
Algorithm Efficiency

(Carrano, Chapter 9)

Measuring the Efficiency of Algorithms

- ◆ Consider two algorithms that perform the same task
- ◆ What does it mean to compare the algorithms?
- ◆ How can we conclude one is better, say, more *efficient* than the other?
- ◆ What are the factors that affect the efficiency of an algorithm?

Analysis of Algorithms

- ◆ Algorithm analysis techniques allows us to contrast the efficiency of different solutions for a problem
- ◆ Algorithms must efficiently use *time* and *memory*
- ◆ Algorithm analysis techniques focuses on gross differences in algorithm efficiency
- ◆ We will focus on time efficiency

Why not just measure program execution time?

- ◆ why don't we simply run the program and measure the time it takes to run?
- ◆ execution time depends upon a number of different factors such as:
 - programming techniques
 - programming language (language features)
 - compiler
 - operating system
 - computer architecture
 - input data
- ◆ measuring *implementations* does not tell us much about the fundamental nature of the program

Issues

- ◆ In analyzing algorithms, we would like to develop fundamentals “laws” that are *independent* of such considerations as
 - programming techniques
 - programming language (language features)
 - compiler
 - operating system
 - computer architecture
 - input data
- ◆ For this course, we are assuming the **von Neumann architecture**. Most computers are based on this architecture. In theory, better performance can sometimes be obtained using multiprocessor architectures, but that is outside the scope of this course

-
- ◆ Consider *linked-list traversal*:

```
ptrType Cur = Head;           (1 assignment)
while (Cur != NULL) {       (N+1 comparisons)
    cout << Cur->Item << endl; (N writes)
    Cur = Cur->Next;         (N assignments)
}
```
 - ◆ If assignments, comparisons, and write operations take a , c , and w time units, respectively, then:
 - list traversal takes: $(N+1) * (a+c) + N * w$ time units
 - That is, time required to write N nodes is proportional to N
 - ◆ Consider our solution to *Towers of Hanoi*:
 - For a problem with N disks, we require $2^N - 1$ moves
 - If each move requires the same time m , the solution takes $(2^N - 1) * m$ time units

-
- ◆ Consider the code segment:
for ($I = 1$ to N)
 for ($J = 1$ to I)
 for ($K = 1$ to 5)
 statement x ; (t time units)

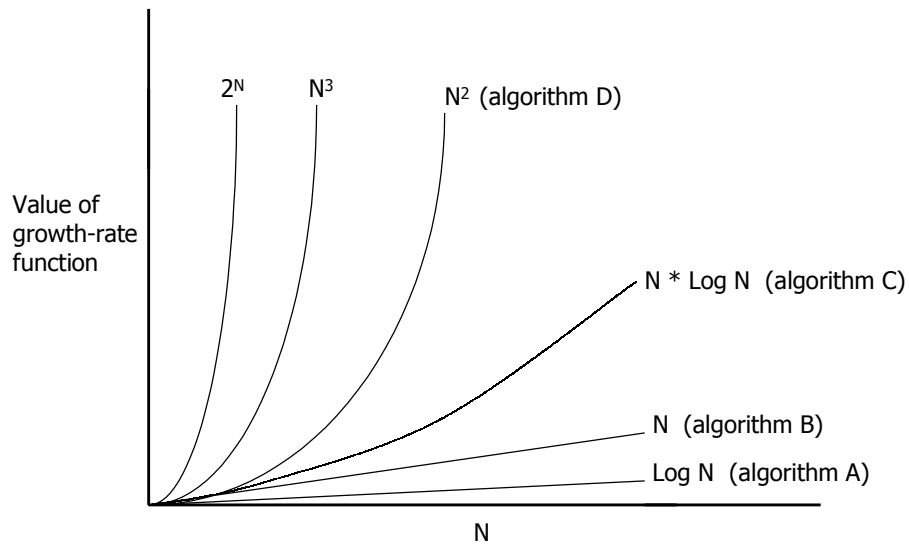
- ◆ Total time:

$$\sum_{i=1}^N 5 \times t \times i = 5 \times t \times (1 + 2 + \dots + N)$$
$$= 5 \times t \times \left(N \times \frac{(N+1)}{2} \right)$$

Algorithm Growth Rates

- ◆ An algorithm's (i.e., a solution's) time requirement can be expressed as a function of the *problem size*
- ◆ Problem size depends on the particular problem.
Examples:
 - number of nodes in a linked-list
 - number of disks in the Tower of Hanoi problem
 - size of an array
 - number of elements in a stack etc.
- ◆ The important thing is to find out how quickly the time of an algorithm grows as a function of the problem size
- ◆ This is called an algorithm's *growth rate*

*Once we know the growth rate of algorithms,
we can do comparison's:*



Keeping our perspective

- ◆ Suppose we take a program and run it on several machines
- ◆ For each machine, we measure the execution time for several values of n
- ◆ Suppose we notice that no matter which (von Neumann) machine we use, the execution time appears to be a quadratic function of n (i.e., $T(n) = an^2 + bn + c$)
- ◆ We will consider this to be a fundamental property of that program or algorithm

-
- ◆ Suppose that the execution time for algorithm A is a quadratic function of n
 - ◆ Suppose that the execution time for algorithm B is a linear function of n
 - ◆ Suppose that the execution time for algorithm C is an exponential function of n
 - ◆ We say that these 3 algorithms belong to 3 different **complexity classes** (based on particular functions of n)

-
- ◆ For large problems (i.e., as n becomes very large), higher-order terms will dominate all of the rest

- ◆ Example: $T(n) = 0.0001724n^2 + 0.4n + 21.3$

We might first think that we can ignore the n -squared term because its coefficient is so small — but *eventually*, this term will be much larger than the others

- ◆ We'd like to characterize an algorithm with a statement such as
"A requires time proportional to 2^n "

O-notation

- ◆ We use a convention called O-notation (or "big-O-notation") to represent different complexity classes
- ◆ Example: $T(n) = an^2 + bn + c$
 - » We say that $T(n)$ is $O(n^2)$, or we say that T is of quadratic complexity
- ◆ Example: $T(n) = an + b$
 - » We say that $T(n)$ is $O(n)$
- ◆ Example: $T(n) = a2^n + c$
 - » We say that $T(n)$ is $O(2^n)$

Definition of O-notation

- ◆ Let n and M be positive integers, and let K be a positive real number
- ◆ We say that $h(n)$ is $O(f(n))$ if there exists positive constants K and M such that:
$$|h(n)| \leq K|f(n)|, \forall n \geq M$$
- ◆ If we say that an algorithm runs to completion in $O(f(n))$ steps, this means that the actual number of steps is no more than a constant times $|f(n)|$, provided n is large enough

Manipulating O-notation

- ◆ Usually, we can do this quickly:
 - identify the dominant term
 - ignore lesser terms
 - ignore any coefficient that the dominant term may have

◆ Example: $T(n) = a2^n + bn^3 + c$

$$O(T(n)) = O(a2^n + bn^3 + c)$$

$$= O(a2^n)$$

$$= O(2^n)$$

$\Rightarrow T$ is of exponential complexity

Some common complexity classes

- | | |
|---------------|---------------|
| ◆ constant | $O(1)$ |
| ◆ logarithmic | $O(\log n)$ |
| ◆ linear | $O(n)$ |
| ◆ $n \log n$ | $O(n \log n)$ |
| ◆ quadratic | $O(n^2)$ |
| ◆ cubic | $O(n^3)$ |
| ◆ exponential | $O(2^n)$ |
| ◆ exponential | $O(10^n)$ |

Consider sequential search

- ◆ Given an array of N items
- ◆ We look at each item starting from the beginning, until we find the desired item or we reach the end
- ◆ *Best case*: the desired item is the first item (a single comparison; hence $O(1)$)
- ◆ *Worst-case*: the desired item is the last one (N comparisons, hence $O(N)$)
- ◆ *Average-case*: the desired item is in the middle ($N/2$ comparisons; hence $O(N)$)

Consider binary search (1)

- ◆ Given an array of N items
- ◆ Inspect the middle item of an array of size N
- ◆ Inspect middle item of an array of size $N/2$
- ◆ Inspect middle item of an array of size $N/2^2$, and so on
- ◆ What is the worst-case complexity?
 - How many inspections will we perform?
 - Let's say $N = 2^k$, for some k
 - Worst-case: k divisions/comparisons
 - Thus, the complexity is $O(k) = O(\log_2 N)$

Complexity of binary search (2)

- ◆ What if N is not a power of 2?
 - We can find: $2^{k-1} < N < 2^k$
 - The algorithm would still require *at most* k divisions. It follows that:
 - $k - 1 < \log_2 N < k$
 - $k < 1 + \log_2 N < k + 1$
 - $k = 1 + \log_2 N$ rounded down
 - Thus, the complexity is still $O(\log_2 N)$
- ◆ In general, the algorithm is $O(\log_2 N)$ in the worst-case for any N

- ◆ We have looked at several algorithms already:

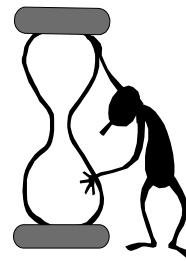
» sequential search:	$O(n)$
» binary search:	$O(\log n)$
» selection sort:	$O(n^2)$
» Towers of Hanoi:	$O(2^n)$

(For the Towers of Hanoi solution, notice that the "size of the problem" is not the length of an array)

Why are we so obsessed with learning the complexity class of an algorithm?

- ◆ Suppose an algorithm takes exactly $T(n)$ microseconds to complete

$T(n)$	$n=16$	$n=256$
$\log_2 n$	$4 \mu s$	$8 \mu s$
n	$16 \mu s$	$256 \mu s$
n^2	$25.6 \mu s$	65.5 ms
2^n	65.5 ms	10^{63} years



Worst-case and average-case analysis

- ◆ Usually we analyze an algorithm's worst-case behavior (it is easier to determine)
- ◆ In some cases, average-case behavior is more useful
(but it is usually much more difficult to evaluate)

Summary

- ◆ The efficiency of an algorithm is determined in terms of its *growth-rate*
- ◆ The growth-rate of an algorithm tells us how quickly the algorithm grows as a function of the "size of the problem"
- ◆ Computational complexity of an algorithm is represented using the O-notation
- ◆ Using O-notation, we can classify algorithms to belong to different "complexity classes"
- ◆ We would like to avoid algorithms that are of exponential complexity