A2 – S4 – 2025/2026 – II.2415
Advanced Algorithms and Programming
Semester Project

LAB_1

Revision of Algorithms' Fundamentals

Session Objectives

By the end of this session, you will be able to:

1. Implement fundamental algorithms using mathematical operations
2. Analyze simple time and space complexity of algorithms
3. Apply appropriate simple data structures to solve common problems
4. Establish effective team collaboration and version control practices

Core Exercises

Exercise 1: Integer Mirror (Digit Reversal)
Objective: Practice mathematical manipulation without string/table conversion.
Given a non-negative integer n, return the number formed by reversing its decimal digits. Do not convert the integer to a string at any point.
Examples:

- Input: 315 → Output: 513
- Input: 400 → Output: 4 (note trailing zeros)
- Input: 7 → Output: 7

Requirements:

1. Implement using only mathematical operations (+, -, *, div, mod)
2. Handle edge cases: zero, single digit, numbers with trailing zeros
3. Analyze the number of operations needed for an n-digit number

Complexity Analysis Questions:

1. How many arithmetic operations are performed for a d-digit number?
2. What is the time complexity (number of expensive operations) in terms of the input value n?

Exercise 2: Balanced Symbol Checker
Objective: Implement stack-based algorithm for pattern matching.
Given a string s containing only the characters ( ) [ ] { }, determine if the string is properly balanced. A string is balanced if:

1. Every opening symbol has a corresponding closing symbol
2. Symbols are properly nested

Examples:

- Input: "({[]})" → Output: True
- Input: "([)]" → Output: False
- Input: "" → Output: True (empty string is balanced)
- Input: "((())" → Output: False

Requirements:

1. Use a stack data structure

2. Return boolean result
3. Count the number of comparison operations

Complexity Analysis Questions:

1. How many comparisons are needed for an n-character string?
2. What happens if we extend to include additional symbol pairs or other different characters?

Exercise 3: Merge Overlapping Intervals

Objective: Practice sorting and interval manipulation algorithms.

Given a collection of intervals where each interval has integer start and end values (with start ≤ end), merge all overlapping intervals and return a list of non-overlapping intervals that cover the same ranges.

Examples:

- Input: [[1,3], [2,6], [15,18]], [8,10], Output: [[1,6], [8,10], [15,18]]
- Input: [[1,4], [4,5]], Output: [[1,5]]
- Input: [[1,4], [0,4]], Output: [[0,4]]

Requirements:

1. Sort intervals appropriately before merging
2. Handle edge cases: empty input, single interval, no overlaps
3. Optimize for both time and memory efficiency

Complexity Analysis Questions:

1. How many comparison operations are needed for n intervals?
2. What is the time complexity of your solution?

Exercise 4: Polynomial Evaluation

Objective: Implement efficient mathematical computation.

Implement a function to evaluate a polynomial at a given point using Horner's method. Given:

- An array of coefficients coeffs where coeffs[i] is the coefficient for x^i
- A real number x (the evaluation point)

Example:

- coeffs = [3, -2, 0, 5] represents $P(x) = 3 - 2x + 0x^2 + 5x^3$
- x = 2.0, Output: $P(2) = 3 - 4 + 0 + 40 = 39$

Requirements:

1. Implement Horner's method (not the naive approach)
2. Handle polynomials of any degree
3. Consider numerical precision for floating-point calculations

Complexity Analysis Questions:

1. How many multiplications are needed for a degree-n polynomial?
2. Compare with the naive approach: what is the improvement?

Exercise 5: Array Rotation Optimization

Objective: Implement efficient array manipulation with in-place operations.

Given an array Vect and an integer k, rotate the array to the right by k positions. The rotation should be done in-place with a constant extra space.

Examples:

- Input: Vect = [1, 2, 3, 4, 5, 6, 7], k = 10, Output: [5, 6, 7, 1, 2, 3, 4]
- Input: Vect = [1], k = 5, Output: [1]

Requirements:

1. Implement three different approaches and compare:
   a) Using temporary array (Linear Space)
   b) Rotate one by one (related to n×k time)
   c) Reverse segments method (Linear time, constant space)
2. Handle k larger than array length
3. Provide time and space complexity for each approach

Complexity Analysis Questions:

1. What is the optimal time complexity achievable?
2. How does k affect the performance?
3. When would each approach be preferable?

Exercise 6: First Unique Character Finder

Objective: Practice hash table/dictionary usage for frequency counting.

Given a string s, find the index of the first non-repeating character. If no such character exists, return -1.

Examples:

- Input: "leetcode" → Output: 0 (character 'l')
- Input: "loveleetcode" → Output: 2 (character 'v')
- Input: "aabb" → Output: -1 (no unique character)
- Input: "dddccdbba" → Output: 8 (character 'a')

Requirements:

1. Implement with linear time complexity
2. Try two different approaches:
   a) Using two passes with frequency dictionary
   b) Using ordered dictionary or linked hash map
3. Handle empty string and single character cases

Complexity Analysis Questions:

1. What is the space complexity of your solution?
2. How many passes through the string are needed?
3. What data structure is most efficient for this problem?

Final Question

Give some potential usage of these LAB_1 developed algorithms in your future semester project.

General Instructions

Repository Setup

- One team member creates Git repository
- All members clone repository
- Create branch: LAB _01_Revision

File Structure:

/LAB _01/
├── exercise1_integer_mirror.xx (xx is dependent of the used programming language)
├── exercise2_balanced_symbols.xx
├── exercise3_merge_intervals.xx
├── exercise4_polynomial_eval.xx
├── exercise5_array_rotation.xx
├── exercise6_first_unique.xx
└── README.md (Team members and assigned exercises, Brief description of each solution, Complexity analysis summary)

Git Requirements:

1. Repository: properly initialized and shared
2. Commits: minimum 2 commits per team member
3. Branch: Work in LAB _01_Revision branch
4. History: clear commit messages showing progress

Complexity Guidelines:

- Constant time (Hash Table Lookup): O(1), Logarithmic (binary search): O(log n)
- Linear (single loop): O(n), Linearithmic (sorting): O(n log n)
- Quadratic (nested loops): $O(n^2)$, Exponential (Knapsack Problem): $O(2^n)$

Documentation & Notation

At the End of the Session	**Before Friday Midnight of the Same Week**
Note: 10	**Note: 10**
• Each Team (of number BB) Upload at the **end of the session** a very short PDF document and name it **TBB_LAB1_AES.pdf**, containing for each exercise: the algorithm in Pseudo-Code (according to the rules explained in lecture 1) – Brief answers of the questions, • You can implement your algorithms to make sure that are working, • NO AI-generated content (will be checked for authenticity), • NO code snippets.	• You finish all the implementation of all exercises, • You finalize the previous report and give it a new name **TBB_LAB1_BFM.pdf** and upload it before **Friday midnight of the same week**, containing for each exercise: same as previous + more complexity analysis and reflection + Test set for edge cases of your Algorithms' implementation, • AI-generated content is not recommended, • A link to the Team's Repository in GitHub.