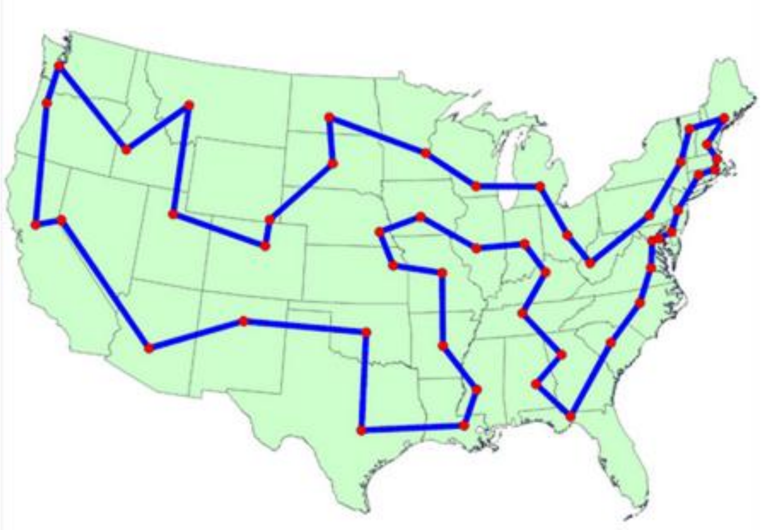


Lecture 2

Introduction to Asymptotic Analysis

Goal: Measuring Code Efficiency

Goal: Measuring Code Efficiency

Intuitive Runtime Characterizations

- Clock Time
- Exact Operation Counting
- Exact Count Exercise

Asymptotic Analysis

- Why Scaling Matters
- Computing Worst Case Order of Growth (Tedious Approach)
- Computing Worst Case Order of Growth (Simplified Approach)

Asymptotic Notation

- Big Theta (a.k.a. Order of Growth)
- Big O and Big Omega

Writing Efficient Programs

An engineer will do for a dime what any fool will do for a dollar.

Efficiency comes in two flavors:

- Programming cost.
  - How long does it take to develop your programs?
  - How easy is it to read, modify, and maintain your code?
    - More important than you might think!
    - Majority of cost is in maintenance, not development!
- Execution cost (from today until end of course).
  - How much time does your program take to execute?
  - How much memory does your program require?

Example of Algorithm Cost

Objective: Determine if a sorted array contains any duplicates.

- Given sorted array A, are there indices i != j where A[i] == A[j]?

-3	-1	2	4	4	8	10	12

Example of Algorithm Cost

Objective: Determine if a sorted array contains any duplicates.

- Given sorted array A, are there indices i != j where A[i] == A[j]?

-3	-1	2	4	4	8	10	12

Silly algorithm: Consider every possible pair, returning true if any match.

- Are (-3, -1) the same? Are (-3, 2) the same? ...

Better algorithm?

Example of Algorithm Cost

Objective: Determine if a sorted array contains any duplicates.

- Given sorted array A, are there indices i != j where A[i] == A[j]?

-3	-1	2	4	4	8	10	12

Silly algorithm: Consider every possible pair, returning true if any match.

- Are (-3, -1) the same? Are (-3, 2) the same? ...

Today's goal: Introduce formal technique for comparing algorithmic efficiency.

Better algorithm?

- For each number A[i], look at A[i+1], and return true the first time you see a match. If you run out of items, return false.

Clock Time

Goal: Measuring Code Efficiency

Intuitive Runtime Characterizations

- **Clock Time**
- Exact Operation Counting
- Exact Count Exercise

Asymptotic Analysis

- Why Scaling Matters
- Computing Worst Case Order of Growth (Tedious Approach)
- Computing Worst Case Order of Growth (Simplified Approach)

Asymptotic Notation

- Big Theta (a.k.a. Order of Growth)
- Big O and Big Omega

How Do I Runtime Characterization?

Our goal is to somehow **characterize the runtimes** of the functions below.

- Characterization should be **simple** and **mathematically rigorous**.
- Characterization should **demonstrate superiority** of dup2 over dup1.

```
public static boolean dup1(int[] A) {
  for (int i = 0; i < A.length; i += 1) {
    for (int j = i + 1; j < A.length; j += 1) {
      if (A[i] == A[j]) {
         return true;
      }
    }
  }
  return false;
}
```

dup1

dup2

```
public static boolean dup2(int[] A) {
  for (int i = 0; i < A.length - 1; i += 1) {
    if (A[i] == A[i + 1]) {
      return true;
    }
  }
  return false;
}
```

Techniques for Measuring Computational Cost

Technique 1: Measure execution time in seconds using a client program.

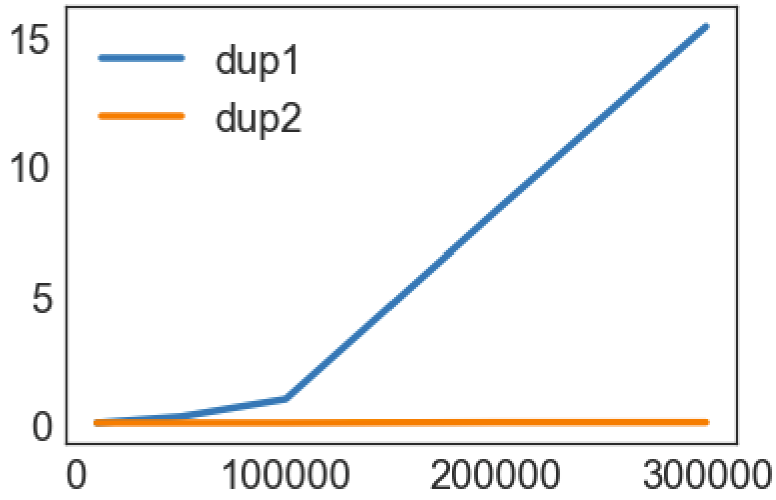
- Tools:
  - Physical stopwatch.
  - Unix has a built in `time` command that measures execution time.
  - Princeton Standard library has a `Stopwatch` class.


```java
public static void main(String[] args) {
  int N = Integer.parseInt(args[0]);
  int[] A = makeArray(N);
  dup1(A);
}
```

Time Measurements for `dup1` and `dup2`

N	dup1	dup2
10000	0.08	0.08
50000	0.32	0.08
100000	1.00	0.08
200000	8.26	0.1
400000	15.4	0.1

Time to complete (in seconds)



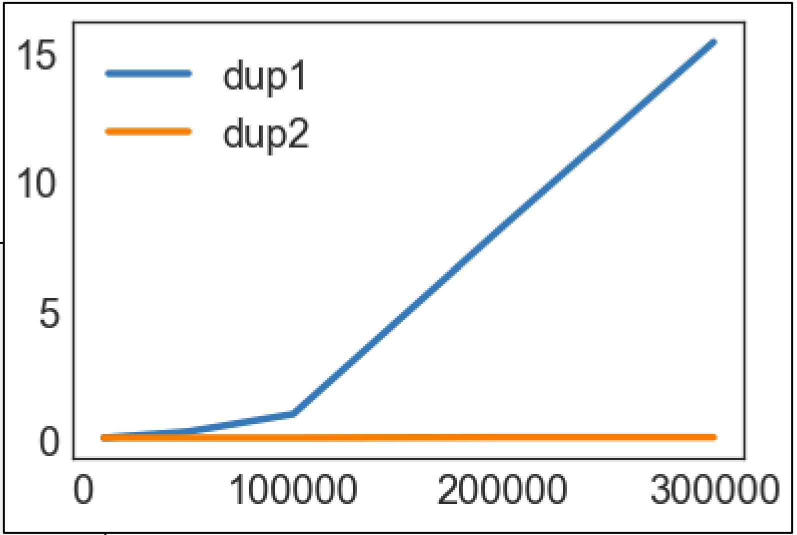
Techniques for Measuring Computational Cost

Technique 1: Measure execution time in seconds using a client program.

- Good: Easy to measure, meaning is obvious.
- Bad: May require large amounts of computation time. Result varies with machine, compiler, input data, programming language, etc.

Interesting observation: If you double the size of the input, dup1 takes ~4x longer, while dup2 takes ~2x longer. True regardless of language and machine.

```java
public static void main(String[] args) {
  int N = Integer.parseInt(args[0]);
  int[] A = makeArray(N);
  dup1(A);
}
```



Exact Operation Counting

Technique 2A: Count possible operations for an array of size N = 10,000.

- Good: Machine independent. Input dependence captured in model.
- Bad: Tedious to compute. Array size was arbitrary. Doesn't tell you actual time.

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

operation	count, N=10000
i = 0	1
j = i + 1	1 to 10000
less than (<)	2 to 50,015,001
increment (+=1)	0 to 50,005,000
equals (==)	1 to 49,995,000
array accesses	2 to 99,990,000

The counts are tricky to compute. Work not shown.

Technique 2B: Count possible operations in terms of input array size N.

- Good: Machine independent. Input dependence captured in model. Tells you how algorithm **scales**.
- Bad: Even more tedious to compute. Doesn't tell you actual time.

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j<A.length; j += 1)
{
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

operation	symbolic count	count, N=10000
i = 0	1	1
j = i + 1	1 to N	1 to 10000
less than (<)	2 to $(N^2+3N+2)/2$	2 to 50,015,001
increment (+=1)	0 to $(N^2+N)/2$	0 to 50,005,000
equals (==)	1 to $(N^2-N)/2$	1 to 49,995,000
array accesses	2 to N^2-N	2 to 99,990,000

Exact Count Exercise

Techniques for Measuring Computational Cost [dup2]

Your turn: Try to come up with rough estimates for the symbolic and exact counts for at least one of the operations.

- Tip: Don't worry about being off by one. Just try to predict the rough magnitudes of each.

```
for (int i = 0; i < A.length - 1; i += 1){
  if (A[i] == A[i + 1]) {
    return true;
  }
}
return false;
```

operation	sym. count	count, N=10000
i = 0	1	1
less than (<)		
increment (+=1)		
equals (==)		
array accesses		

Your turn: Try to come up with rough estimates for the symbolic and exact counts for at least one of the operations.

```
for (int i = 0; i < A.length - 1; i += 1) {
  if (A[i] == A[i + 1]) {
    return true;
  }
}
return false;
```

operation	symbolic count	count, N=10000
i = 0	1	1
less than (<)	1 to N	0 to 10000
increment (+=1)	0 to N - 1	0 to 9999
equals (==)	1 to N - 1	1 to 9999
array accesses	2 to 2N - 2	2 to 19998

Especially observant folks may notice we didn't count everything, e.g. "- 1" and "+ 1" operations. We'll see why this omission is not a problem very shortly.

If you did this exercise but were off by one, that's fine. The exact numbers aren't that important.

Why Scaling Matters

Goal: Measuring Code Efficiency

Intuitive Runtime Characterizations

- Clock Time
- Exact Operation Counting
- Exact Count Exercise

Asymptotic Analysis

- **Why Scaling Matters**
- Computing Worst Case Order of Growth (Tedious Approach)
- Computing Worst Case Order of Growth (Simplified Approach)

Asymptotic Notation

- Big Theta (a.k.a. Order of Growth)
- Big O and Big Omega

Comparing Algorithms

Which algorithm is better? Why?

operation	symbolic count	count, N=10000
i = 0	1	1
j = i + 1	1 to N	1 to 10000
less than (<)	2 to $(N^2+3N+2)/2$	2 to 50,015,001
increment (+=1)	0 to $(N^2+N)/2$	0 to 50,005,000
equals (==)	1 to $(N^2-N)/2$	1 to 49,995,000
array accesses	2 to N^2-N	2 to 99,990,000

dup1

operation	symbolic count	count, N=10000
i = 0	1	1
less than (<)	0 to N	0 to 10000
increment (+=1)	0 to N - 1	0 to 9999
equals (==)	1 to N - 1	1 to 9999
array accesses	2 to 2N - 2	2 to 19998

dup2

Comparing Algorithms

Which algorithm is better? dup2. Why?

- Fewer operations to do the same work [e.g. 50,015,001 vs. 10000 operations].
- Better answer: Algorithm **scales better** in the worst case. $(N^2+3N+2)/2$ vs. N.
- Even better answer: Parabolas (N^2) grow faster than lines (N).

operation	symbolic count	count, N=10000
i = 0	1	1
j = i + 1	1 to N	1 to 10000
less than (<)	2 to $(N^2+3N+2)/2$	2 to 50,015,001
increment (+=1)	0 to $(N^2+N)/2$	0 to 50,005,000
equals (==)	1 to $(N^2-N)/2$	1 to 49,995,000
array accesses	2 to N^2-N	2 to 99,990,000

dup1

operation	symbolic count	count, N=10000
i = 0	1	1
less than (<)	0 to N	0 to 10000
increment (+=1)	0 to N - 1	0 to 9999
equals (==)	1 to N - 1	1 to 9999
array accesses	2 to 2N - 2	2 to 19998

dup2

Asymptotic Behavior

In most cases, we care only about asymptotic behavior, i.e. what happens for very large N.

- Simulation of billions of interacting particles.
- Social network with billions of users.
- Logging of billions of transactions.
- Encoding of billions of bytes of video data.

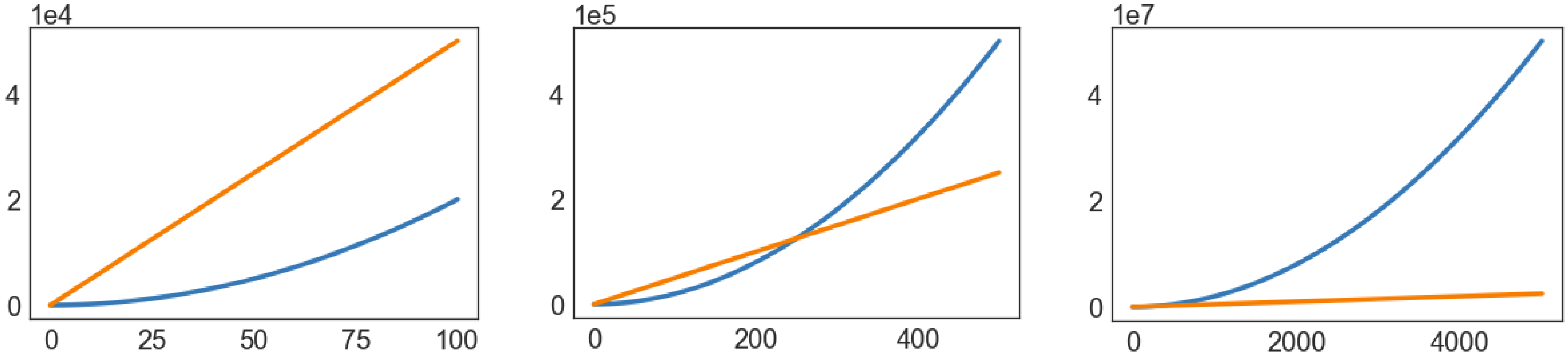
Algorithms which scale well (e.g. look like lines) have better asymptotic runtime behavior than algorithms that scale relatively poorly (e.g. look like parabolas).

Parabolas vs. Lines

Suppose we have two algorithms that zerpify a collection of N items.

- `zerp1` takes $2N^2$ operations.
- `zerp2` takes 500N operations.

For small N, `zerp1` might be faster, but as dataset size grows, the parabolic algorithm is going to fall farther and farther behind (in time it takes to complete).



Scaling Across Many Domains

We'll informally refer to the "shape" of a runtime function as its ***order of growth*** (will formalize soon).

- Effect is dramatic! Often determines whether a problem can be solved at all.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

(from Algorithm Design: Tardos, Kleinberg)

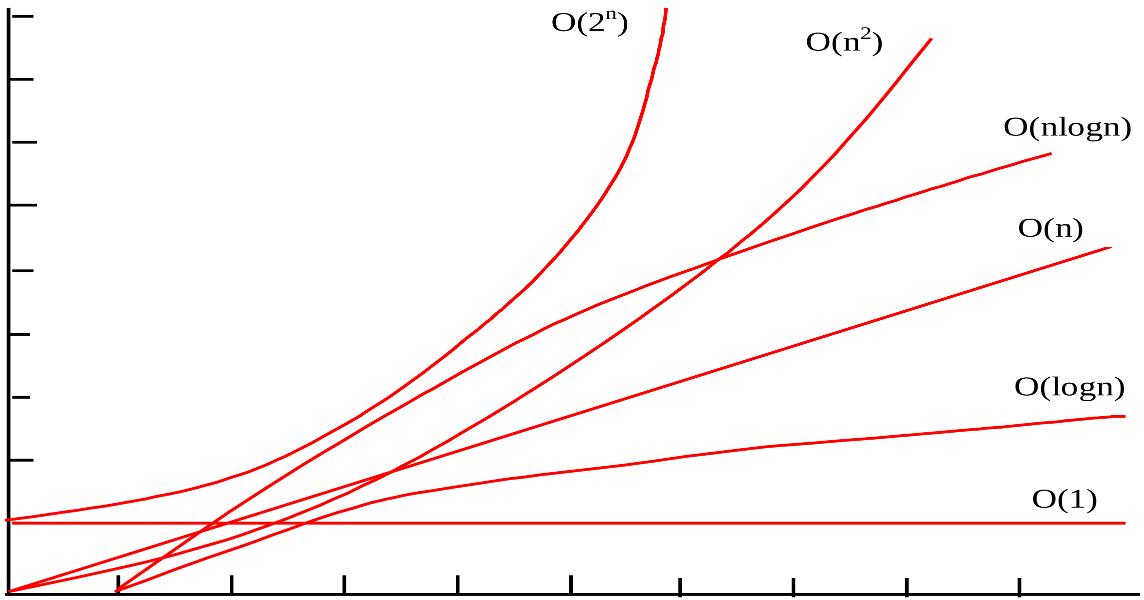
Comparing Common Growth Functions

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

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

Comparing Common Growth Functions

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$



Duplicate Finding

Our goal is to somehow **characterize the runtimes** of the functions below.

- Characterization should be **simple** and **mathematically rigorous**.
- Characterization should **demonstrate superiority** of dup2 over dup1.

operation	symbolic count
i = 0	1
j = i + 1	1 to N
less than (<)	2 to $(N^2+3N+2)/2$
increment (+=1)	0 to $(N^2+N)/2$
equals (==)	1 to $(N^2-N)/2$
array accesses	2 to N^2-N

dup1: parabolic, a.k.a. quadratic

operation	symbolic count
i = 0	1
less than (<)	0 to N
increment (+=1)	0 to N - 1
equals (==)	1 to N - 1
array accesses	2 to 2N - 2

dup2: linear

Computing Worst Case Order of Growth (Tedious Approach)

Duplicate Finding

Our goal is to somehow **characterize the runtimes** of the functions below.

- Characterization should be **simple** and **mathematically rigorous**.

operation	count
i = 0	1
j = i + 1	1 to N
less than (<)	2 to $(N^2+3N+2)/2$
increment (+=1)	0 to $(N^2+N)/2$
equals (==)	1 to $(N^2-N)/2$
array accesses	2 to N^2-N

operation	count
i = 0	1
less than (<)	0 to N
increment (+=1)	0 to N - 1
equals (==)	1 to N - 1
array accesses	2 to 2N - 2

Let’s be more careful about what we mean when we say the left function is “like” a parabola, and the right function is “like” a line.

Intuitive Simplification 1: Consider Only the Worst Case

Simplification 1: Consider only the worst case.

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

operation	count
i = 0	1
j = i + 1	~~1 to N~~
less than (<)	~~2 to $(N^2+3N+2)/2$~~
increment (+=1)	~~0 to $(N^2+N)/2$~~
equals (==)	~~1 to $(N^2-N)/2$~~
array accesses	~~2 to N^2-N~~

Intuitive Simplification 1: Consider Only the Worst Case

Simplification 1: Consider only the worst case.

- **Justification**: When comparing algorithms, we often care only about the worst case [but we will see exceptions in this course].

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

We’re effectively focusing on the case where there are no duplicates, because this is where there is a performance difference.

operation	worst case count
i = 0	1
j = i + 1	N
less than (<)	$(N^2+3N+2)/2$
increment (+=1)	$(N^2+N)/2$
equals (==)	$(N^2-N)/2$
array accesses	N^2-N

Consider the algorithm below. What do you expect will be the **order of growth** of the runtime for the algorithm?

A. N [linear]
B. N^2 [quadratic]
C. N^3 [cubic]
D. N^6 [sextic]

operation	count
less than (<)	$100N^2 + 3N$
greater than (>)	$2N^3 + 1$
and (&&)	5,000

In other words, if we plotted total runtime vs. N, what shape would we expect?

Intuitive Order of Growth Identification

Consider the algorithm below. What do you expect will be the **order of growth** of the runtime for the algorithm?

A. N [linear]
B. N^2 [quadratic]
C. N^3 [cubic]
D. N^6 [sextic]

operation	count
less than (<)	$100N^2 + 3N$
greater than (>)	$2N^3 + 1$
and (&&)	5,000

Argument:

- Suppose < takes α nanoseconds, > takes β nanoseconds, and && takes γ nanoseconds.
- Total time is $\alpha(100N^2 + 3N) + \beta(2N^3 + 1) + 5000\gamma$ nanoseconds.
- For very large N, the $2\beta N^3$ term is much larger than the others.

Extremely important point.
Make sure you understand it!

Intuitive Simplification 2: Eliminate low order terms

Simplification 2: Ignore lower order terms

- Eventually, $0.000000000001N^{2.00000001}$ will grow bigger than $10000000000N^2$
- So for sufficiently large N, only the largest term will actually matter

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

operation	worst case count
i = 0	1
j = i + 1	N
less than (<)	$(N^2\cancel{+3N+2})/2$
increment (+=1)	$(N^2\cancel{+N})/2$
equals (==)	$(N^2\cancel{-N})/2$
array accesses	$N^2\cancel{-N}$

(Not as) Intuitive Simplification 3: Eliminate multiplicative constants

Simplification 3: Ignore any coefficients

- Coefficients don't affect the "shape" of the function
- Often can change depending on what you consider "one operation"
- There are some branches of runtime analysis which care about coefficients. But it's much harder, because...

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

operation	worst case count
i = 0	1
j = i + 1	N
less than (<)	$N^2/\cancel{2}$
increment (+=1)	$N^2/\cancel{2}$
equals (==)	$N^2/\cancel{2}$
array accesses	N^2

Intuitive Simplification 4: Combine all operations

Simplification 4: Treat all operations as taking "1 unit of time"

- Even if an increment takes 1 ns and an array access takes 1000000 ns, those are still basically coefficients
- Also lets as pick what we count as a "primitive" operation arbitrarily
- Assumes that operations (ex. addition) take constant time regardless of input; this is known as the "cost model".

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

operation	worst case count
i = 0	1
j = i + 1	N
less than (<)	N^2
increment (+=1)	N^2
equals (==)	N^2
array accesses	N^2

Simplification Summary

Simplifications:

1. Only consider the worst case.
2. Ignore lower order terms.
3. Ignore any coefficients.
4. All operations take the same time.

These three simplifications are OK because we only care about the “**order of growth**” of the runtime.

operation	count
i = 0	1
j = i + 1	1 to N
less than (<)	2 to $(N^2+3N+2)/2$
increment (+=1)	0 to $(N^2+N)/2$
equals (==)	1 to $(N^2-N)/2$
array accesses	2 to N^2-N

operation	worst case o.o.g.
Total	N^2

Worst case order of growth of runtime: N^2

Simplification Summary: Repeating the Process for `dup2`

Simplifications:

1. Only consider the worst case.
2. Ignore lower order terms.
3. Ignore any coefficients.
4. All operations take the same time.

operation	count
i = 0	1
less than (<)	0 to N
increment (+=1)	0 to N - 1
equals (==)	1 to N - 1
array accesses	2 to 2N - 2

These three simplifications are OK because we only care about the "**order of growth**" of the runtime.

operation	worst case o.o.g.

Worst case order of growth of runtime:

Repeating the Process for dup2

Simplifications:

1. Only consider the worst case.
2. Ignore lower order terms.
3. Ignore any coefficients.
4. All operations take the same time.

This simplification is OK because we specifically only care about worst case.

These three simplifications are OK because we only care about the "**order of growth**" of the runtime.

operation	count
i = 0	1
less than (<)	0 to N
increment (+=1)	0 to N - 1
equals (==)	1 to N - 1
array accesses	2 to 2N - 2

operation	worst case o.o.g.
Total	N

Worst case order of growth of runtime: N

Summary of Our (Painful) Analysis Process

One thing to note: If N -> N^2, then 2N -> $(2N)^2 = 4N^2$; doubling the size of the input means 4x longer runtime

This is what we observed earlier! So despite all our simplifications, this theoretical analysis matches our experimental values.

operation	count
i = 0	1
j = i + 1	1 to N
less than (<)	2 to $(N^2+3N+2)/2$
increment (+=1)	0 to $(N^2+N)/2$
equals (==)	1 to $(N^2-N)/2$
array accesses	2 to N^2-N

operation	worst case o.o.g.
Total	N^2

Worst case order of growth of runtime: N^2

Summary of Our (Painful) Analysis Process

Our process:

- Construct a table of exact counts of all possible operations.
- Convert table into a worst case order of growth using 4 simplifications.

operation	count
i = 0	1
j = i + 1	1 to N
less than (<)	2 to $(N^2+3N+2)/2$
increment (+=1)	0 to $(N^2+N)/2$
equals (==)	1 to $(N^2-N)/2$
array accesses	2 to N^2-N

operation	worst case o.o.g.
Total	N^2

Worst case order of growth of runtime: N^2

By using our simplifications from the outset, we can avoid building the table at all!

Computing Worst Case Order of Growth (Simplified Approach)

Goal: Measuring Code Efficiency

Intuitive Runtime Characterizations

- Clock Time
- Exact Operation Counting
- Exact Count Exercise

Asymptotic Analysis

- Why Scaling Matters
- Computing Worst Case Order of Growth (Tedious Approach)
- **Computing Worst Case Order of Growth (Simplified Approach)**

Asymptotic Notation

- Big Theta (a.k.a. Order of Growth)
- Big O and Big Omega

Simplified Analysis Process

Rather than building the entire table, we can instead:

- Treat anything that takes constant time (relative to N) as a single operation
- Figure out the order of growth for the count of that operation by either:
  - Making an exact count, then discarding the unnecessary pieces.
  - Using intuition and inspection to determine order of growth (only possible with lots of practice).

Let's redo our analysis of `dup1` with this new process.

- This time, we'll show all our work.

Analysis of Nested For Loops (Based on Exact Count)

Find the order of growth of the worst case runtime of dup1.

```
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
      if (A[i] == A[j])
         return true;
return false;
```

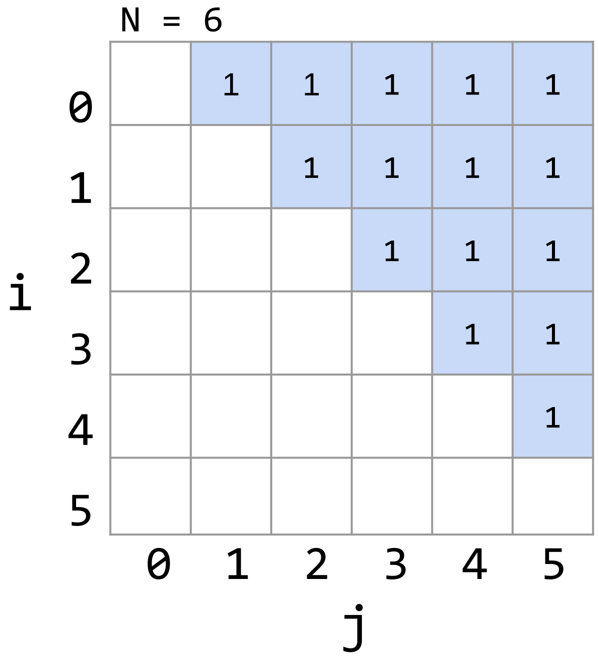
Analysis of Nested For Loops (Based on Exact Count)

Find the order of growth of the worst case runtime of dup1.

```
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
          //1 unit of time
return false;
```

Analysis of Nested For Loops (Based on Exact Count)

Find the order of growth of the worst case runtime of dup1.




```java
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
            //1 unit of time
return false;
```

Worst case number of steps:

$$C = 1 + 2 + 3 + \ldots + (N - 3) + (N - 2) + (N - 1)$$

$$C = (N - 1) + (N - 2) + (N - 3) + \ldots + 3 + 2 + 1$$

$$2C = \underbrace{N + N + \ldots + N}_{\text{N-1 of these}} \qquad = N(N - 1)$$

$$\therefore C = N(N - 1)/2$$

Analysis of Nested For Loops (Based on Exact Count)

Find the order of growth of the worst case runtime of `dup1`.

N = 6

i \ j	0	1	2	3	4	5
0		1	1	1	1	1
1			1	1	1	1
2				1	1	1
3					1	1
4						1
5						

```java
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
          //1 unit of time
return false;
```

Worst case number of steps:

$$C = 1 + 2 + 3 + \ldots + (N - 3) + (N - 2) + (N - 1) = N(N-1)/2$$

operation	worst case o.o.g.
==	N^2

Worst case order of growth of runtime: N^2

Analysis of Nested For Loops (Simpler Geometric Argument)

Find the order of growth of the worst case runtime of dup1.

N = 6

i \ j	0	1	2	3	4	5
0		1	1	1	1	1
1			1	1	1	1
2				1	1	1
3					1	1
4						1
5						

```
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
              //1 unit of time
return false;
```

Worst case number of steps:

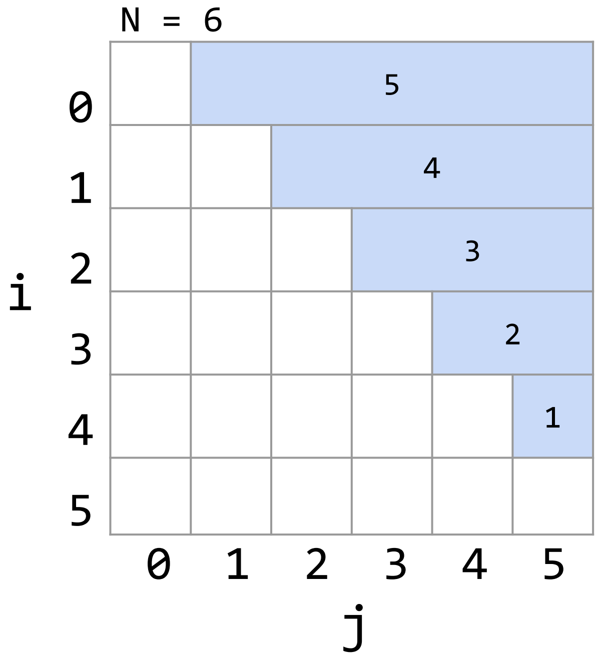
- Given by area of right triangle of side length N-1.
- Order of growth of area is N^2.

operation	worst case o.o.g.
==	N^2

Worst case order of growth of runtime: N^2

Loops Example 1: Based on Exact Count

Find the order of growth of the worst case runtime of dup1.




```
int N = A.length;
for (int i = 0; i < N; i += 1)
    // N-i-1 units of work
return false;
```

Worst case number of operations:

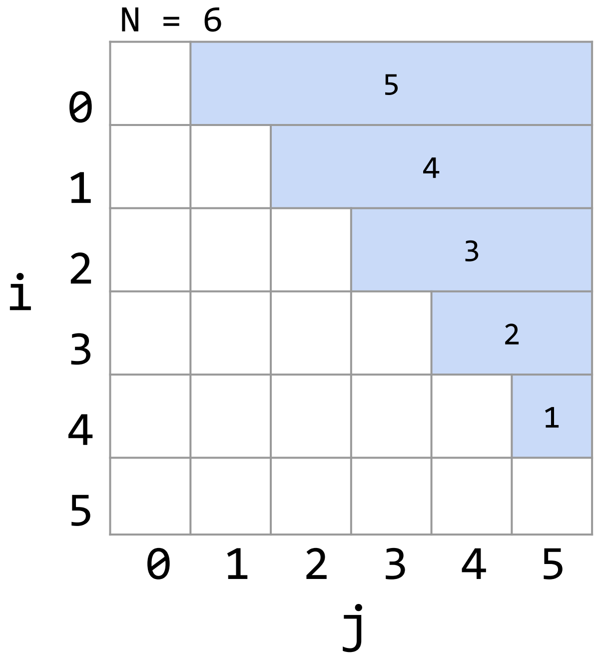
$$C = 1 + 2 + 3 + \ldots + (N - 3) + (N - 2) + (N - 1) = N(N-1)/2$$

operation	worst case count
Total	$\Theta(N^2)$

Worst case runtime: $\Theta(N^2)$

Loops Example 1: Based on Exact Count

Find the order of growth of the worst case runtime of dup1.




```
int N = A.length;
for (int i = 0; i < N; i += 1)
    // N-i-1 units of work
return false;
```

Worst case number of operations:

- Given by area of right triangle of side length N-1.
- Area is $\Theta(N^2)$.

operation	worst case count
Total	$\Theta(N^2)$

Worst case runtime: $\Theta(N^2)$

Big Theta (a.k.a. Order of Growth)

Goal: Measuring Code Efficiency

Intuitive Runtime Characterizations

- Clock Time
- Exact Operation Counting
- Exact Count Exercise

Asymptotic Analysis

- Why Scaling Matters
- Computing Worst Case Order of Growth (Tedious Approach)
- Computing Worst Case Order of Growth (Simplified Approach)

Asymptotic Notation

- **Big Theta (a.k.a. Order of Growth)**
- Big O and Big Omega

Formalizing Order of Growth

Given a function Q(N), we can apply our last two simplifications (ignore low orders terms and multiplicative constants) to yield the order of growth of Q(N).

- Example: $Q(N) = 3N^3 + N^2$
- Order of growth: N^3

Let's finish out this lecture by moving to a more formal notation called Big-Theta.

- The math might seem daunting at first.
- … but the idea is exactly the same! Using "Big-Theta" instead of "order of growth" does not change the way we analyze code at all.

Order of Growth Exercise

Consider the functions below.

- Informally, what is the “shape” of each function for very large N?
- In other words, what is the order of growth of each function?

function	order of growth
$N^3 + 3N^4$	
$1/N + N^3$	
$1/N + 5$	
$Ne^N + N$	
$40 \sin(N) + 4N^2$	

Order of Growth Exercise

Consider the functions below.

- Informally, what is the “shape” of each function for very large N?
- In other words, what is the order of growth of each function?

function	order of growth
$N^3 + 3N^4$	N^4
$1/N + N^3$	N^3
$1/N + 5$	1
$Ne^N + N$	Ne^N
$40 \sin(N) + 4N^2$	N^2

Big-Theta

Suppose we have a function R(N) with order of growth f(N).

- In "Big-Theta" notation we write this as $R(N) \in \Theta(f(N))$.
- Examples:
  - $N^3 + 3N^4 \in \Theta(N^4)$
  - $1/N + N^3 \in \Theta(N^3)$
  - $1/N + 5 \in \Theta(1)$
  - $Ne^N + N \in \Theta(Ne^N)$
  - $40 \sin(N) + 4N^2 \in \Theta(N^2)$

function R(N)	order of growth
$N^3 + 3N^4$	N^4
$1/N + N^3$	N^3
$1/N + 5$	1
$Ne^N + N$	Ne^N
$40 \sin(N) + 4N^2$	N^2

Big-Theta: Formal Definition

$$R(N) \in \Theta(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.

i.e. very large N

Example: $40 \sin(N) + 4N^2 \in \Theta(N^2)$

- $R(N) = 40 \sin(N) + 4N^2$
- $f(N) = N^2$
- k1 = 3
- k2 = 5

Big-Theta: Formal Definition

$$R(N) \in \Theta(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$$

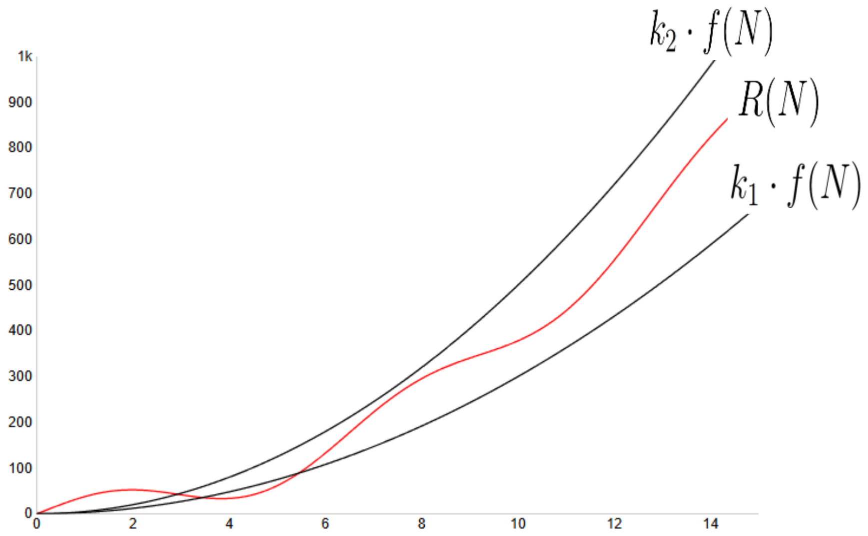
for all values of N greater than some N_0.

i.e. very large N

Example: $40 \sin(N) + 4N^2 \in \Theta(N^2)$

- $R(N) = 40 \sin(N) + 4N^2$
- $f(N) = N^2$
- k1 = 3
- k2 = 5

R(N):	4*N^2+40*sin(N)	f(N):	N^2
k1:	3	k2:	5
maxN:	15	maxY:	1000



Big-Theta Challenge

Suppose $R(N) = (4N^2+3N*\ln(N))/2$.

- Find a simple $f(N)$ and corresponding k_1 and k_2.

$$\boxed{R(N) \in \Theta(f(N))}$$

means there exist positive constants k_1 and k_2 such that:

$$\boxed{k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)}$$

for all values of N greater than some N_0.

i.e. very large N

Big-Theta Challenge

Suppose $R(N) = (4N^2+3N*\ln(N))/2$.

- $f(N) = N^2$
- $k_1 = 1$
- $k_2 = 3$

$$\boxed{R(N) \in \Theta(f(N))}$$

means there exist positive constants k_1 and k_2 such that:

$$\boxed{k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)}$$

for all values of N greater than some N_0.

i.e. very large N

Using Big-Theta doesn't change anything about runtime analysis (no need to find k_1 or k_2 or anything like that).

- The only difference is that we use the Θ symbol anywhere we would have said "order of growth".

operation	worst case count
i = 0	1
j = i + 1	$\Theta(N)$
less than (<)	$\Theta(N^2)$
increment (+=1)	$\Theta(N^2)$
equals (==)	$\Theta(N^2)$
array accesses	$\Theta(N^2)$

operation	worst case count
Total	$\Theta(N^2)$

Worst case runtime: $\Theta(N^2)$

Big O and Big Omega

Goal: Measuring Code Efficiency

Intuitive Runtime Characterizations

- Clock Time
- Exact Operation Counting
- Exact Count Exercise

Asymptotic Analysis

- Why Scaling Matters
- Computing Worst Case Order of Growth (Tedious Approach)
- Computing Worst Case Order of Growth (Simplified Approach)

Asymptotic Notation

- Big Theta (a.k.a. Order of Growth)
- **Big O and Big Omega**

Big Theta

We used Big Theta to describe the order of growth of a function.

function R(N)	order of growth
$N^3 + 3N^4$	$\Theta(N^4)$
$1/N + N^3$	$\Theta(N^3)$
$1/N + 5$	$\Theta(1)$
$Ne^N + N$	$\Theta(Ne^N)$
$40 \sin(N) + 4N^2$	$\Theta(N^2)$

We also used Big Theta to describe the rate of growth of the runtime of a piece of code.

Big O and Big Omega

Whereas Big Theta can informally be thought of as something like “equals”, Big O can be thought of as “less than or equal” and Big Omega can be thought of as "greater than or equal"

Example, the following are all true:

- $N^3 + 3N^4 \in \Theta(N^4)$
- $N^3 + 3N^4 \in O(N^4)$
- $N^3 + 3N^4 \in O(N^6)$
- $N^3 + 3N^4 \in O(N^{N!})$
- $N^3 + 3N^4 \in \Omega(N^4)$
- $N^3 + 3N^4 \in \Omega(N^2)$
- $N^3 + 3N^4 \in \Omega(1)$

Big-Theta: Formal Definition

$$R(N) \in \Theta(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.

i.e. very large N

Big O

$$R(N) \in O(f(N))$$

means there exists a positive constant k_2 such that:

$$R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.

i.e. very large N

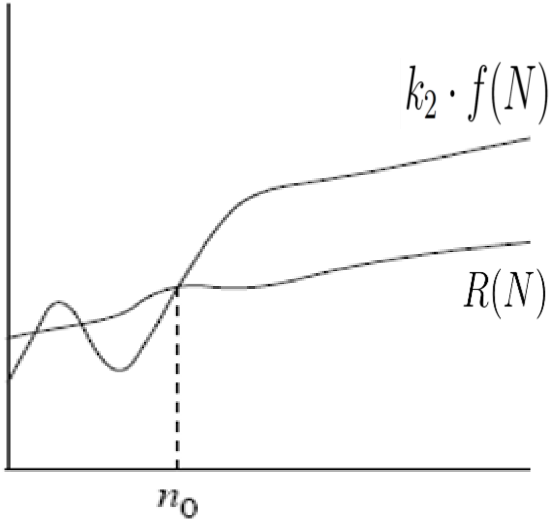
Big O

$$R(N) \in O(f(N))$$

means there exists a positive constant k_2 such that:

$$R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.



i.e. very large N

More Examples

Examples of functions in $O(n^2)$:

n^2

$n^2 + n$

$n^2 + 1000n$

$1000n^2 + 1000n$

Also,

n

$n/1000$

$n^{1.99999}$

$n^2 / \lg \lg \lg n$

Big Omega

$$R(N) \in \Omega(f(N))$$

means there exists a positive constant k_1 such that:

$$k_1 \cdot f(N) \le R(N)$$

for all values of N greater than some N_0.

i.e. very large N

Big Omega

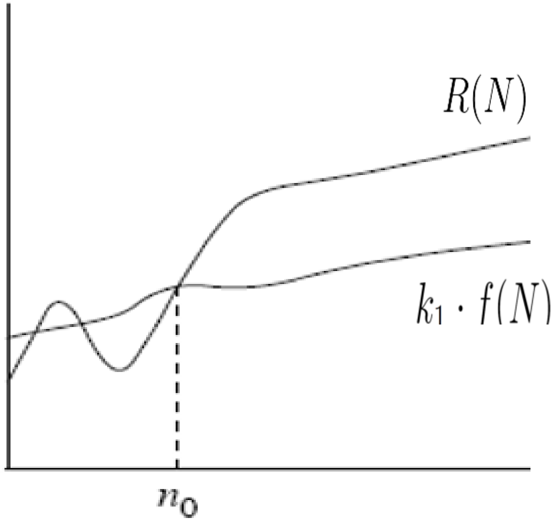
$$R(N) \in \Omega(f(N))$$

means there exists a positive constant k_1 such that:

$$k_1 \cdot f(N) \leq R(N)$$

for all values of N greater than some N_0.

i.e. very large N



More Examples

Examples of functions in $\Omega(n^2)$:

n^2

$n^2 + n$

$n^2 - n$

$1000n^2 + 1000n$

$1000n^2 - 1000n$

Also,

n^3

$n^{2.00001}$

$n^2 \lg \lg \lg n$

2^{2^n}

Big Theta vs. Big O

We will see why big O is practically useful in the upcoming lectures.

	Informal meaning:	Family	Family Members
Big Theta $\Theta(f(N))$	Order of growth is f(N).	$\Theta(N^2)$	$N^2/2$ $2N^2$ $N^2 + 38N + N$
Big O $O(f(N))$	Order of growth is less than or equal to f(N).	$O(N^2)$	$N^2/2$ $2N^2$ $\lg(N)$
Big Omega $\Omega(f(N))$	Order of growth is greater than or equal to f(N).	$\Omega(N^2)$	$N^2/2$ $2N^2$ $N^{N!}$

Summary

Given a code snippet, we can express its runtime as a function R(N), where N is some property of the input of the function (often the size of the input).

Rather than finding R(N) exactly, we instead usually only care about the order of growth of R(N).

One approach (not universal):

- Reduce constant time operations to "1 unit of time", and let C(N) be the count of how many times that operation occurs as a function of N.
- Determine order of growth f(N) for C(N), i.e. $C(N) \in \Theta(f(N))$
  - Often (but not always) we consider the worst case count.
- Can use O as an alternative for Θ. O is used for upper bounds. Ω isn't used often in practical settings, but is often used in theoretical CS for lower bounds.