Analysis of Algorithms That Solve the Maximum Sub-array Problem

Abstract— **This document is a summary of the comparison of three algorithms that solve the maximum sub-array problem, namely; the Brute force, divide and conquer method and Kadane's algorithm which were implemented in a C++ program to output the time taken by each algorithm to solve the problem, and to observe and analyze the theoretical and empirical time complexities of each algorithm.**

Key Words: C++, algorithm, sub-array, theoretical, empirical, complexity.

I. INTRODUCTION

Programming follows the sole practice of analysing a problem and creating the simplest and most efficient solution to that problem. An example of this is in the implementation of different sorting algorithms to solve the maximum sub-array of a size defined array. The efficiency of the algorithms can then be assessed and quantified using time complexities, denoted by $O(\textit{time factor})$[1]. These time complexities fall into a theoretical and empirical range, where theoretical accounts for the worst case scenario of the algorithm, and empirical is the realistic time taken by the algorithm.

II. HARDWARE AND SOFTWARE SPECIFICATIONS

The program was run on a computer running Ubuntu 64-bit 14.04 LTS, featuring an Intel i3 dual core processor with a clock speed of 2.2 GHz, 4GB of DDR3-1333MHz RAM and a 5400 RPM SATA3 hard drive. The program was compiled with the use of the GNU GCC compiler, and the output of the program was plotted using GNUplot.

The initial test system featured an Intel i7 quad core overclocked to 4.8GHz, 16GB of DDR3-2133MHz, and 256GB solid state drive. It was noticed that the system seemed to process the program faster than the implemented *clock()* function could tick. Thus a large lack of precision was obtained, and a slower test system had to be used. The implementation of the experimental *high resolution clock* was considered, but it did not provide consistent results.

III. REVIEW OF ALGORITHMS

A. Brute force algorithm

This algorithm, by definition, is one in which every element in a set of values/literals is compared to every other element. In this portion of the assignment, we implemented the search for the largest sub-array by using two variables to iterate simultaneously and monitor the position of the values being analyzed. The maximum sub-array was initialized to the first value of the array before any comparisons were done to ensure that no errors were 'thrown' in the case of the array only containing one value whilst comparisons were done to an index out of bounds[2].

A temporary summation of the sub-array was monitored using the indices of the iterating variables. This summation value was then compared to the maximum sum already stored wherein the value of the maximum sum was set to the temporary sum, if temporary sum was bigger. The 'while loop' that was implemented to make comparisons of the temporary sum to the maximum sum was terminated using a decrementing counter of 1 as the array was traversed in reverse. For a set of N elements there are $(N+(N-1)+(N-2)+(N-3)+(N-4)+.....+(N-N))$ possible subsets, including the Null/empty set $\{\}$. The general mathematical expression can be evaluated to $Sm = N^2 - 2N-$(other polynomials of order < 2 which depend on N). Thus we can conclude that the runtime of this algorithm can be averaged at $O(N^2)$.

B. Divide and conquer algorithm

This algorithm uses recursion to find the maximum sub-array in an array containing either both negative and positive numbers or just numbers that are all negative or positive. The base case for this recursion is an array containing only one element. This element is returned as the maximum sub-array. In an array that contains only positive numbers, the maximum sub-array will be the sum of all the numbers in the array[3].

This algorithm divides the array into two and finds the maximum sub-array in each array then compares the two maximum sub-arrays. It might happen though that the maximum sub-array is between both the right half of the array and the left half of the array. This is the maximum crossing sub-array in the right half of the midpoint (also through maximum sub-array function) and the maximum crossing sum across all elements of the array (using maximum crossing function). The maximum of the three is returned via the second utility function. The maximum sub-array found when comparing the left half and the right half is then compared to the maximum crossing array. The result of this is returned as the maximum sub-array. The theoretical time complexity is approximated to be or the order $O(nlogn)$.

C. Kadane's Algorithm

This algorithm is implemented such that it finds the maximum continuous subsequence in a one dimensional sequence. The code is basic. The array of n integers from $\{-50, 50\}$ is stored. The user is allowed to define the size of the array. The array is populated randomly. A function called *Kadane's Algorithm* is called. The original array is passed to this function and the algorithm finds the highest sum of elements in the array by keeping a running total of the highest value and comparing it to the previous highest value. It's theoretical complexity is $O(n)$. The singular disadvantage of using this algorithm is that it is unable to handle an array of only negative numbers. At least one value in the array must be positive[4].

IV. RESULTS

The output of the program produced the time taken by each algorithm to sort through each array. The results immediately conveyed a relationship that was expected from the theoretical complexities. These results were then plotted against a log-log graph using *gnuplot*, which allowed for the empirical time complexities to be compared. Initial plotting gave rise to issues with regard to inconsistent spikes in the plot. This was accommodated by utilising the *smooth bezier* function to allow for a "line of best fit" to produce a better correlation of results as seen below[Fig.1][5].

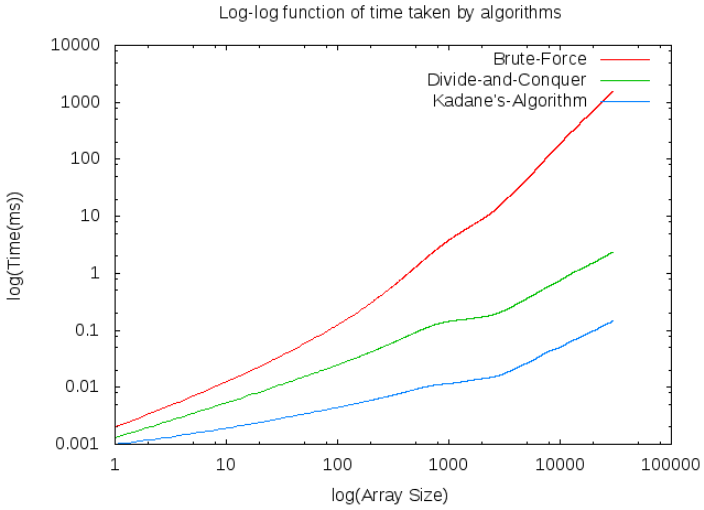

Fig. 1. Comparison of empirical time complexities of each algorithm.

From the figure above, there is an immediate trend in empirical time complexity of each algorithm. *Brute force* appears to be linear for small array sizes but very quickly becomes an exponential function of time. Both *divide and conquer* as well as *Kadane's* show a linear relationship, however the latter of the two clearly shows much better efficiency as the size of the array increases.

V. CONCLUSION

Through analysis of both theoretical as well as empirical data. It is clearly obvious that Kadane's algorithm offers the most time efficient and least performance intensive sorting method when compared to the likes of the brute force, and divide and conquer alternatives. For small array sizes, personal preference of either of the three algorithms may take place, but real world scenarios such as cryptography and super computing would deal with massive amounts of data, and thus *Kadane's algorithm* would be the optimal solution in those applications.

REFERENCES

[1] Adamchik, V. 2009. *Algorithmic Complexity*, online, available at: http://www.cs.cmu.edu/ãdamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html. [Accessed 24 August 15].

[2] Stoimen. 2012. *Computer Algorithms: Brute Force String Matching*, online, available at: http://www.stoimen.com/blog/2012/03/27/computer-algorithms-brute-force-string-matching/. [Accessed 23 August 15].

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, 2009.

[4] Bentley, J. 1984, *Programming pearls: algorithm design techniques, Communications of the ACM 27*. 9th Revision, pgs 865?873.

[5] Kawano. 2004. *About 2-Dimensional Plot*, online, available at: http://lowrank.net/gnuplot/plot2-e.html. [Accessed 24 August 15].