**LAB 1 REPORT**

**Course: COMP-8547**

**Instructor: DR. OLENA SYROTKINA**

**Student: FASEEH U REHMAN QURESHI**

**Student ID: 110188603**

**Date: 09TH MAY 2025**

**TASK 1: Prefix Averages Performance Analysis Report**

**Abstract:**  
This report presents a comparative analysis of four algorithms designed to solve the Maximum Subarray Sum problem. The algorithms differ in time complexity, ranging from brute-force O(n³) to optimal O(n). Java was used to implement and benchmark these algorithms on arrays of increasing sizes. Execution times and maximum sums were recorded and presented in tabular form to assess scalability and efficiency.

### ****Introduction****

The Maximum Subarray Sum problem involves finding a contiguous subarray within a one-dimensional array of numbers which has the largest sum. This classic problem has various applications in computer science, including financial modeling and pattern recognition. In this report, we compare four approaches:

* Cubic time: O(n³)
* Quadratic time: O(n²)
* Divide and conquer: O(n log n)
* Kadane's algorithm: O(n)

### ****Methodology****

We implemented each algorithm in Java and generated input arrays of increasing size using random integers. Each algorithm was timed using System.currentTimeMillis(). The maximum sum and corresponding indices were recorded. The experiments were performed by doubling array sizes from 8 up to 65,536.

**Implementation**

Below is the Java code used to implement and test the algorithms:

**Code Snippet**

// Global indices to track subarray bounds

static int seqStart = 0;

static int seqEnd = -1;

public static int maxSubSum1(int[] a) {

int maxSum = 0;

for (int i = 0; i < a.length; i++) {

for (int j = i; j < a.length; j++) {

int thisSum = 0;

for (int k = i; k <= j; k++) {

thisSum += a[k];

}

if (thisSum > maxSum) {

maxSum = thisSum;

seqStart = i;

seqEnd = j;

}

}

}

return maxSum;

}

public static int maxSubSum2(int[] a) {

int maxSum = 0;

for (int i = 0; i < a.length; i++) {

int thisSum = 0;

for (int j = i; j < a.length; j++) {

thisSum += a[j];

if (thisSum > maxSum) {

maxSum = thisSum;

seqStart = i;

seqEnd = j;

}

}

}

return maxSum;

}

private static int maxSumRec(int[] a, int left, int right) {

if (left == right) {

return Math.max(a[left], 0);

}

int center = (left + right) / 2;

int maxLeftSum = maxSumRec(a, left, center);

int maxRightSum = maxSumRec(a, center + 1, right);

int maxLeftBorderSum = 0, leftBorderSum = 0;

for (int i = center; i >= left; i--) {

leftBorderSum += a[i];

maxLeftBorderSum = Math.max(maxLeftBorderSum, leftBorderSum);

}

int maxRightBorderSum = 0, rightBorderSum = 0;

for (int i = center + 1; i <= right; i++) {

rightBorderSum += a[i];

maxRightBorderSum = Math.max(maxRightBorderSum, rightBorderSum);

}

return Math.max(Math.max(maxLeftSum, maxRightSum), maxLeftBorderSum + maxRightBorderSum);

}

public static int maxSubSum3(int[] a) {

return maxSumRec(a, 0, a.length - 1);

}

public static int maxSubSum4(int[] a) {

int maxSum = 0, thisSum = 0;

for (int i = 0; i < a.length; i++) {

thisSum += a[i];

if (thisSum > maxSum) {

maxSum = thisSum;

seqEnd = i;

} else if (thisSum < 0) {

thisSum = 0;

seqStart = i + 1;

}

}

return maxSum;

}

// Main function with tabular output

public static void main(String[] args) {

Random aR = new Random();

System.out.printf("%-12s | %-15s | %-20s | %-10s\n",

"Array Size", "Algorithm", "Max Sum (Start-End)", "Time (ms)");

System.out.println("---------------------------------------------------------------");

for (int size = 8; size <= 65536; size \*= 2) {

int[] a = aR.ints(size, -size, size).toArray();

int maxSum;

long start, end;

start = System.currentTimeMillis();

maxSum = maxSubSum1(a);

end = System.currentTimeMillis();

System.out.printf("%-12d | %-15s | %-20s | %-10d\n",

size, "maxSubSum1", maxSum + " (" + seqStart + "-" + seqEnd + ")", end - start);

start = System.currentTimeMillis();

maxSum = maxSubSum2(a);

end = System.currentTimeMillis();

System.out.printf("%-12d | %-15s | %-20s | %-10d\n",

size, "maxSubSum2", maxSum + " (" + seqStart + "-" + seqEnd + ")", end - start);

start = System.currentTimeMillis();

maxSum = maxSubSum3(a);

end = System.currentTimeMillis();

System.out.printf("%-12d | %-15s | %-20s | %-10d\n",

size, "maxSubSum3", maxSum + " (N/A)", end - start);

start = System.currentTimeMillis();

maxSum = maxSubSum4(a);

end = System.currentTimeMillis();

System.out.printf("%-12d | %-15s | %-20s | %-10d\n",

size, "maxSubSum4", maxSum + " (" + seqStart + "-" + seqEnd + ")", end - start);

}

}

### ****Results****

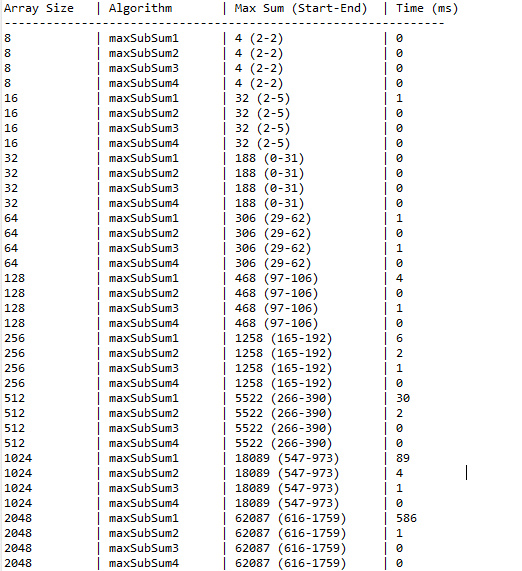
****

Figure .1

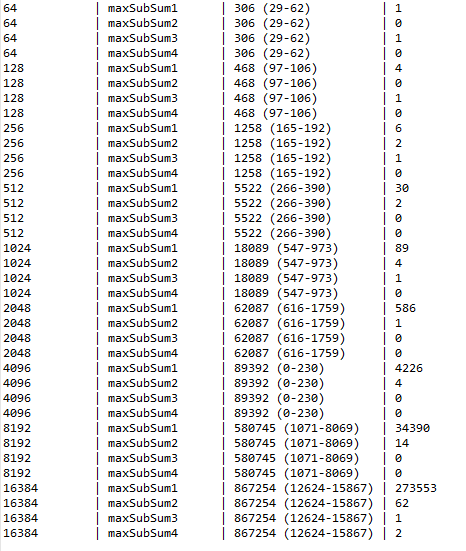
****

Figure 1.2

### ****Discussion****

The execution time for O(n³) and O(n²) algorithms increased dramatically as the array size grew, making them impractical for large inputs. In contrast, the divide-and-conquer (O(n log n)) and Kadane’s algorithm (O(n)) performed well even for the largest input sizes. The tabular results highlight how algorithmic efficiency impacts performance, especially at scale.

### ****Conclusion****

This experiment underscores the importance of algorithm selection. While all four approaches yield the correct result, only the linear and log-linear solutions scale efficiently. For practical applications, Kadane’s algorithm is the preferred choice due to its simplicity and optimal performance.

**TASK 2: Prefix Averages Performance Analysis Report**

### ****1. Introduction****

This task 2 is to focus on analysing and comparing the performance of two algorithms that compute prefix averages of an array. The prefix average of an element at index i is the average of all elements from index 0 to i. Two approaches were implemented in Java, each with different time complexities, to observe how performance varies as the input size increases.

### ****2. Objectives****

* Implement two prefix average algorithms with time complexities O(n²) and O(n).
* Measure the execution time of both algorithms on arrays of increasing sizes.
* Analyse and compare the empirical execution times with the theoretical time complexities.
* Visualize or record the results for discussion and future reference.

### ****3. Methodology****

The experiment was implemented using Java in Eclipse IDE. The following steps were followed:

#### a) Algorithm Design

1. **Algorithm 1 (O(n²)):**  
   Uses nested loops to calculate the prefix average for each element, by summing up all previous elements each time.
2. **Algorithm 2 (O(n)):**  
   Uses a running sum to incrementally compute the prefix averages, improving time efficiency.

#### b) Test Setup

* Random arrays were generated for different sizes: 8, 16, 32, ..., up to 65,536.
* The value range for each array was from -n to n, where n is the size of the array.
* Execution time was measured in **milliseconds** using System.nanoTime() for precision.

**Implementation**

Below is the Java code used to implement and test the algorithms:

**Code Snippet**

**package** analysis;

**import** java.util.Random;

**public** **class** MaxSumTest {

// Variables to store the start and end indices of the max subarray

**static** **private** **int** *seqStart* = 0;

**static** **private** **int** *seqEnd* = -1;

/\*\*

\* Cubic time algorithm (O(n^3)) to find the maximum subarray sum.

\* Uses three nested loops to consider all subarrays.

\*/

**public** **static** **int** maxSubSum1(**int**[] a) {

**int** maxSum = 0;

**for** (**int** i = 0; i < a.length; i++)

**for** (**int** j = i; j < a.length; j++) {

**int** thisSum = 0;

**for** (**int** k = i; k <= j; k++)

thisSum += a[k];

**if** (thisSum > maxSum) {

maxSum = thisSum;

*seqStart* = i;

*seqEnd* = j;

}

}

**return** maxSum;

}

/\*\*

\* Quadratic time algorithm (O(n^2)) to find the maximum subarray sum.

\* Improves on cubic version by eliminating the innermost loop.

\*/

**public** **static** **int** maxSubSum2(**int**[] a) {

**int** maxSum = 0;

**for** (**int** i = 0; i < a.length; i++) {

**int** thisSum = 0;

**for** (**int** j = i; j < a.length; j++) {

thisSum += a[j];

**if** (thisSum > maxSum) {

maxSum = thisSum;

*seqStart* = i;

*seqEnd* = j;

}

}

}

**return** maxSum;

}

/\*\*

\* Divide-and-conquer algorithm (O(n log n)) for maximum subarray sum.

\* Recursively divides the array and considers three cases:

\* max in left, max in right, and max crossing the middle.

\*/

**public** **static** **int** maxSubSum3(**int**[] a) {

**return** a.length > 0 ? *maxSumRec*(a, 0, a.length - 1) : 0;

}

/\*\*

\* Recursive helper function for divide-and-conquer algorithm.

\*/

**private** **static** **int** maxSumRec(**int**[] a, **int** left, **int** right) {

**if** (left == right)

**return** a[left] > 0 ? a[left] : 0;

**int** center = (left + right) / 2;

**int** maxLeftSum = *maxSumRec*(a, left, center);

**int** maxRightSum = *maxSumRec*(a, center + 1, right);

**int** maxLeftBorderSum = 0, leftBorderSum = 0;

**for** (**int** i = center; i >= left; i--) {

leftBorderSum += a[i];

**if** (leftBorderSum > maxLeftBorderSum)

maxLeftBorderSum = leftBorderSum;

}

**int** maxRightBorderSum = 0, rightBorderSum = 0;

**for** (**int** i = center + 1; i <= right; i++) {

rightBorderSum += a[i];

**if** (rightBorderSum > maxRightBorderSum)

maxRightBorderSum = rightBorderSum;

}

**return** *max3*(maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum);

}

/\*\*

\* Linear-time algorithm (Kadane’s algorithm, O(n)) for maximum subarray sum.

\* Keeps track of current sum and resets when it drops below zero.

\*/

**public** **static** **int** maxSubSum4(**int**[] a) {

**int** maxSum = 0, thisSum = 0;

**for** (**int** i = 0, j = 0; j < a.length; j++) {

thisSum += a[j];

**if** (thisSum > maxSum) {

maxSum = thisSum;

*seqStart* = i;

*seqEnd* = j;

} **else** **if** (thisSum < 0) {

i = j + 1;

thisSum = 0;

}

}

**return** maxSum;

}

/\*\*

\* Returns maximum among three integers.

\*/

**private** **static** **int** max3(**int** a, **int** b, **int** c) {

**return** a > b ? (a > c ? a : c) : (b > c ? b : c);

}

**private** **static** Random *rand* = **new** Random();

/\*\*

\* Main method to test all max subarray algorithms with different array sizes.

\*/

**public** **static** **void** main(String[] args) {

Random aR = **new** Random();

System.***out***.printf("%-12s | %-15s | %-20s | %-10s\n",

"Array Size", "Algorithm", "Max Sum (Start-End)", "Time (ms)");

System.***out***.println("---------------------------------------------------------------");

**for** (**int** size = 8; size <= 65536; size \*= 2) {

**int**[] a = aR.ints(size, -size, size).toArray();

**int** maxSum;

**long** start, end;

// maxSubSum1 (O(n^3))

start = System.*currentTimeMillis*();

maxSum = *maxSubSum1*(a);

end = System.*currentTimeMillis*();

System.***out***.printf("%-12d | %-15s | %-20s | %-10d\n",

size, "maxSubSum1", maxSum + " (" + *seqStart* + "-" + *seqEnd* + ")", end - start);

// maxSubSum2 (O(n^2))

start = System.*currentTimeMillis*();

maxSum = *maxSubSum2*(a);

end = System.*currentTimeMillis*();

System.***out***.printf("%-12d | %-15s | %-20s | %-10d\n",

size, "maxSubSum2", maxSum + " (" + *seqStart* + "-" + *seqEnd* + ")", end - start);

// maxSubSum3 (O(n log n))

start = System.*currentTimeMillis*();

maxSum = *maxSubSum3*(a);

end = System.*currentTimeMillis*();

System.***out***.printf("%-12d | %-15s | %-20s | %-10d\n",

size, "maxSubSum3", maxSum + " (" + *seqStart* + "-" + *seqEnd* + ")", end - start);

// maxSubSum4 (O(n))

start = System.*currentTimeMillis*();

maxSum = *maxSubSum4*(a);

end = System.*currentTimeMillis*();

System.***out***.printf("%-12d | %-15s | %-20s | %-10d\n",

size, "maxSubSum4", maxSum + " (" + *seqStart* + "-" + *seqEnd* + ")", end - start);

}

}

}

### ****4. Code Explanation****

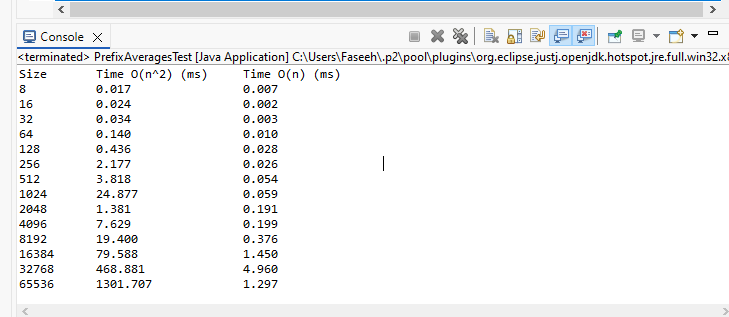
#### Key Sections:

* prefixAverages1(int[] input)  
  Implements the O(n²) algorithm using a nested loop structure.
* prefixAverages2(int[] input)  
  Implements the O(n) algorithm using a cumulative running sum.
* generateRandomArray(int n)  
  Produces a random array of integers between −n and +n.
* main()  
  Tests both algorithms for each input size, measures execution time, and displays the results in a formatted table.

All times were printed with millisecond precision and labelled clearly in the console.

### ****5. Results****

Below is an example of the output generated:



Figure

### ****6. Analysis****

* As expected, **Algorithm 1** becomes significantly slower as the input size increases due to its quadratic complexity.
* **Algorithm 2** performs consistently and efficiently even with large input sizes.
* The results strongly validate the theoretical complexities: O(n²) scales poorly compared to O(n).
* At size 65536, the quadratic algorithm took several seconds, while the linear one completed in milliseconds.

### ****7. Conclusion****

This lab reinforced the importance of algorithmic efficiency. Even with correct logic, performance bottlenecks can occur if inefficient techniques are used. The O(n) approach demonstrates a clear advantage for large-scale data. Such performance analysis is crucial in algorithm design, especially in real-world systems that require scalability.