Zakho Technical College

Department of Computer Information System

Data Structures and Algorithms

2. Analysis (array and stack)

Lecturers:

Sipan M. Hameed

Ahmed Jamil

www.sipan.dev

2024-2025

1. Table of Contents

2. Analysis

Analysis of Algorithms is a fundamental aspect of computer science that involves evaluating performance of algorithms and programs. Efficiency is measured in terms of **time** and **space**.

2.1 Big O Notation

Big O notation is a way to describe **how the time or space requirements of an algorithm grow** as the input size increases. It answers:

"How does my algorithm slow down as the input gets very large?"

Big O notation is a way to describe how the runtime or space requirements of an algorithm grow as the input size grows. It's like a way of classifying algorithms based on their efficiency. It doesn't tell you the exact time an algorithm takes, but it tells you how the time scales with the input size.

Think of it this way: you have a recipe (an algorithm) for baking a cake. Big O notation tells you how the amount of time you spend baking changes as you make more and more cakes (larger input size).

2.2 Why It Matters

- Helps compare algorithm efficiency.
- Predicts scalability (e.g., "Will this code work for 1 million users?").
- Focuses on the **worst-case scenario** (pessimistic but safe estimate).

2.3 Key Concepts

1. **Ignore Constants**:
   $O(2n) \rightarrow O(n)$
   $O(500) \rightarrow O(1)$
   (Constants don't matter for large inputs).
2. **Focus on Dominant Terms**:
   $O(n^2 + n) \rightarrow O(n^2)$
   $O(\log n + n) \rightarrow O(n)$
   (Only the fastest-growing term matters).

2.4 Big O Notation examples:

Big O	Description	Example Operation	Operations (n=10)	Operations (n=100)	Operations (n=1000)	Operations (n=10,000)	Operations (n=100,000)
$O(1)$	Constant	Check if the first name is "Alice"	1	1	1	1	1
$O(\log n)$	Logarithmic	Find a name in a sorted list using binary search	~3	~7	~10	~13	~17
$O(n)$	Linear	Go through the list and print each name	10	100	1000	10,000	100,000
$O(n \log n)$	Linearithmic	Sort the list alphabetically (efficiently)	~33	~664	~9970	~133,000	~1,660,000
$O(n^2)$	Quadratic	Compare each name to every other name in the list	100	10,000	1,000,000	100,000,000	10,000,000,000
$O(2^n)$	Exponential	Try all possible combinations of names from the list	1024	1.27×10^{30}	1.07×10^{301}	1.79×10^{3010}	3.23×10^{30103}

2.5 When to Use Big O

- **Optimization:** Avoid $O(n^2)$ for large datasets.
- **Choosing Data Structures:** Use a **hash table (O(1))** instead of a **list (O(n))** for fast lookups.
- **Designing Systems:** Ensure your code scales for future growth.

2.6 C# array operations

1. Access by Index

- **Time Complexity:** O(1)
- **Explanation:** Directly access an element using its index.
- **Code Example:**

```
int[] arr = { 10, 20, 30, 40, 50 };
int element = arr[2]; // Accesses 30 (O(1))
```

2. Linear Search

- **Time Complexity:** O(n)
- **Explanation:** Iterate through the array to find an element.
- **Code Example:**

```
int index = Array.IndexOf(arr, 30); // Returns 2 (O(n))
```

3. Binary Search

- **Time Complexity:** O(log n)
- **Explanation:** Requires a **sorted array**. Repeatedly divides the search interval in half.
- **Code Example:**

```
Array.Sort(arr); // Sort first (O(n log n))
int index = Array.BinarySearch(arr, 30); // Returns 2 (O(log n))
```

4. Insertion at Specific Index

- **Time Complexity:** O(n)
- **Explanation:** Requires creating a new array and copying elements, shifting subsequent elements.
- **Code Example:**

```
int[] arr = { 10, 20, 30, 40, 50 };
int[] newArr = new int[arr.Length + 1];
int insertIndex = 2;
int newElement = 25;

// Copy elements before the insertion point
Array.Copy(arr, 0, newArr, 0, insertIndex);
newArr[insertIndex] = newElement;
// Copy elements after the insertion point
Array.Copy(arr, insertIndex, newArr, insertIndex + 1, arr.Length - insertIndex);
arr = newArr; // New array: { 10, 20, 25, 30, 40, 50 }
```

5. Deletion at Specific Index

- **Time Complexity:** O(n)
- **Explanation:** Requires creating a new array and copying elements, skipping the deleted element.
- **Code Example:**

```
int[] arr = { 10, 20, 30, 40, 50 };
int deleteIndex = 2;
int[] newArr = new int[arr.Length - 1];

// Copy elements before the deletion point
Array.Copy(arr, 0, newArr, 0, deleteIndex);
// Copy elements after the deletion point
Array.Copy(arr, deleteIndex + 1, newArr, deleteIndex, arr.Length -
deleteIndex - 1);
arr = newArr; // New array: { 10, 20, 40, 50 }
```

6. Copying an Array

- **Time Complexity:** O(n)
- **Explanation:** Copies all elements to a new array.
- **Code Example:**

```
int[] copy1 = (int[])arr.Clone(); // Shallow copy
int[] copy2 = new int[arr.Length];
Array.Copy(arr, copy2, arr.Length);
```

7. Resizing an Array

- **Time Complexity:** O(n)
- **Explanation:** Creates a new array and copies elements to it.
- **Code Example:**

```
Array.Resize(ref arr, arr.Length + 1); // Adds a new element (default 0)
```

8. Sorting

- **Time Complexity:** O(n log n) (average case for `Array.Sort`)
- **Explanation:** Uses an optimized sorting algorithm (e.g., QuickSort).
- **Code Example:**

```
Array.Sort(arr); // Sorts in-place
```

9. Reversing

- **Time Complexity:** O(n)
- **Explanation:** Reverses the order of elements.
- **Code Example:**

```
Array.Reverse(arr); // Reverses in-place
```

2.7 Summary

Operation	Big O Notation	Description
Accessing an Element	O(1)	Constant time access using an index.
Searching for an Element	O(n)	Linear time search for an element.
Searching for an Element (Binary search)	O(log n)	Binary time search for an element.
Inserting an Element	O(n)	Linear time insert requires shifting elements.
Deleting an Element	O(n)	Linear time delete requires shifting elements.
Iterating Over an Array	O(n)	Linear time to perform operations on each element.

2.8 stack

The Stack<T> class in C# provides a last-in, first-out (LIFO) collection. It's useful when you need to process items in the reverse order they were added. Here's a breakdown of how to use it, along with common operations and examples:

```
using System;
using System.Collections.Generic;
```

1. Push (Add to Top)

- **Time Complexity: O(1)** *(amortized)*
  - **Explanation:** Adding to the top of the stack is usually O(1). If the internal array is full, it doubles in size (O(n) for resizing), but this cost is "amortized" over many operations, making the average O(1).
- **Code Example:**

```
Stack<int> stack = new Stack<int>();
stack.Push(10);  // O(1)
stack.Push(20);  // O(1)
```

2. Pop (Remove from Top)

- **Time Complexity: O(1)**
  - **Explanation:** Removes the top element directly (no resizing needed).
- **Code Example:**

```
int top = stack.Pop(); // Removes and returns 20 (O(1))
```

3. Peek (Inspect Top)

- **Time Complexity: O(1)**
  - **Explanation:** Returns the top element without removing it.
- **Code Example:**

```
int currentTop = stack.Peek(); // Returns 10 (O(1))
```

4. Contains (Check Existence)

- **Time Complexity: O(n)**
  - **Explanation:** Searches the entire stack in the worst case.
- **Code Example:**

```
bool hasTen = stack.Contains(10); // True (O(n))
```

5. Count (Get Size)

- **Time Complexity: O(1)**
  - **Explanation:** The count is tracked internally as a property.
- **Code Example:**

```
int size = stack.Count; // Returns 1 (O(1))
```

6. Clear (Remove All Elements)

- **Time Complexity: O(n)**
  - **Explanation:** Resets the internal array (requires iterating through elements to clear references if applicable).
- **Code Example:**

```
stack.Clear(); // Stack is now empty (O(n))
```

7. ToArray (Copy to New Array)

- **Time Complexity: O(n)**
  - **Explanation:** Copies all elements to a new array.
- **Code Example:**

```
int[] arrayCopy = stack.ToArray(); // O(n)
```

2.8.1 Stack Key Notes

- **Underlying Implementation:** C# `Stack<T>` uses an **array internally**, so resizing during `Push` can occasionally take O(n) time, but this is averaged to O(1) over many operations (amortized analysis).
- **No Index Access:** Unlike arrays, you can't access elements by index (e.g., `stack[0]` is invalid). Use `ToList()` or `ToArray()` first (O(n)).
- **Use Cases:** Stacks are ideal for LIFO (Last-In-First-Out) scenarios like undo/redo functionality, backtracking algorithms, or parsing expressions.

2.8.2 Stack Common Use Cases for Stacks:

- **Undo/Redo Functionality:** Stacks are perfect for implementing undo/redo features in applications.
- **Function Call Stack:** The runtime environment uses a stack to manage function calls.
- **Expression Evaluation:** Stacks are used in evaluating mathematical expressions (e.g., converting infix to postfix notation).
- **Backtracking Algorithms:** Stacks are helpful in algorithms that involve exploring different paths (e.g., depth-first search).