Maximum Subarray and Variants – Biotech

William Jordan
Lane Goss
Patrick Galloway
Michael Cates

Project Overview:

Our team will explore the Maximum Subarray Problem along with its variants. The maximum subarray problem involves searching through an array of numbers for the contiguous subarray (a smaller group of elements in the main array) that has the largest sum of any subarray that can be made from the array. By studying different algorithms to solve this problem, we hope to understand their efficiency and practical performance.

Algorithms and Methods

We plan to explore three main algorithms:

1. Divide-and-conquer
    - This method recursively splits the array into smaller subarrays, finds the maximum subarray in each part, and combines the results.
    - Expected complexity: $O(n \log n)$
2. Greedy Algorithm
    - This method iterates through the array just once, keeping track of the maximum sum ending at each position.
    - Expected complexity: $O(n)$
3. 2D Expansion with Dynamic Programming
    - This extends the maximum subarray problem to 2D matrices.
    - Expected complexity: $O(n)^3$

Potential Methods of Experimentation and Analysis

To gain an enriched understanding of the Maximum Subarray Problem, both empirical and theoretical methods could be used:

1. Testing on Various Input Sizes
    - The primary input of a Maximum Subarray algorithm is an array of n numbers. Tests could be run using arrays of varying sizes to determine what effect, and to what extent this affects the algorithm's runtime.
2. Summation Analysis of Maximum Subarray Algorithms
    - To determine the theoretical efficiency and order of growth of a Maximum Subarray algorithm, its code, or a pseudocode representation of it, can be analyzed using summation to determine the expected number of operations that would occur during runtime for an array of size n.
3. Comparing Multiple Algorithm Methods

- There are multiple methods that can be used to find a maximum subarray. By running tests using each of them, with a standardized set of inputs acting as control values, we can compare their runtimes to find the most efficient algorithm among them.

Research Questions

The following questions will guide our research:

1. Which of the three algorithms appears to handle the maximum subarray problem most efficiently both theoretically and in practice?
2. Are there cases where a slower-looking algorithm performs competitively or better on certain inputs?
3. How do different input conditions (all negative numbers, all positive numbers, or mixed values) influence the performance of each algorithm, or do they at all?
4. Does the choice of algorithm significantly affect memory usage, or are all three roughly comparable in practice?
5. For the 2D version of the problem, is the DP-based method noticeably better than brute force, or do both struggle as input size grows?

Minimal / Low Complexity Experiment

A realistic experiment that we can perform is to implement all three algorithms (divide-and-conquer, greedy, and 2D expansion with dynamic programming) and run them on very small inputs with the goal of verifying that each algorithm produces the correct maximum subarray sum.

In doing this, we will measure the following:

- Running times on these small inputs, even if the differences aren't dramatic yet
- Memory footprint observations (Recursive calls vs. Iterative approach)

This experiment ensures the algorithms are implemented correctly and gives the team its first data points for comparison, without requiring large-scale computation.

Intermediate / Ambitious Experiment

A more ambitious experiment that we may perform, time permitting, is to empirically compare the three approaches against straightforward brute-force baselines to discover when and why one method may outperform another.

This would be useful for the following:

- To discover actual performances differences rather than assuming them
- Test sensitivity to input characteristics
- Verify algorithm correctness

- Connect theory with practice
- Build empirical research skills

Potential Roadblocks:

As a group, we may face some time constraints, considering that two members of our group work part-time in the evenings. However, if we manage our time effectively, we should have more than enough time to complete it. I do not believe there would be any lack of knowledge that would slow us down.

Our team has a strong combined skill set, with most members experienced in coding and mathematics. Connor excels at pseudocode, Patrick is skilled with summations, Lane is great at tracking loop outputs, and Michael is good at filling in any gaps that might occur error wise. Overall, there should be no concerns within the group that can’t be dealt with swiftly.