Serving Education for Last 40 Years

NEERAJ®

MCS-208
Data Structures and Algorithms

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 ₹ 280/-

Content

DATA STRUCTURES AND ALGORITHMS

■■

QUESTION PAPER

June – 2024

(Solved)

DATA STRUCTURES AND ALGORITHMS MCS-208

Time: 3 Hours] *[Maximum Marks : 100*

Weightage : 70%

Note: Question No. 1 is compulsory. Attempt any three questions from the rest. All algorithms should be written nearer to 'C' language.

Q. 1. ***(a)*** **Write an algorithm for implementation of a Stack using Arrays.**

Ans. Implementation of Stack Using Arrays: You're correct that while both arrays and stacks store ordered elements, there are important differences between them. A stack is a data structure that follows the LIFO (Last In, First Out) principle, where elements are added or removed from the top of the stack. On the other hand, an array is a fixed-size collection of elements, and its size must be declared at the time of initialization.

When using an array to implement a stack, one would typically define an array with a maximum size (to prevent overflow), and a variable to track the current top of the stack.

```
#include <stdio.h>
int choice, stack[10], top = 0, element;
void push();
void pop();
void showelements();
int main() {
   while (1) {
         printf("\nEnter one of the following options:\n");
         printf("1. PUSH\n2. POP\n3. SHOW ELEMENTS\n4. EXIT\n");
      scanf("%d", &choice);
      switch (choice) {
         case 1:
            push();
            break;
         case 2:
            pop();
            break;
         case 3:
            showelements();
            break;
         case 4:
            printf("Exiting program.\n");
            return 0;
         default:
               printf("Invalid choice. Please enter 1–4.\n");
      }
   }
}
void push() {
   if (top < 10) {
      printf("Enter the element to be pushed to stack:\n");
      scanf("%d", &element);
      stack[top] = element;
      top++;
   } else {
      printf("Stack is full.\n");
   }
}
void pop() {
   if (top > 0) {
      top--;
      element = stack[top];
      printf("Popped element: %d\n", element);
   } else {
      printf("Stack is empty.\n");
   }
}
void showelements() {
   if (top == 0) {
      printf("Stack is empty.\n");
   } else {
      printf("Stack elements:\n");
      for (int i = 0; i < top; i++) {
         printf("%d\n", stack[i]);
      }
   }
}
```

A stack of size 10 is used to store integers with four operations: push, pop, showelements, and exit. push adds an element, pop removes the top element, showelements displays all elements, and exit ends the program. The top variable tracks the next free position, element holds the value to be pushed or popped, and choice lets the user select an operation.

***(b)* Write an algorithm for multiplication of two matrices.**

Ans. Matrix Multiplication Algorithm (Pseudocode)

Let matrix A be of size m × n and matrix B be of size n × p. The result matrix C will be of size m × p.

```
For each row i in matrix A:
    For each column j in matrix B:
        Initialize C[i][j] = 0
        For each element k from 0 to n-1:
            C[i][j] += A[i][k] * B[k][j]
```

C Program for Matrix Multiplication

```
#include <stdio.h>

int main() {
    int A[10][10], B[10][10], C[10][10];
    int m, n, p, q, i, j, k;

    // Input size of matrices
    printf("Enter rows and columns of matrix A (m n): ");
    scanf("%d %d", &m, &n);

    printf("Enter rows and columns of matrix B (p q): ");
    scanf("%d %d", &p, &q);

    if (n != p) {
        printf("Matrix multiplication not possible. Columns of A must equal rows of B.\n");
        return 0;
    }

    // Input elements of matrix A
    printf("Enter elements of matrix A:\n");
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &A[i][j]);

    // Input elements of matrix B
    printf("Enter elements of matrix B:\n");
    for (i = 0; i < p; i++)
        for (j = 0; j < q; j++)
            scanf("%d", &B[i][j]);

    // Multiply matrices
    for (i = 0; i < m; i++) {
        for (j = 0; j < q; j++) {
            C[i][j] = 0;
            for (k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    }

    // Output result
    printf("Resultant Matrix C:\n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < q; j++)
            printf("%d ", C[i][j]);
        printf("\n");
    }

    return 0;
}
```

***(c)* Write an association for implementation of Quick Sort.**

Ans. Ref.: See Chapter-9, Page No. 102, 'Quick Sort'.

***(d)* Convert the following expression to postfix:**

a + b*c + d/e*f

Ans. Given Infix Expression:

*a + b * c – d/e * f*

Step-by-Step Conversion:

Start with *b * c* → postfix: *b c **

Then *a* + (*b * c*) → postfix: *a b c * +*

Next, *d/e* → postfix: *d e /*

Then (*d/e*) * f → postfix: *d e / f **

Finally, the full expression is: (*a* + (*b * c*)) – ((*d* / *e*) * *f*)

Final Postfix Expression:

*a b c * + d e / f * –*

Q. 2. *(a)* Explain the process of implementing two queues in an array.

Ans. Ref.: See Chapter-2, Page No. 15, Q. No. 5.

***(b)* Explain the process of calculation of storage complexity with an example.**

Ans. Ref.: See Chapter-1, Page No. 3, 'Calculation of Storage Complexity'.

Q. 3. *(a)* What are circular linked lists? Write an algorithm for insertion of an element into a circular linked list.

DATA STRUCTURES AND ALGORITHMS

Analysis of Algorithms

1

INTRODUCTION

Many people believe that computers can do anything, but this is a misconception. In reality, a computer can only execute a limited set of predefined instructions. These instructions, when organized in a specific sequence, are known as an algorithm. When an algorithm is written in a programming language, it becomes a program.

The study of algorithms and how efficiently they perform is a key area of computer science. Even with the development of high-speed computers, designing time-efficient algorithms remains crucial. This is where complexity theory comes in – it explores the resources required to solve problems, primarily focusing on time (the number of steps) and space (amount of memory used).

It's important to distinguish complexity theory from computability theory. While computability theory examines whether a problem can be solved by any algorithm at all, complexity theory is concerned with how efficiently that solution can be achieved.

The field known as Analysis of Algorithms seeks to understand algorithmic complexity. Although much of the research emphasizes worst-case scenarios, there is also significant attention paid to average-case analysis. Over time, the focus in computing has shifted – from hardware to programming, and now to algorithm design itself, which lies at the core of effective problem solving.

CHAPTER AT A GLANCE

MATHEMATICAL BACKGROUND

Analyzing an algorithm involves measuring the resources it uses, such as time and memory, to execute. Since algorithms often handle inputs of varying lengths, analysis helps estimate their efficiency. This process is a key part of computational complexity theory, which guides the search for more efficient solutions by offering theoretical resource bounds.

Definition of Algorithm: An algorithm is a sequence of definite and effective steps that takes input, produces output, and always terminates. Its key characteristics are: Input, Output, Definiteness, Effectiveness, and Termination.

Complexity classes: Complexity classes group decision problems by their computational difficulty.

1. Class P includes problems solvable by a deterministic machine in polynomial time – efficient in the worst case.

2. Class NP includes problems solvable by a non-deterministic machine in polynomial time, such as SAT, Hamiltonian Path, and Vertex Cover.

What is Complexity?: Complexity measures how time or storage grows with problem size. To avoid machine-dependent factors, we use asymptotic analysis, which describes growth in terms of proportionality to known functions. This focuses on the complexity of algorithms, not the problems themselves.

Asymptotic Analysis: Asymptotic analysis describes algorithm complexity in terms of growth relative to input size, using notations like 'Big O', 'Omega', and 'Theta'. These express upper, lower, and tight bounds on performance. While 'Little o' is conceptually similar to 'Big O', it is rarely used in practice.

Tradeoff between space and time complexity: Sometimes, we trade space for time by choosing data structures that use more memory to speed up computation. To make such choices wisely, we rely on complexity analysis.

Instead of counting exact steps or time, we use asymptotic analysis to estimate algorithm performance for large inputs. This is done using Big O, Omega (Ω), and Theta (Θ) notations, with Big O being the most common. These notations help us understand how an algorithm's time or space grows with input size, independent of machine or runtime variations.

The Θ-Notation (Tight Bound): The Theta (Θ) **notation** provides a tight bound on a function, meaning it grows at the same rate as another function, within constant factors. We write $f(n) = \Theta(g(n))$ if there are constants n_0, c_1, and c_2 such that for all $n \geq n_0$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$**.** This means $g(n)$ is an asymptotically tight bound for $f(n)$.

The big O notation (Upper Bound): Big O notation gives an upper bound on a function's growth. We write $f(n) = O(g(n))$ if there are constants n_o and c such that for all $n \geq n_o$, $f(n) \leq c \cdot g(n)$. This means $f(n)$ grows no faster than ***g(n)***, up to a constant factor, for large n.

Mathematically, for a given function $g(n)$, the set $O(g(n))$ includes all functions $f(n)$ such that:

$O(g(n)) = \{f(n)$: there exist constants $c > 0$ and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_o\}$.

This defines an upper bound on $f(n)$ using $g(n)$ scaled by a constant.

While Big O gives an upper limit, Theta (Θ) is tighter because it bounds a function from both above and below. For example:

$f(n) = an + c$ is $O(n)$ (tight bound) and also $O(n^2)$ (loose bound), but $O(n)$ better reflects its true growth rate.

The Ω-Notation (Lower Bound): Omega (Ω) notation provides a lower bound on a function's growth. We write $f(n) = \Omega(g(n))$ if there are constants $c > 0$ and n_0 such that for all $n \geq n_o$, $f(n) \geq c \cdot g(n)$.

Mathematically: $\Omega(g(n)) = \{f(n) :$ there exist constants $c > 0$ and $n_0 \geq 0$ such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0\}$ Ω-notation is often used to represent the best-case performance of an algorithm.

Asymptotic notation: Here are examples illustrating the use of asymptotic notation:

Example 1: $f(n) = 3n^3 + 2n^2 + 4n + 3$

$4n + 3$ is $O(n)$

So, $f(n) = 3n^3 + 2n^2 + O(n)$

$2n^2 + O(n)$ is $O(n^2)$

Hence, $f(n) = 3n^3 + O(n^2)$

Therefore, $f(n) = O(n^3)$

Here are some key rules of Big-O notation to remember:

1. For any function $f(n)$, $f(n) = O(f(n))$.

2. Any polynomial of degree k can be written as $O(n^k)$. **Example:** $akn^k + ... + a0 = O(n^k)$ (lower-order terms are ignored).

3. Logarithmic bases are ignored in Big–O: $\log_a n = O(\log_b n)$. We typically write it as $O(\log n)$.

4. Logarithmic functions grow slower than any positive polynomial: $\log_b n = O(n^c)$ for any $c > 0$.

5. Polynomial functions grow slower than exponential functions: $n^k = O(b^n)$.

6. Exponential functions grow slower than factorial functions: $a^n = O(n!)$ for any constant a.

PROCESS OF ANALYSIS

The goal is to determine an algorithm's efficiency, based on the resources it uses, such as: CPU utilization (Time complexity), Memory utilization (Space complexity), Disk usage (I/O) and Network usage (Bandwidth).

Performance: Refers to actual resource usage when a program runs. It depends on the algorithm, machine, compiler, etc.

Complexity: Describes how resource needs grow with input size. For example, summing 1000 numbers takes more time/memory than summing 2.

Time Complexity: The maximum time a Turing machine takes to process any input of length n.

Space Complexity: The amount of memory required, typically expressed using Big-O. For instance, $O(n^2)$ means doubling input size multiplies memory use by four.

Determination of Time Complexity

The RAM Model: Developed by John von Neumann, the RAM model is used to study algorithms independently of machine or language.

Key Assumptions:

1. Each simple operation (+, –, =, if, call) takes 1 step.
2. Memory access takes 1 step.
3. Loops and subroutines depend on data size.

Time Complexity (using Big-O):

O(1): Constant time, independent of input size.

O(*n*): Linear time, grows proportionally with input size.

O(*n*²): Quadratic time, grows with the square of input size.

Example 1: Simple Sequence of Statements

If a program has **k simple statements**:

Statement 1;

Statement 2;

...

Statement *k*;

Each takes constant time if it involves basic operations.

Total time = O(1 + 1 + ... + 1) = O*(k)* = O(1) (since *k* is a fixed constant).

Exact analysis of insertion sort:

1. for $j = 2$ to length[A] $\rightarrow c1$, executed $(n - 1) + 1$ times
2. key = A[j] $\rightarrow c2$, executed $(n - 1)$ times
3. $i = j - 1$ $\rightarrow c3$, executed $(n - 1)$ times
4. while ($i > 0$) and (A[i] > key) $\rightarrow c4$, executed $\sum(Tj)$ times
5. A[$i + 1$] = A[i] $\rightarrow c4$, executed $\sum(Tj - 1)$ times
6. $i = i - 1$ $\rightarrow c5$, executed $\sum(Tj - 1)$ times
7. A[$i + 1$] = key $\rightarrow c6$, executed $(n - 1)$ times

Where:

1. Tj is the number of times the while-loop runs for each value of j (from 2 to n).
2. n is the length of the array A.

The total time T*(n)* is the sum of times for each line multiplied by its cost factor:

$$T(n) = c_1 n + c_2 (n - 1) + c_3 (n - 1) + c_4 \sum_{j=2}^{n} T_j + c_5 \sum_{j=2}^{n} (T_j - 1) + c_6 \sum_{j=2}^{n} (Tj - 1) + c_7 (n - 1)$$

Worst Case

- Worst case is an upper bound, ensuring the algorithm won't exceed this time.
- Occurs when input is reverse sorted.
- In this case, A[i] > key is always true, so:

$$\sum_{j=2}^{n} T_j = \frac{n(n+1)}{2} - 1$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

Best Case

- Occurs when input is already sorted.
- A[i] <= key at line 4, so the inner loop doesn't run.

Then,

$T(n) = c_1 n + c_2 (n - 1) + c_3 (n - 1) + c_4 (n - 1) = O(n)$

Average Case: Average over all input cases of size n. Still results in O(**n^2**) complexity.

$$T(n)_{\text{best}} < T(n)_{\text{Avg.}} < T(n)_{\text{worst}}$$

CALCULATION OF STORAGE COMPLEXITY

As memory becomes cheaper, runtime complexity gains more attention, but space complexity remains crucial. If a program exceeds memory limits, it can't run at all, making space usage more critical than runtime in some cases. Fortunately, memory is often reused during execution.

Iterative Programs: Space complexity is easier to analyze – based on variable declarations and data structures (e.g., arrays).

Recursive Programs: Space complexity is more complex. Active recursive calls stack up, using space for: Local variables, Arguments, Return addresses.

For n recursive calls, space complexity is typically O*(n)*.

Example: Binary Recursion

1. If $n == 0$ or $1 \rightarrow$ return 1
2. Recur for $f(n - 1)$
3. Recur for $f(n-2)$
4. Return sum of step 2 and 3

This binary recursion creates two calls per step, leading to **O(2^n)** time and **O*(n)*** space complexity due to the call stack.

CALCULATION OF TIME COMPLEXITY

Example 1: Simple Code

$x = 4y + 3$

$z = z + 1$

$p = 1$

All variables are scalar, and each line takes constant time regardless of values. So, runtime = O(1) for each line.

Example 2: Binary Search

In a sorted list, binary search halves the search space in each step:

N, N/2, N/4, ..., 1

This continues until the interval size is 1.

Taking log base 2 of each step gives:

$\log_2 N, \log_2 N - 1, ..., 0$

Total iterations = $\log_2 N + 1$, hence time complexity = O(log N).

The Complexity Ladder:

1. **O(1) – Constant Time:** Time doesn't grow with input size. *Example: Array access A[i]*
2. **O(log *n*) – Logarithmic Time:** Grows slowly with input. *Example: Binary Search*
3. **O*(n)* – Linear Time:** Grows directly with input size. *Example: Looping over an array*
4. **O(*n* log *n*) – Linearithmic Time:** Grows faster than linear but less than quadratic. *Example: Merge Sort*
5. **O(n^k) – Polynomial Time:** Grows with the k-th power of n. *Example: Selection Sort (O(n^2))*
6. **O(2^n)–Exponential Time:** Grows extremely fast; impractical for large inputs. *Example: Some recursive algorithms*

CHECK YOUR PROGRESS

Q. 1. The function $9n + 12$ and $1000n + 400000$ are both $O(n)$. (True/False)

Ans. True.

Q. 2. If a function $f(n) = O(g(n))$ and $h(n) = O(g(n))$, then $f(n) + h(n) = O(g(n))$. (True/False)

Ans. True.

Q. 3. If $f(n) = n^2 + 3n$ and $g(n) = 6000n + 34000$ then $O(f(n)) < O(g(n))$.) (True/False)

Ans. False.

Q. 4. The asymptotic complexity of algorithms depends on hardware and other factors.

Ans. False.

Q. 5. Give simplified big-O notation for the following growth functions:

1. $30n^2$

2. $10n^3 + 6n^2$

3. $5nlog\ n + 30n$

4. $\log n + 3n$

5. $\log n + 32$

Ans. 1. $30n^2$

Dominant term: n^2

Constants are ignored in Big-O.

Big-O: $O(n^2)$

2. $10n^3 + 6n^2$

Dominant term: n^3 (because it grows faster than n^2)

Drop lower-order terms and constants.

Big-O: $O(n^3)$

3. $5n \log n + 30n$

Dominant term: $n \log n$ (grows faster than n)

Drop lower-order terms and constants.

Big-O: $O(n \log n)$

4. $\log n + 3n$

Dominant term: n (grows faster than $\log n$)

Drop lower-order term.

Big-O: $O(n)$

5. $\log n + 32$

Dominant term: $\log n$ (since 32 is a constant)

Big–O: $O(\log n)$

Table

Function	**Big-O**
$30n^2$	$O(n^2)$
$10n^3 + 6n^2$	$O(n^3)$
$5n \log n + 30n$	$O(n \log n)$
$\log n + 3n$	$O(n)$
$\log n + 32$	$O(\log n)$

Q. 6. The set of algorithms whose order is O (1) would run in the same time. True/False

Ans. True.

Q. 7. Find the complexity of the following program in big O notation:

```
printMultiplicationTable(int max){
for(int i = 1 ; i <= max ; i + +)
{
for(int j = 1 ; j <= max ; j + +)
cout << (i * j) << " " ;
cout << endl ;
} //for
```

Ans.
```
printMultiplicationTable(int max) {
  for (int i = 1; i <= max; i++) {        // Outer loop
    for (int j = 1; j <= max; j++) {    // Inner loop
        cout << (i * j) << " ";          // Constant-
time operation
    }
    cout << endl;                       // Constant-time
operation
  }
}
```

Understanding the Loops:

1. Outer Loop:

- Runs from $i = 1$ to $i <= \max$.
- This means it runs **max times** in total.

2. Inner Loop (nested inside outer loop):

- For each value of i, j runs from 1 to max.
- So it also runs max times for every iteration of the outer loop.

3. Total Iterations: Since the inner loop runs max times inside the outer loop which itself runs max times, the total number of operations becomes:

4. Work Done Inside the Loops: cout << (i * j) << " " is a simple arithmetic and output operation, which we consider to take constant time: O(1).

Total Time Complexity: $O(\max \times \max \times 1) = \boxed{(O(\max^2))}$

This is called quadratic time complexity, meaning the program's runtime grows proportional to the square of the input size (`max`).

Q. 8. Consider the following program segment:

```
for (i = 1; i <= n; i *= 2)
  {
  j = 1;
  }
```

What is the running time of the above program segment in big O notation?