Northeastern University

Lecture 14: The Efficiency of Algorithms

Prof. Chen-Hsiang (Jones) Yu, Ph.D.

College of Engineering

Materials are edited by Prof. Jones Yu from

Data Structures and Abstractions with Java, 5th edition. By Frank M. Carrano and Timothy M. Henry.
ISBN-13 978-0-13-483169-5 © 2019 Pearson Education, Inc.

What is Algorithms?

- An algorithm is a set of rules that must be followed to solve a specific problem.

Oxford Learner's Dictionaries

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

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

Algorithm A	Algorithm B	Algorithm C

`long sum = 0;`
`for (long i = 1; i <= n; i++)`
`  sum = sum + i;`
	`sum = 0;`
	`for (long i = 1; i <= n; i++)`
	`{`
	`  for (long j = 1; j <= i; j++)`
	`     sum = sum + 1;`
	`} // end for`
		`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

Program with Execution Time

```java
public class Computing {
  public static void main(String[] args){
    long startTime, endTime, totalTime;
    long n = 100000;

    //Algorithm A
    startTime = System.currentTimeMillis();
    long sum = 0;
    for(long i=1;i<=n;i++){
      sum = sum + i;
    }
    endTime = System.currentTimeMillis();
    totalTime = endTime - startTime;
    System.out.println("sum = " + sum + "; time = " + totalTime + " milliseconds");

    //Algorithm B
    startTime = System.currentTimeMillis();
    sum = 0;
    for(long i=1;i<=n;i++){
      for(long j=1;j<=i;j++){
        sum = sum + 1;
      }
    }
    endTime = System.currentTimeMillis();
    totalTime = endTime - startTime;
    System.out.println("sum = " + sum + "; time = " + totalTime + " milliseconds");

    //Algorithm C
    startTime = System.currentTimeMillis();
    sum = n * (n+1)/2;
    endTime = System.currentTimeMillis();
    totalTime = endTime - startTime;
    System.out.println("sum = " + sum + "; time = " + totalTime + " milliseconds");
  }
}
```

Results

```
sum = 5000050000; time = 2 milliseconds
sum = 5000050000; time = 2028 milliseconds
sum = 5000050000; time = 0 milliseconds
```

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

	`long sum = 0;`
	`for (long i = 1; i <= n; i++)`
	`  sum = sum + i;`
		`sum = 0;`
		`for (long i = 1; i <= n; i++)`
		`{`
		`  for (long j = 1; j <= i; j++)`
		`     sum = sum + 1;`
		`} // end for`
			`sum = n * (n + 1) / 2;`

Additons	n	$n(n + 1)/2$	1
Multiplications	0	0	1
Divisions	0	0	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

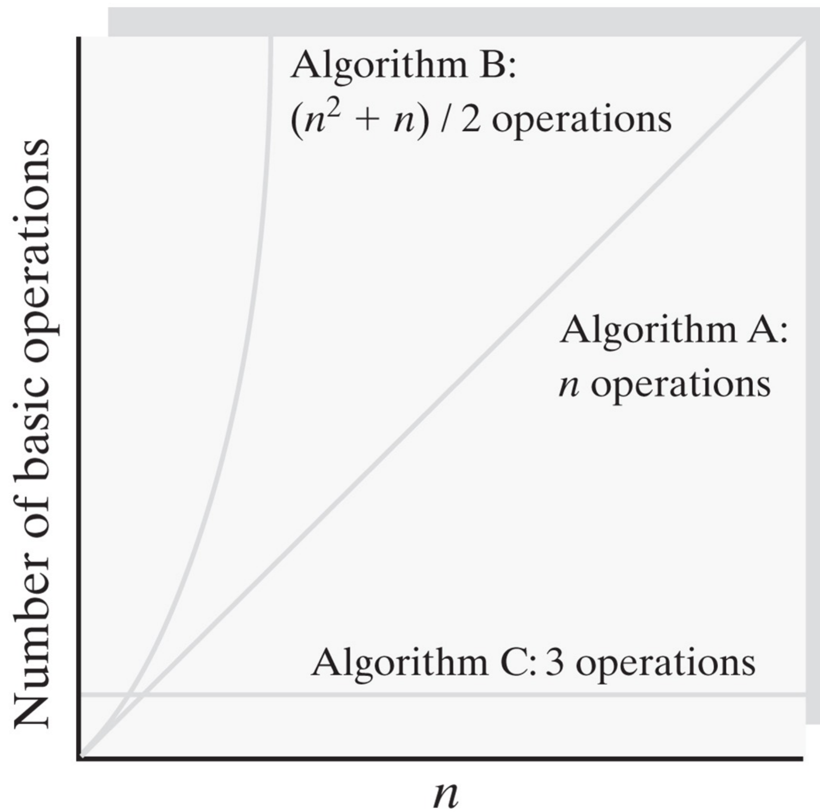


Figure 4-3: The number of basic operations required by the algorithms in Figure 4-1 as a function of *n*

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	1,000	9,966	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	100,00	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}	$10^{301,301}$	$10^{2,933,369}$

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 $g(n)$
- $f(n)$ is $O(g(n))$ - if
  - Positive constants c and 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

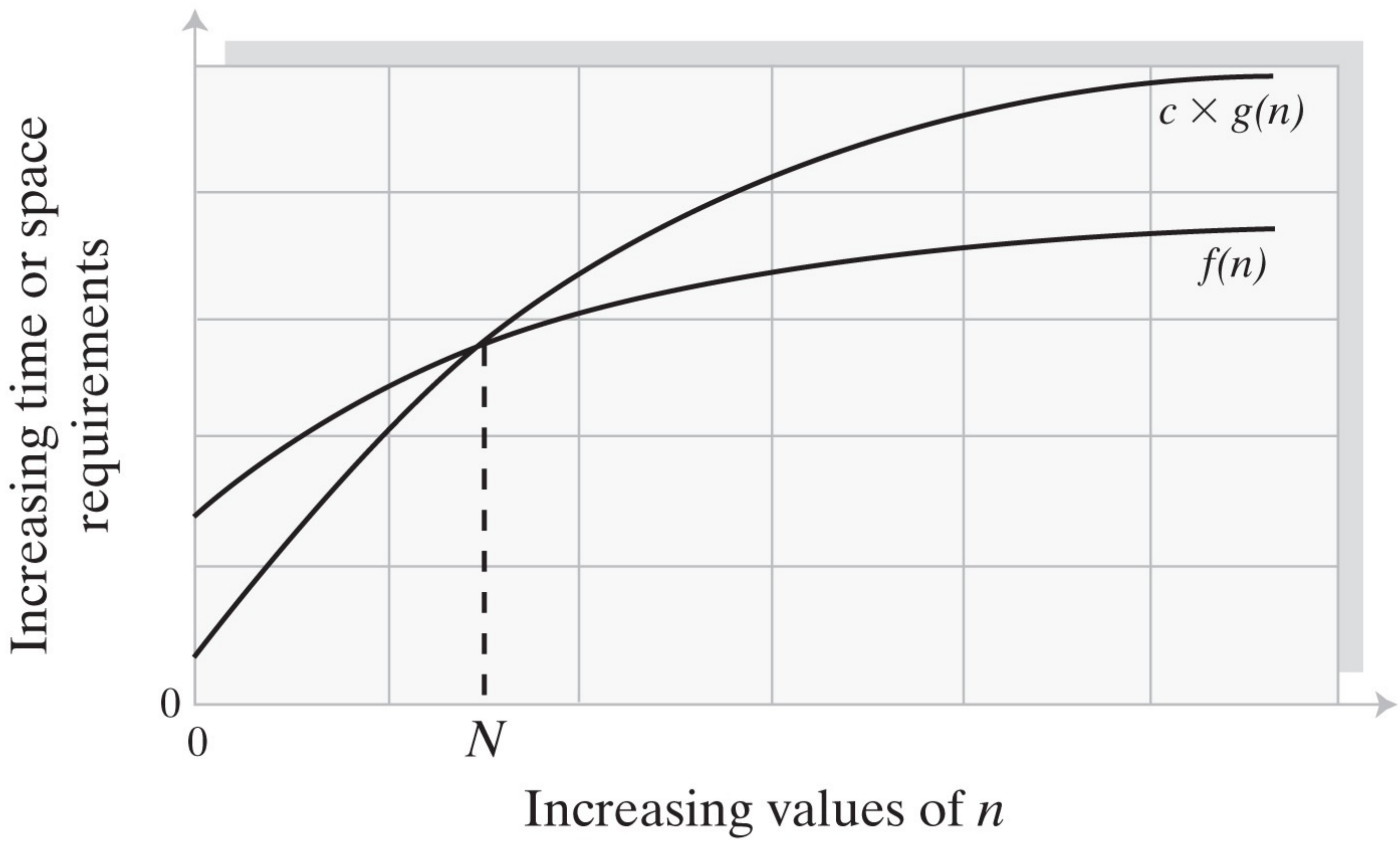


Figure 4-5: An illustration of the definition of Big Oh

Exercise

- Show that $3n^2 + 2^n$ is $O(2^n)$. What values of c and N did you use?

Answer

- $3n^2 + 2^n \leq 2^n + 2^n = 2 \times 2^n$ when $n \geq 8$
- So $3n^2 + 2^n = O(2^n)$, using $c = 2$ and $N = 8$

Big Oh Notation

$O(k\ g(n)) = O(g(n))$ for a constant k

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

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

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

$O(max(g_1(n), g_2(n), . . ., g_m(n))$

$O(max(g_1(n), g_2(n), . . ., g_m(n))) =$

$max(O(g_1(n)), O(g_2(n)), . . ., O(g_m(n)))$

Identities for Big Oh Notation

Complexities of Program Constructs

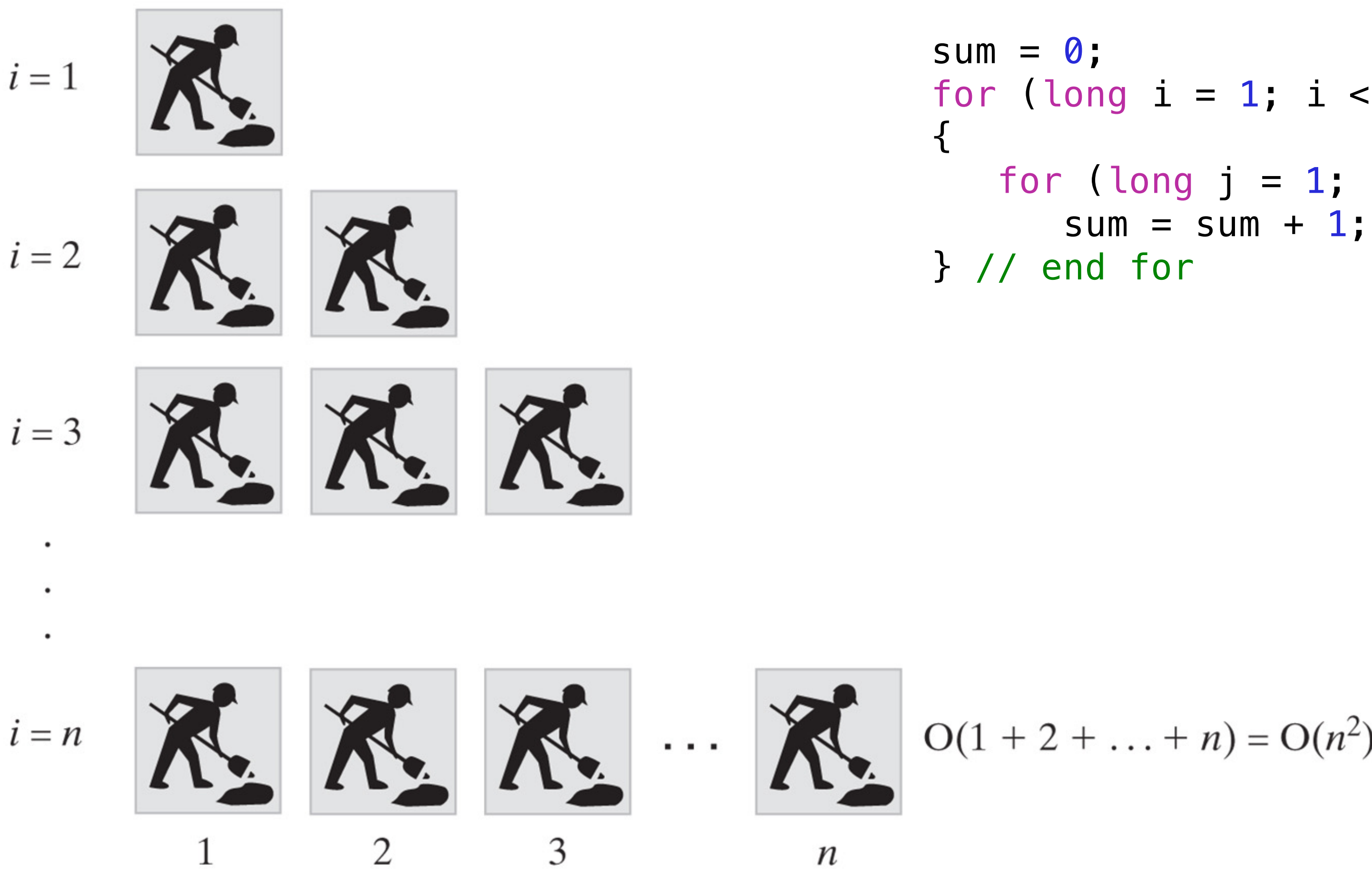
Construct	Time Complexity
Consecutive program segments $S_1, S_2, \ldots, S_k$ whose growth-rate functions are $g_1, \ldots, g_k$, respectively	$\max(O(g_1), O(g_2), \ldots, O(g_k))$
An `if` statement that chooses between program segments S_1 and S_2 whose growth-rate functions are $g_1$and g_2, respectively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

Picturing Efficiency

```
long sum = 0;
for (long i = 1; i <= n; i++)
   sum = sum + i;
```

Figure 4-6: An O(n) algorithm

Picturing Efficiency




```
sum = 0;
for (long i = 1; i <= n; i++)
{
   for (long j = 1; j <= n; j++)
      sum = sum + 1;
} // end for
```

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

Picturing Efficiency


```
sum = 0;
for (long i = 1; i <= n; i++)
{
   for (long j = 1; j <= n; j++)
      sum = sum + 1;
} // end for
```

$O(n \times n) = O(n^2)$

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	$(2n)^3$	Multiples 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 (cont.)

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

Big Omega Notation

- A function $f(n)$ is of order at least $g(n)$ - that is, $f(n)$ is $\Omega(g(n))$ - if $g(n)$ is $O(f(n))$.

- In other words, $f(n)$ is $\Omega(g(n))$ if
  - Positive constants c and N exist such that $f(n) \geq c \times g(n)$ for all $n \geq N$
  - That is, the time requirement $f(n)$ is not smaller than $c \times g(n)$, its lower bound.

Big Theta Notation

- A function $f(n)$ is of order $g(n)$ - that is, $f(n)$ is $\Theta(g(n))$ - if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$.
- Actually, we can say $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.
- The time requirement $f(n)$ is the same as $g(n)$. $c \times g(n)$ is both a lower bound and an upper bound on $f(n)$.
- A big theta analysis assures us that the time estimate is as good as possible.

Exercise

- The following algorithm finds out whether an array contains duplicate entries within its first *n* elements.
- What is the Big Oh of this algorithm in the worst case?

```
Algorithm hasDuplicates(array, n)

for index = 0 to n - 2
    for rest = index + 1 to n - 1
        if (array[index] equals array[rest])
            return true
return false
```

Answer

- Let's tabulate the maximum number of times the inner loop executes for various values of index:

index	**Inner Loop Iterations**
0	n - 1
1	n - 2
2	n - 3
. . . .	
n - 2	1

- As you can see, the maximum number of times the inner loop executes is $1 + 2 + \ldots + n - 1$, which is $n\,(n - 1)\,/\,2$. Thus, the algorithm is $O(n^2)$ in the worst case.

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