Maximum Subarray and Variants in Biotech Signal Detection

CSC 2400 – Design of Algorithms, Fall 2025 Semester

Patrick Galloway

Lane Goss

William Jordan

Problem and Methods Overview

The Maximum Subarray problem seeks to identify a contiguous region within an array that yields the largest possible sum. In biological datasets, such as gene expression curves, protein activation levels, and RNA sequencing coverage, these numeric sequences can exhibit significant "hotspot" regions associated with biological functions or disease mechanisms. As such, the ability to automate the detection of such zones is of immense value to the biotech industry.

In this project, we have applied three algorithmic approaches to detect these high-signal regions:

1. Divide-and-Conquer – recursively splits the array and combines partial solutions.
2. Greedy (Kadane's Algorithm) - linear-time dynamic strategy that tracks best subarrays ending at each position.
3. 2D Dynamic Programming – applies a row-collapsing technique and Kadane's algorithm to identify maximum-sum submatrices in 2D biological heatmaps

A brute-force reference method was also used for ensuring algorithmic correctness using small datasets of predetermined values.

Research Objectives

From the outset, our goal was to conclusively determine which of these three methods is the most efficient solution to the maximum subarray problem, and if their documented orders of growth provide an accurate answer to this question. With an order of growth of $\theta(n)$, the greedy method (Kadane's algorithm) is expected to be the most efficient, as it is below divide-and-conquer, $\theta(n\log n)$, and the 2D dynamic programming method, $\theta(n^3)$ in terms of scale.

Additionally, we wished to determine if either of those methods could perform better than the greedy one for the same number of elements at certain, very small amounts, despite being theoretically slower. Some algorithms, when at low input sizes, cease to be so

inefficient as to be impractical, and this is an avenue of investigation that could not be overlooked.

Experimental Procedure

For this experiment, our test program was written in Python 3.13 using the PyCharm IDE. It also draws upon multiple public libraries from Python's extensive catalog, such as NumPy, time, csv, and matplotlib. Each of these are used to add features to the program that enable a faster, more efficient experimental experience. After the primary code (final.py) was completed, it was then arranged into a Jupyter Notebook (.ipynb) file for ease of demonstration.

Each of the three Maximum Subarray methods were tested using multiple synthetically generated datasets representing a range of input sizes. For the Divide-and-Conquer and Greedy (Kadane's algorithm) methods, one-dimensional arrays of floats, positive and negative, with sizes 1000, 2000, 5000, and 10000 were used to test how they reacted to inputs of different sizes. For the 2D technique, matrices of sizes 30x30 (900 elements), 50x50 (2500 elements), and 80x80 (6400 elements) were used. During the execution of the program, the execution time of each method was measured using Python's time library. Additionally, each method was run with these settings' multiple times (ten for the 1D methods, 5 for the 2D) so that the measured execution times can be averaged for the sake of a fair comparison. For each method, a brute-force test was used to ensure their correctness, with a simple and small dataset and a provided solution. Each method proved to be functional.

For reference, the machine that produced the subsequent results had the following specifications:

- CPU: AMD Ryzen 5 3600 (6-core, 12-thread, 3.6 GHz)
- RAM: 32GB DDR4
- GPU: NVIDIA RTX 3060

Experimental Results

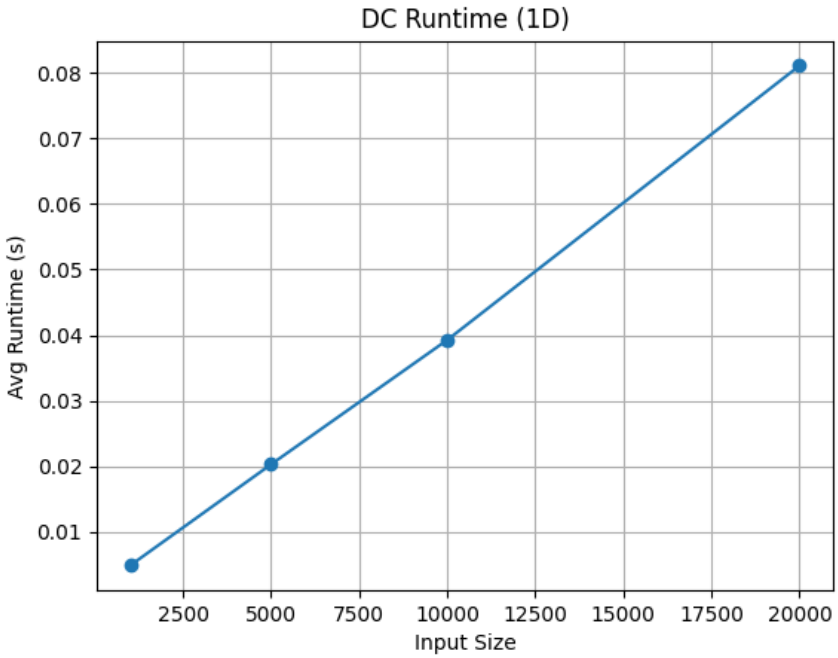

Figure 1. The results of the divide-and-conquer method. This method's θ(nlogn) order-of-growth is less apparent at this input size, closely resembling a linear one. However, at higher element counts (i.e. >200,000), the upward curve becomes more visible. This scale is not used in the experiment because it took multiple minutes to fully calculate on all team members' systems.

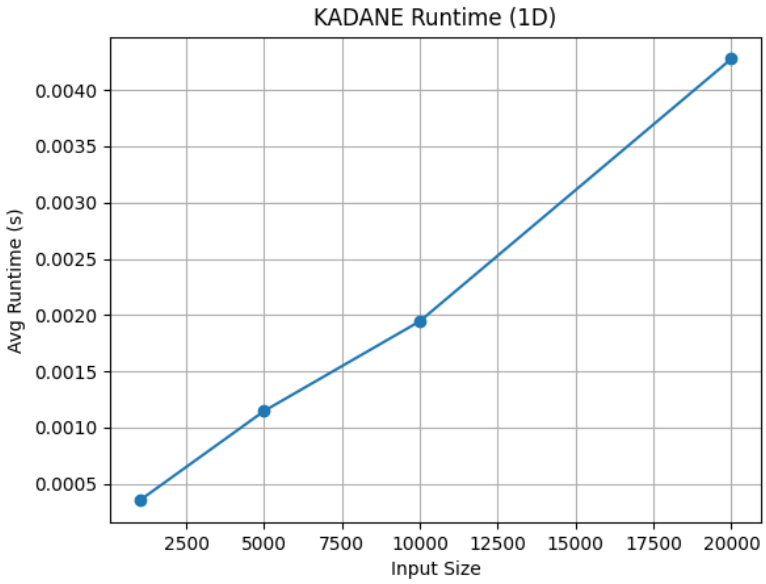

Figure 2. The results of the greedy (Kadane's algorithm) method.

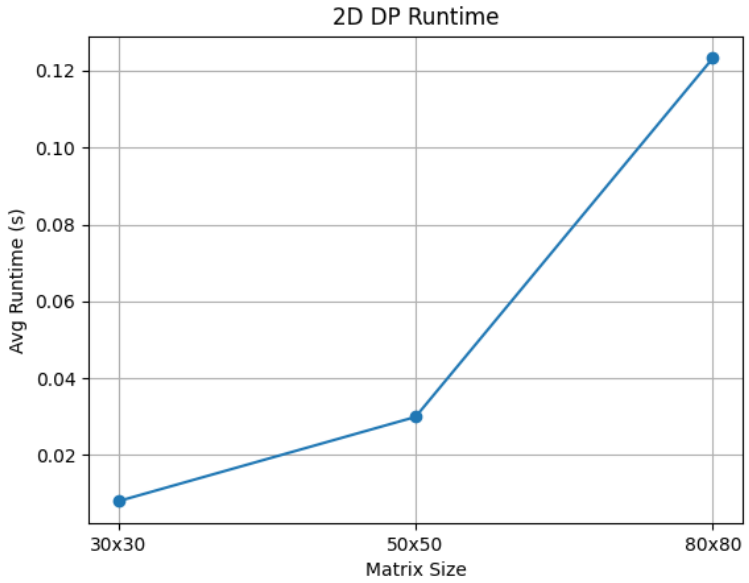

Figure 3. The results of the 2D Dynamic Programming method. This method's $\theta(n^3)$ order-of-growth is already apparent at this scale.

```
=== KADANE (1D) RESULTS ===
Size        Average Time (s)
---------------------------------
1000        0.000146
5000        0.000710
10000       0.001562
20000       0.003287

=== DC (1D) RESULTS ===
Size        Average Time (s)
---------------------------------
1000        0.005371
5000        0.026699
10000       0.053969
20000       0.158550

=== DP2D (2D DP) RESULTS ===
Matrix Size        Average Time (s)
-----------------------------------
30x30             0.010089
50x50             0.042974
80x80             0.153155

Process finished with exit code 0
```

Figure 4. The output of the final program in its entirety, produced by a different system than the preceding figures.

Analysis of Results and Reflection

Via empirical analysis, we can draw several conclusions from the results. The most immediate takeaway in the above graphs is that despite having a theoretical order of growth of θ(nlogn), the divide-and-conquer method produced a surprisingly similar slope to the greedy (Kadane's algorithm) method, which expresses the linear order of growth expected of it (θ(n)). However, this is because of the input sizes used. When tested at even higher counts, such as 200,000 elements and above, the increasingly steep curve of an θ(nlogn) became more visible. However, this was deemed impractical for the live demonstration of the code due to taking over a full minute to calculate on the team members' machines.

When considering the average time that each operation took to produce the maximum subarray, it becomes clear that all the methods behaved as expected when compared to one another. The greedy method is fastest due to having a linear (θ(n)) order of growth, with the divide-and-conquer method occupying second place with its slightly larger order (θ(nlogn)), and the least efficient being the 2D Dynamic Programming method with its cubic order of growth ($\theta(n^3)$). The greedy method was so efficient, in fact, that even at the full test capacity of 20,000 elements, it took less than one one-hundredth of a second to find the maximum subarray. The other methods were both over this amount of time for most of their input sizes.

The primary limitation of our experiments was the limited computing power of our personal systems. Were these constraints not in place, an even larger range of input sizes could have been tested in a timely manner to determine the extent of the methods' curves and average computation times. If the project could be done over, more time could have been devoted to the method algorithms themselves – specifically, making it so that for each calculation, the starting and ending indices of each maximum subarray would be measured and displayed as well as the elapsed time, although this would not have affected the presentation of the results in the final project.

Sources

The authors acknowledge the utilization of ChatGPT, a language model developed by OpenAI, in the preparation of this assignment. ChatGPT was employed in the following manner(s) within this assignment: grammatical correction and formatting of final.py and notebooks_Intro.ipynb.

All other sources consisted of in-class material, such as lecture slides and textbook excerpts:

Levitin, A. *Introduction to The Design and Analysis of Algorithms*. Pearson, 2012, 3rd Edition.

Table of Responsibilities

Team Member	Brainstorming	Coding	Experiment Design	Visualization/Results	Communication
Galloway, Patrick	33%	60%	60%	0%	10%
Goss, Lane	33%	0%	0%	0%	45%
Jordan, William	33%	40%	40%	100%	45%

Team Member	Submission	Report Writing	Presentation
Galloway, Patrick	100%	60%	35%
Goss, Lane	0%	20%	35%
Jordan, William	0%	20%	30%

Setbacks and Challenges

The primary challenge in this project, of course, was the coding itself. It was something of a learning process to figure out how to apply our knowledge of the various programming techniques to solve a real industry problem. The testing phase for it alone took multiple hours, involving some trial-and-error for formatting and a great deal of work on the synthetic dataset generation methods. In the earlier stages of the program, it was considered desirable for the maximum subarray methods to return the start and end indices of the subarray as well, although this idea was shelved when time constraints prevailed and it turned out that the location of the maximum subarray did not matter to the time averages so long as its value was correct. Additionally, procrastination proved to be a limiting factor in the project's overall quality, as much of it was developed and completed over the course of the final week of November.

GitHub Repository Link

https://github.com/ComputerConnor/group2-radian-001.git