

An Analysis of Finding the Order of Time Complexities

S. Ismail Mohideen^{1*} and B. Rajesh²

¹*Associate Professor, PG and Research Department of Mathematics, Jamal Mohamed College Tiruchirappalli- 620 020. India. Email: simohideen@yahoo.co.in*

²*Department of Mathematics, Anna University Tiruchirappalli - Pattukkottai Campus, Pattukkottai- 614 701. India. Email: rajesh_rab@yahoo.co.in*

**Corresponding author*

Abstract

Analysis of an algorithm is to determine the amount of resources such as time and storage necessary to execute it. The efficiency or complexity of an algorithm is based on the function relating the input length to the number of steps required to execute the algorithm. In this paper, the computational complexities of different algorithms are analyzed.

Keywords: Time Complexity, Algorithms, Order, Big O, Spanning tree.

1. Introduction

An algorithm is a list of instructions that solves every instance of a problem in a finite number of steps. Analyzing an algorithm determines the amount of “time” that the algorithm takes to execute on a particular input size. Here, the amount of time means an approximation of the number of operations that an algorithm performs. For example, if the input to an algorithm is a graph, the number of nodes and edges in the graph can describe the input size.

The key idea to measure time and space as a function of the length of the input came in the early 1960's by Hartmanis and Stearns [1] and thus computational complexity research started.

In the analysis of algorithms, it is not important to know exactly how many operations an algorithm does. The general behavior, that is, the over all growth rate of the algorithm is an important factor. The growth rate of the algorithm is the rate of increase in operations for an algorithm as the size of the problem increases. As the rate of growth of an algorithm is determined by the largest term in an equation, the terms that grow more slowly are discarded. There are three categories: 1. Big Omega

2. Big Theta and 3. Big Oh to determine the computational complexity [2]. The computational complexities of all the algorithms discussed in this paper are determined, using Big Oh notation.

2. Terminology

Big O:

The function $f(n) = O(g(n))$ if and only if there exist a positive constant c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

3. Growth rate of the algorithm

The growth rate of the algorithm is the rate of increase in operations for an algorithm as the size of the problem increases. To discuss about the growth rates, we use Big O. Definition of Big O states only that $g(n)$ is an upper bound on value of $f(n)$, for all $n, n \leq n_0$.

Consider the below functions.

$3n + 2 = O(n)$ as $3n + 2 \leq 4n$ for all $n \geq 2$

$3n + 2 \neq O(1)$ as $3n + 2 > c$ for any constant c , and for all $n \geq n_0$

$10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$.

$O(1)$ means computing time is constant. $O(n)$ is called linear. $O(n^2)$ is called quadratic. $O(n^3)$ is called cubic. $O(2^n)$ is called exponential. If algorithm takes time $O(\log n)$, it is faster for large n , than it had taken for $O(n)$. Similarly $O(n \log n)$ is better than $O(n^2)$ but, not good as $O(n)$.

The function 2^n grows very rapidly with n . If an algorithm needs 2^n steps for execution, then, When $n = 40$, the number of steps needed is approximately 1.1×10^{12} . On a computer performing one billion steps per second,

When $n = 40$, the algorithm require about 18.3 minutes.

When $n = 50$, the same algorithm would run for 13 days on this computer.

When $n = 60$, the same algorithm would run for 310.56 years.

When $n = 100$, the same algorithm would run for 4×10^{13} years.

In general, the utility of algorithm with exponential complexity is limited to small value of n ($n \leq 40$).

4. Analysis of algorithm

The computational complexity for four different algorithms is analyzed in the following sections.

4.1 Algorithm 1

* To find the *largest* of n given numbers

Step1. Let $Max = x_1$

Step2. For $i = 2, 3, 4, \dots, n$

If $x_i > Max$, then $Max = x_i$

Otherwise do nothing.

Step3. Finally number in Max is the largest of n numbers.

4.2 Computational complexity and execution time of Algorithm 1

Each of the numbers $x_2, x_3, x_4, \dots, x_n$ is compared with Max once. If c is the time taken to compare two numbers, then the total time taken to execute the program is **(n-1)c**. Here c is the quantity that depends on efficiency of computer.

In general, we can say that, the time it takes to execute the algorithm is proportional to n-1. The computational complexity for algorithm-1 is **O(n)**.

4.3 Algorithm 2

* To find the *largest* of n given numbers

Step1. For $i= 1$ to $n-1$

Step2. Compare x_i and x_{i+1} and place the larger of two numbers in x_{i+1} and smaller in x_i .

Step3. Finally x_n is the largest of n numbers.

4.4 Computational complexity and execution time of algorithm 2

Here the total number of comparisons required is n-1. Therefore the time it takes to execute the algorithm is proportional to n-1. The computational complexity for the algorithm-2 is **O(n)**.

4.5 Difference and similarities of algorithm1 and algorithm2

Algorithm1 does not disturb the contents stored in register and also places the largest element in Max. Algorithm2 rearranges the contents of other register and places the largest element in x_n .

Both algorithms have the same computational complexities.

4.6 Algorithm 3

* To sort the elements in ascending order (using bubble sort).

Step1. For $i = n, n-1, n-2, \dots, 3, 2, 1$

Use Algorithm 2 to place in x_i the largest of i numbers.

Step2. Finally the numbers $x_1, x_2, x_3, x_4, \dots, x_n$ are in ascending order.

X_4	1	1	1	4	4	4	4
X_3	4	4	4	1	1	3	3
X_2	2	3	3	3	3	1	2
X_1	3	2	2	2	2	2	1

4.7 computational complexity and execution time of algorithm 3

First execution of Step1 uses (n-1) comparisons,

Second execution of Step1 uses (n-2) comparisons,

⋮
 ⋮
 ⋮

(n-1)st execution of Step1 uses 1 comparison.

Therefore total number of comparisons used = $1 + 2 + 3 + \dots + (n-2) + (n-1) = \mathbf{n(n-1) / 2}$

The computational complexity for the algorithm-3 is $\mathbf{O(n^2)}$.

4.8 Algorithm 4

To determine both the largest and smallest of n numbers.

Consider n numbers.

Using Algorithm 2, we can find the largest number in (n-1) comparisons.

Again using the similar algorithm, we can find the smallest of remaining (n-1) numbers in (n-2) comparisons. Therefore total number of comparisons used is $(n-1) + (n-2) = \mathbf{2n-3}$

The computational complexity of the algorithm-4 is $\mathbf{O(n)}$.

4.8 Algorithm 5

To determine the computational complexity of Kruskal's algorithm[3] in finding out the minimum weight spanning tree.

To compute the number of comparisons and number of iterations involved in Kruskal's algorithm for a network with n vertices, first let us consider a graph with three vertices and calculate the number of comparisons and number of iterations involved in it. Then it can be extended to five vertices and finally it can be generalized for n vertices.

If there are three vertices, then the first iteration will result in 2+1 comparisons. Second iteration will result in 1+1 comparisons and subsequently the third iteration will have one comparison. Hence the total no of iterations will be 3 and the total number of comparisons will be $3+2+1=6$.

If there are five vertices, then the first iteration will have 4+3+2+1 comparisons. Subsequently second iteration will have 3+3+2+1 comparisons, third iteration will have 2+3+2+1 comparisons, fourth iteration will have 1+3+2+1 comparisons, fifth iteration will result in 3+2+1 comparisons, sixth iteration will have 2+2+1 comparisons, seventh iteration will have 1+2+1 comparisons, eighth iteration will have 2+1 comparisons, ninth iteration will have 1+1 comparisons and finally tenth iteration will have one comparison. Hence the total no of iterations is 10 and the total number of comparisons is 55.

In general, for n vertices, the number of iterations will be $n(n-1)/2$, that is nC_2 iterations and the number of comparisons will be $1+2+3+ \dots + n(n-1)/2$, which results in $O(n^2)$ as computational complexity.

Therefore the computational complexity of Kruskal's algorithm [3] in finding out the minimum weight spanning tree is $O(n^2)$. More problems can be found in [4].

5. Conclusion

From this analysis it is clear that rate of growth of algorithm plays an important role than the number of operations in an algorithm. This analysis also provides background information on the efficiency of a program.

6. References

- [1] J. Hartmanis and R. Stearns, 1965, “On the computational complexity of algorithms”. Transactions of the American Mathematical Society, 117:285-306.
- [2] Sastry, V.N., Janakiraman, T.N. and Ismail Mohideen, S., “ New Polynomial Time Algorithm to Compute a Set of Pareto Optimal Paths for Multi Objective Shortest Path Problem”, International Journal of Computer Mathematics, Vol. 82, No. 3, 289 – 300, March 2005.
- [3] L.R.Foulds, 1992, “Graph Theory Applications”, Springer-Verlag New York, Inc, 234-236.
- [4] Horowitz,E., Sahni,S., and Rajasekaran,S., 2000, Computer Algorithms, Galgotia Publications, New Delhi.

