Algorithm Efficiency: A Review

Prashant Agrawal
Associate Professor
KIET Group of Institutions, Ghaziabad
prashant.agarwal@gmail.com

Naresh Chandra
Assistant Professor
KIET Group of Institutions, Ghaziabad
naresh.chandra@gmail.com

***Abstract*- As establishing the amount of computing resources needed to execute the algorithm. The Resources needed are: memory space and running time. The paper basically focuses on analyzing the efficiency of an algorithm (or the complexity of an algorithm). Such an analysis is useful to establish if a given algorithm uses a reasonable amount of resources to solve a problem. If the algorithm needs too many resources then, even if it is correct, it cannot be used in practice. Such an algorithm is not an efficient one**
Although, this objective of this paper is to emphasize that the analysis concerns itself primarily with significant differences in efficiency – differences that you can usually obtain only through superior methods of solution and rarely through clever tricks in coding.

***Keywords*- Computing resources, sorting, algorithm analysis, comparison.**

I. INTRODUCTION

Usually, the amount of computing resources depends on the dimension of input data and this dimension can be expressed as the number of bits used to codify the input data.
If the algorithm analyzes the properties of a natural number, the input's size can be considered for example, if number = n then input size = $[\log_2(n)] + 1$. The most important fact which may relate to the algorithm's efficiency is in terms of some problem statements [1]:

1. How are the algorithms coded? Does one algorithm run faster than another because of better programming? We should not compare implementations rather than the algorithms. Implementations are sensitive to factors such as programming style that cloud the issue.

2. What computer should you use? The only fair way would be to use the same computer for both programs. But even then, the particular operations that one algorithm uses may be faster or slower than the other – and may be just the reverse on a different computer. In short, we should compare the efficiency of the algorithms independent of a particular computer.

3. What data should the programs use? There is always a danger that we will select instances of the problem for which one of the algorithms runs uncharacteristically fast. For example, when comparing a sequential search and a binary search of a sorted array. If the test case happens to be that we are searching for an item that happens to be the smallest in the array, the sequential search will find the item more quickly than the binary search.

In Section II, we are taking an example of calculating the time it takes to execute a piece of code. This example derives an algorithm's time requirements as a function of problem size. The way to measure a problem's size depends on the application.

In Section III, we will discuss how about the system specifications of current computer affect the algorithm efficiency. Does it make a difference to run the program when we change the system specifications.

In Section IV, we will emphasize on the choice of data influenced by the operations involved in the algorithm. In simple terms,
Data Interface + Algorithms = Efficient Programs

II. ALGORITHM CODED

In order to code an algorithm we need to remember that there is an important difference between a program and the underlying algorithm that the program is representing. To explore this difference, lets take an example:

```
for(i=1; i<=N; i++)
        for(j=1; j<=i; ++j)
                for(k=0; k<5; ++k)
                        TASK T;
```

If task T requires t time units, the innermost loop on K requires 5*t time units. We have to calculate the total time to execute this code, which in turn 5*t*N*(N+1)/2 time units. This algorithm basically derives that the problem size depends on the application.

Algorithm analysis is concerned with comparing algorithms based upon the amount of computing resources that each algorithm uses. Through this derivation, we will be able to say that which algorithm would be better than the others because it is more efficient in its use of those resources or perhaps because it simply uses fewer [12].

III. SYSTEM SPECIFICATION

An increase in hardware performance, the running time of the program will affect the algorithm. Meaning if you have computer A which is overall 2 times slower than computer B. Than your algorithm will be twice as fast on computer B. Twice as fast though really makes hardly no difference when you consider big input values to an algorithm though [10].

System specification is very important, and selection of algorithms requires knowledge of the system specification and the size of the input data [11].

Algorithms are usually described using the Big O Notation. This notation describes the asymptotic behavior of an algorithm; it describes the behavior when the input data is very very large. So for example, we have two algorithms for sorting.

Algo A with O(n)
Algo B with $O(n^2)$
And let's take two PCs:

PC1
PC2 100 times faster than PC1
And we have two setups:

PC1 running Algo A
PC2 running Algo B

When n is very very large (like billions) PC2 will beat PC1. (see FIGURE 1)

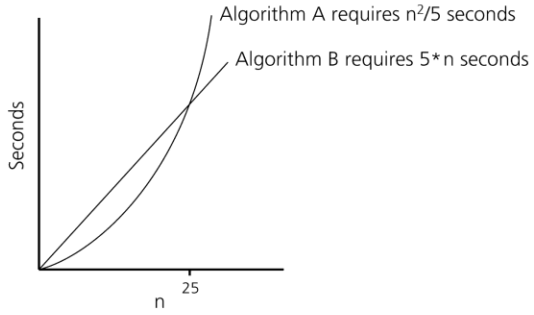

FIGURE 1: Executing of Algorithm A & B

IV. DATA IMPACT ON ALGORITHM

As the efficiency is influenced by the manner of representing the data, in solving a problem, it is necessary to choose an abstraction of reality, i.e., to define a set of data that is to represent the real situation. This choice must be guided by the problem to be solved which then follows a choice of representation of this information [2]. The complexity of an algorithm is describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. To arrange the data according to their values, one of the most common applications in computer science is sorting. Based on the data size, sorts are generally classified as either external sorting or internal sorting [3][6]. An internal sort is the sort in which all the data are held in the primary memory during the sorting process. An external sort uses primary memory for the data currently being sorted and secondary storage for any data that will not fit in the primary memory. Based on the information about the data, sort can be comparison and non-comparison-based sorting. Comparison based algorithm orders a sorting array by weighing the value of one element against the value of other elements. [9] Algorithms such as quick sort, merge sort, heap sort, bubble sort, and insertion sort are comparison based. Whereas, Non-comparison based algorithm sorts an array without consideration of pair wise data elements. Bucket sort, radix sort are example of non comparison based. Efficient sorting is important to optimizing the use of other algorithms that require sorted lists to work correctly; it is also often useful for producing human-readable output [11].

All sorting algorithms are nearly problem specific. How one can predict a suitable sorting algorithm for a particular problem? What makes good sorting algorithms? Speed is probably the top consideration, but other factors of interest include versatility in handling various data types, consistency of performance, memory requirements, length and complexity of code, and stability factor (preserving the original order of records that have equal keys) [8]. For example, sorting a database which is so big that cannot fit into memory all at once is quite different from sorting an array of 100 integers. Not only will the implementation of the algorithm be quite different, naturally, but it may even be that the same algorithm which is fast in one case is slow in the other. Also sorting an array may be different from sorting a linked list [4][5].

In order to judge suitability of a sorting algorithm to a particular problem we need to see the following statements:

1. Are the data that application needs to sort tending to have some pre existing order

2. What are properties of data being sorted
3. Do we need a stable sort

NOTE: Generally the more we know about the properties of data to be sorted, the faster we can sort them.

For some strange reason, sorting the data miraculously makes the code almost six times faster.

Let us consider the following C++ code:

```
#include <algorithm>
#include <ctime>
#include <iostream>

int main()
{
   // Generate data
   const unsigned arraySize = 32768;
   int data[arraySize];

   for (unsigned c = 0; c < arraySize; ++c)
      data[c] = std::rand() % 256;

   // !!! With this, the next loop runs faster
   std::sort(data, data + arraySize);

   // Test
   clock_t start = clock();
   long long sum = 0;

   for (unsigned i = 0; i < 100000; ++i)
   {
      // Primary loop
      for (unsigned c = 0; c < arraySize; ++c)
      {
         if (data[c] >= 128)
            sum += data[c];
      }
   }

   double elapsedTime = static_cast<double>(clock() -
start) / CLOCKS_PER_SEC;

   std::cout << elapsedTime << std::endl;
   std::cout << "sum = " << sum << std::endl;
}
```

Here in the above example without std::sort(data, data + arraySize), the code runs in 11.54 seconds. But with the sorted data, the code runs in 1.93 seconds [7].

Again, if tried in JAVA code then,

```
import java.util.Arrays;
import java.util.Random;

public class Main
{
   public static void main(String[] args)
   {
      // Generate data
      int arraySize = 32768;
      int data[] = new int[arraySize];

      Random rnd = new Random(0);
      for (int c = 0; c < arraySize; ++c)
         data[c] = rnd.nextInt() % 256;

      // !!! With this, the next loop runs faster
      Arrays.sort(data);

      // Test
      long start = System.nanoTime();
      long sum = 0;

      for (int i = 0; i < 100000; ++i)
      {
         // Primary loop
         for (int c = 0; c < arraySize; ++c)
         {
            if (data[c] >= 128)
               sum += data[c];
         }
      }

      System.out.println((System.nanoTime() - start) /
1000000000.0);
      System.out.println("sum = " + sum);
   }
}
```

Again the results are same.

V. CONCLUSION

To explore the efficiency, the analyze and comparison of the algorithms based on the specification of data, to programmed the code will track the actual execution of algorithm. This paper reviews the alternative differences which may be the benchmark on the analysis of algorithm. Although this paper may help Computer Scientist to determine the best use of computing resources for a particular problem to improve the efficiency.

REFERENCES

[1] Harry R. Louis, "The efficiency of Algorithms" IEEE Algo-data, 26(2): 40-49, 2003
[2] Wirth, N., 1976. "Algorithms + Data Structures = Programs": Prentice-Hall, Inc. Englewood Cliffs, N.J.K. Mehlhorn. Sorting and Searching. Springer Verlag,

Berlin, 1984.
[3] Knuth, D."The Art of Computer programming Sorting and Searching", 2nd edition, vol.3. Addison- Wesley, 1998.
[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein."Introduction to Algorithms". MIT Press, Cambridge, MA, 2nd edition, 2001
[5] Aho, A.V., Hopcroft, J.E., Ullman, J. D., 1974."The Design and Analysis of Computer Algorithms" Addison-Wesley.
[6] A. Aho, J. Hopcroft, J. Ullman, "Data Structures and Algorithms", Pearson India reprint, 2000
[7] Geoffrey I. Webb, "OPUS: An Efficient Admissible Algorithm for Unordered Search", Journal of Artificial Intelligence Research 3 (1995) 431-465
[8] Pearl, J. (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley, Reading, Mass
[9] Schlimmer, J. C. (1993). Efficiently inducing determinations: A complete and systematic search algorithm that uses optimal pruning. In Proceedings of the 1993 International Conference on Machine Learning, pp. 284–290 San Mateo, Ca. Morgan Kaufmann.
[10] Chung-Yang (Ric) Huang "Fundamentals of algorithms" University of California, Santa Barbara, California
[11] Alfred Strohmeier "Algorithms and Data Structures" Swiss Federal Institute of Technology in Lausanne, Software Engineering Laboratory
[12] Michael McMilan "Data Structures and Algorithms using C#", Cambridge University Press.