**CMPSC122 Lab Section 001**

Lab 12

Algorithm Analysis

Prepared By

Armin Rejaie

**Instructor: Jonathan Liaw**

**Date: 04/15/2025**

**Table of Contents**

[1. Objective 3](#_Toc196665446)

[2. Introduction 3](#_Toc196665447)

[3. Procedure 3](#_Toc196665448)

[4. Discussion 4](#_Toc196665449)

[5. Conclusion 9](#_Toc196665450)

[6. Appendix 10](#_Toc196665451)

# 1. Objective

This project aims to examine three different algorithms to determine their maximum contiguous subsequent sums and discover the algorithms’ efficiencies. The professor has provided the following algorithms:

* Algorithm #1: Brute Force (the Blue function)
* Algorithm #2: Divide and Conquer (the Green function)
* Algorithm #3: Linear scan, dynamic programming (the Red function)

Since only the functions are being analyzed, no improvements or changes to the functions will be discussed. Both theoretical and actual results will be examined, proving their effectiveness in practice.

# 2. Introduction

To elaborate on what the maximum contiguous subsequence sum problem is, it is where a given sequence of numbers such as A1, A2, … An (containing integers of positive or negative values) contain a subsequence Aj, …, Ak that the sum of its elements is as large as possible among all possible contiguous subsequences. In simpler terms, the subsequence and sum can be found by taking a portion of the sequence without leaving any gaps. For example, given an array: [-2, 11, -4, 13, -5, 2], the maximum contiguous subsequence is [11, -4, 13] with a sum of 20. Likewise, given an array: [3, -2, 5, -1, 6, -3], the maximum contiguous subsequence is [3, -2, 5, -1, 6] with a sum of 11. Yet, there is a special case when a sequence only contains negative integers, leaving a sum of 0. If given an array: [-5, -1, -8, -3], all the possible contiguous subsequences have negative sums, defining a maximum sum of 0. This theory will contribute to the runtime of the algorithms, as the functions with better efficiency will have smaller maximum contiguous subsequent sums and expand more slowly over time.

# 3. Procedure

The steps that will need to be taken for this project are as follows. First, analyze each of the three algorithms and theoretically determine their efficiency by identifying their Big-O-notation in its source code and their maximum contiguous subsequence sums. Then, the algorithms will be tested and run in a C++ compiler which will evaluate the runtime of each function. The same loop sizes, which is 10, will be used for the functions to fairly compare the results between the different algorithms. Finally, both the theoretical and actual results of the experiment will be compared with each other and give statistics to analyze which algorithms are most efficient in practice. This experiment will be conducted on a personal computer with an operating system of 26100.3775, a processor of 12th Gen Intel(R) Core(TM) i7-12650H 2.30 GHz and installed RAM (memory) of 16.0 GB (15.6 GB usable).

# 4. Discussion

Analysis of Algorithm #1: Brute Force (The Blue function)

int MaxSublistSum\_Blue(int array[], int size) {

int currentMax = 0; // 1

for (int i = 0; i < size; i++) { // 1+n+(n-1) = 2n

int curSum = 0; // (n-1)

for (int j = 0; j < size - i; j++) { // ½(n^2-n)

curSum += array[i + j]; // (n-1)

if (curSum > currentMax) { // (n-1)

currentMax = curSum; // (n-1)

}

}

}

return currentMax; // 1

}

T(n) = 1 + 2n + (n-1) + ½(n^2-n) + (n-1) + (n-1) + (n-1) + 1

Simplified: 11/2n + 1/2n^2 – 1  
Removing the constants: n + n^2  
T(n) = **O(n^2)**

Analysis of Algorithm #2: Divide and Conquer (The Green function)

int MaxSublistSum\_Green(int array[], int size) {

return MaxSublistSum\_DnC\_Rec(array, 0, size - 1);

}

int MaxSublistSum\_DnC\_Rec(int array[], int start, int end) {

if (start > end) {

return 0;

}

else if (start == end) {

return max(0, array[start]);

}

else {

int mid = (start + end) / 2;

int leftMax = MaxSublistSum\_DnC\_Rec(array, start, mid);

int rightMax = MaxSublistSum\_DnC\_Rec(array, mid + 1, end);

int inBetweenMax = 0;

int leftCurSum = 0;

for (int i = mid; i >= start; i--) {

leftCurSum += array[i];

if (leftCurSum > inBetweenMax) {

inBetweenMax = leftCurSum;

}

}

int rightCurSum = inBetweenMax;

for (int i = mid + 1; i <= end; i++) {

rightCurSum += array[i];

if (rightCurSum > inBetweenMax) {

inBetweenMax = rightCurSum;

}

}

return max(max(leftMax, rightMax), inBetweenMax);

}

}

T(n) = **O(nlogn)**

The derivation of the Big-O notation for this algorithm makes sense because due to the recursion, it splits the array into two-halves every time there is a recursion call, creating a sum of up to O(n) for each level of recursion.

Analysis of Algorithm #3: Linear scan, dynamic programming (The Red function)

int MaxSublistSum\_Red(int array[], int size) {

int\* maxSumEndingAt = new int[size]; // 1

maxSumEndingAt[0] = array[0]; // 1

int currentMax = max(0, maxSumEndingAt[0]); // 1

for (int i = 1; i < size; i++) { // 1+n+(n-1)= 2n

if (maxSumEndingAt[i - 1] <= 0) { // (n-1)

maxSumEndingAt[i] = array[i]; // (n-1)

}

else { // (n-1)

maxSumEndingAt[i] = maxSumEndingAt[i - 1] + array[i]; //(n-1)

}

if (maxSumEndingAt[i] > currentMax) { // (n-1)

currentMax = maxSumEndingAt[i]; // (n-1)

}

}

delete[] maxSumEndingAt; // 1

return currentMax; // 1

}  
  
T(n) = 1 + 1 + 1 + 2n + (n-1) + (n-1) + (n-1) + (n-1) + (n-1) + (n-1) + 1 + 1

Simplified: 8n – 1

Removing the constants: n – 1

T(n) = **O(n)**

Experimental Results of Algorithm #1: Brute Force (The Blue function)

|  |  |
| --- | --- |
| Array Size | Max Sum Sequence |
| 500 | 577 |
| 1000 | 577 |
| 2000 | 1209 |
| 4000 | 1209 |
| 8000 | 1752 |
| 16000 | 3869 |
| 32000 | 5954 |
| 64000 | 4128 |
| 128000 | 17383 |
| 256000 | 9626 |

Experimental Results of Algorithm #2: Divide and Conquer (The Green function)

|  |  |
| --- | --- |
| Array Size | Max Sum Sequence |
| 500 | 1106 |
| 1000 | 1493 |
| 2000 | 1493 |
| 4000 | 1921 |
| 8000 | 2463 |
| 16000 | 5012 |
| 32000 | 8597 |
| 64000 | 9000 |
| 128000 | 9000 |
| 256000 | 10469 |

Experimental Results of Algorithm #3: Linear scan, dynamic programming (The Red function)

|  |  |
| --- | --- |
| Array Size | Max Sum Sequence |
| 500 | 1099 |
| 1000 | 1601 |
| 2000 | 1601 |
| 4000 | 1601 |
| 8000 | 1771 |
| 16000 | 5417 |
| 32000 | 7654 |
| 64000 | 7654 |
| 128000 | 7654 |
| 256000 | 7654 |

# 5. Conclusion

In all, after theoretically analyzing and then compiling the code of the three algorithms, the results match well. In the first function, the Blue function, the Big-O notation was analyzed to be O(n^2), an algorithm that expands quickly in a short amount of time. Hence, it is why compared to the other two functions, it has significantly longer run times to determine the maximum contiguous subsequence sums at higher array sizes. On the other hand, the Red and Green functions both ran exponentially faster and expanded more slowly. Despite this, it can be determined that the Red function has the upper hand because of its smaller expansion speed. The Big-O notation for the Red function is O(n), compared to the Big-O notation of the Green function, which is O(nlogn), a faster expanding function. Additionally, the maximum contiguous subsequence sums of each algorithm can be analyzed, which shows smaller results in the Red function compared to the others. In all, the ranking of most efficient algorithms from best to worst would be: Red -> Green -> Blue. In this experiment, there were not many errors to be observed as all the code compiled and ran as expected.

# 6. Appendix

One note that can be taken from this experiment is that the run time for algorithms can depend on computer to computer. This was all conducted on a gaming laptop, which may explain the extremely fast runtime for the Red and Green functions. Despite this, the Blue function was still time-consuming to achieve its data. For any future experiments, especially with computers of the same specifications, it is important not to choose a number too high to loop as the information can take a long time to compile.