Zakho Technical College

Department of Computer Information System

Data Structures and Algorithms

1. Introduction to Data Structures and Algorithms

Lecturers:

Sipan M. Hameed

Ahmed Jamil

www.sipan.dev

2024-2025

Contents

1.1 DSA (D**ata S**tructures and A**lgorithms)**

A data structure is a way to store data.is the study of organizing data efficiently using data structures like arrays, stacks, and trees, paired with step-by-step procedures (or algorithms) to solve problems effectively. Data structures manage how data is stored and accessed, while algorithms focus on processing this data.

1.2 **different kinds of data structures**

in Computer Science there are two different kinds of data structures.

Primitive Data Structures are basic data structures provided by programming languages to represent single values, such as integers, floating-point numbers, characters, and Booleans.

Abstract Data Structures are higher-level data structures that are built using primitive data types and provide more complex and specialized operations. Some common examples of abstract data structures include arrays, linked lists, stacks, queues, trees, and graphs.

1.3 **Why to Learn DSA?**

- Learning DSA boosts your problem-solving abilities and make you a stronger programmer.

- DSA is foundation for almost every software like GPS, Search Engines, AI ChatBots, Gaming Apps, Databases, Web Applications, etc

- Top Companies like **Google, Microsoft, Amazon, Apple, Meta** and many other heavily focus on DSA **i**n interviews.

1.4 **What are Algorithms?**

An algorithm is a set of step-by-step instructions to solve a given problem or achieve a specific goal. When we talk about algorithms in Computer Science, the step-by-step instructions are written in a programming language, an algorithm uses data structures.

Algorithms are fundamental to computer programming as they provide step-by-step instructions for executing tasks. An efficient algorithm can help us to find the solution we are looking for, and to transform a slow program into a faster one.

By studying algorithms, developers can write better programs.

Algorithm examples:

- Finding the fastest route in a GPS navigation system
- Navigating an airplane or a car (cruise control)
- Finding what users search for (search engine)
- Sorting, for example sorting movies by rating

1.5 Data Structures together with Algorithms

Data structures and algorithms (DSA) go hand in hand. A data structure is not worth much if you cannot search through it or manipulate it efficiently using algorithms, and the algorithms in this tutorial are not worth much without a data structure to work on.

DSA is about finding efficient ways to store and retrieve data, to perform operations on data, and to solve specific problems.

By understanding DSA, you can:

- Decide which data structure or algorithm is best for a given situation.
- Make programs that run faster or use less memory.
- Understand how to approach complex problems and solve them in a systematic way.

1.6 Where is Data Structures and Algorithms Needed?

Data Structures and Algorithms (DSA) are used in virtually every software system, from operating systems to web applications:

- For managing large amounts of data, such as in a social network or a search engine.
- For scheduling tasks, to decide which task a computer should do first.
- For planning routes, like in a GPS system to find the shortest path from A to B.
- For optimizing processes, such as arranging tasks so they can be completed as quickly as possible.
- For solving complex problems: From finding the best way to pack a truck to making a computer 'learn' from data.

DSA is fundamental in nearly every part of the software world:

- Operating Systems
- Database Systems
- Web Applications
- Machine Learning
- Video Games
- Cryptographic Systems
- Data Analysis
- Search Engines

1.7 Theory and Terminology

As we go along in this tutorial, new theoretical concepts and terminology (new words) will be needed so that we can better understand the data structures and algorithms we will be working on.

These new words and concepts will be introduced and explained properly when they are needed, but here is a list of some key terms, just to get an overview of what is coming:

Term	Description
Algorithm	A set of step-by-step instructions to solve a specific problem.
Data Structure	A way of organizing data so it can be used efficiently. Common data structures include arrays, linked lists, and binary trees.
Time Complexity	A measure of the amount of time an algorithm takes to run, depending on the amount of data the algorithm is working on.
Space Complexity	A measure of the amount of memory an algorithm uses, depending on the amount of data the algorithm is working on.
Big O Notation	A mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. Used in this tutorial to describe the time complexity of an algorithm.
Recursion	A programming technique where a function calls itself.
Divide and Conquer	A method of solving complex problems by breaking them into smaller, more manageable sub-problems, solving the sub-problems, and combining the solutions. Recursion is often used when using this method in an algorithm.
Brute Force	A simple and straight forward way an algorithm can work by simply trying all possible solutions and then choosing the best one.

1.8 Key Considerations when choosing a Data Structure

In the realm of computer science, data is the lifeblood of any program. But raw, unorganized data is of limited use. We need ways to organize and manage this data efficiently so that we can process it meaningfully. This is where *data structures* come into play.

A data structure is a specialized way of organizing, storing, and retrieving data in a computer's memory. It's a blueprint for how data is arranged, allowing for efficient access and manipulation. Think of it like choosing the right container for your belongings. You wouldn't store books in a shoe box or shoes in a bookshelf. Similarly, different data structures are suited for different types of data and operations.

Key Considerations when choosing a Data Structure:

- **Type of data:** Is it a collection of numbers, text, or more complex objects?
- **Operations:** What kind of operations will be performed on the data (e.g., searching, sorting, inserting, deleting)?
- **Efficiency:** How quickly do these operations need to be performed?
- **Memory usage:** How much memory is required to store the data?

1.3 The Importance of Data Structures and Algorithms

Data structures and algorithms are fundamental to computer science. They are the building blocks of virtually every software application. Here's why they are so important:

- **Efficiency:** Choosing the right data structure and algorithm can dramatically improve the performance of a program. Efficient algorithms can reduce execution time and memory usage, leading to faster and more responsive applications.
- **Organization:** Data structures provide a way to organize and manage large amounts of data, making it easier to search, retrieve, and process.
- **Problem Solving:** DSA provides a framework for thinking about and solving problems computationally. It equips you with the tools to analyze problems, design solutions, and implement them effectively.
- **Scalability:** Well-designed data structures and algorithms are essential for building scalable applications that can handle increasing amounts of data and traffic.
- **Software Development:** A strong understanding of DSA is crucial for any software developer. It enables you to write better code, design efficient systems, and solve complex problems.
- **Job Interviews:** DSA is a common topic in technical interviews for software engineering roles. Demonstrating proficiency in DSA is often a requirement for landing a job in the tech industry.

1.9 Types of Data Structures

Data structures can be broadly classified into various categories, including:

- **Linear Data Structures:** Data elements are arranged in a sequential or linear fashion. Examples include:
    - **Arrays:** A fixed-size collection of elements of the same data type.
    - **Linked Lists:** A dynamic collection of elements, where each element points to the next.
    - **Stacks:** A LIFO (Last-In, First-Out) data structure.
    - **Queues:** A FIFO (First-In, First-Out) data structure.
- **Non-Linear Data Structures:** Data elements are not arranged sequentially. Examples include:
    - **Trees:** Hierarchical data structures with a root node and child nodes.
    - **Graphs:** A collection of vertices (nodes) and edges that connect them.
    - **Hash Tables:** Data structures that use a hash function to map keys to values.

1.10 Types of algorithms

Algorithms are like recipes for solving problems with a computer. They are step-by-step instructions that tell a computer how to perform a task. Just like there are many different ways to cook a meal, there are many different types of algorithms, each with its own strengths and weaknesses. Here's a breakdown of some common algorithm types:

1. Sorting Algorithms
Examples:

- **Bubble Sort:** Simple but inefficient for large datasets.
- **Insertion Sort:** Efficient for small or nearly sorted lists.
- **Merge Sort:** Efficient for large datasets, uses a "divide and conquer" strategy.
- **Quick Sort:** Also efficient for large datasets, uses a "divide and conquer" strategy.

2. Searching Algorithms

These algorithms find a specific item in a dataset.

- **Examples:**
    - **Linear Search:** Checks each item one by one (like looking for a book in a randomly organized library).
    - **Binary Search:** Efficient for sorted datasets, repeatedly divides the search space in half (like using an index to find a word in a dictionary).

3. Graph Algorithms

These algorithms work with data represented as a network of nodes (vertices) and connections (edges).

- **Examples:**
    - **Breadth-First Search (BFS):** Explores the graph layer by layer (like exploring a maze by trying all paths at the same time).
    - **Depth-First Search (DFS):** Explores the graph by going as deep as possible along each branch before backtracking (like exploring a maze by following one path until you hit a dead end, then going back).
    - **Dijkstra's Algorithm:** Finds the shortest path between two nodes in a graph (like finding the fastest route on a map).

4. Dynamic Programming Algorithms

These algorithms solve problems by breaking them down into smaller overlapping subproblems, solving each subproblem only once, and storing their solutions to avoid redundant computations.

- **Example:** Calculating the Fibonacci sequence efficiently.

5. Greedy Algorithms

These algorithms make locally optimal choices at each step in the hope of finding a globally optimal solution. They don't always guarantee the best solution, but they are often efficient and provide good approximations.

- **Example:** Making change with the fewest number of coins.

6. Divide and Conquer Algorithms

- These algorithms solve problems by recursively breaking them down into smaller subproblems, solving each subproblem independently, and combining the solutions to solve the original problem.
- **Example:** Merge Sort (sorting algorithm).

7. Backtracking Algorithms

These algorithms explore potential solutions by trying different options and undoing them (backtracking) if they don't lead to a solution.

- **Example:** Solving a Sudoku puzzle.

8. Brute-Force Algorithms

These algorithms try all possible solutions until the correct one is found. They are simple but can be inefficient for large problem spaces.

- **Example:** Trying every possible password to crack a code.

9. Randomized Algorithms

These algorithms use randomness in their steps. They may not always give the exact solution, but they can provide good approximations with a high probability.

- **Example:** QuickSort (sorting algorithm).

10. Machine Learning Algorithms

These algorithms learn patterns from data and use those patterns to make predictions or decisions.

- **Examples:**
  - **Linear Regression:** Predicting a value based on a linear relationship with other values.
  - **Decision Trees:** Making decisions based on a tree-like structure of rules.
  - **Neural Networks:** Complex algorithms inspired by the human brain, used for tasks like image recognition and natural language processing.

1.11 Choosing the Right Algorithm

The choice of algorithm depends on several factors, including:

- **The nature of the problem:** What kind of problem are you trying to solve?
- **The size of the input:** How much data will the algorithm be processing?
- **The desired level of accuracy:** Do you need an exact solution or an approximation?
- **The available resources:** How much time and memory do you have available?

Understanding the different types of algorithms and their characteristics will help you choose the most appropriate one for your specific needs.

1.12 Practical Examples:

1.13 Array

1. Introduction to Arrays

Arrays store multiple elements of the same type in contiguous memory. They have a fixed size once created.

2. Single-Dimensional Arrays

Declaration & Initialization:

```
// Declaration
int[] numbers1 = new int[5]; // 5 elements (0-4), initialized to 0
int[] numbers2 = { 1, 2, 3, 4, 5 }; // Implicit initialization
```

```
// Accessing elements
numbers1[0] = 10; // First element
Console.WriteLine(numbers2[3]); // Output: 4
```

Looping:

```
// For loop
for (int i = 0; i < numbers2.Length; i++)
{
    Console.WriteLine(numbers2[i]);
}
// Foreach loop
foreach (int num in numbers2)
{
    Console.WriteLine(num);
}
```

3. Multidimensional Arrays (Rectangular)

2D Array Example:

```
int[,] matrix = new int[2, 3] { { 1, 2, 3 }, { 4, 5, 6 } };
```

```
// Accessing
Console.WriteLine(matrix[1, 2]); // Output: 6
```

```
// Looping
for (int i = 0; i < matrix.GetLength(0); i++)
{
    for (int j = 0; j < matrix.GetLength(1); j++)
    {
        Console.Write(matrix[i, j] + " ");
    }
    Console.WriteLine();
}
```

4. Jagged Arrays (Array of Arrays)

```
int[][] jagged = new int[3][];
jagged[0] = new int[] { 1, 2 };
jagged[1] = new int[] { 3 };
jagged[2] = new int[] { 4, 5, 6 };
```

```
// Accessing
Console.WriteLine(jagged[2][1]); // Output: 5
```

```
// Looping
foreach (int[] arr in jagged)
{
    foreach (int num in arr)
    {
        Console.Write(num + " ");
    }
    Console.WriteLine();
}
```

5. Common Methods & Properties

Length & GetLength:

```
int[] arr = { 1, 2, 3 };
Console.WriteLine(arr.Length); // 3 (total elements)
```

```
int[,] grid = new int[2, 4];
Console.WriteLine(grid.GetLength(0)); // 2 (rows)
Console.WriteLine(grid.GetLength(1)); // 4 (columns)
```

Sorting & Reversing:

```

Array.Sort(arr); // Sorts in ascending order
Array.Reverse(arr); // Reverses element order
```

Copying:

```
int[] copy1 = (int[])arr.Clone(); // Shallow copy
int[] copy2 = new int[3];
arr.CopyTo(copy2, 0); // Copies to target array
```

6. Working with Arrays

Passing to Methods:

```
static void SquareElements(int[] arr)
{
    for (int i = 0; i < arr.Length; i++)
    {
        arr[i] *= arr[i];
    }
}
```

// Usage:

```
int[] nums = { 2, 3, 4 };
SquareElements(nums); // Modifies original array
```

7. Default Values & Initialization

Value types: 0 for numeric, false for bool

Reference types: null

```
bool[] bools = new bool[3]; // [false, false, false]
string[] names = new string[2]; // [null, null]
```

Fixed size: Can't dynamically resize (use List<T> for dynamic collections)

8. Resizing Arrays

Use Array.Resize (creates a new array):

```
int[] numbers = { 1, 2, 3 };
Array.Resize(ref numbers, 5); // New size: 5
// numbers becomes [1, 2, 3, 0, 0]
```

9. Conclusion

Use single-dimensional arrays for simple lists.

Multidimensional arrays for matrices/grids.

Jagged arrays for variable-length rows.

Prefer List<T> when dynamic resizing is needed.

Final Tips:

Initialize jagged array sub-arrays before use.

Use foreach for read-only iteration and for for modification.

1.14 Basic Array Operations

Example 1: Full lifecycle of an array

```
using System;

namespace ConsoleApp14
{
    internal class Program
    {
        static void Main(string[] args)
        {
            // Declaration and initialization
            string[] fruits = new string[4] {
"Date","Banana","Cherry","Apple"   };

            // Modification
            fruits[2] = "Cantaloupe"; // Change 3rd element
            Array.Resize(ref fruits, 5); // Add extra capacity
            fruits[4] = "Elderberry";

            // Iteration and output
            Console.WriteLine("Fruits Collection:");
            foreach (string fruit in fruits)
            {
                Console.WriteLine($"- {fruit}");
            }

            // Search operation
            int index = Array.IndexOf(fruits, "Date");
            Console.WriteLine($"Index of 'Date': {index}"); // Output: 3

            // Sorting
            Array.Sort(fruits);
            Console.WriteLine("\nSorted Fruits:");
            Console.WriteLine(string.Join(", ", fruits));
            // Apple, Banana, Cantaloupe, Date, Elderberry

        }
    }
}
```

2. Working with User Input:

```
using System;

namespace ConsoleApp14
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter the number of scores:");
            int numScores = int.Parse(Console.ReadLine());
            // Get the number of scores from the user

            int[] scores = new int[numScores];
            // Create an array of the specified size

            for (int i = 0; i < numScores; i++)
            {
                Console.WriteLine($"Enter score {i + 1}:");
                scores[i] = int.Parse(Console.ReadLine());
                // Get each score from the user
            }

            Console.WriteLine("\nScores entered:");
            for (int i = 0; i < numScores; i++)
            {
                Console.WriteLine($"Score {i + 1}: {scores[i]}");
            }

            // Calculate the average:
            int sum = 0;
            for (int i = 0; i < numScores; i++)
            {
                sum += scores[i];
            }
            double average = (double)sum / numScores;
            // Important: Cast to double for accurate division
            Console.WriteLine($"Average score: {average}");



        }
    }
}
```

3. Searching an Array:

```
using System;

namespace ConsoleApp14
{
    internal class Program
    {
        static void Main(string[] args)
        {
            string[] names = { "Alice", "Bob", "Charlie", "David", "Eve" };

            Console.WriteLine("Enter a name to search for:");
            string searchName = Console.ReadLine();

            // Linear search (simple but not efficient for large arrays):
            bool found = false;
            for (int i = 0; i < names.Length; i++)
            {
                if (names[i].Equals(searchName,
StringComparison.OrdinalIgnoreCase))
                { // Case-insensitive comparison
                    Console.WriteLine($"{searchName} found at index {i}");
                    found = true;
                    break; // Exit the loop once found
                }
            }

            if (!found)
            {
                Console.WriteLine($"{searchName} not found.");
            }

            // Using Array.IndexOf (more concise for simple searches):
            int index = Array.IndexOf(names, searchName);
            if (index != -1)
            {
                Console.WriteLine($"{searchName} found at index {index}");
            }
            else
            {
                Console.WriteLine($"{searchName} not found.");
            }

        }
    }
}
```