, c)
{
        return a + b + b*c/(a+b+4.0);
}
```
C = Space needed for a, b, c and result; $S_P(abc) = 0$

2.
```
Algorithm Sum(a, n)
{
        s = 0;
        for(i = 1 to n)
                s = s + a[i];
        return s;
}
```
Space required for

- formal parameters a and n
- local variables s, i and constant 0
- instruction space.

This amount of space needed does not depend on value of n.

$S_{sum}(n) = 0$

Since a is actually the address of the first element in a[](i.e., a[0]), the space needed by it is also constant

3.
```
Algorithm rsum(int a[], int n)
{
        if(n>0)
                return rsum(a, n-1)+a[n-1];
        return 0;
}
```
Let reference of a = 4bytes; value of n = 4 bytes; return address = 4 bytes are stored on recursion stack for each recursion call.

Each recursive call require 12 bytes

Depth of recursion = (n+1)

recursion stack space = 12(n+1)

$S_{rsum}(n) = 12\ (n+1)$

3. Time Complexity

3.1 Time Complexity

Time taken by a program P is sum of compile time and runtime

$T(P) = C + T_P(I)$

C (compile time) is independent of instance characteristics ($\therefore$ constant)

$T_P(I)$ (Run time) is dependent on instance characteristics.

(i) However, **analytical approach to determine the exact runtime is complicated**

- Since runtime depends on machine dependent issues like i) Type of processor, ii) Access rate (rate/write operations), iii) Architecture and machine independent issue iv) input size.

(ii) Run time expression should be machine-independent.

Therefore, **we estimate runtime as function of input size.** i.e., we find rate of growth of time with respect to input size.

Running time = f(input size)