CSIS 3103

More
Algorithm Analysis

Efficiency Analysis

- With an array of n elements, Sequential Search makes $f(n) = n/2$ comparisons, on average, if the target is in the array
- This algorithm runs in *linear time*, because $n/2$ is *of the same order* as the linear function $g(n) = n$. (The graph is a *line*.)
- This is abbreviated: $O(n)$

Big-O Notation

An abbreviation of "order of magnitude"

$T(n) = O(f(n))$

- There are positive constants, n_0 and c *such that for all* $n > n_0$, $cf(n) \geq T(n)$
- $cf(n)$ is an upper bound on $T(n)$
- If T is a measure of the performance of an algorithm, it will never be worse than $cf(n)$

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
     Simple Statement 0
  }
}
for (int i = 0; i < n; i++) {
     Simple Statement 1
     Simple Statement 2
     Simple Statement 3
     Simple Statement 4
     Simple Statement 5
}
 Simple Statement 6
 Simple Statement 7
...
 Simple Statement 30
```

Nested loops execute *Simple Statement 0* n^2 times

Loop execute 5 *Simple Statements* n times

25 *Simple Statements* executed, once each

How much work is required?

Big-O Notation

The growth rate of f(n) is determined by the fastest growing term - the one with the largest exponent

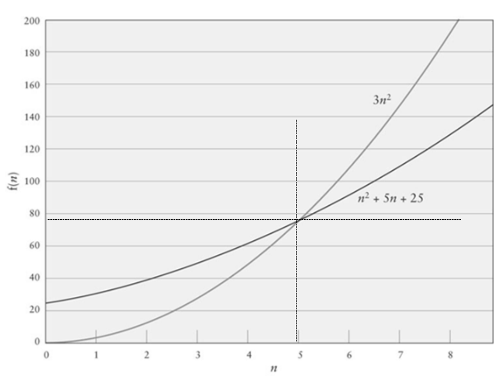
In the example, an algorithm of

$$O(n^2 + 5n + 25)$$

is more simply expressed as

$$O(n^2)$$

In general, it is safe to ignore all constants and to drop the lower-order terms when determining the order of magnitude



Efficiency Analysis

Goal: Simplify as much as possible by getting rid of unnecessary information

- Rounding: 1,000,001 ≈ 1,000,000
- Suppose it takes
  - 50,000 ms for Windows to boot up
  - 10 ms to process some transaction
- n transactions take (50,000 + 10n) ms
- 10n becomes more important as n gets large

Big-O Notation

A simple way to determine the big-O notation of an algorithm is to look at the loops and to see whether the loops are nested

Assuming a loop body consists only of simple statements,

a single loop is $O(n)$

a pair of nested loops is $O(n^2)$

a nested pair of loops inside another is $O(n^3)$

. . .

Reasoning about algorithms

$O(n)$ algorithm,

- For 5,000 elements takes 3.2 seconds
- For 10,000 elements takes 6.4 seconds
- For 15,000 elements takes ….?

$O(n^2)$ algorithm

- For 5,000 elements takes 3.2 seconds
- For 10,000 elements takes 12.8 seconds
- For 15,000 elements takes …?

Consider:

```
for (int i = 1; i < n; i++) {
  for (int j = i; j < n; j++) {
      3 simple statements
  }
}
```

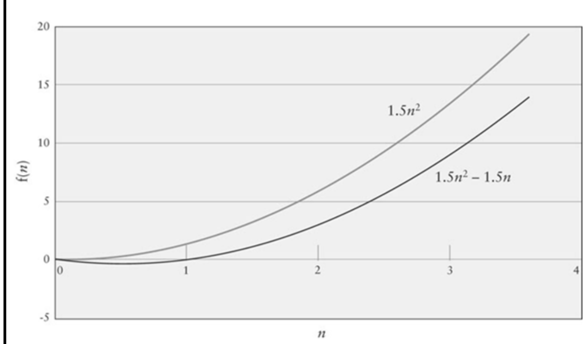
$$T(n) = 3(n-1) + 3(n-2) + \ldots + 3$$
$$= 3(n - 1 + n - 2 + n - 3 + \ldots + 1)$$
$$= 3(1 + 2 + \ldots + n - 1) = 3(n \times (n-1))/2$$
$$= 3(n^2 - n) / 2$$

Therefore $T(n) = 1.5n^2 - 1.5n$

- When $n = 0$, the polynomial has the value 0
- For values of $n > 1$,
  $1.5n^2 > 1.5n^2 - 1.5n$

Therefore, using $n_0 = 1$ and $c = 1.5$ we conclude that

$T(n)$ is $O(n^2)$



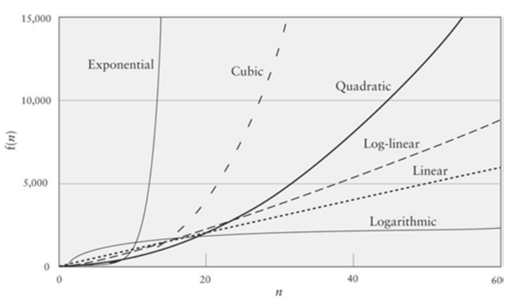
Notation

Symbol	Meaning
$T(n)$	The time that a method or program takes as a function of the number of inputs, n. We may not be able to measure or determine this exactly.
$f(n)$	Any function of n. Generally, $f(n)$ will represent a simpler function than $T(n)$, for example, n^2 rather than $1.5n^2 - 1.5n$.
$O(f(n))$	Order of magnitude. $O(f(n))$ is the set of functions that grow no faster than $f(n)$. We say that $T(n) = O(f(n))$ to indicate that the growth of $T(n)$ is bounded by the growth of $f(n)$.

Common Growth Rates

Big-O	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Common Growth Rates



Effects of Different Growth Rates

$O(f(n))$	$f(50)$	$f(100)$	$f(100)/f(50)$
$O(1)$	1	1	1
$O(\log n)$	5.64	6.64	1.18
$O(n)$	50	100	2
$O(n \log n)$	282	664	2.35
$O(n^2)$	2500	10,000	4
$O(n^3)$	12,500	100,000	8
$O(2^n)$	1.126×10^{15}	1.27×10^{30}	1.126×10^{15}
$O(n!)$	3.0×10^{64}	9.3×10^{157}	3.1×10^{93}

A Caution

- Beware of very large constant factors
- An algorithm running in time 1,000,000 N is still $O(N)$
- But it might be less efficient on your data set than one running in time $2N^2$, which is $O(N^2)$

Algorithms with Exponential and Factorial Growth Rates

Given an $O(2^n)$ algorithm, if 100 inputs takes an hour then,

- 101 inputs will take 2 hours
- 105 inputs will take 32 hours
- 114 inputs will take 16,384 hours (almost 2 years!)

When Worse is Better

Some cryptographic algorithms can be broken in $O(2^n)$ time, where n is the number of bits in the key

- A key length of 40 is considered breakable by a modern computer,
- A key length of 100 bits will take a billion-billion (10^{18}) times longer than a key length of 40

Performance of `KWArrayList`

- The `set` and `get` methods execute in constant time: $O(1)$
- Inserting or removing general elements is linear time: $O(n)$
- Adding at the end is (usually) constant time: $O(1)$
- With our reallocation technique the average is $O(1)$
  - The worst case is $O(n)$ because of reallocation