Algorithm Analysis

COMP 2210 – Dr. Hendrix

AUBURN UNIVERSITY

SAMUEL GINN COLLEGE OF ENGINEERING

Algorithm Analysis

Algorithm analysis is an approach to describing certain efficiency characteristics of an algorithm in terms of certain problem characteristics.

Typically this means describing the time/work or space requirements of an algorithm in terms of its input size.

Linear search on an array of size **N** will require **N** elements to be examined in the worst case.

(problem) input size **work requirements**

Binary search on an array of size **N** will require approximately $\log_2 N$ elements to be examined in the worst case.

Algorithm analysis allows us to **predict** the performance of algorithms and programs, **compare** competing solutions to the same problem, **guarantee** performance from our software, gain **insight** into techniques for designing efficient algorithms, . . .

Performance guarantees

```
public class Arrays extends Object

public static <T> void sort(T[] a, Comparator<? super T> c)
    ...
```

Implementation note: This implementation is a stable, adaptive, iterative mergesort that **requires far fewer than n lg(n) comparisons** when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation **requires approximately n comparisons**. Temporary **storage requirements vary from a small constant** for nearly sorted input arrays **to n/2 object references** for randomly ordered input arrays.

```
public static void sort(int[] a)
    ...
```

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm **offers O(n log(n)) performance** on many data sets that cause other quicksorts to **degrade to quadratic performance**, and is typically faster than traditional (one-pivot) Quicksort implementations.

Performance bugs

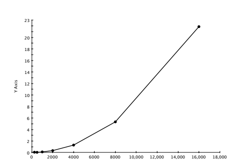
A good understanding of algorithm analysis also allows us to avoid **performance bugs** in our software. A poor understanding of algorithm analysis can lead directly to software that:

- Performs too slowly, especially as input sizes increase
- Consumes too much memory
- Opens security vulnerabilities, like denial of service attacks

Example: The “beck” exploit in the Apache web server.

```
/a///b//c/////d/page.html
  ↓ no2slash(char* name)
/a/b/c/d/page.html
```

Quadratic time complexity



This function required time proportional to the square of the number of ‘/’ characters in the string.

References: http://bit.ly/oEdWvn http://bit.ly/T27uyU

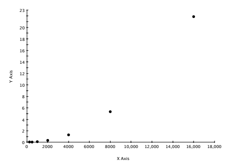
Approaches to algorithm analysis

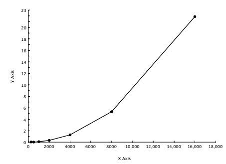
Empirical analysis Analyze running time based on observations and experiments.

Apply the scientific method:

1. Make observations and gather data.
2. Hypothesize an explanation for the observations.
3. Make predictions based on the hypothesis.
4. Formulate an experiment to see if the predictions occur.
5. Analyze experimental results to corroborate or falsify the hypothesis.

N	Elapsed Time	Ratio
250	0.061	N/A
500	0.042	1.35
1000	0.112	2.67
2000	0.340	3.04
4000	1.298	3.82
8000	5.334	4.11
16000	21.88	4.10





~N²

Approaches to algorithm analysis

Mathematical analysis Develop a cost model that includes costs for individual operations.

Computation model: Simple one-processor random-access machine with non-hierachical memory

Total running time:

$$\sum (c_i \times f_i)$$

$$T(N) = c_1 N + c_2(N-1) + c_3(N-1) + c_4 \sum_{j=2}^{N} t_j + c_5 \sum_{j=2}^{N} (t_j - 1)$$

$$T(N) = c_1 N + c_2(N-1) + c_3(N-1) + c_4\left(\frac{N(N+1)}{2} - 1\right) + c_5\left(\frac{N(N-1)}{2}\right)$$

$$T(N) \propto aN^2 + bN + c$$

~N²

Empirical analysis

Make observations: Run the program multiple times, systematically increasing the input size for each run.

Gather data: Time each program run and record the elapsed time along with the associated input size.

Suggestion: Since many of the programs we will deal with have running time proportional to N^k where N is the problem size, it's often useful to successively double the input size and look at the ratio of the elapsed time of successive program runs.

$$T(N) \propto N^k$$

$$\frac{T(2N_i)}{T(N_i)} \propto \frac{(2N_i)^k}{N_i^k} = \frac{2^k N_i^k}{N_i^k} = 2^k$$

N	T(N)	Ratio
N	T(N)	-
2N	T(2N)	T(2N)/T(N)
4N	T(4N)	T(4N)/T(2N)
8N	T(8N)	T(8N)/T(4N)
. . .	. . .	. . .

Ratio approaches a constant: 2^k

Empirical analysis

Example: Experimental analysis of the beck exploit in Apache.

Observations

N	Elapsed Time	Ratio
250	0.061	N/A
500	0.042	1.35
1000	0.112	2.67
2000	0.340	3.04
4000	1.298	3.82
8000	5.334	4.11
16000	21.88	4.10

Since we corroborated our hypothesis, we can say that the ratio will converge to ~4 = 2^k = N^k. So, running time is growing in proportion to N^2.

Hypothesis

As N doubles, running time is increasing by a factor of 4.

Prediction

For N = 32000, elapsed time will be ~87.52 seconds and for N = 64000, elapsed time will be ~350.08 seconds.

Experimentally test hypothesis

N	Elapsed Time	Ratio
32000	85.763	3.92
64000	345.634	4.03

Participation

Q: Given the following table of timing data, what is the most reasonable conclusion regarding the time complexity of the program?

Run	N	Time	Ratio	lg Ratio
0	8	0.00		
1	16	0.01	5.50	2.46
2	32	0.02	1.36	0.45
3	64	0.03	1.87	0.90
4	128	0.14	5.04	2.33
5	256	1.01	7.19	2.85
6	512	8.11	8.00	3.00
7	1024	65.75	8.10	3.02
8	2048	521.65	7.93	2.99

A. The program requires time proportional to N^2
B. The program requires time proportional to N^3
C. The program requires time proportional to N^4
D. The program requires time proportional to N^8

Mathematical analysis

Mathematical analysis Develop a cost model that includes costs for individual operations.

Computation model: Simple one-processor random-access machine with non-hierachical memory

Total running time = $\sum (c_i \times f_i)$

cost of executing operation i
(a property of the underlying system)

frequency of execution of operation i
(a property of the algorithm)

Mathematical analysis

Don Knuth is the “father” and pioneer of algorithm analysis. TAOCP is requisite bookshelf material.

Mathematical analysis

Total running time = $\sum (c_i \times f_i)$

cost of executing operation i
(a property of the underlying system)

frequency of execution of operation i
(a property of the algorithm)

```
public int sumA(int N) {
   int sum;
   sum = N*(N+1)/2;
   return sum;
}
```

Operation	Cost (c_i)	Freq (f_i)
variable declaration	1.5 ns	1
assignment	1.7 ns	1
int addition	2.1 ns	1
int multiplication	2.5 ns	1
int division	5.4 ns	1
return statement	2.0 ns	1

Varies from system to system. Hard to measure precisely.

Running time of sumA is 15.2 ns

Not entirely useful information

Abstraction decision #1:

Treat the cost of primitive operations and simple statements as some unspecified constant.

Running time of sumA is constant.

(Same for any value of N.)

Mathematical analysis

Total running time = $\sum(c_i \times f_i)$

cost of executing operation i (a property of the underlying system)

frequency of execution of operation i (a property of the algorithm)

```
public int sumB(int N) {
  int sum = 0;
  for (int i = 1; i <= N; i++) {
    sum = sum + i;
  }
  return sum;
}
```

Operation	Cost (c_i)	Freq (f_i)
sum = sum + i	c	N

The running time of sumB is c*N.

Running time of sumB is linear.

(Increases in direct proportion to N.)

Abstraction decision #2:

Focus only on "core" operations instead of counting every single operation that is performed.

Mathematical analysis

Total running time = $\boxed{\sum(c_i \times f_i)}$

cost of executing operation i (a property of the underlying system)

frequency of execution of operation i (a property of the algorithm)

```
public int sumC(int N) {
  int sum = 0;
  for (int i = 1; i <= N; i++) {
     for (int j = 1; j <= i; j++ {
       sum = sum + 1;
     }
  }
  return sum;
}
```

Operation	Cost (c_i)	Freq (f_i)
sum = sum + i	c	N(N+1)/2

The running time of sumC is $(c/2)(N^2 + N)$

Running time of sumC is quadratic.

(Increases in direct proportion to N^2.)

Abstraction decision #3:

Focus only on the highest-order term and ignore coefficients, constants, and low-order terms.

We are characterizing time complexity by the running time function's **growth rate**.

(more to come ...)

Mathematical analysis

```
public int sumA(int N) {
   int sum;
   sum = N*(N+1)/2;
   return sum;
}
```

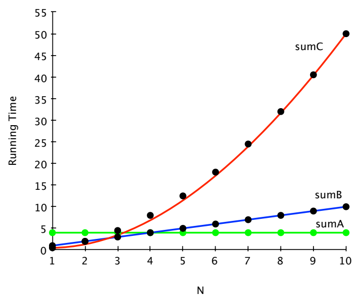
Performs a **constant** amount of work.

```
public int sumB(int N) {
  int sum = 0;
  for (int i = 1; i <= N; i++) {
     sum = sum + i;
  }
  return sum;
}
```

Performs a **linear** (~N) amount of work.

```
public int sumC(int N) {
  int sum = 0;
  for (int i = 1; i <= N; i++) {
     for (int j = 1; j <= i; j++ {
        sum = sum + 1;
     }
  }
  return sum;
}
```

Performs a **quadratic** (~N^2) amount of work.



Analysis of binary search

Total running time = $\sum(c_i \times f_i)$

cost of executing operation i

frequency of execution of operation i

```
public int search(int[] a, int target) {
   int left = 0, right = a.length-1;
   while (left <= right) {
      int middle = (left + right) / 2;
      if (target < a[middle])
         right = middle - 1;
      else if (target > a[middle])
         left = middle + 1;
      else
         return middle;
   }
   return -1;
}
```

Worst case analysis

As a function of the array size (N), what is the maximum number of times that this statement could execute?

Analysis of binary search

```
public int search(int[] a, int target) {
   int left = 0, right = a.length-1;
   while (left <= right) {
      int middle = (left + right) / 2;
      if (target < a[middle])
         right = middle - 1;
      else if (target > a[middle])
         left = middle + 1;
      else
         return middle;
   }
   return -1;
}
```

Worst case analysis

target = 25

statement executions = 4

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

COMP 2210 • Dr. Hendrix • 17

Analysis of binary search

```
public int search(int[] a, int target) {
   int left = 0, right = a.length-1;
   while (left <= right) {
      int middle = (left + right) / 2;
      if (target < a[middle])
         right = middle - 1;
      else if (target > a[middle])
         left = middle + 1;
      else
         return middle;
   }
   return -1;
}
```

Worst case analysis

target = 1

statement executions = 3

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

COMP 2210 • Dr. Hendrix • 18

Analysis of binary search

```
public int search(int[] a, int target) {
   int left = 0, right = a.length-1;
   while (left <= right) {
      int middle = (left + right) / 2;
      if (target < a[middle])
         right = middle – 1;
      else if (target > a[middle])
         left = middle + 1;
      else
         return middle;
   }
   return -1;
}
```

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

Worst case analysis

target = 13

statement executions = 3

Analysis of binary search

```
public int search(int[] a, int target) {
   int left = 0, right = a.length-1;
   while (left <= right) {
      int middle = (left + right) / 2;
      if (target < a[middle])
         right = middle – 1;
      else if (target > a[middle])
         left = middle + 1;
      else
         return middle;
   }
   return -1;
}
```

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

Worst case analysis

target = 15

statement executions = 4

Analysis of binary search

```
public int search(int[] a, int target) {
   int left = 0, right = a.length-1;
   while (left <= right) {
      int middle = (left + right) / 2;
      if (target < a[middle])
         right = middle - 1;
      else if (target > a[middle])
         left = middle + 1;
      else
         return middle;
   }
   return -1;
}
```

Worst case analysis

target = **value not in array**

statement executions = **$\sim\log_2 N$**

$N = N/2^0$

$N/2 = N/2^1$ *after 1 operation*

$N/4 = N/2^2$ *after 2 operations*

$N/8 = N/2^3$ *after 3 operations*

After k operations…

$N/2^k$

1

$N/2^k = 1$

$N = 2^k$

$k = \log_2 N$

Binary search requires time proportional to $\log_2$N in the worst case.

Analysis of binary search

```
public int search(int[] a, int target) {
   int left = 0, right = a.length-1;
   while (left <= right) {
      int middle = (left + right) / 2;
      if (target < a[middle])
         right = middle - 1;
      else if (target > a[middle])
         left = middle + 1;
      else
         return middle;
   }
   return -1;
}
```

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

Worst case analysis

target = 13

statement executions = 3 ⟵ $\log_2(12) = 3.58$

Analysis of binary search

```
public int search(int[] a, int target) {
   int left = 0, right = a.length-1;
   while (left <= right) {
      int middle = (left + right) / 2;
      if (target < a[middle])
         right = middle – 1;
      else if (target > a[middle])
         left = middle + 1;
      else
         return middle;
   }
   return -1;
}
```

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

Worst case analysis

target = 15

statement executions = 4

$\log_2(12) = 3.58$

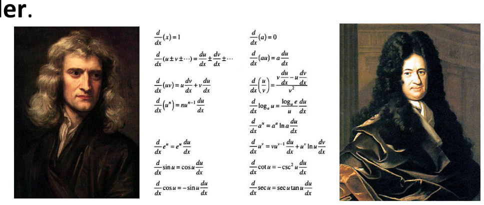
COMP 2210 • Dr. Hendrix • 23

Growth rate

The **growth rate** of a time complexity function is a measure of how the amount of work an algorithm does changes as its input (problem) size changes.

The growth rate of a function is sometimes called its **order**.

Without turning to calculus, a useful way of thinking about growth rate for a time complexity function T(N) is to think about the change in T(N) as N doubles.



Quadratic growth rate

N	N^2	Ratio	$(N^2+N)/2$	Ratio	$15N^2+50N+500$	Ratio
64	4096		2080		65140	
128	16384	4	8256	3.96	252660	3.87
256	65536	4	32896	3.98	996340	3.94
512	262144	4	131328	3.99	3958260	3.97
1024	1048576	4	524800	3.99	15780340	3.98

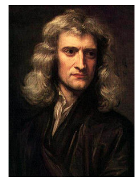
COMP 2210 • Dr. Hendrix • 24

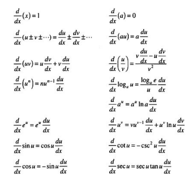
Growth rate

The **growth rate** of a time complexity function is a measure of how the amount of work an algorithm does changes as its input (problem) size changes.

The growth rate of a function is sometimes called its **order**.

Without turning to calculus, a useful way of thinking about growth rate for a time complexity function T(N) is to think about the change in T(N) as N doubles.





Logarithmic growth rate

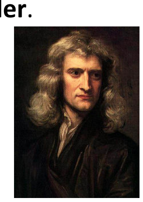
N	$\log_2 N$	Ratio	$\log_3 N$	Ratio	$\log_{10} N$	Ratio
64	6		3.79		1.81	
128	7	1.17	4.42	1.17	2.12	1.17
256	8	1.14	5.05	1.14	2.41	1.14
512	9	1.13	5.68	1.13	2.71	1.13
1024	10	1.11	6.31	1.11	3.01	1.11

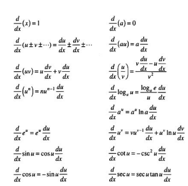
Growth rate

The **growth rate** of a time complexity function is a measure of how the amount of work an algorithm does changes as its input (problem) size changes.

The growth rate of a function is sometimes called its **order**.

Without turning to calculus, a useful way of thinking about growth rate for a time complexity function T(N) is to think about the change in T(N) as N doubles.





NlogN growth rate

N	Nlo_2N	Ratio	$5N\log_2 N + N$	Ratio	$15\log_2 N + N + 50$	Ratio
64	384		1984		5874	
128	896	2.33	4608	2.32	13618	2.37
256	2048	2.29	10496	2.28	31026	2.28
512	4608	2.25	23552	2.24	69682	2.25
1024	10240	2.22	52224	2.22	154674	2.22

Big-Oh notation

We will describe the growth rate of an algorithm's time complexity function in terms of **big-Oh**.

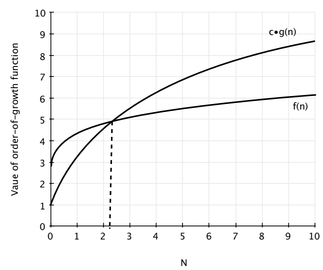
We will make statements like this:

The time complexity of this algorithm is O(N²).

A function that describes the running time in terms of the problem size (N).

Read "order of N²"

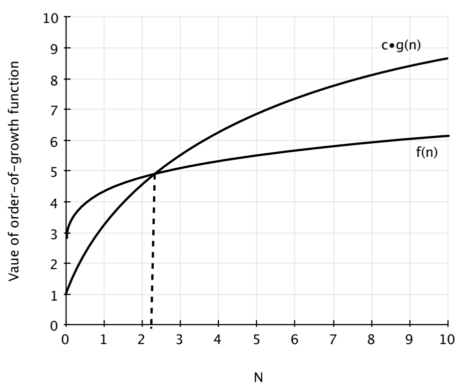
Big-Oh describes an upper bound on growth rate.



COMP 2210 • Dr. Hendrix • 27

Big-Oh notation

Let f(n) and g(n) be functions defined on the nonnegative integers. We say "f(n) is O(g(n))" if and only if there exists a nonnegative integer n_0 and a constant c > 0 such that for all integers $n \geq n_0$ we have $f(n) \leq c \cdot g(n)$.



Example:

$f(N) = N^2 + 2N + 1$ is $O(N^2)$ since for $N \geq 1$ $f(N) \leq 4N^2$.

Since big-Oh describes an upper bound, it would be technically correct to say that f(N) in the above example is $O(N^3)$.

That is improper use of the notation, however. We always want to state big-Oh in terms of the function with the slowest growth rate that satisfies the definition.

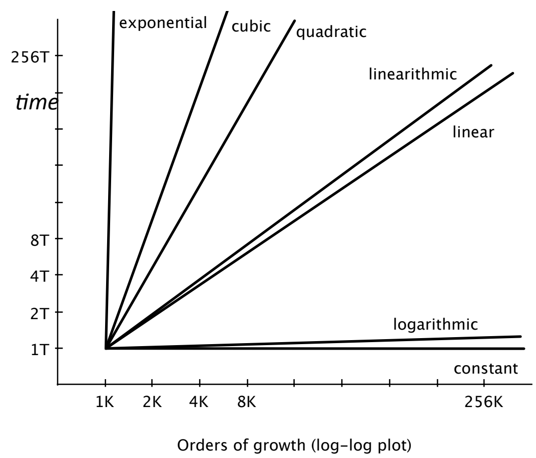
Just use the fastest growing term in the function and drop any coefficients it may have.

COMP 2210 • Dr. Hendrix • 28

Common orders of growth

A small set of functions describe order-of-growth for typical algorithms.

$$1, \quad \log N, \quad N, \quad N \log N, \quad N^2, \quad N^3, \quad 2^N$$



Practical implications

Assume a machine executing at one billion instructions per second.

	1,000	10,000	100,000	1,000,000	10,000,000
O(log N)	< 1 ns	10 ns	132 ns	166 ns	199 ns
O(N)	1 ns	10 ns	100 ns	1 ms	10 ms
O(NlogN)	10 ns	100 ns	2 ms	20 ms	0.2 sec
$O(N^2)$	1 ms	0.1 sec	10 sec	17 min	28 hours
$O(N^3)$	1 sec	17 min	12 days	32 yrs	32,000 yrs
$O(N^4)$	17 min	4 months	3,200 yrs	3.2 M yrs	3.17E15 yrs
$O(2^N)$	3.4E284 yrs	??	??	??	??
O(N!)	??	??	??	??	??

Reasonable for most inputs

Enables new technologies.

(e.g., FFT)

Unreasonable for all inputs

Big-Oh analysis

Big-Oh analysis refers to analyzing an algorithm or program and expressing its running time in terms of big-Oh.

Types of analysis

Best case – Provides a lower bound on cost. Defined by the “easiest” input. Provides a goal for other inputs/cases.

Average case – Provides an expectation of cost. Typically defined by a model for random input. Provides a way to predict performance.

Worst case – Provides an upper bound on cost. Defined by the “most difficult” input. Provides a guarantee on performance.

We will almost always perform a worst case analysis.

Calculating worst case big-Oh

We will use a simple syntax-based approach to calculating worst-case big-Oh.

1. All simple statements and primitive operations have constant cost.
2. The cost of a sequence of statements is the sum of the costs of each individual statement.
3. The cost of a selection statement is the cost of the most expensive branch.
4. The cost of a loop is the cost of the body multiplied by the by the maximum number of iterations that the loop makes.

We won’t formally address the analysis of recursive algorithms until COMP 3270.

Example: loops

```
for (int i = 0; i < N; i++)
{
    a = a + i;
    b++;
    System.out.println(a, b);
}
```

N iterations

constant cost

Cost of the loop = cost of the body x number of iterations

= constant x N

= ~cN

O(N)

Example: loops

```
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
    {
        System.out.println(i*j);
    }

}
```

N iterations

N iterations

constant cost

Cost of the i loop = (cost of the j loop body x number of j loop iterations) x number of i loop iterations

= (constant x N) x N

= ~cN^2

O(N^2)

Example: loops

```
for (int i = 0; i < N; i++)
{
    for (int j = i; j < N; j++)
    {
        System.out.println(i*j);
    }

}
```

N iterations

N iterations (max)

constant cost

Cost of the i loop = (cost of the j loop body x number of j loop iterations) x number of i loop iterations

= (constant x N) x N

= $\sim cN^2$

$O(N^2)$

Example: sequence

c_1

```
int count = 0;
int k = n;
```

$c_2 \bullet \log_2 n$

```
while (k > 1) {
   k = k / 2;
   count++;
}
```

$c_3 \bullet n^2$

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      System.out.println(i*j);
   }
}
```

$c_4 \bullet n$

```
for (int i = 0; i < n; i++) {
   count = count + i;
}
```

$T(n) = \sim c_3 \bullet n^2 + c_4 \bullet n + c_2 \bullet \log_2 n + c_1$

$O(N^2)$

Example: selection

$c_1 \bullet \log_2 n$

$c_2 \bullet n$

```
if (a < b) {

   while (k > 1) {
      k = k / 2;
      count++;
   }

}
else {

   for (int i = 0; i < n; i++) {
      count = count + i;
   }

}
```

O(N)

COMP 2210 • Dr. Hendrix • 37

Participation

Q: What is the big-Oh time complexity of the following code?

```
int count = 0;
for (int i = 1; i <= n; i++) {
   int j = 1;
   while (j < n) {
      for (int k = 1; k < n; k = k + 2) {
         count++;
      }
      j = j * 2;
   }
}
```

A. O(NlogN)
B. O(N^2logN)
C. O(N^2)
D. O(N^3)

COMP 2210 • Dr. Hendrix • 38

Examples

```
int count = 0;
for (int i = 1; i <= n; i++)
   for (int j = n; j > 1; j--)
      for (int k = 1; k < n; k = k + 2)
         count++;
```

```
int count = 0;
for (int i = 1; i <= n; i++)
   for (int j = n; j > 1; j--)
      for (int k = 1; k < 1000; k = k + 2)
         count++;
```

```
int count = 0;
for (int i = 1; i <= 1000000; i++)
   for (int j = i; j > 500; j--)
      for (int k = 1; k < 10500; k = k + 2)
         count++;
```


Examples

```
int count = 0;
int j = 1;
for (int i = 1; i < n; i++) {
   while (j < n) {
      j++;
      count++;
   }
   j = 1;
 }
```

```
int count = 0;
int i = n;
while (i > 1) {
   count++;
   i = i / 2;
}
```

Examples

```
int count = 0;
int i = 1;
while (i < n) {
   count++;
   i = i * 2;
}
```

```
int count = 0;
for (int i = 1; i <= n; i++) {
   int j = n;
   while (j > 1) {
      for (int k = 1; k < n; k = k + 2) {
         count++;
      }
      j = j / 2;
   }
}
```