CMPSC122 Lab Section 004L

Lab 12

Lab Proj 12 – Algorithm Analysis

Prepared By

Khang N. Vu

Instructor: Ziyun Huang, Ph.D.

Jonathan Liaw, M.S.

Date: April 29th, 2025

Table of Contents

1 Objective (or Abstract)

Objective of this lab report:

1. Analyze each algorithm's time complexity theoretically using Big-O notation.
2. Compare their actual performance through practical testing on varying input sizes.
3. Determine the most efficient algorithm for real-world applications

2 Introduction

In this case study, I will provide a report on how different algorithms calculate the maximum sum for a contiguous subset in a larger set. There will be three algorithms analyzed in this report: Brute force, divide and conquer using the recursion method, and linear scan using dynamic programming.

I will first provide an analysis of each algorithm, as well as a hypothesis of how well each algorithm will run

3 Procedure

Equipment used for the experiment:

- Laptop Asus ROG Zephyrus G16 (2024) GU605
    - OS: Windows 11 Home - 23H2
    - CPU: Intel(R) Core(TM) Ultra 9 185H
        - 16 Cores, 22 Threads
        - 5.1 GHz – 115W
    - GPU: NVIDIA GeForce RTX 4070 Laptop GPU
        - VRAM: 8GB GDDR6X
    - RAM: 16 GB
        - DDR5 – 7467 MT/s

During the testing period, the program (in Visual Studio Code) was reported by Task Manager to use roughly 10 ± 2% of the CPU and 520 ± 10 MB of RAM

4 Discussion

1. Hypothesis

Brute force (Blue)

This algorithm (Blue) uses a nested loop to check all possible continuous subsets of numbers.

Code analysis:

- The outer loop runs from i = 0 to i = size - 1.

- The inner loop runs from j = 0 to j = size - i - 1, computing the sum of the subsequence starting at i and of length j + 1.
- The maximum sum encountered is stored in currentMax

Big-O analysis:

- The outer loop runs O(n) times.
- The inner loop runs O(n) times in the worst case (when i = 0, it runs n times).
- ⇨ In conclusion, time complexity is $O(n^2)$

Divide and conquer (Green)

This algorithm (Green) uses a recursive divide-and-conquer approach.

Code analysis:

1. Base case
   - If start > end, return 0.
   - If start == end, return max(0, array[start])
2. Divide
   - Split the array into left and right halves around mid = (start + end) / 2.
   - Recursively compute the max sublist sum for the left and right halves.
3. Recursive
   - Compute the max sum that crosses the midpoint by:
   - Extending leftwards from mid to start to find the best left suffix.
   - Extending rightwards from mid + 1 to end to find the best right prefix.
   - The max of the left half, right half, and crossing sum is returned.

Big-O analysis:

- The recurrence relation is: $T(n) = 2T(n/2) + O(n)$. (The O(n) term comes from the two linear scans to compute the crossing sum)
- ⇨ Time complexity is O(n log n)

Linear scan (Red)

The Red function computes the maximum subset sum in one pass.

Code analysis:

- If maxSumEndingAt[i-1] is negative, start a new sublist at i (since adding a negative reduces the sum).
- Otherwise, extend the previous sublist by including array[i].
- Update currentMax if the current sublist sum is greater.

Big-O analysis:

- The loop runs exactly n-1 times from i = 1 to i = size – 1.
- ⇨ Total time complexity is O(n)

2. Expected performance

ALGORITHM	APPROACH	TIME COMPLEXITY	# OPERATIONS TO FINISH N = 64000
BLUE	Brute force	$O(n^2)$	$64000^2 \approx$ 4 Billion
GREEN	Recursion	O(n log n)	64000 * 16 ≈ 1 Million
RED	Linear	O(n)	64000

3. Actual running time

The number of loops is set at 20, which will test n = 500, 1000, 2000, 4000, …, 262144000.

ALGORITHM	TIME COMPLEXITY	TOTAL TIME TAKEN (SECONDS)
BLUE	$O(n^2)$	1009.494 *
GREEN	O(n log n)	34.211
RED	O(n)	1.248

* The Blue algorithm is only tested for 12 loops (n = 500, 1000, 2000, …, 1024000) due to time constraints, but a comprehensive conclusion can still be surmised.

5 Conclusion

- The Blue algorithm ($O(n^2)$) is substantially slower compared to the other two algorithms and is extremely slow for large inputs. This matches with the time complexity where for each time n doubles, the time it takes to calculate quadruple.

⇨ Not suitable for real-world use beyond very small datasets

- The Green algorithm (O(n log n)) is significantly faster than the Blue algorithm but still slightly slower than the Red algorithm, because the log n fuction grows increadibly slowly, slower than n (in the Blue algorithm), but it is still more than 1 (in the Red algorithm). This is a viable but still generally not a recommended choice for calculating.

⇨ Usable but not recommended.

- The Red algorithm (O(n)) is the best-performing algorithm in practice, matching theoretical expectations for a linear function; matching the expectation for a linear time complexity.

⇨ Ideal for real-world use, even for very large datasets.