PRINCIPLES OF PROGRAMMING AND ANALYSIS OF ALGORITHMS

Chapter 2: Principles of Programming and Analysis of Algorithms

Engineering / Diploma — Study Notes

2.1 Algorithms

An algorithm is a finite, well-defined, step-by-step sequence of instructions designed to solve a specific problem or perform a specific task within a finite amount of time. It takes zero or more inputs, processes them through a clearly specified set of operations, and produces one or more outputs. An algorithm is essentially the logical blueprint of a program — it describes WHAT steps need to be carried out, independent of any particular programming language.

Every algorithm must satisfy certain essential properties in order to be considered correct and usable. These properties distinguish a true algorithm from a vague or informal set of instructions, and they ensure that the algorithm can be reliably translated into a working program.

Characteristics/properties of a good algorithm:

- Finiteness — the algorithm must terminate after a finite number of steps; it cannot run forever.
- Definiteness — each step must be precisely and unambiguously defined, leaving no room for confusion.
- Input — an algorithm should have zero or more well-defined inputs taken from a specified set.
- Output — an algorithm must produce one or more outputs that have a specified relationship to the inputs.
- Effectiveness — every operation in the algorithm must be basic enough to be carried out, in principle, exactly and in a finite amount of time.
- Feasibility/Practicality — the algorithm should be practically implementable with the available resources.

Ways of representing an algorithm:

- Natural language description — writing the steps in plain English (or another language).
- Pseudocode — a language-independent, code-like notation used to describe the logic clearly.
- Flowchart — a diagrammatic/graphical representation of the steps and decisions using standard symbols.

2.2 Different Approaches for Designing an Algorithm

There is often more than one way to solve a given problem, and computer scientists have developed several general strategies, known as algorithm design techniques or paradigms, that can be applied across a wide range of problems. Choosing the right design approach can greatly affect the efficiency, simplicity, and correctness of the resulting algorithm.

Major algorithm design approaches:

- Brute Force Approach — solves a problem in the most straightforward, direct manner by trying all possible solutions; simple to design but often inefficient for large inputs (e.g., linear search).
- Divide and Conquer — breaks a problem into smaller sub-problems of the same type, solves each sub-problem recursively, and then combines their solutions to solve the original problem (e.g., merge sort, quick sort, binary search).
- Greedy Approach — builds up a solution step by step, at each step choosing the option that looks best (locally optimal) at that moment, without reconsidering earlier choices (e.g., Kruskal's and Prim's algorithms for minimum spanning tree).
- Dynamic Programming — solves complex problems by breaking them into overlapping sub-problems, solving each sub-problem only once, and storing the results to avoid redundant computation (e.g., Fibonacci series, knapsack problem).
- Backtracking — builds a solution incrementally and abandons (backtracks from) a partial solution as soon as it determines that it cannot lead to a valid complete solution (e.g., N-Queens problem, maze solving).
- Branch and Bound — a systematic method for solving optimization problems by dividing the solution space into smaller branches and using bounds to eliminate branches that cannot produce a better solution.

The choice among these approaches depends on the nature of the problem, the acceptable time and space complexity, and whether an exact or approximate solution is required.

2.3 Complexity

The complexity of an algorithm is a measure of the amount of resources — mainly time and memory (space) — that the algorithm requires to run as a function of the size of its input. Analyzing complexity allows programmers to predict how an algorithm will perform as the input size grows, and to compare different algorithms that solve the same problem in order to choose the most efficient one.

Types of complexity:

- Time Complexity — the amount of computational time an algorithm takes to run, expressed as a function of the input size (n). It measures the number of basic operations performed, not the actual clock time, since actual time depends on hardware and compiler.
- Space Complexity — the amount of memory space an algorithm requires to run to completion, including space for input values, auxiliary variables, and any additional data structures used.

Cases considered while analyzing time complexity:

- Best Case — the minimum time required for the algorithm to complete, occurring under the most favourable input conditions.
- Average Case — the expected time required, averaged over all possible inputs of a given size.
- Worst Case — the maximum time required for the algorithm to complete, occurring under the most unfavourable input conditions; this is generally the most important case for reliability analysis.

2.4 Big 'O' Notation

Big O notation is a mathematical notation used to describe the upper bound (worst-case growth rate) of an algorithm's running time or space requirement as the input size (n) becomes very large. It expresses how the resource requirement grows relative to the input size, while ignoring constant factors and lower-order terms

that become insignificant for large n. Big O is the most widely used notation in algorithm analysis because it gives a clear, standardized way to compare the efficiency of different algorithms.

Other related asymptotic notations:

- Big Omega (Ω) notation — describes the lower bound (best-case growth rate) of an algorithm.
- Big Theta (Θ) notation — describes a tight bound, when the upper and lower bounds are of the same order, giving the average/exact growth rate.

Common time complexities (from fastest to slowest):

Notation	Name	Example
O(1)	Constant time	Accessing an array element by index
O(log n)	Logarithmic time	Binary search
O(n)	Linear time	Linear search, single loop traversal
O(n log n)	Linearithmic time	Merge sort, Quick sort (average case)
$O(n^2)$	Quadratic time	Bubble sort, Selection sort, Insertion sort
$O(2^n)$	Exponential time	Recursive Fibonacci (naive), Subset generation
O(n!)	Factorial time	Solving Travelling Salesman by brute force

In Big O notation, only the dominant (fastest-growing) term is retained, and constant multipliers are dropped. For example, an algorithm that performs $3n^2 + 5n + 2$ operations is simply written as $O(n^2)$, because as n grows very large, the n^2 term dominates the growth of the function.

2.5 Algorithm Analysis

Algorithm analysis is the process of determining the amount of time and space resources required by an algorithm to solve a problem, so that its efficiency can be evaluated and compared with alternative algorithms. It is a fundamental step in algorithm design because it helps predict how an algorithm will behave as input size increases, well before the algorithm is actually implemented and executed on a computer.

Objectives of algorithm analysis:

- To predict the resources (time and memory) an algorithm will require before it is implemented.
- To compare different algorithms designed for the same problem and select the most efficient one.
- To identify bottlenecks and areas where an algorithm's performance can be improved.
- To ensure the algorithm will scale acceptably as the size of the input grows.

Steps/approach in algorithm analysis:

1. Identify the input size (n) that affects the algorithm's running time.
2. Identify the basic operation(s) whose execution count determines the running time.
3. Determine how many times the basic operation is executed as a function of n (this may differ for best, average, and worst cases).
4. Express the result using asymptotic notation (Big O, Omega, or Theta).

Two algorithms solving the same problem can be compared purely by their time and space complexity, without needing to implement and run both on a computer. This theoretical, machine-independent comparison is one of

the most valuable outcomes of algorithm analysis, and it forms the basis for choosing appropriate algorithms and data structures in real-world software design.

Multiple Choice Questions (MCQs)

1. Which of the following is NOT a characteristic of a good algorithm?

A) Finiteness
B) Definiteness
C) Ambiguity
D) Effectiveness

2. An algorithm must terminate after a finite number of steps. This property is called:

A) Definiteness
B) Finiteness
C) Input
D) Output

3. Which of the following is a diagrammatic representation of an algorithm?

A) Pseudocode
B) Flowchart
C) Machine code
D) Assembly code

4. Merge Sort is based on which algorithm design approach?

A) Greedy approach
B) Divide and Conquer
C) Backtracking
D) Brute force

5. Which approach makes the locally optimal choice at each step, hoping it leads to a global optimum?

A) Dynamic Programming
B) Divide and Conquer
C) Greedy approach
D) Backtracking

6. Dynamic programming is most useful when a problem has:

A) No sub-problems
B) Overlapping sub-problems
C) Only one possible solution
D) No recursive structure

7. The N-Queens problem is typically solved using:

A) Greedy approach
B) Backtracking
C) Divide and Conquer
D) Brute force only

8. Time complexity of an algorithm mainly measures:

A) The actual clock time in seconds
B) The number of basic operations as a function of input size
C) The size of the source code
D) The number of programmers required

9. Which case represents the maximum time an algorithm can take for a given input size?

A) Best case
B) Average case
C) Worst case
D) Amortized case

10. Big O notation describes the:

A) Exact running time
B) Lower bound of an algorithm
C) Upper bound (worst-case growth rate) of an algorithm
D) Space used by the compiler

11. Which notation represents the lower bound (best case) of an algorithm?

A) Big O
B) Big Omega
C) Big Theta
D) Big Sigma

12. Binary search has a time complexity of:

A) O(1)
B) O(n)
C) O(log n)
D) $O(n^2)$

13. Bubble sort has a worst-case time complexity of:

A) O(n)
B) O(log n)
C) $O(n^2)$
D) O(n log n)

14. In the expression $3n^2 + 5n + 2$, the Big O notation is:

A) O(n)
B) $O(n^2)$
C) O(5n)
D) $O(3n^2)$

15. Which of the following is an objective of algorithm analysis?

A) Increasing the length of the code
B) Comparing algorithms and predicting resource requirements
C) Making the algorithm machine-dependent
D) Removing the need for testing

Answer Key

1. C 2. B 3. B 4. B 5. C 6. B 7. B 8. B 9. C 10. C 11. B 12. C 13. C 14. B 15. B

Broad / Long Answer Questions

5. Define an algorithm. Explain the essential characteristics that every algorithm must satisfy.
6. Explain, with examples, the different ways of representing an algorithm.
7. Describe the Divide and Conquer approach of algorithm design. Give two examples of algorithms based on this approach.
8. Differentiate between the Greedy approach and Dynamic Programming approach with suitable examples.
9. What is Backtracking? Explain with the help of the N-Queens problem or a similar example.
10. Explain time complexity and space complexity. Why are both important while analyzing an algorithm?
11. Explain the best case, average case, and worst case time complexity of an algorithm with a suitable example.
12. What is Big O notation? Explain its significance in algorithm analysis with examples.
13. Differentiate between Big O, Big Omega, and Big Theta notations.
14. Explain the steps involved in the analysis of an algorithm, and discuss why algorithm analysis is important before implementation.