Algorithm Efficiency

Measuring the efficiency of algorithms

The topic of algorithms is a topic that is central to computer science. Measuring an algorithm's efficiency is important because your choice of an algorithm for a given application often has a great impact. Word processors, ATMs, video games and life support systems all depend on efficient algorithms.

Consider two searching algorithms. What does it mean to compare the algorithms and conclude that one is better than the other?

The **analysis of algorithms** is the area of computer science that provides tools for contrasting the efficiency of different methods of solution. Notice the use of the term *methods of solution* rather than *programs*; it is important 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.

Although the efficient use of both time and space is important, inexpensive memory has reduced the significance of space efficiency. Thus, we will focus primarily on time efficiency. How do you compare the time efficiency of two algorithms that solve the same problem? One possible approach is to implement the two algorithms in C++ and run the programs. There are three difficulties with this approach:

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.

To overcome these difficulties, computer scientists employ mathematical techniques that analyze algorithms independently of specific implementations, computers or data. You begin this analysis by counting the number of significant operations in a particular solution.

As an example of calculating the time it takes to execute a piece of code, consider the nested for loops below:

```
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 will discuss how to calculate the total time, which is: 5*t*N*(N+1)/2 time units.

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. The searches we have discussed depend on the size of the array we are searching. The most important thing to learn is how quickly the algorithm's time requirement grows as a function of the problem size. A statement such as:

Algorithm A requires time proportional to f(N)

enables you to compare algorithm A with another algorithm B which requires *g(N)* time units.

Algorithm A is said to be **order *f(N)***, which is denoted as ***O(f(N))***; *f(N)* is called the algorithm's **growth rate function**. Because the notation uses the capital letter O to denote order, it is called **Big O notation**. If a problem of size *N* requires time that is directly proportional to *N*, the problem is *O(N)* – that is, order *N*. If the time requirement is directly proportional to N^2, the problem is $O(N^2)$, and so on.

Definition of the Order of an Algorithm: Algorithm *A* is order *f(N)* – denoted *O(f(N))* – if constants *c* and N_0 exist such that *A* requires no more than *c*f(N)* time units to solve a problem of size $N >= N_0$. That is, *g(N)* is *O(f(N))* if the constants *c* and N_0 exist such that $g(N) < c*f(N)$ for $N >= N_0$. If *g(N)* is the time required to run Algorithm *A*, the *A* is *O(F(N))*.

The requirement $N >= N_0$ in the definition of *O(f(N))* means that the time estimate is correct for sufficiently large problems.

These growth-rate functions have the following intuitive interpretations:

1	A growth-rate functions of 1 implies a problem whose time requirement is constant and, therefore, independent of the problem's size.
$\log_2 N$	The time requirement for a **logarithmic** algorithm increases slowly as the problem size increases. The binary search algorithm has this behavior.
N	The time requirement for a **linear** algorithm increases directly with the size of the problem.
$N\log_2 N$	The time requirement increases more rapidly than a linear algorithm. Such algorithms usually divide a problem into smaller problems that are each solved separately.

N^2	The time requirement for a **quadratic** algorithm increases rapidly with the size of the problem. Algorithms that use two nested loops are often quadratic.
N^3	The time requirement for a **cubic** algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm. Algorithms that use three nested loops are often cubic and are practical only for small problems.
2^N	As the size of a problem increases, the time requirement for an exponential algorithm usually increases too rapidly to be practical.

If algorithm *A* requires time that is proportional to function *f* and algorithm *B* requires time that is proportional to a slower-growing function *g*, it is apparent that *B* will always be significantly more efficient than *A* for large enough problems. For large problems, the proportional growth rate dominates all other factors in determining an algorithm's efficiency.

Some properties of Big O notation help to simplify the analysis of an algorithm. You should keep in mind the *O(f(N))* means "is of order *f(N)*" or "has order *f(N)*." O is not a function.

1. You can ignore low order terms in an algorithm's growth-rate function.

2. You can ignore a multiplicative constant in the high-order term of an algorithm's growth rate function.

3. $O(f(N)) + O(g(N)) = O(f(N) + g(N))$. You can combine growth-rate functions.

These properties imply that you need only an estimate of the time requirement to obtain an algorithm's growth rate; you do not need an exact statement of an algorithm's time requirement, which is fortunate because deriving the exact time requirement is often difficult and sometimes impossible.

Worst-case analysis. A particular algorithm might require different times to solve different problems of the same size. For example, the time an algorithm requires to search *N* items might depend on the nature of the items. Usually you consider the maximum amount of time that an algorithm can require to solve a problem of size *N* – that is, the worst case. Although worst-case analysis can produce a pessimistic time estimate, such an estimate does not mean that your algorithm will always be slow. Instead, you have shown that the algorithm will never be slower than your estimate. An algorithm's worst case might happen rarely, if at all, in practice.

Tightness. We want the "tightest" big-O upper bound we can prove. If $f(N)$ is $O(N^2)$, we want to say so even though the statement $f(N)$ is $O(N^3)$ is technically true but "weaker."

Simplicity. We will generally regard $f(N)$ as "simple" if it is a single term and the coefficient of that term is 1. N^2 is simple, $2N^2$ and $N^2 + N$ are not.

We want simple and tight bounds to describe the order of our algorithms.