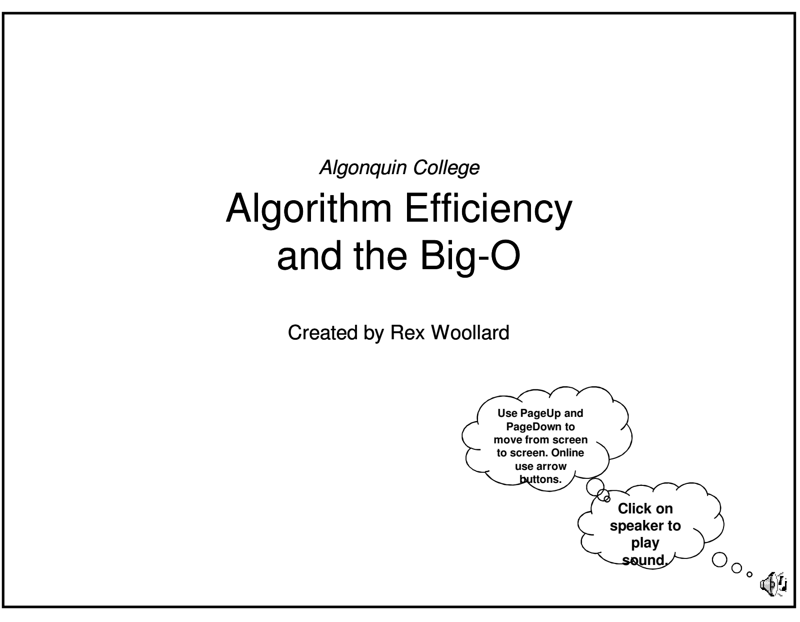


The Quest for Efficiency

- Why does it matter?
- How do you compare different algorithms?
- How do you measure?

Click on each speaker.

Linear Loops: Case 1

Counting by 1

```
for (int i = 0; i < 1000; i++)
  // do something
```

Linear Loops: Case 1: Doubling

Counting by 1

```
for (int i = 0; i < 2000; i++)
  // do something
```

Absolute Measure of Efficiency: f(n) = n

Value of *n*	Program Effort (Iterations)	Effect of Doubling
1000	1000	
2000	2000	Twice Effort
4000	4000	Twice Effort

Linear Loops: Case 2

Counting by 2

```
for (int i = 0; i < 1000; i = i + 2)
  // do something
```

Linear Loops: Case 2: Doubling

Counting by 2

```
for (int i = 0; i < 2000; i = i + 2)
  // do something
```

Absolute Measure of Efficiency: f(*n*) = *n*

Value of *n*	Program Effort (Iterations)	Effect of Doubling *n*
1000	500	
2000	1000	Twice Effort
4000	2000	Twice Effort

Big-O Principle

- Big-O measures the effect as you scale to larger values of *n.*
- We don't care that a loop required 1000 iterations.
- We do care that the loop required twice as many iterations when we doubled *n.*
- Thus, the previous two cases are deemed to have the same Big-O.

Logarithmic Loops: Multiply

Double on Each Iteration

```
for (int i = 1; i <= 1000; i = i * 2)
  // do something
```

Iteration	Value of *i*
1	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128
9	256
10	512
exit	1024

Logarithmic Loops: Multiply: Doubling

Double on Each Iteration

```
for (int i = 1; i <= 2000; i = i * 2)
  // do something
```

Absolute Measure of Efficiency: f(n) = $\log_2 n$

Value of *n*	Program Effort (Iterations)	Effect of Doubling *n*
1000	10	
2000	11	+1
4000	12	+1
8000	13	+1
16000	14	+1
32000	15	+1

Logarithmic Loops: Divide

Divide in Half on Each Iteration

```
for (int i = 1000; i >= 1; i = i / 2)
  // do something
```

Iteration	Value of *i*
1	1000
2	500
3	250
4	125
5	62
6	31
7	15
8	7
9	3
10	1
exit	0

Logarithmic Loops: Divide: Doubling

Divide in Half on Each Iteration

```
for (int i = 2000; i >= 1; i = i / 2)
  // do something
```

Absolute Measure of Efficiency: f(n) = $\log_2 n$

Value of *n*	Program Effort (Iterations)	Effect of Doubling *n*
1000	10	
2000	11	+1
4000	12	+1
8000	13	+1
16000	14	+1
32000	15	+1

So What?

- Loop goes from 1000 to 1.
- Obviously, loop that subtracts 1 takes longer than a loop that divides by 2.
- Loops are doing different things! Right!
- Wrong!
- Same goal for both. Move from 1000 to 1.
- Algorithm is different for each loop, but goal is the same.

So What?

- Consider a common problem.
- Coil a rope.

Coiling Rope: Two Solutions

- Two different approaches (algorithms):
- 1: Loop the rope around your arm, one coil at a time.
- 2: Divide the rope in half again and again.

Coiling Rope: Two Solutions: Effort

- Assume 100 metre rope.
- Each loop in coil about 1 metre.
- 1: *Loop rope around arm*: 100 iterations.
- 2: *Divide rope in half*: 6 or 7 iterations.

Radically Different Efficiencies

- Same goal.
- Different algorithms.
- Different efficiencies.

Nested Loops: Quadratic: Doubling

```
for (int i = 0; i < 1000; i++)
  for (int j = 0; j < 1000; j++)
    // do something
```

Absolute Measure of Efficiency: $f(n) = n^2$

Value of n	Program Effort (Iterations)	Effect of Doubling
1000	1,000,000	
2000	4,000,000	4 times effort
4000	16,000,000	4 times effort
8000	64,000,000	4 times effort
16000	256,000,000	4 times effort
32000	1,024,000,000	4 times effort

Nested Loops: Quadratic: Tripling

```
for (int i = 0; i < 1000; i++)
  for (int j = 0; j < 1000; j++)
    // do something
```

Absolute Measure of Efficiency: $f(n) = n^2$

Value of n	Program Effort (Iterations)	Effect of Tripling
1000	1,000,000	
3000	9,000,000	9 times effort
9000	81,000,000	9 times effort
27000	729,000,000	9 times effort
81000	6,561,000,000	9 times effort

Nested Loops: Quadratic: Quadrupling

```
for (int i = 0; i < 1000; i++)
  for (int j = 0; j < 1000; j++)
    // do something
```

Absolute Measure of Efficiency: $f(n) = n^2$

Value of n	Program Effort (Iterations)	Effect of Quadrupling n
1000	1,000,000	
4000	16,000,000	16 times effort
16000	256,000,000	16 times effort
64000	4,096,000,000	16 times effort

Calculating Efficiency with Nesting

- Each single iteration through outer loop requires complete set of iterations of inner loop.
- Outer loop had efficiency of n.
- Inner loop had efficiency of n.
- Combined efficiency: multiply each nested loop, thus n^2.

Nested Loops: Linear Logarithmic

```
for (int i = 0; i < 1000; i++)
  for (int j = 1000; j >= 1; j = j / 2)
    // do something
```

- Outer loop has efficiency of *n*, since it is a simple counting loop.
- Inner loop has efficiency of $\log_2 n$, since it is a divide by 2 loop.
- Multiply the two efficiencies because of the nesting.

Absolute Measure of Efficiency: $f(n) = n \log_2 n$

Dealing with Increasing Complexity

- Simple examples so far.
- Consider something more complex.
- Dependent Quadratic

```
for (int i = 1; i <= 10; i++)
  for (int j = 1; j <= i; j++)
    // do something
```

Control of inner loop is dependent on value of outer loop variable.

Nested Loops: Dependent Quadratic

```
for (int i = 1; i <= 10; i++)
  for (int j = 1; j <= i; j++)
    // do something
```

$1 + 2 + 3 + . . . + 10 = 55$

Absolute Measure of Efficiency: $f(n) = n ((n + 1) / 2)$

- Need to simplify in order to compare algorithms

The Big-O

- What really matters in measuring efficiency?
- Order of Magnitude

Efficiency: $f(n) = n^2$

Written in Big-O: $O(n^2)$

The Big-O: Simplifying: 1

Commonly Used Big-O Values to Express Algorithm Performance
Ranked from Best (Fastest) to Worst (Slowest)
O(1) O($\log_2 n$) O(n) O($n \log_2 n$) O(n^2) O(n^3) . . . O(n^k) O(2^n) O($n!$)

- In each term, set coefficient of term to 1.
- Keep largest term and discard others.

- How would you simplify Dependent Quadratic?

Absolute Measure of Efficiency: f(n) = n ((n + 1) / 2)

The Big-O: Simplifying: 2

- How would you simplify Dependent Quadratic?
- Expand to express all terms without parentheses.

Absolute Measure of Efficiency: f(n) = n ((n + 1) / 2)

n ((n + 1) / 2)

The Big-O: Simplifying: 3

- How would you simplify Dependent Quadratic?
- Expand to express all terms without parentheses.

Absolute Measure of Efficiency: f(n) = n ((n + 1) / 2)

n ((n + 1) / 2)

n (n / 2 + 1 / 2)

The Big-O: Simplifying: 4

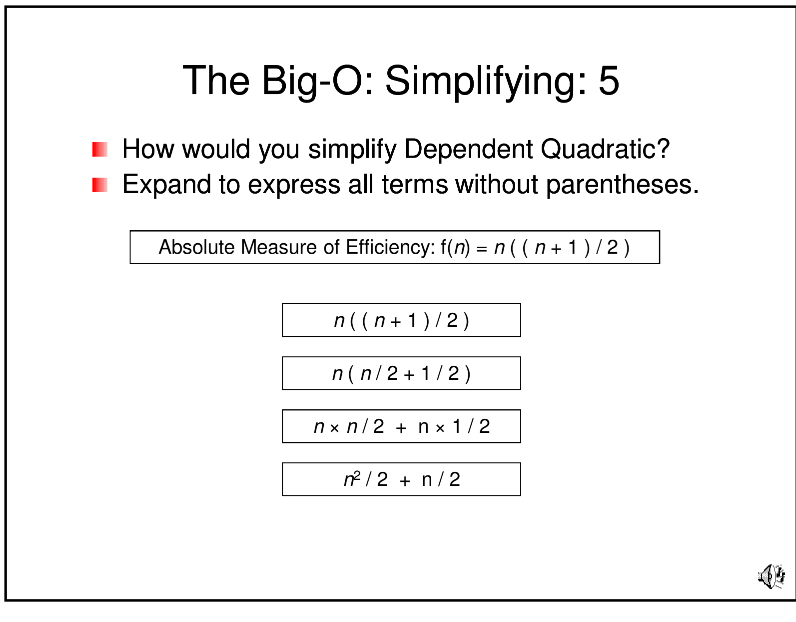
- How would you simplify Dependent Quadratic?
- Expand to express all terms without parentheses.

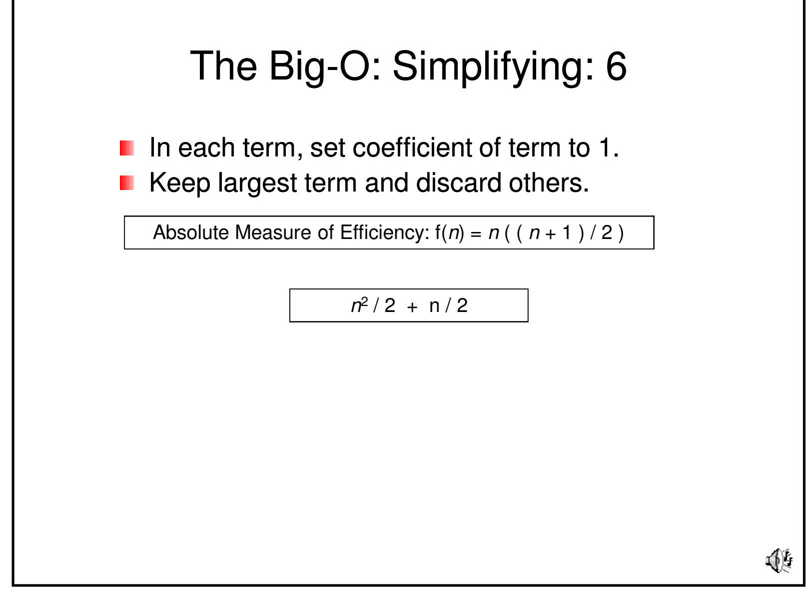
Absolute Measure of Efficiency: f(n) = n ((n + 1) / 2)

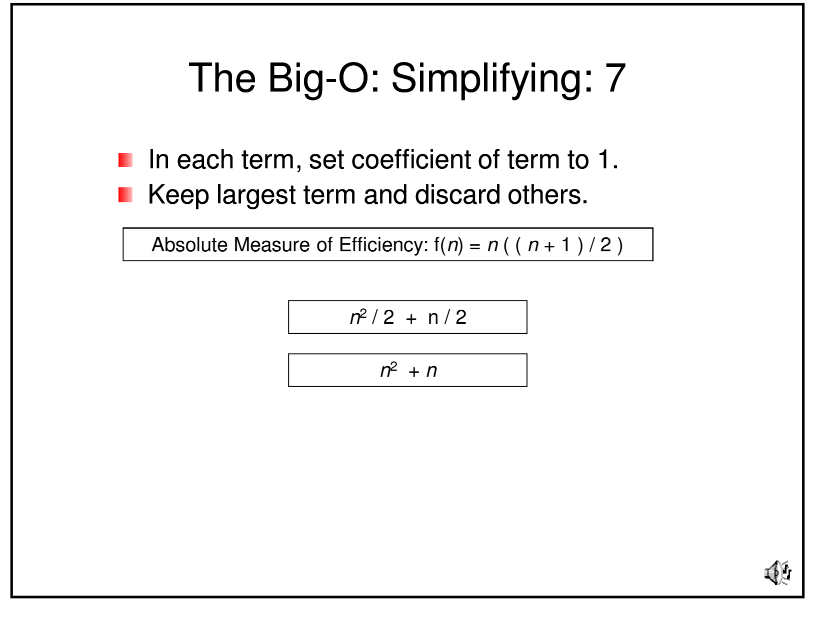
n ((n + 1) / 2)

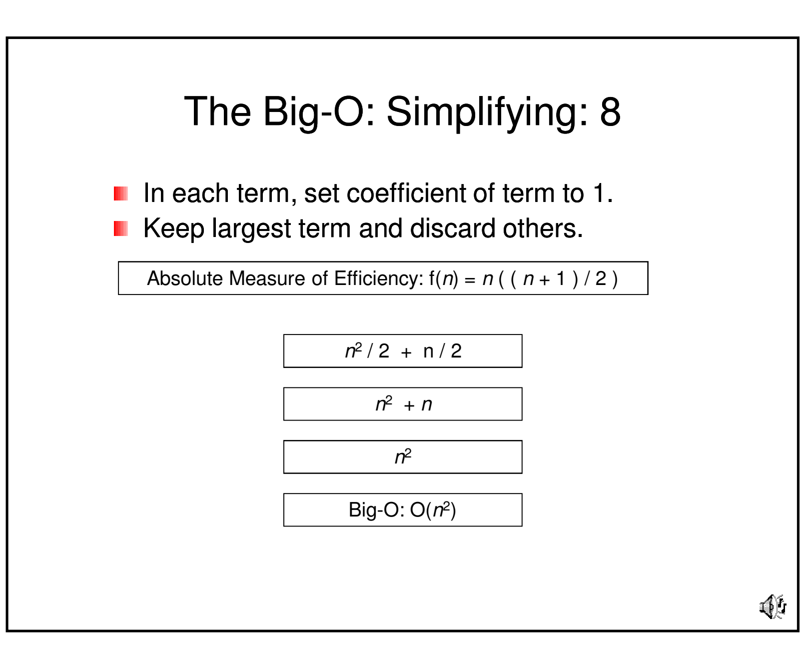
n (n / 2 + 1 / 2)

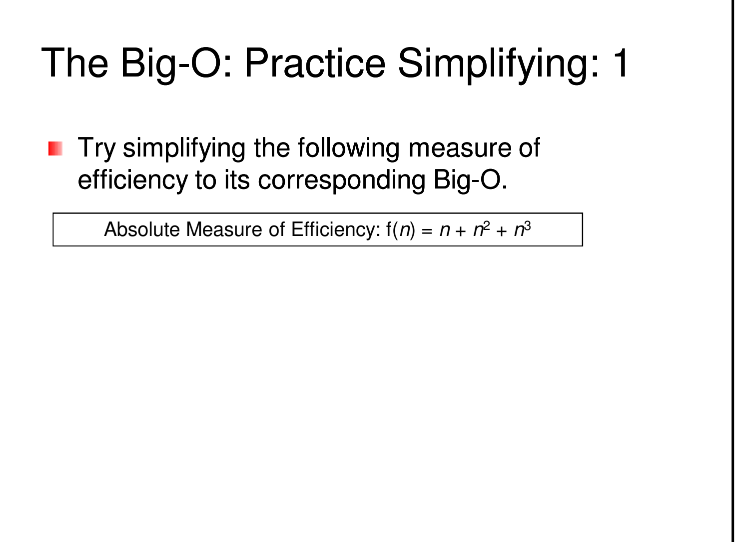
$n \times n / 2 + n \times 1 / 2$

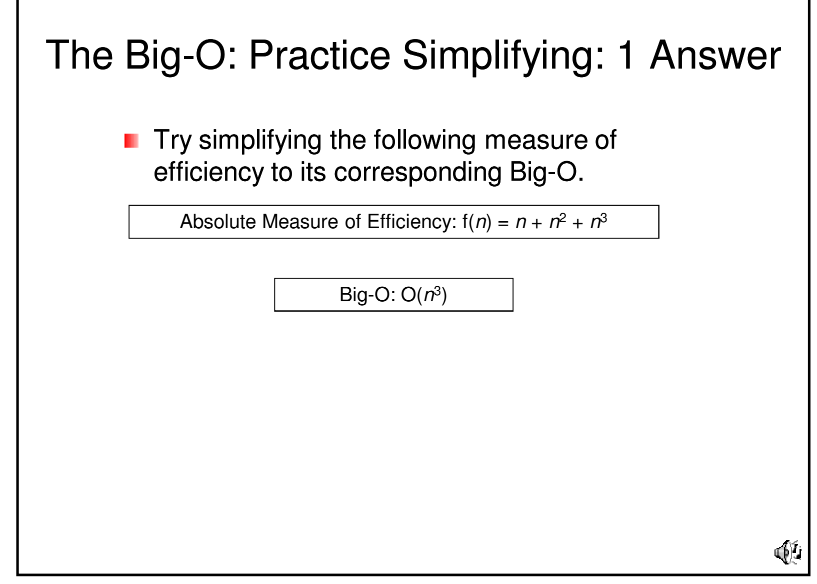

The Big-O: Simplifying: 5
How would you simplify Dependent Quadratic?
Expand to express all terms without parentheses.
Absolute Measure of Efficiency: f(n) = n ((n + 1) / 2)
n ((n + 1) / 2)
n (n / 2 + 1 / 2)
n × n / 2 + n × 1 / 2
n2 / 2 + n / 2

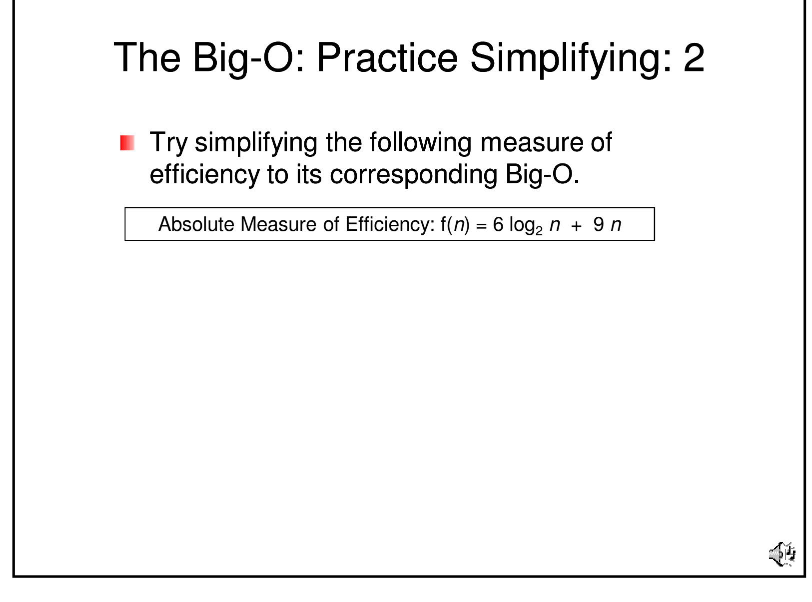

The Big-O: Simplifying: 6
In each term, set coefficient of term to 1.
Keep largest term and discard others.
Absolute Measure of Efficiency: f(n) = n ((n + 1) / 2)
n2 / 2 + n / 2

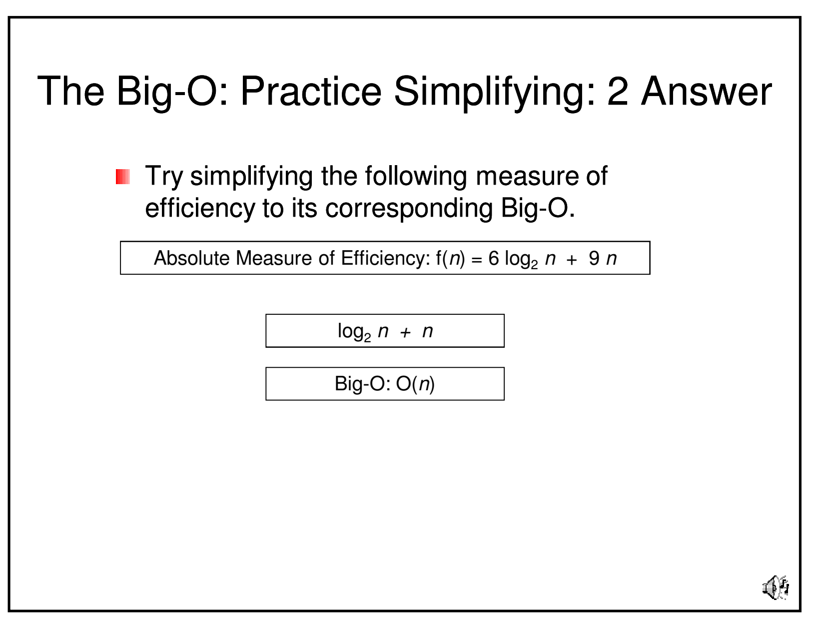

The Big-O: Simplifying: 7
In each term, set coefficient of term to 1.
Keep largest term and discard others.
Absolute Measure of Efficiency: f(n) = n ((n + 1) / 2)
n2 / 2 + n / 2
n2 + n


The Big-O: Simplifying: 8
In each term, set coefficient of term to 1.
Keep largest term and discard others.
Absolute Measure of Efficiency: f(n) = n ((n + 1) / 2)
n2 / 2 + n / 2
n2 + n
n2
Big-O: O(n2)


The Big-O: Practice Simplifying: 1
Try simplifying the following measure of efficiency to its corresponding Big-O.
Absolute Measure of Efficiency: f(n) = $n + n^2 + n^3$


The Big-O: Practice Simplifying: 1 Answer
Try simplifying the following measure of efficiency to its corresponding Big-O.
Absolute Measure of Efficiency: f(n) = $n + n^2 + n^3$
Big-O: O(n^3)


The Big-O: Practice Simplifying: 2
Try simplifying the following measure of efficiency to its corresponding Big-O.
Absolute Measure of Efficiency: f(n) = $6 \log_2 n + 9n$


The Big-O: Practice Simplifying: 2 Answer
Try simplifying the following measure of efficiency to its corresponding Big-O.
Absolute Measure of Efficiency: f(n) = $6 \log_2 n + 9n$
$\log_2 n + n$
Big-O: O(n)

Finding the Big-O: 1

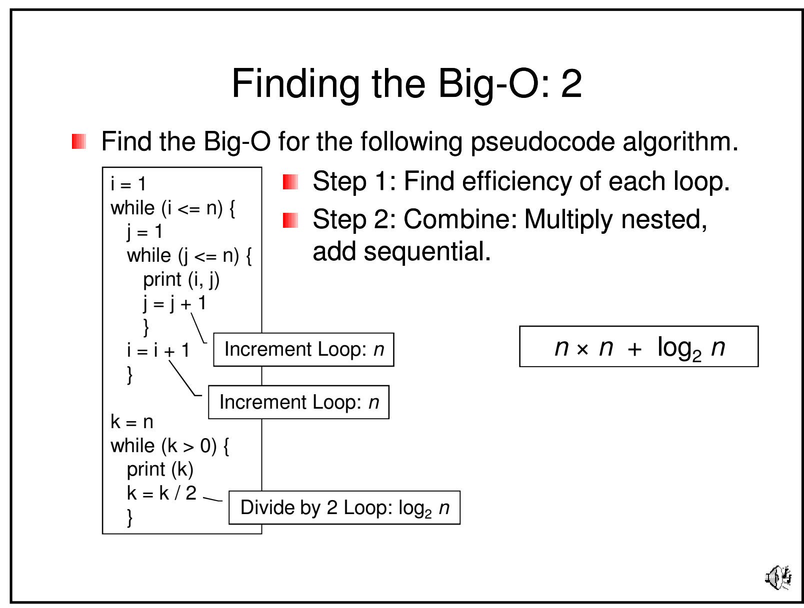
- Find the Big-O for the following pseudocode algorithm.
  - Step 1: Find efficiency of each loop.

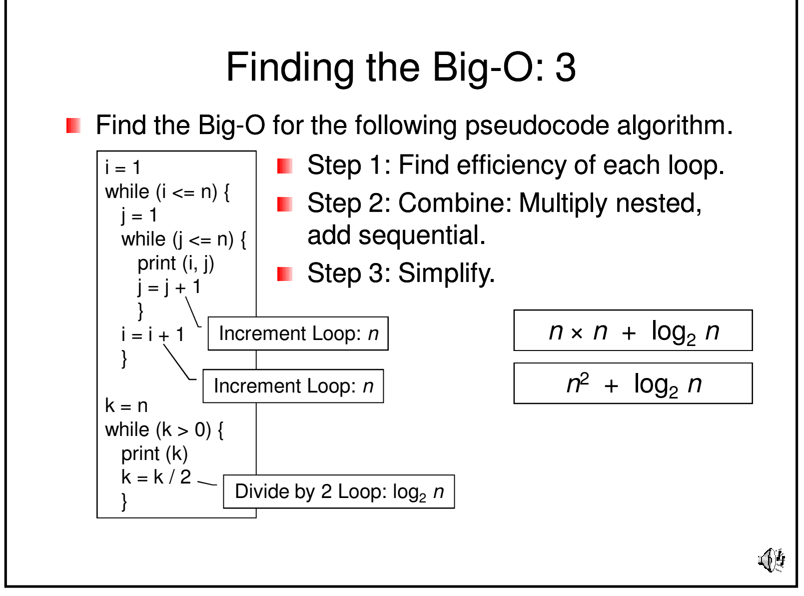
```
i = 1
while (i <= n) {
  j = 1
  while (j <= n) {
    print (i, j)
    j = j + 1
    }
  i = i + 1
  }
k = n
while (k > 0) {
  print (k)
  k = k / 2
  }
```

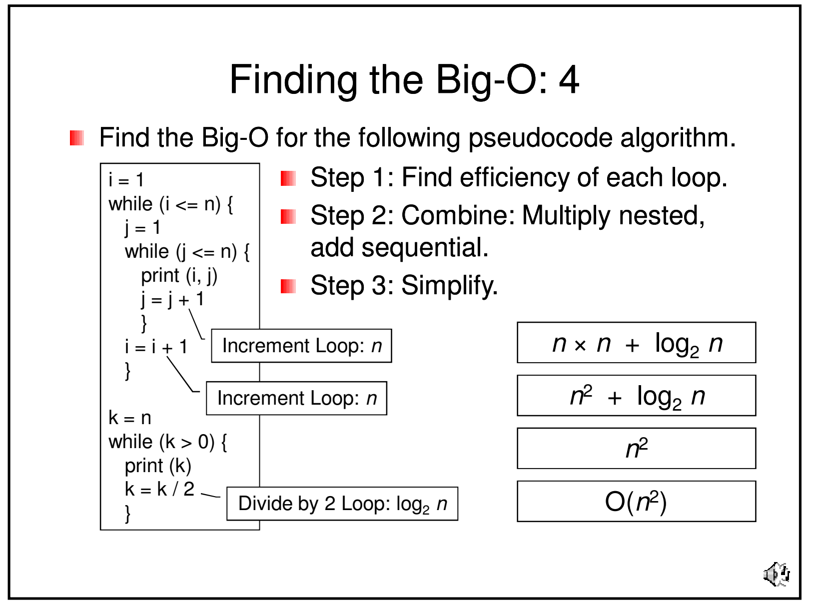
Increment Loop: n

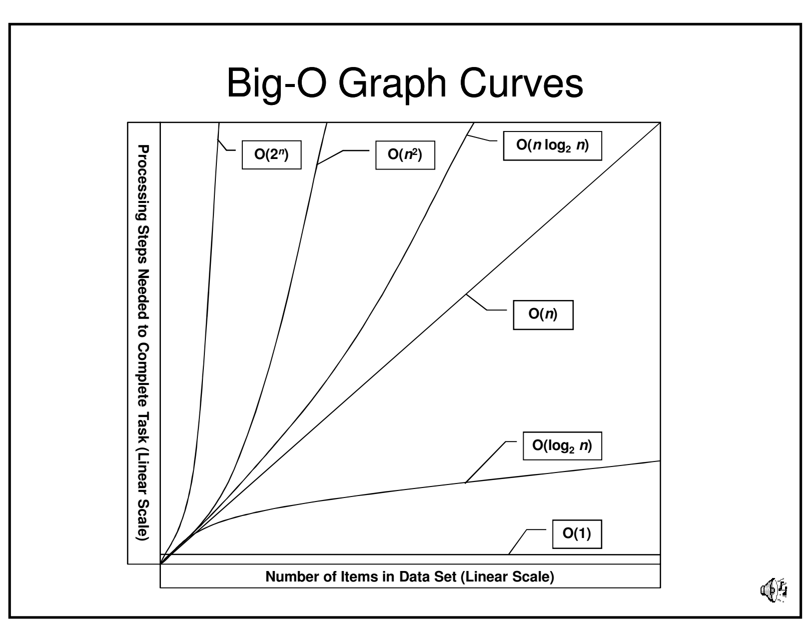
Increment Loop: n

Divide by 2 Loop: $\log_2 n$









Algorithm Efficiency for *n* of 10,000

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log_2 n)$	14	microseconds
Linear	$O(n)$	10,000	0.1 seconds
Linear Logarithmic	$O(n \log_2 n)$	140,000	2 seconds
Quadratic	$O(n^2)$	$10{,}000^2$	15-20 minutes
Polynomial	$O(n^k)$	$10{,}000^k$	hours
Exponential	$O(2^n)$	$2^{10{,}000}$	intractable
Factorial	$O(n!)$	10,000!	intractable

Algorithm Efficiency for *n* of 10,000

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log_2 n)$	14	microseconds
Linear	$O(n)$	10,000	0.1 seconds
Linear Logarithmic	$O(n \log_2 n)$	140,000	2 seconds
Quadratic	$O(n^2)$	$10{,}000^2$	15-20 minutes
Polynomial	$O(n^k)$	$10{,}000^k$	hours
Exponential	$O(2^n)$	$2^{10{,}000}$	intractable
Factorial	$O(n!)$	10,000!	intractable

Exponential case with *n* equal to only 1,000 (not 10,000) would take 3.39e+288 years. The age of the universe is estimated to be 1.5e+10 years.