The Efficiency of Algorithms

Chapter 4

Data Structures and Abstractions with Java, 4e, Global Edition
Frank Carrano

Why Efficient Code?

- Computers are faster, have larger memories
  - So why worry about efficient code?
- And … how do we measure efficiency?

Example

- Consider the problem of summing

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + n$$

Algorithm A	Algorithm B	Algorithm C

`sum = 0`
`for i = 1 to n`
`    sum = sum + i`
	`sum = 0`
	`for i = 1 to n`
	`{`
	`    for j = 1 to i`
	`        sum = sum + 1`
	`}`
		`sum = n * (n + 1) / 2`

FIGURE 4-1 Three algorithms for computing the sum 1 + 2 + . . . + n for an integer n > 0

Example

```java
// Computing the sum of the consecutive integers from 1 to n:
long n = 10000; // Ten thousand

// Algorithm A
long sum = 0;
for (long i = 1; i <= n; i++)
   sum = sum + i;
System.out.println(sum);

// Algorithm B
sum = 0;
for (long i = 1; i <= n; i++)
{
   for (long j = 1; j <= i; j++)
       sum = sum + 1;
} // end for
System.out.println(sum);

// Algorithm C
sum = n * (n + 1) / 2;
System.out.println(sum);
```

Java code for the three algorithms

What is “best”?

- An algorithm has both time and space constraints – that is complexity
  - Time complexity
  - Space complexity
- This study is called analysis of algorithms

Counting Basic Operations

- A basic operation of an algorithm
  - The most significant contributor to its total time requirement

	Algorithm A	Algorithm B	Algorithm C
Additions	n	$n(n+1)/2$	1
Multiplications			1
Divisions			1
Total basic operations	n	$(n^2+n)/2$	3

FIGURE 4-2 The number of basic operations required by the algorithms in Figure 4-1

Counting Basic Operations

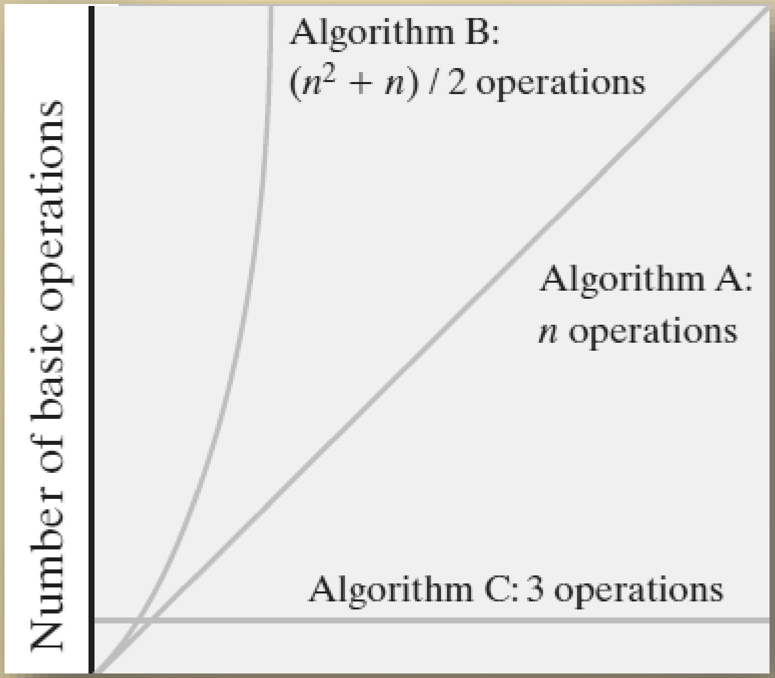

FIGURE 4-3 The number of basic operations required by the algorithms in Figure 4-1 as a function of *n*

Counting Basic Operations

n	$\log(\log n)$	$\log n$	$\log^2 n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	2	3	11	10	33	10^2	10^3	10^3	10^5
10^2	3	7	44	100	664	10^4	10^6	10^{30}	10^{94}
10^3	3	10	99	1000	9966	10^6	10^9	10^{301}	10^{1435}
10^4	4	13	177	10,000	132,877	10^8	10^{12}	10^{3010}	$10^{19,335}$
10^5	4	17	276	100,000	1,660,964	10^{10}	10^{15}	$10^{30,103}$	$10^{243,338}$
10^6	4	20	397	1,000,000	19,931,569	10^{12}	10^{18}	$10^{301,030}$	$10^{2,933,369}$

FIGURE 4-4 Typical growth-rate functions evaluated at increasing values of *n*

Best, Worst, and Average Cases

- For some algorithms, execution time depends only on size of data set
- Other algorithms depend on the nature of the data itself
  - Here we seek to know best case, worst case, average case

Big Oh Notation

- A function f(n) is of order at most g(n)
- That is, f(n) is O(g(n))—if
  - A positive real number c and positive integer N exist …
  - Such that $f(n) \leq c \times g(n)$ for all $n \geq N$
  - That is, $c \times g(n)$ is an upper bound on $f(n)$ when n is sufficiently large

Big Oh Notation

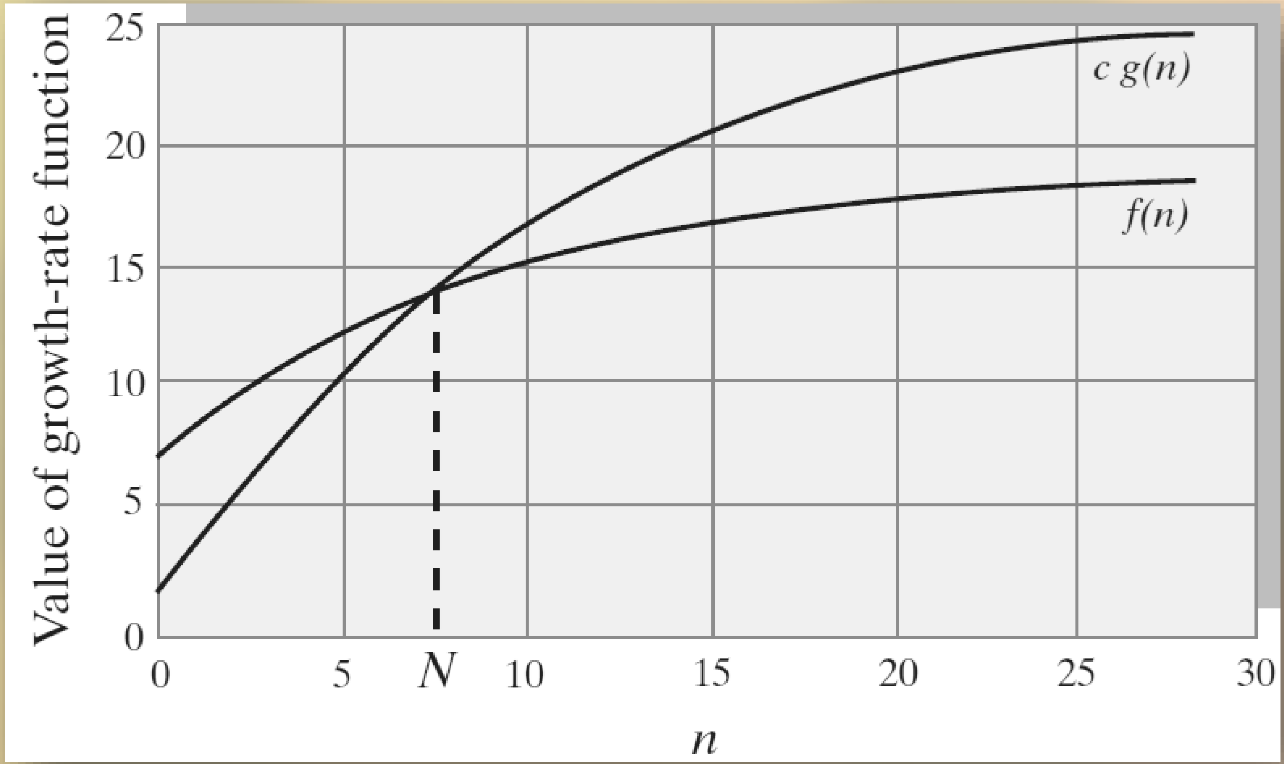

FIGURE 4-5 An illustration of the definition of Big Oh

Big Oh Notation

The following identities hold for Big Oh notation:

$$O(k\, g(n)) = O(g(n)) \text{ for a constant } k$$

$$O(g_1(n)) + O(g_2(n)) = O(g_1(n) + g_2(n))$$

$$O(g_1(n)) \times O(g_2(n)) = O(g_1(n) \times g_2(n))$$

$$O(g_1(n) + g_2(n) + \ldots + g_m(n)) = O(\max(g_1(n), g_2(n), \ldots, g_m(n)))$$

$$O(\max(g_1(n), g_2(n), \ldots, g_m(n))) = \max(O(g_1(n)), O(g_2(n)), \ldots, O(g_m(n)))$$

By using these identities and ignoring smaller terms in a growth-rate function, you can usually find the order of an algorithm's time requirement with little effort. For example, if the growth-rate function is $4n^2 + 50n - 10$,

$$\begin{aligned} O(4n^2 + 50n - 10) &= O(4n^2) && \text{by ignoring the smaller terms} \\ &= O(n^2) && \text{by ignoring the constant multiplier} \end{aligned}$$

Identities for Big Oh Notation

Complexities of Program Constructs

Construct	Time Complexity
Consecutive program segments $S_1, S_2, \ldots, S_k$ whose growth-rate functions are $g_1, \ldots, g_k$, respectively	$\max(O(g_1), O(g_2), \ldots, O(g_k))$
An `if` statement that chooses between program segments S_1 and S_2 whose growth-rate functions are g_1 and g_2, respectively	$O(condition) + \max(O(g_1), O(g_2))$
A loop that iterates m times and has a body whose growth-rate function is g	$m \times O(g(n))$

Picturing Efficiency

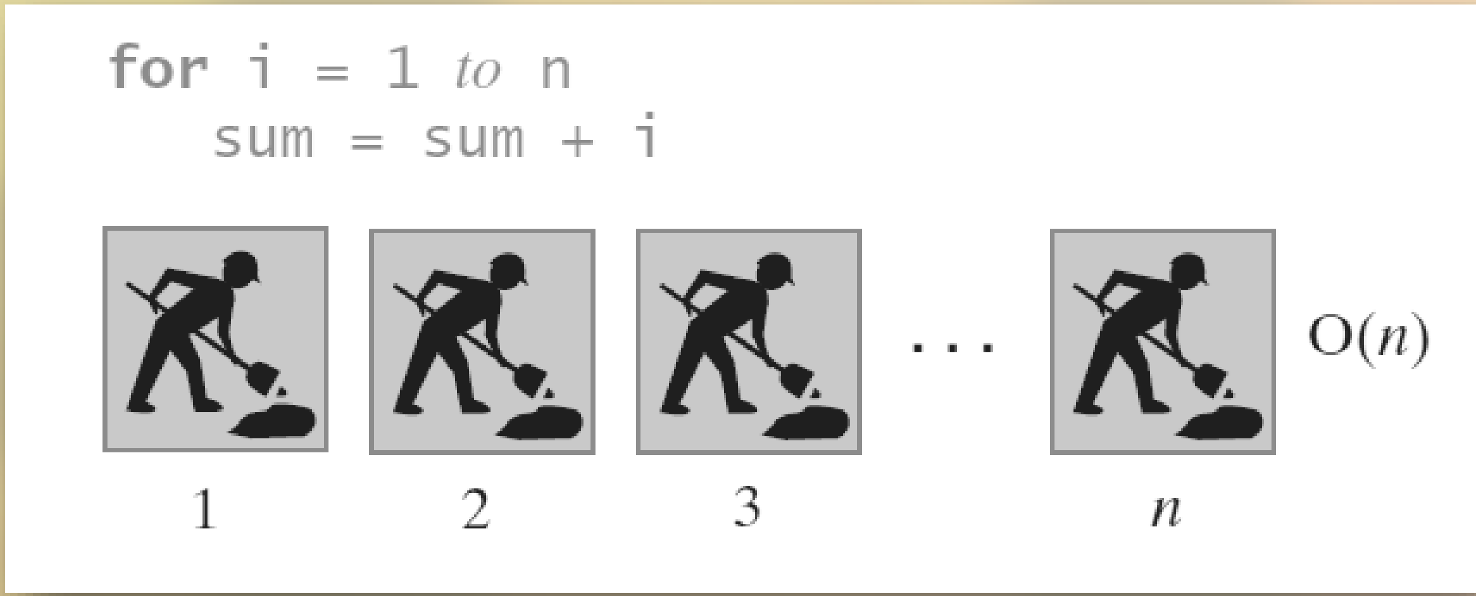

FIGURE 4-6 An O(n) algorithm

Picturing Efficiency

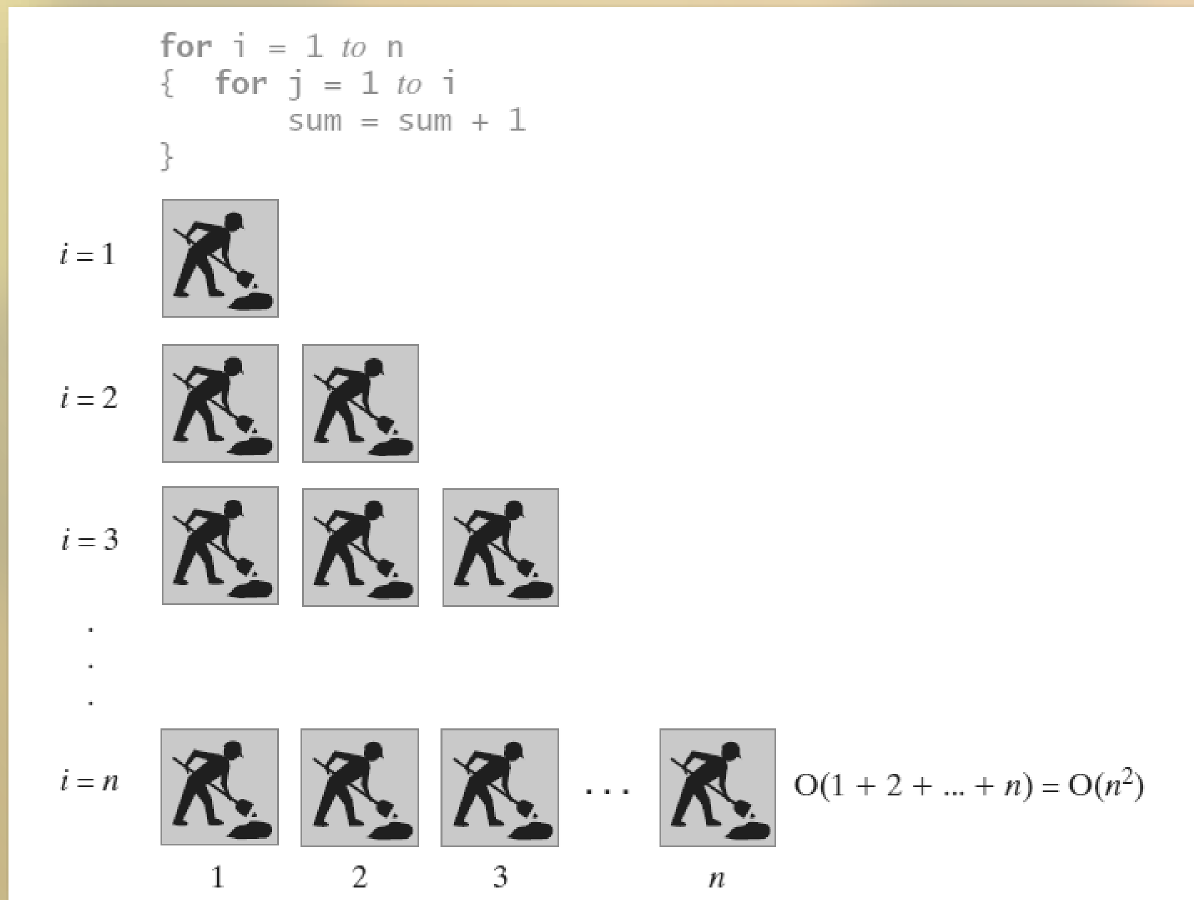

FIGURE 4-7 An $O(n^2)$ algorithm

Picturing Efficiency

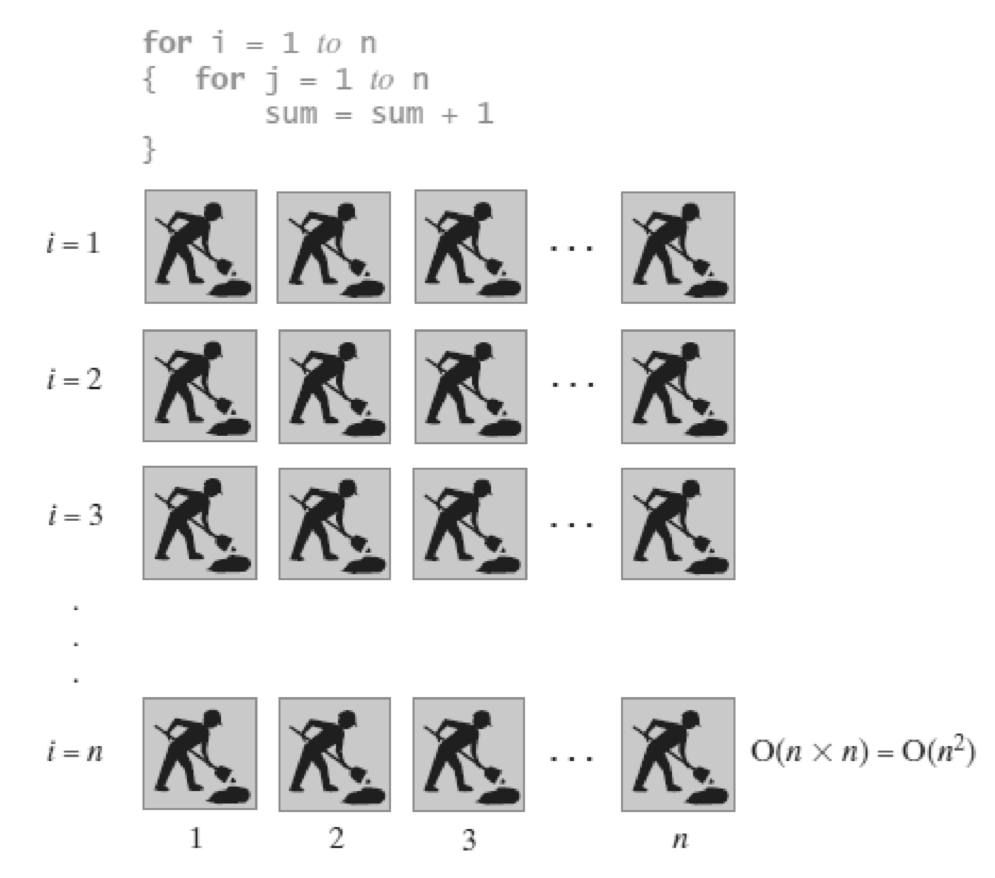

FIGURE 4-8 Another $O(n^2)$ algorithm

Picturing Efficiency

Growth-Rate Function for Size n Problems	Growth-Rate Function for Size $2n$ Problems	Effect on Time Requirement
1	1	None
$\log n$	$1 + \log n$	Negligible
n	$2n$	Doubles
$n \log n$	$2n \log n + 2n$	Doubles and then adds $2n$
n^2	$(2n)^2$	Quadruples
n^3	$(2n)^3$	Multiplies by 8
2^n	2^{2n}	Squares

FIGURE 4-9 The effect of doubling the problem size on an algorithm's time requirement

Picturing Efficiency

Growth-Rate Function g	$g(10^6) / 10^6$
$\log n$	0.0000199 seconds
n	1 second
$n \log n$	19.9 seconds
n^2	11.6 days
n^3	31,709.8 years
2^n	$10^{301,016}$ years

FIGURE 4-10 The time required to process one million items by algorithms of various orders at the rate of one million operations per second

Efficiency of Implementations of ADT Bag

Operation	Fixed-Size Array	Linked
`add(newEntry)`	O(1)	O(1)
`remove()`	O(1)	O(1)
`remove(anEntry)`	O(1), O(n), O(n)	O(1), O(n), O(n)
`clear()`	O(n)	O(n)
`getFrequencyOf(anEntry)`	O(n)	O(n)
`contains(anEntry)`	O(1), O(n), O(n)	O(1), O(n), O(n)
`toArray()`	O(n)	O(n)
`getCurrentSize(), isEmpty()`	O(1)	O(1)

FIGURE 4-11 The time efficiencies of the ADT bag operations for two implementations, | expressed in Big Oh notation

End

Chapter 4