# CSC3310 Algorithms

## Asymptotic Notation

William Retert

Milwaukee School of Engineering

# Example—CONTAINS

Given an array $A$ and a value $v$, determine whether the value is stored in the array.

# Example—CONTAINS

Given an array $A$ and a value $v$, determine whether the value is stored in the array.

**Input** An array $A[0, \ldots, n-1]$ of $n$ elements and a value $v$
**Output** True iff $v$ is one of the elements of the array; false otherwise

# Example—CONTAINS

Given an array $A$ and a value $v$, determine whether the value is stored in the array.

**Input** An array $A[0, \ldots, n-1]$ of $n$ elements and a value $v$
**Output** True iff $v$ is one of the elements of the array; false otherwise

```
procedure CONTAINS(A[0,...,n-1], v)
    check ← 0
    while check ≠ n and A[check] ≠ v do
        check ← check + 1
    end while
    return check ≠ n
end procedure
```

# Example—CONTAINS

```
procedure CONTAINS(A[0,...,n-1], v)
    check ← 0
    while check ≠ n and A[check] ≠ v do
        check ← check + 1
    end while
    return check ≠ n
end procedure
```

We can define the loop invariant as

$$\forall i < check, A[i] \neq v$$

This is trivially true on the first iteration, and inductively true for subsequent iterations. (From the loop test for the current iteration and the induction for the previous ones)
When the loop ends, then either $check = n$ or $A[check] = v$. If $check = n$, then the loop invarient implies that $v$ is not in the array. If $check \neq n$ then $A[check] = v$, so $v$ is in the array.

# Example—CONTAINS

```
procedure CONTAINS(A[0,...,n-1], v)
    check ← 0
    while check ≠ n and A[check] ≠ v do
        check ← check + 1
    end while
    return check ≠ n
end procedure
```

Define $T(n)$ as the number of single-step operations peformed on an array of $n$ elements.

$$T(n) = \begin{cases} 3n+3 & v \text{ is not in } A \\ 3(\text{indexof}(v))+4 & v \text{ in } A \end{cases}$$

# Example—CONTAINS

Define $T(n)$ as the number of single-step operations peformed on an array of $n$ elements.

$$T(n) = \begin{cases} 3n+3 & v \text{ is not in } A \\ 3(\text{indexof}(v))+4 & v \text{ in } A \end{cases}$$

The time depends on $n$, but also on whether and where $v$ is in the array. We therefore consider cases:

Best case $v$ is the first thing in the array. We find it in 4 steps
Worst case $v$ is not in the array. We perform $3n+3$ steps looking for it.

**N.B.** The size of $n$ is not a case! All cases are for values of $T(n)$, where the $n$ is passed in from outside.

# Time Complexity

**Benchmark** Implement and measure

- Hardware?
- Software Environment?

**Exact Analysis** Count operations

- Laborious
- What is an operation?

**Asymptotic Analysis** Estimate number of primitive operations within a constant factor

- Relative to size of input
- Ignore fine details of implementation

# Asymptotic Notation

Five kinds of notation:

- $\Theta$ “Big Theta” — Upper and Lower Bound
- $O$ “Big O” — Inclusive Upper Bound
- $\Omega$ “Big Omega” — Inclusive Lower Bound
- $o$ “Little O” — Exclusive Upper Bound
- $\omega$ “Little Omega” — Exclusive Lower Bound

# Asymptotic Analysis

```
Scanner in = new Scanner(System.in);
String msg = "How many friends do you want to say hello to?";
System.out.println(msg);
int numFriends = Integer.parseInt(in.nextLine());
String s = "Hello to " + numFriends + " friends!";
System.out.println(s);
```

Operations:

# Asymptotic Analysis

```
Scanner in = new Scanner(System.in);
String msg = "How many friends do you want to say hello to?";
System.out.println(msg);
int numFriends = Integer.parseInt(in.nextLine());
String s = "Hello to " + numFriends + " friends!";
System.out.println(s);
```

Operations:

3

# Asymptotic Analysis

```
Scanner in = new Scanner(System.in);
String msg = "How many friends do you want to say hello to?";
System.out.println(msg);
int numFriends = Integer.parseInt(in.nextLine());
String s = "Hello to " + numFriends + " friends!";
System.out.println(s);
```

Operations:

$$3 + 1$$

# Asymptotic Analysis

```
Scanner in = new Scanner(System.in);
String msg = "How many friends do you want to say hello to?";
System.out.println(msg);
int numFriends = Integer.parseInt(in.nextLine());
String s = "Hello to " + numFriends + " friends!";
System.out.println(s);
```

Operations:

$$3 + 1 + 3$$

# Asymptotic Analysis

```
Scanner in = new Scanner(System.in);
String msg = "How many friends do you want to say hello to?";
System.out.println(msg);
int numFriends = Integer.parseInt(in.nextLine());
String s = "Hello to " + numFriends + " friends!";
System.out.println(s);
```

Operations:

$$3 + 1 + 3 + 5 + ??$$

# Asymptotic Analysis

```
Scanner in = new Scanner(System.in);
String msg = "How many friends do you want to say hello to?";
System.out.println(msg);
int numFriends = Integer.parseInt(in.nextLine());
String s = "Hello to " + numFriends + " friends!";
System.out.println(s);
```

Operations:

$$\cancel{3}O(1) + O(1)\cancel{1} + O(1)\cancel{3} + O(1)$$

# Asymptotic Analysis

```
Scanner in = new Scanner(System.in);
String msg = "How many friends do you want to say hello to?";
System.out.println(msg);
int numFriends = Integer.parseInt(in.nextLine());
String s = "Hello to " + numFriends + " friends!";
System.out.println(s);
```

Operations:

$$O(1) + O(1) + O(1) + O(1) + O(1)$$

# Asymptotic Analysis

```
Scanner in = new Scanner(System.in);
String msg = "How many friends do you want to say hello to?";
System.out.println(msg);
int numFriends = Integer.parseInt(in.nextLine());
String s = "Hello to " + numFriends + " friends!";
System.out.println(s);
```

Operations:

$$O(1) + O(1) + O(1) + O(1) + O(1) + O(1)$$

# Asymptotic Analysis

```
Scanner in = new Scanner(System.in);
String msg = "How many friends do you want to say hello to?";
System.out.println(msg);
int numFriends = Integer.parseInt(in.nextLine());
String s = "Hello to " + numFriends + " friends!";
System.out.println(s);
```

Operations:

$$O(1) + O(1) + O(1) + O(1) + O(1) + O(1) = 6 \times O(1)$$

# Asymptotic Analysis

```
Scanner in = new Scanner(System.in);
String msg = "How many friends do you want to say hello to?";
System.out.println(msg);
int numFriends = Integer.parseInt(in.nextLine());
String s = "Hello to " + numFriends + " friends!";
System.out.println(s);
```

Operations:

$$O(1) + O(1) + O(1) + O(1) + O(1) + O(1) = O(1)$$

# Loops

```
public static int find(int[] a, int target) {
  int result = -1;
  for(int i = 0; i < a.length; ++i) {
    if( a[i] == target ) {
      result = i;
    }
  }
  return result;
}
```

# Loops

```
public static int find(int[] a, int target) {
  int result = -1;
  for(int i = 0; i < a.length; ++i) {
    if( a[i] == target ) {
      result = i;
    }
  }
  return result;
}
```

Operations: $O(1)$

# Loops

```
public static int find(int[] a, int target) {
  int result = -1;
  for(int i = 0; i < a.length; ++i) {
    if( a[i] == target ) {
      result = i;
    }
  }
  return result;
}
```

Operations: $O(1) + O(1)$

# Loops

```
public static int find(int[] a, int target) {
  int result = -1;
  for(int i = 0; i < a.length; ++i) {
    if( a[i] == target ) {
      result = i;
    }
  }
  return result;
}
```

Operations: $O(1) + O(1) + \sum_{i=0}^{n-1}()$

# Loops

```
public static int find(int[] a, int target) {
  int result = -1;
  for(int i = 0; i < a.length; ++i) {
    if( a[i] == target ) {
      result = i;
    }
  }
  return result;
}
```

Operations: $O(1) + O(1) + \sum_{i=0}^{n-1}(O(1))$

# Loops

```
public static int find(int[] a, int target) {
  int result = -1;
  for(int i = 0; i < a.length; ++i) {
    if( a[i] == target ) {
      result = i;
    }
  }
  return result;
}
```

Operations: $O(1) + O(1) + \sum_{i=0}^{n-1}(O(1) + O(1))$

# Loops

```
public static int find(int[] a, int target) {
  int result = -1;
  for(int i = 0; i < a.length; ++i) {
    if( a[i] == target ) {
      result = i;
    }
  }
  return result;
}
```

Operations: $O(1) + O(1) + \sum_{i=0}^{n-1}(O(1) + O(1) + O(1))$

# Loops

```
public static int find(int[] a, int target) {
  int result = -1;
  for(int i = 0; i < a.length; ++i) {
    if( a[i] == target ) {
      result = i;
    }
  }
  return result;
}
```

Operations: $O(1) + O(1) + \sum_{i=0}^{n-1}(O(1) + O(1) + O(1) + O(1))$

# Loops

```
public static int find(int[] a, int target) {
  int result = -1;
  for(int i = 0; i < a.length; ++i) {
    if( a[i] == target ) {
      result = i;
    }
  }
  return result;
}
```

Operations: $O(1) + O(1) + \sum_{i=0}^{n-1}(O(1) + O(1) + O(1) + O(1)) + O(1)$

# Loops

```
public static int find(int[] a, int target) {
  int result = -1;
  for(int i = 0; i < a.length; ++i) {
    if( a[i] == target ) {
      result = i;
    }
  }
  return result;
}
```

Operations: $O(1) + O(1) + \sum_{i=0}^{n-1}(O(1) + O(1) + O(1) + O(1)) + O(1) + O(1)$

# Loops

```
public static int find(int[] a, int target) {
  int result = -1;
  for(int i = 0; i < a.length; ++i) {
    if( a[i] == target ) {
      result = i;
    }
  }
  return result;
}
```

Operations: $O(1) + O(1) + \sum_{i=0}^{n-1}(O(1) + O(1) + O(1) + O(1)) + O(1) + O(1)$

$$= 4nO(1) + 4O(1)$$

# Loops

```
public static int find(int[] a, int target) {
  int result = -1;
  for(int i = 0; i < a.length; ++i) {
    if( a[i] == target ) {
      result = i;
    }
  }
  return result;
}
```

Operations: $O(1) + O(1) + \sum_{i=0}^{n-1}(O(1) + O(1) + O(1) + O(1)) + O(1) + O(1)$

$$= 4nO(1) + 4O(1)$$

$$= O(n)$$

# Simplifying Big-O

- If $d(n)$ is $O(f(n))$ then $ad(n)$ is $O(f(n))$ for constant $a > 0$
- If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$ then $d(n) + e(n)$ is $O(f(n) + g(n))$
- If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$ then $d(n)e(n)$ is $O(f(n)g(n))$
- If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$ then $d(n)$ is $O(g(n))$
- If $f(n)$ is a polynomial of degree $d$, then $f(n)$ is $O\left(n^d\right)$

- $n^x$ is $O(a^n)$ for any fixed $x > 0$ and $a > 1$
- $\log(n^x)$ is $O(\log n)$ for any fixed $x > 0$
- $\log^x n$ is $O(n^y)$ for any fixed $x > 0$ and $y > 0$

**N.B.** Including constant factors and lower-order terms in big-O notation is considered poor taste.

# Ranking Families

"Largest" to "smallest":

| | |
|---|---|
| | $O(n!)$ |
| Exponential | $O(2^n)$ |
| | $O(n^3)$ |
| | $O(n^2 \log n)$ |
| Quadratic | $O(n^2)$ |
| | $O(n \log n)$ |
| Linear | $O(n)$ |
| Logarithmic | $O(\log n)$ |
| Constant | $O(1)$ |

- Functions which are $O(n^k)$ for some $k \geq 1$ are called *polynomial*
- Functions which are $O(a^n)$ for some $a > 1$ are called *exponential*

# Ranking Families

"Largest" to "smallest":

$\Theta(n!)$

Exponential $\Theta(2^n)$

$\dots$

$\Theta(n^3)$

$\Theta(n^2 \log n)$

Quadratic $\Theta(n^2)$

$\Theta(n \log n)$

Linear $\Theta(n)$

Logarithmic $\Theta(\log n)$

Constant $\Theta(1)$

- Functions which are $\Theta(n^k)$ for some $k \geq 1$ are called *polynomial*
- Functions which are $\Theta(a^n)$ for some $a > 1$ are called *exponential*

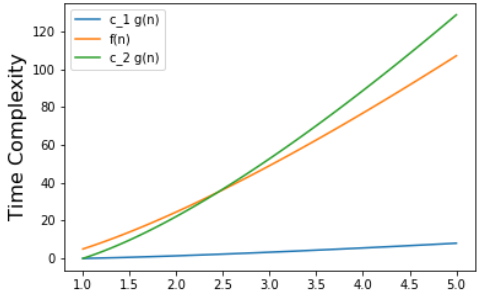
# Asymptotic Notation

Five kinds of notation:

- $\Theta$ “Big Theta” — Upper and Lower Bound
- $O$ “Big O” — Inclusive Upper Bound
- $\Omega$ “Big Omega” — Inclusive Lower Bound
- $o$ “Little O” — Exclusive Upper Bound
- $\omega$ “Little Omega” — Exclusive Lower Bound

# Θ Notation

- $\Theta(g(n))$ asymptotically-tight bound
  - Upper bound of $c_2 g(n)$ (grows slower than $c_2 g(n)$)
  - Lower bound of $c_1 g(n)$ (grows faster than $c_1 g(n)$)
- We get to choose $c_1$ and $c_2$
- Only holds for large enough $n$



# Θ Notation

$$\Theta(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

# Θ Notation

$$\Theta(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

$\{x|condition\}$ is the set of **all** $x$'s for which *condition* is true.

- $x$ is in the set **iff** the condition is true
  - if *condition* is true for $x$, then $x$ is in the set
  - if $x$ is in the set, then *condition* is true for $x$

# Θ Notation

$$\Theta(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

$\{x|condition\}$ is the set of **all** $x$'s for which *condition* is true.

- $x$ is in the set **if and only if** the condition is true
  - if *condition* is true for $x$, then $x$ is in the set
  - if $x$ is in the set, then *condition* is true for $x$

# Θ Notation

$$\Theta(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Here, $f(n) \in \Theta(g(n))$ if and only if $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$
That is, $\Theta(g(n))$ defines a set of functions.

- Some functions are in the set, some are not.
- We can find $c_1$, $c_2$, and $n_0$ for any function in the set.

# Θ Notation

$$\Theta(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Here, $f(n) \in \Theta(g(n))$ if and only if $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$
Let $f(n) = 5n^2 - 3n + 2$. We can show that $f(n) \in \Theta\left(n^2\right)$

# Θ Notation

$$\Theta(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Here, $f(n) \in \Theta(g(n))$ if and only if $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$
Let $f(n) = 5n^2 - 3n + 2$. We can show that $f(n) \in \Theta\left(n^2\right)$ To do this, we must show that

$$c_1 n^2 \leq 5n^2 - 3n + 2 \leq c_2 n^2$$

# Θ Notation

$$\Theta(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Here, $f(n) \in \Theta(g(n))$ if and only if $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$
Let $f(n) = 5n^2 - 3n + 2$. We can show that $f(n) \in \Theta\left(n^2\right)$ To do this, we must show that

$$c_1 n^2 \leq 5n^2 - 3n + 2 \leq c_2 n^2$$

Let's split that up:

$c_1 n^2 \leq 5n^2 - 3n + 2$
$5n^2 - 3n + 2 \leq c_2 n^2$

# Θ Notation

$$\Theta(g(n)) = \left\{ f(n) \mid \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Let $f(n) = 5n^2 - 3n + 2$. We can show that $f(n) \in \Theta\left(n^2\right)$ To do this, we must show that

$$c_1 n^2 \leq 5n^2 - 3n + 2 \leq c_2 n^2$$

Let's split that up:

$c_1 n^2 \leq 5n^2 - 3n + 2$ This is true for $c_1 = 4$ if $n \geq 2$

$$4n^2 \leq 5n^2 - 3n + 2$$
$$3n \leq n^2 + 2$$

$5n^2 - 3n + 2 \leq c_2 n^2$

# Θ Notation

$$\Theta(g(n)) = \left\{ f(n) \mid \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Let $f(n) = 5n^2 - 3n + 2$. We can show that $f(n) \in \Theta\left(n^2\right)$ To do this, we must show that

$$c_1 n^2 \leq 5n^2 - 3n + 2 \leq c_2 n^2$$

Let's split that up:

$c_1 n^2 \leq 5n^2 - 3n + 2$ This is true for $c_1 = 4$ if $n \geq 2$

$$\begin{aligned} 4n^2 &\leq 5n^2 - 3n + 2 \\ 3n &\leq n^2 + 2 \end{aligned}$$

$5n^2 - 3n + 2 \leq c_2 n^2$ This is true for $c_2 = 5$

$$\begin{aligned} 5n^2 - 3n + 2 &\leq 5n^2 \\ 2 &\leq 3n \end{aligned}$$

# Θ Notation

$$\Theta(g(n)) = \left\{ f(n) \mid \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Let $f(n) = 5n^2 - 3n + 2$. We can show that $f(n) \in \Theta\left(n^2\right)$ To do this, we must show that

$$c_1 n^2 \leq 5n^2 - 3n + 2 \leq c_2 n^2$$

Let's split that up:

$c_1 n^2 \leq 5n^2 - 3n + 2$ This is true for $c_1 = 4$ if $n \geq 2$

$$\begin{aligned} 4n^2 &\leq 5n^2 - 3n + 2 \\ 3n &\leq n^2 + 2 \end{aligned}$$

$5n^2 - 3n + 2 \leq c_2 n^2$ This is true for $c_2 = 5$

$$\begin{aligned} 5n^2 - 3n + 2 &\leq 5n^2 \\ 2 &\leq 3n \end{aligned}$$

unless $n = 0 \ldots$

# Θ Notation

$$\Theta(g(n)) = \left\{ f(n) \mid \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Let $f(n) = 5n^2 - 3n + 2$. We can show that $f(n) \in \Theta\left(n^2\right)$ To do this, we must show that

$$c_1 n^2 \leq 5n^2 - 3n + 2 \leq c_2 n^2$$

Let's split that up:

$c_1 n^2 \leq 5n^2 - 3n + 2$ This is true for $c_1 = 4$ if $n \geq 2$

$$\begin{aligned} 4n^2 &\leq 5n^2 - 3n + 2 \\ 3n &\leq n^2 + 2 \end{aligned}$$

$5n^2 - 3n + 2 \leq c_2 n^2$ This is true for $c_2 = 5$ and $n \geq 2$

$$\begin{aligned} 5n^2 - 3n + 2 &\leq 5n^2 \\ 2 &\leq 3n \end{aligned}$$

# Θ Notation

$$\Theta(g(n)) = \left\{ f(n) \mid \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Let $f(n) = 5n^2 - 3n + 2$. We can show that $f(n) \in \Theta\left(n^2\right)$ To do this, we must show that

$$c_1 n^2 \leq 5n^2 - 3n + 2 \leq c_2 n^2$$

Let's split that up:

$c_1 n^2 \leq 5n^2 - 3n + 2$ This is true for $c_1 = 4$ if $n \geq 2$

$$4n^2 \leq 5n^2 - 3n + 2$$
$$3n \leq n^2 + 2$$

$5n^2 - 3n + 2 \leq c_2 n^2$ This is true for $c_2 = 5$ and $n \geq 2$

$$5n^2 - 3n + 2 \leq 5n^2$$
$$2 \leq 3n$$

So we can choose $c_1 = 4$, $c_2 = 5$, and $n_0 = 3$ to meet the condition. Therefore, $5n^2 - 3n + 2 \in \Theta\left(n^2\right)$

# Θ Notation

$$\Theta(g(n)) = \left\{ f(n) \mid \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Let $f(n) = 5n^2 - 3n + 2$. We can show that $f(n) \in \Theta\left(n^2\right)$ To do this, we must show that

$$c_1 n^2 \leq 5n^2 - 3n + 2 \leq c_2 n^2$$

Let's split that up:

$c_1 n^2 \leq 5n^2 - 3n + 2$ This is true for $c_1 = 1$

$$1n^2 \leq 5n^2 - 3n + 2$$
$$3n \leq 4n^2 + 2$$

$5n^2 - 3n + 2 \leq c_2 n^2$ This is true for $c_2 = 12$ and $n \geq 40$

$$5n^2 - 3n + 2 \leq 12n^2$$
$$2 \leq 7n^2 + 3n$$

So we can choose $c_1 = 4$, $c_2 = 5$, and $n_0 = 3$ to meet the condition. We can also choose $c_1 = 1$, $c_2 = 12$ and $n_0 = 40$. Therefore, $5n^2 - 3n + 2 \in \Theta\left(n^2\right)$

# Example Proof: $5n + 2$ is $\Theta(n)$

# Example Proof: $5n + 2$ is $\Theta(n)$

By definition $5n + 2 \in \Theta(n)$ if $c_1 n \leq 5n + 2 \leq c_2 n$ for $n \geq n_0$.

# Example Proof: $5n + 2$ is $\Theta(n)$

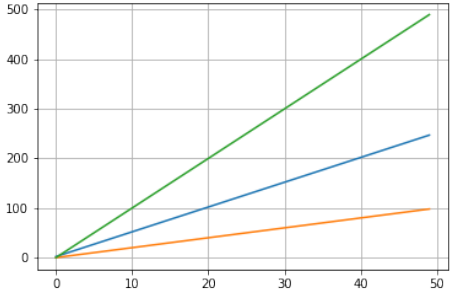
By definition $5n + 2 \in \Theta(n)$ if $c_1 n \leq 5n + 2 \leq c_2 n$ for $n \geq n_0$.
Let $c_1 = 2$, $c_2 = 10$, and $n_0 = 10$

# Example Proof: $5n + 2$ is $\Theta(n)$

By definition $5n + 2 \in \Theta(n)$ if $c_1 n \leq 5n + 2 \leq c_2 n$ for $n \geq n_0$.
Let $c_1 = 2$, $c_2 = 10$, and $n_0 = 10$
Is $2n \leq 5n + 2 \leq 10n$ true for $n \geq 10$?

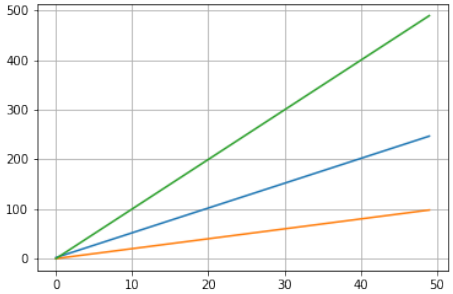
# Example Proof: $5n + 2$ is $\Theta(n)$

By definition $5n + 2 \in \Theta(n)$ if $c_1 n \leq 5n + 2 \leq c_2 n$ for $n \geq n_0$.
Let $c_1 = 2$, $c_2 = 10$, and $n_0 = 10$
Is $2n \leq 5n + 2 \leq 10n$ true for $n \geq 10$?



# Example Proof: $5n + 2$ is $\Theta(n)$

By definition $5n + 2 \in \Theta(n)$ if $c_1 n \leq 5n + 2 \leq c_2 n$ for $n \geq n_0$.
Let $c_1 = 2$, $c_2 = 10$, and $n_0 = 10$
As $2n \leq 5n + 2 \leq 10n$ for $n \geq 10$, $5n + 2 \in \Theta(n)$



# Example Proof: $5n + 2 \notin \Theta\left(n^2\right)$

# Example Proof: $5n + 2 \notin \Theta\left(n^2\right)$

Proof by contradiction: Assume the opposite and derive a contradiction.

# Example Proof: $5n + 2 \notin \Theta\left(n^2\right)$

Proof by contradiction: Assume that $5n + 2 \in \Theta\left(n^2\right)$

# Example Proof: $5n + 2 \notin \Theta\left(n^2\right)$

Proof by contradiction: Assume that $5n + 2 \in \Theta\left(n^2\right)$
Then, by the definition, $c_1 n^2 \leq 5n + 2 \leq c_2 n^2$ when $n \geq n_0$ for some $c_1, c_2, n_0$. That is:

- $c_1 n^2 \leq 5n + 2$
- $5n + 2 \leq c_2 n^2$

Which of these is more likely to lead to a contradiction?

# Example Proof: $5n + 2 \notin \Theta\left(n^2\right)$

Proof by contradiction: Assume that $5n + 2 \in \Theta\left(n^2\right)$
Then, by the definition, $c_1 n^2 \leq 5n + 2 \leq c_2 n^2$ when $n \geq n_0$ for some $c_1, c_2, n_0$. That is:

- $c_1 n^2 \leq 5n + 2$
- $5n + 2 \leq c_2 n^2$ True for $c_2 = 5, n \geq 5$

# Example Proof: $5n + 2 \notin \Theta\left(n^2\right)$

Proof by contradiction: Assume that $5n + 2 \in \Theta\left(n^2\right)$
Then, by the definition, $c_1 n^2 \leq 5n + 2 \leq c_2 n^2$ when $n \geq n_0$ for some $c_1, c_2, n_0$. That is:

- $c_1 n^2 \leq 5n + 2$
  Suppose this is true for $c_1 = k$
- $5n + 2 \leq c_2 n^2$

# Example Proof: $5n + 2 \notin \Theta\left(n^2\right)$

Proof by contradiction: Assume that $5n + 2 \in \Theta\left(n^2\right)$
Then, by the definition, $c_1 n^2 \leq 5n + 2 \leq c_2 n^2$ when $n \geq n_0$ for some $c_1, c_2, n_0$. That is:

- $c_1 n^2 \leq 5n + 2$
  Suppose this is true for $c_1 = k$

$$kn^2 \leq 5n + 2 \text{ for all } n \geq n_0$$

- $5n + 2 \leq c_2 n^2$

# Example Proof: $5n + 2 \notin \Theta\left(n^2\right)$

Proof by contradiction: Assume that $5n + 2 \in \Theta\left(n^2\right)$
Then, by the definition, $c_1 n^2 \leq 5n + 2 \leq c_2 n^2$ when $n \geq n_0$ for some $c_1, c_2, n_0$. That is:

- $c_1 n^2 \leq 5n + 2$
  Suppose this is true for $c_1 = k$

$$kn^2 \leq 5n + 2 \text{ for all } n \geq n_0$$
$$n \leq \frac{5}{k} + \frac{2}{kn} \text{ for all } n \geq n_0$$

# Example Proof: $5n + 2 \notin \Theta\left(n^2\right)$

Proof by contradiction: Assume that $5n + 2 \in \Theta\left(n^2\right)$
Then, by the definition, $c_1 n^2 \leq 5n + 2 \leq c_2 n^2$ when $n \geq n_0$ for some $c_1, c_2, n_0$. That is:

- $c_1 n^2 \leq 5n + 2$
  Suppose this is true for $c_1 = k$

$$kn^2 \leq 5n + 2 \text{ for all } n \geq n_0$$
$$n \leq \frac{5}{k} + \frac{2}{kn} \text{ for all } n \geq n_0$$

Let $n = \max(\frac{5}{k} + \frac{2}{kn_0}, n_0 + 1)$.

# Example Proof: $5n + 2 \notin \Theta\left(n^2\right)$

Proof by contradiction: Assume that $5n + 2 \in \Theta\left(n^2\right)$

Then, by the definition, $c_1 n^2 \leq 5n + 2 \leq c_2 n^2$ when $n \geq n_0$ for some $c_1, c_2, n_0$. That is:

- $c_1 n^2 \leq 5n + 2$
  Suppose this is true for $c_1 = k$

$$kn^2 \leq 5n + 2 \text{ for all } n \geq n_0$$

$$n \leq \frac{5}{k} + \frac{2}{kn} \text{ for all } n \geq n_0$$

Let $n = \max(\frac{5}{k} + \frac{2}{kn_0}, n_0 + 1)$. Therefore, $n \geq \frac{5}{k} + \frac{2}{kn_0}$ and $n > n_0$. As $n > n_0$,

$$\frac{5}{k} + \frac{2}{kn_0} \leq n \leq \frac{5}{k} + \frac{2}{kn}$$

$$\cancel{\frac{5}{k}} + \frac{2}{kn_0} \leq \cancel{\frac{5}{k}} + \frac{2}{kn}$$

$$\frac{\cancel{2}}{\cancel{k}n_0} \frac{\cancel{k}n\cancel{n_0}}{\cancel{2}} \leq \frac{\cancel{2}}{\cancel{k}\cancel{n}} \frac{\cancel{k}\cancel{n}n_0}{\cancel{2}}$$

$$n \leq n_0$$

which is a contradiction

# Example Proof: $5n + 2 \notin \Theta\left(n^2\right)$

Proof by contradiction: Assume that $5n + 2 \in \Theta\left(n^2\right)$
Then, by the definition, $c_1 n^2 \leq 5n + 2 \leq c_2 n^2$ when $n \geq n_0$ for some $c_1, c_2, n_0$. That is:

- $c_1 n^2 \leq 5n + 2$
  Suppose this is true for $c_1 = k$

$$kn^2 \leq 5n + 2 \text{ for all } n \geq n_0$$

$$n \leq \frac{5}{k} + \frac{2}{kn} \text{ for all } n \geq n_0$$

$$\infty = \lim_{n \to \infty} n$$

$$\frac{5}{k} = \lim_{n \to \infty} \frac{5}{k} + \frac{2}{kn}$$

# Simplifying Big-Θ

- If $d(n)$ is $\Theta(f(n))$ then $ad(n)$ is $\Theta(f(n))$ for constant $a > 0$
- If $d(n)$ is $\Theta(f(n))$ and $e(n)$ is $\Theta(g(n))$ then $d(n) + e(n)$ is $\Theta(f(n) + g(n))$
- If $d(n)$ is $\Theta(f(n))$ and $e(n)$ is $\Theta(g(n))$ then $d(n)e(n)$ is $\Theta(f(n)g(n))$
- If $d(n)$ is $\Theta(f(n))$ and $f(n)$ is $\Theta(g(n))$ then $d(n)$ is $\Theta(g(n))$
- If $f(n)$ is a polynomial of degree $d$, then $f(n)$ is $\Theta\left(n^d\right)$

- If $f(n) \in \Theta(g(n))$, then $g(n) \in \Theta(f(n))$

**N.B.** Including constant factors and lower-order terms in big-Θ notation is considered poor taste.

# Asymptotic Notation

Five kinds of notation:

- $\Theta$ “Big Theta” — Upper and Lower Bound
- $O$ “Big O” — Inclusive Upper Bound
- $\Omega$ “Big Omega” — Inclusive Lower Bound
- $o$ “Little O” — Exclusive Upper Bound
- $\omega$ “Little Omega” — Exclusive Lower Bound

# Big-O, Formally

$$\mathrm{O}\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

# Big-O, Formally

$$\mathrm{O}\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

This should look familiar...

$$\Theta\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- As with Θ, O defines a set of functions.
- A function is in the set if we can find appropriate $c$ and $n_0$

# Big-O, Formally

$$\mathrm{O}\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- As with $\Theta$, O defines a set of functions.
- A function is in the set if we can find appropriate $c$ and $n_0$

$$5n^2 - 3n + 2 \in \mathrm{O}\left(n^2\right)$$

because we can find $c = 5$ and $n_0 = 2$:

# Big-O, Formally

$$\mathrm{O}\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- As with Θ, O defines a set of functions.
- A function is in the set if we can find appropriate $c$ and $n_0$

$$5n^2 - 3n + 2 \in \mathrm{O}\left(n^2\right)$$

because we can find $c = 5$ and $n_0 = 2$:

$$5n^2 - 3n + 2 \leq 5n^2$$

$$\cancel{5n^2} - 3n + 2 \leq \cancel{5n^2}$$

$$2 \leq 3n \text{ for all } n \geq 2$$

# Big-O Examples

$$64 \in O(1)$$
$$\log n - \log\log n + 2 \in O(\log n)$$
$$\log n - \log\log n + 2 \in O(n)$$
$$5n + \log n \in O(n)$$
$$5n^2 + n\log n \in O\left(n^2\right)$$
$$5n^2 + n\log n \in O\left(n^3\right)$$

# Big-O Examples

$$64 \in O(1)$$
$$\log n - \log\log n + 2 \in O(\log n)$$
$$\log n - \log\log n + 2 \in O(n)$$
$$5n + \log n \in O(n)$$
$$5n^2 + n\log n \in O\left(n^2\right)$$
$$5n^2 + n\log n \in O\left(n^3\right)$$
$$5n^2 + n\log n \in O\left(n^4\right)$$
$$5n^2 + n\log n \in O\left(n^5\right)$$
$$5n^2 + n\log n \in O\left(2^n\right)$$

# Big-O Examples

$$
\begin{aligned}
64 &\in O(1) \\
\log n - \log\log n + 2 &\in O(\log n) \\
\log n - \log\log n + 2 &\in O(n) \\
5n + \log n &\in O(n) \\
5n^2 + n\log n &\in O\left(n^2\right) \\
5n^2 + n\log n &\in O\left(n^3\right) \\
5n^2 + n\log n &\in O\left(n^4\right) \\
5n^2 + n\log n &\in O\left(n^5\right) \\
5n^2 + n\log n &\in O\left(2^n\right)
\end{aligned}
$$

$$O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n\log n) \subseteq O\left(n^2\right) \subseteq O\left(n^2\log n\right) \subseteq O\left(n^3\right) \subseteq O\left(2^n\right) \subseteq O(n!)$$

# Big-O Examples

$$5n^2 + n\log n \in O\left(n^2\right)$$
$$5n^2 + n\log n \in O\left(n^3\right)$$
$$5n^2 + n\log n \in O\left(n^4\right)$$
$$5n^2 + n\log n \in O\left(n^5\right)$$
$$5n^2 + n\log n \in O\left(2^n\right)$$

$$O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n\log n) \subseteq O\left(n^2\right) \subseteq O\left(n^2\log n\right) \subseteq O\left(n^3\right) \subseteq O\left(2^n\right) \subseteq O(n!)$$

- Conventionally, use the *tightest* ("smallest") bound
  - Provides the most useful information

# Prove $\frac{1}{2}n^2 - 3n \in O\left(n^2\right)$

# Prove $\frac{1}{2}n^2 - 3n \in O\left(n^2\right)$

Definition:

$$O\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

# Prove $\frac{1}{2}n^2 - 3n \in O\left(n^2\right)$

Definition:

$$O\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Need to find $c$, $n_0$ such that $0 \leq \frac{1}{2}n^2 - 3n \leq cn^2$ for all $n \geq n_0$

# Prove $\frac{1}{2}n^2 - 3n \in O\left(n^2\right)$

Definition:

$$O\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Need to find $c$, $n_0$ such that $0 \leq \frac{1}{2}n^2 - 3n \leq cn^2$ for all $n \geq n_0$

$$0 \leq \frac{1}{2}n^2 - 3n \leq cn^2$$
$$0 \leq n^2 - 6n \leq 2cn^2$$

# Prove $\frac{1}{2}n^2 - 3n \in O\left(n^2\right)$

Definition:

$$O\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Need to find $c$, $n_0$ such that $0 \leq \frac{1}{2}n^2 - 3n \leq cn^2$ for all $n \geq n_0$

$$\begin{aligned} 0 \leq \frac{1}{2}n^2 - 3n &\leq cn^2 \\ 0 \leq n^2 - 6n &\leq 2cn^2 \end{aligned}$$

Let $n_0 = 10$ and $c = 6$

# Asymptotic Notation

Five kinds of notation:

- $\Theta$ “Big Theta” — Upper and Lower Bound
- $O$ “Big O” — Inclusive Upper Bound
- $\Omega$ “Big Omega” — Inclusive Lower Bound
- $o$ “Little O” — Exclusive Upper Bound
- $\omega$ “Little Omega” — Exclusive Lower Bound

# Big-Ω, Formally

$$\Omega\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

# Big-Ω, Formally

$$\Omega\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

$$\Theta\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- As with Θ, Ω defines a set of functions.
- A function is in the set if we can find appropriate $c$ and $n_0$

# Big-Ω, Formally

$$\Omega\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- As with Θ, Ω defines a set of functions.
- A function is in the set if we can find appropriate $c$ and $n_0$

$$5n^2 - 3n + 2 \in \Omega\left(n^2\right)$$

because we can find $c = \frac{1}{5}$ and $n_0 = 2$

# Prove $\frac{1}{2}n^2 - 3n \in \Omega\left(n^2\right)$

# Prove $\frac{1}{2}n^2 - 3n \in \Omega\left(n^2\right)$

Definition:

$$\Omega\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

# Prove $\frac{1}{2}n^2 - 3n \in \Omega\left(n^2\right)$

Definition:

$$\Omega\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Need to find $c$, $n_0$ such that $0 \leq cn^2 \leq \frac{1}{2}n^2 - 3n$ for all $n \geq n_0$

# Prove $\frac{1}{2}n^2 - 3n \in \Omega\left(n^2\right)$

Definition:

$$\Omega\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Need to find $c$, $n_0$ such that $0 \leq cn^2 \leq \frac{1}{2}n^2 - 3n$ for all $n \geq n_0$

$$0 \leq cn^2 \leq \frac{1}{2}n^2 - 3n$$
$$0 \leq 2cn^2 \leq n^2 - 6n$$

# Prove $\frac{1}{2}n^2 - 3n \in \Omega\left(n^2\right)$

Definition:

$$\Omega\left(g(n)\right) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Need to find $c$, $n_0$ such that $0 \leq cn^2 \leq \frac{1}{2}n^2 - 3n$ for all $n \geq n_0$

$$0 \leq cn^2 \leq \frac{1}{2}n^2 - 3n$$
$$0 \leq 2cn^2 \leq n^2 - 6n$$

Let $n_0 = 14$ and $c = \frac{1}{4}$
$6n < \frac{1}{2}n^2$ for $n \geq 14$. Therefore, $n^2 - 6n \geq \frac{1}{2}n^2$

# Big-Ω Examples

$$64 \in \Omega(1)$$
$$\log n - \log\log n + 2 \in \Omega(\log n)$$
$$\log n - \log\log n + 2 \in \Omega(1)$$
$$5n + \log n \in \Omega(n)$$
$$5n^2 + n\log n \in \Omega\left(n^2\right)$$
$$5n^2 + n\log n \in \Omega(n\log n)$$

# Big-Ω Examples

$$64 \in \Omega(1)$$
$$\log n - \log\log n + 2 \in \Omega(\log n)$$
$$\log n - \log\log n + 2 \in \Omega(1)$$
$$5n + \log n \in \Omega(n)$$
$$5n^2 + n\log n \in \Omega\left(n^2\right)$$
$$5n^2 + n\log n \in \Omega(n\log n)$$
$$5n^2 + n\log n \in \Omega(n)$$
$$5n^2 + n\log n \in \Omega(\log n)$$
$$5n^2 + n\log n \in \Omega(1)$$

# Big-Ω Examples

$$64 \in \Omega(1)$$
$$\log n - \log\log n + 2 \in \Omega(\log n)$$
$$\log n - \log\log n + 2 \in \Omega(1)$$
$$5n + \log n \in \Omega(n)$$
$$5n^2 + n\log n \in \Omega\left(n^2\right)$$
$$5n^2 + n\log n \in \Omega(n\log n)$$
$$5n^2 + n\log n \in \Omega(n)$$
$$5n^2 + n\log n \in \Omega(\log n)$$
$$5n^2 + n\log n \in \Omega(1)$$

$$\Omega(n!) \subset \Omega(2^n) \subset \Omega\left(n^3\right) \subset \Omega\left(n^2\log n\right) \subset \Omega\left(n^2\right) \subset \Omega(n\log n) \subset \Omega(n) \subset \Omega(\log n) \subset \Omega(1)$$

# Big-Ω Examples

$$5n^2 + n\log n \in \Omega\left(n^2\right)$$
$$5n^2 + n\log n \in \Omega\left(n\log n\right)$$
$$5n^2 + n\log n \in \Omega\left(n\right)$$
$$5n^2 + n\log n \in \Omega\left(\log n\right)$$
$$5n^2 + n\log n \in \Omega\left(1\right)$$

$$\Omega\left(n!\right) \subset \Omega\left(2^n\right) \subset \Omega\left(n^3\right) \subset \Omega\left(n^2\log n\right) \subset \Omega\left(n^2\right) \subset \Omega\left(n\log n\right) \subset \Omega\left(n\right) \subset \Omega\left(\log n\right) \subset \Omega\left(1\right)$$

- Conventionally, use the *tightest* ("largest") bound
  - Provides the most useful information

# Relationship between Θ, O and Ω

**Theorem**

*For any two functions $f(n)$ and $g(n)$ $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$*

# Relationship between Θ, O and Ω

**Theorem**

*For any two functions $f(n)$ and $g(n)$ $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$*

$$\Theta(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

$$O(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

$$\Omega(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{there exist positive constants } c, \text{ and } n_0 \\ \text{such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

# CSC3310 Algorithms

## Correctness

William Retert

Milwaukee School of Engineering

# Algorithms

**Algorithm** a step-by-step procedure for performing a task in a finite amount of time.

# Algorithms

**Algorithm** a step-by-step procedure for solving a computational problem in a finite amount of time.

**Computational Problem** Specifies in general terms relationship between input and output

# Algorithms

**Algorithm** a step-by-step procedure for solving a computational problem in a finite amount of time.

**Computational Problem** Specifies in general terms relationship between input and output

- Brief description
- Constraints on input
- Desired output

# Correctness

- An algorithm is *partially correct* if it always generates a correct result if/when it halts
  - "correct" meaning that the output matches the problem's specification
- An algorithm is totally correct if it is partially correct, and can be proven to terminate

  Thus, correctness proofs tend to have two parts
  - A proof of partial correctness
  - A proof of termination

# Correctness

- An algorithm is *partially correct* if it always generates a correct result if/when it halts
  - “correct” meaning that the output matches the problem’s specification
- An algorithm is totally correct if it is partially correct, and can be proven to terminate
  Thus, correctness proofs tend to have two parts
  - A proof of partial correctness
  - A proof of termination

# Example: Absolute Value

| | |
|---|---|
| Name | ABSOLUTE VALUE |
| Input | $a \in \mathbb{Z}$ |
| Output | $\lvert a \rvert$ |

1: **if** $a < 0$ **then**
2: $\quad r \leftarrow -a$
3: **else**
4: $\quad r \leftarrow a$
5: **end if**

# Example: Absolute Value

**Name** ABSOLUTE VALUE
**Input** $a \in \mathbb{Z}$
**Output** $|a|$

1: **if** $a < 0$ **then** ▷ $a < 0$
2: $\quad r \leftarrow -a$
3: **else** ▷ $a \geq 0$
4: $\quad r \leftarrow a$
5: **end if**

# Example: Absolute Value

**Name** ABSOLUTE VALUE
**Input** $a \in \mathbb{Z}$
**Output** $|a|$

1: **if** $a < 0$ **then** ▷ $a < 0$
2: $\quad r \leftarrow -a$ ▷ $a < 0$ and $r = -a$
3: **else** ▷ $a \geq 0$
4: $\quad r \leftarrow a$
5: **end if**

# Example: Absolute Value

Name ABSOLUTE VALUE
Input $a \in \mathbb{Z}$
Output $|a|$

1: **if** $a < 0$ **then** ▷ $a < 0$
2: $r \leftarrow -a$ ▷ $a < 0$ and $r = -a$
3: **else** ▷ $a \geq 0$
4: $r \leftarrow a$ ▷ $a \geq 0$ and $r = a$
5: **end if**

# Example: Absolute Value

**Name** ABSOLUTE VALUE
**Input** $a \in \mathbb{Z}$
**Output** $|a|$

1: **if** $a < 0$ **then** ▷ $a < 0$
2: $\quad r \leftarrow -a$ ▷ $a < 0$ and $r = -a \Rightarrow r = |a|$
3: **else** ▷ $a \geq 0$
4: $\quad r \leftarrow a$ ▷ $a \geq 0$ and $r = a \Rightarrow r = |a|$
5: **end if**

# Example: Absolute Value

**Name** ABSOLUTE VALUE
**Input** $a \in \mathbb{Z}$
**Output** $|a|$

1: **if** $a < 0$ **then** ▷ $a < 0$
2: $\quad r \leftarrow -a$ ▷ $a < 0$ and $r = -a \Rightarrow r = |a|$
3: **else** ▷ $a \geq 0$
4: $\quad r \leftarrow a$ ▷ $a \geq 0$ and $r = a \Rightarrow r = |a|$
5: **end if** ▷ $r = |a|$

# Example: Maximum Value

**Name** MAXIMUMVALUE
**Description** Find a value in an array that is $\geq$ every other value in the array
**Input** Array $A$
**Output** $a \in A$ such that $a \geq a_i$ for all $a_i \in A$

```
procedure FINDMAX(A)
    m ← A[0]
    i ← 1
    while i < A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

**Name** MAXIMUMVALUE
**Description** Find a value in an array that is $\geq$ every other value in the array
**Input** Array $A$
**Output** $a \in A$ such that $a \geq a_i$ for all $a_i \in A$

```
procedure FINDMAX(A)
    m ← A[0]                          ▷ m = A[0]
    i ← 1
    while i < A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

**Name** MAXIMUMVALUE
**Description** Find a value in an array that is $\geq$ every other value in the array
**Input** Array $A$
**Output** $a \in A$ such that $a \geq a_i$ for all $a_i \in A$

```
procedure FINDMAX(A)
    m ← A[0]                          ▷ m = A[0]
    i ← 1                             ▷ i = 1 and m = A[0]
    while i < A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

**Name** MAXIMUMVALUE
**Description** Find a value in an array that is $\geq$ every other value in the array
**Input** Array $A$
**Output** $a \in A$ such that $a \geq a_i$ for all $a_i \in A$

```
procedure FINDMAX(A)
    m ← A[0]                                          ▷ m = A[0]
    i ← 1                                             ▷ i = 1 and m = A[0]
    while i < A.length do                             ▷ i < A.length and i = 1 and m = A[0]
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

**Name** MAXIMUMVALUE
**Description** Find a value in an array that is $\geq$ every other value in the array
**Input** Array $A$
**Output** $a \in A$ such that $a \geq a_i$ for all $a_i \in A$

```
procedure FINDMAX(A)
    m ← A[0]                                              ▷ m = A[0]
    i ← 1                                                 ▷ i = 1 and m = A[0]
    while i < A.length do                                 ▷ i < A.length and i = 1 and m = A[0]
        if m < A[i] then                                  ▷ i < A.length and i = 1 and m = A[0] and m < A[1]
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

**Name** MAXIMUMVALUE
**Description** Find a value in an array that is $\geq$ every other value in the array
**Input** Array $A$
**Output** $a \in A$ such that $a \geq a_i$ for all $a_i \in A$

```
procedure FINDMAX(A)
    m ← A[0]                                              ▷ m = A[0]
    i ← 1                                                 ▷ i = 1 and m = A[0]
    while i < A.length do                                 ▷ i < A.length and i = 1 and m = A[0]
        if m < A[i] then                                  ▷ i < A.length and i = 1 and m = A[0] and A[0] < A[1]
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

**Name** MAXIMUMVALUE
**Description** Find a value in an array that is $\geq$ every other value in the array
**Input** Array $A$
**Output** $a \in A$ such that $a \geq a_i$ for all $a_i \in A$

```
procedure FINDMAX(A)
    m ← A[0]                                              ▷ m = A[0]
    i ← 1                                                 ▷ i = 1 and m = A[0]
    while i < A.length do                                 ▷ i < A.length and i = 1 and m = A[0]
        if m < A[i] then                                  ▷ i < A.length and i = 1 and m = A[0] and A[0] < A[1]
            m ← A[i]                                      ▷ i < A.length and i = 1 and m = A[1] and A[0] < A[1]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

**Name** MAXIMUMVALUE
**Description** Find a value in an array that is $\geq$ every other value in the array
**Input** Array $A$
**Output** $a \in A$ such that $a \geq a_i$ for all $a_i \in A$

**procedure** FINDMAX($A$)
  $m \leftarrow A[0]$ ▷ $m = A[0]$
  $i \leftarrow 1$ ▷ $i = 1$ and $m = A[0]$
  **while** $i <$ $A$.length **do** ▷ $i <$ $A$.length and $i = 1$ and $m = A[0]$
    **if** $m < A[i]$ **then** ▷ $i <$ $A$.length and $i = 1$ and $m = A[0]$ and $A[0] < A[1]$
      $m \leftarrow A[i]$ ▷ $i <$ $A$.length and $i = 1$ and $m = A[1]$ and $A[0] < A[1]$
    **end if** ▷ $i <$ $A$.length and $i = 1$ and $[(m = A[0]$ and $A[0] \geq A[1])$ or $(m = A[1]$ and $A[0] < A[1])]$
    $i \leftarrow i + 1$
  **end while**
  **return** $m$
**end procedure**

# Example: Maximum Value

**Name** MAXIMUMVALUE
**Description** Find a value in an array that is $\geq$ every other value in the array
**Input** Array $A$
**Output** $a \in A$ such that $a \geq a_i$ for all $a_i \in A$

```
procedure FINDMAX(A)
    m ← A[0]                                                          ▷ m = A[0]
    i ← 1                                                             ▷ i = 1 and m = A[0]
    while i < A.length do                                             ▷ i < A.length and i = 1 and m = A[0]
        if m < A[i] then                                              ▷ i < A.length and i = 1 and m = A[0] and A[0] < A[1]
            m ← A[i]                                                  ▷ i < A.length and i = 1 and m = A[1] and A[0] < A[1]
        end if          ▷ i < A.length and i = 1 and [(m = A[0] and A[0] ≥ A[1]) or (m = A[1] and A[0] < A[1])]
        i ← i + 1       ▷ i < A.length and i = 2 and [(m = A[0] and A[0] ≥ A[1]) or (m = A[1] and A[0] < A[1])]
    end while
    return m
end procedure
```

# Example: Maximum Value

**Name** MAXIMUMVALUE
**Description** Find a value in an array that is $\geq$ every other value in the array
**Input** Array $A$
**Output** $a \in A$ such that $a \geq a_i$ for all $a_i \in A$

```
procedure FINDMAX(A)
    m ← A[0]                                                          ▷ m = A[0]
    i ← 1                                                             ▷ i = 1 and m = A[0]
    while i < A.length do                                             ▷
i < A.length and i = 2 and [(m = A[0] and A[0] ≥ A[1]) or (m = A[1] and A[0] < A[1])]
        if m < A[i] then                                              ▷ i < A.length and i = 1 and m = A[0] and A[0] < A[1]
            m ← A[i]                                                  ▷ i < A.length and i = 1 and m = A[1] and A[0] < A[1]
        end if                ▷ i < A.length and i = 1 and [(m = A[0] and A[0] ≥ A[1]) or (m = A[1] and A[0] < A[1])]
        i ← i + 1             ▷ i < A.length and i = 2 and [(m = A[0] and A[0] ≥ A[1]) or (m = A[1] and A[0] < A[1])]
    end while
    return m
end procedure
```

# Example: Maximum Value

**Name** MAXIMUMVALUE
**Description** Find a value in an array that is $\geq$ every other value in the array
**Input** Array $A$
**Output** $a \in A$ such that $a \geq a_i$ for all $a_i \in A$

**procedure** FINDMAX($A$)
  $m \leftarrow A[0]$ ▷ $m = A[0]$
  $i \leftarrow 1$ ▷ $i = 1$ and $m = A[0]$
  **while** $i <$ $A$.length **do** ▷ $i <$ $A$.length and $i = 2$ and [($m = A[0]$ and $A[0] \geq A[1]$) or ($m = A[1]$ and $A[0] < A[1]$)]
    **if** $m < A[i]$ **then** ▷ $i <$ $A$.length and $i = 2$ and [($m = A[0]$ and $A[0] \geq A[1]$) or ($m = A[1]$ and $A[0] < A[1]$)] and $m < A[2]$
      $m \leftarrow A[i]$ ▷ $i <$ $A$.length and $i = 1$ and $m = A[1]$ and $A[0] < A[1]$
    **end if** ▷ $i <$ $A$.length and $i = 1$ and [($m = A[0]$ and $A[0] \geq A[1]$) or ($m = A[1]$ and $A[0] < A[1]$)]
    $i \leftarrow i + 1$ ▷ $i <$ $A$.length and $i = 2$ and [($m = A[0]$ and $A[0] \geq A[1]$) or ($m = A[1]$ and $A[0] < A[1]$)]
  **end while**
  **return** $m$
**end procedure**

# Example: Maximum Value

**Name** MAXIMUMVALUE
**Description** Find a value in an array that is $\geq$ every other value in the array
**Input** Array $A$
**Output** $a \in A$ such that $a \geq a_i$ for all $a_i \in A$

**procedure** FINDMAX($A$)
  $m \leftarrow A[0]$ ▷ $m = A[0]$
  $i \leftarrow 1$ ▷ $i = 1$ and $m = A[0]$
  **while** $i < A$.length **do** ▷
$i < A$.length and $i = 2$ and [($m = A[0]$ and $A[0] \geq A[1]$) or ($m = A[1]$ and $A[0] < A[1]$)]
    **if** $m < A[i]$ **then** ▷
$i < A$.length and $i = 2$ and [($m = A[0]$ and $A[0] \geq A[1]$) or ($m = A[1]$ and $A[0] < A[1]$)] and $m < A[2]$
      $m \leftarrow A[i]$ ▷ $i < A$.length and $i = 1$ and $m = A[1]$ and $A[0] < A[1]$
    **end if** ▷ $i < A$.length and $i = 1$ and [($m = A[0]$ and $A[0] \geq A[1]$) or ($m = A[1]$ and $A[0] < A[1]$)]
    $i \leftarrow i + 1$ ▷ $i < A$.length and $i = 2$ and [($m = A[0]$ and $A[0] \geq A[1]$) or ($m = A[1]$ and $A[0] < A[1]$)]
  **end while**
  **return** $m$
**end procedure**

We need a way to *generalize* loop behavior

# Loop Invariants

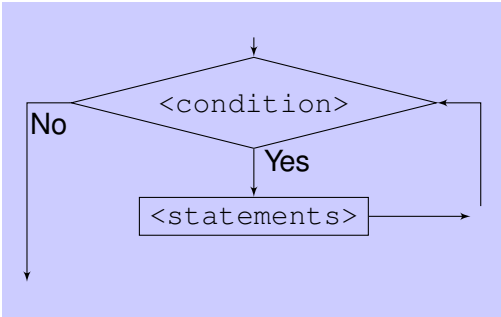
- Property that is **true** before each iteration of a loop
  - Can be broken in the loop body
  - Must be restored by end of loop
- Describe state during loop execution
  - What is the loop doing?
  - How does it accomplish that?

  Describe the state *at that moment!*
  - Do not summarize the behavior of the loop
- Inductive
  - Generally in terms of the loop counter
  - $i$ out of $n$ elements processed after iteration $i$

# Loop Invariants

- Property that is **true** before each iteration of a loop
  - Can be broken in the loop body
  - Must be restored by end of loop
- Describe state during loop execution
  - What is the loop doing?
  - How does it accomplish that?

  Describe the state *at that moment!*
  - Do not summarize the behavior of the loop
- Inductive
  - Generally in terms of the loop counter
  - *i* out of *n* elements processed after iteration *i*



# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i < A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]                                  ▷ m = A[0]
    i ← 1
    while i < A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]                          ▷ m = A[0]
    i ← 1                             ▷ i = 1 and m = A[0]
    while i < A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]                                  ▷ m = A[0]
    i ← 1                                     ▷ i = 1 and m = A[0]

    while i < A.length do                     ▷ Invariant: i < n + 1
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]                                  ▷ m = A[0]
    i ← 1                                     ▷ i = 1 and m = A[0]

    while i < A.length do                     ▷ Invariant: i < n + 1
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1                             ▷ i < A.length and i < n + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]                                  ▷ m = A[0]
    i ← 1                                     ▷ i = 1 and m = A[0]

    while i < A.length do                     ▷ Invariant: i < n + 1
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1                             ▷ i < n and i < n + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]                                   ▷ m = A[0]
    i ← 1                                      ▷ i = 1 and m = A[0]

    while i < A.length do                      ▷ Invariant: i < n + 1
        if m < A[i] then                       ▷ i < n and i < n + 1
            m ← A[i]
        end if
        i ← i + 1                              ▷ i − 1 < n and i − 1 < n + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]                                  ▷ m = A[0]
    i ← 1                                     ▷ i = 1 and m = A[0]

    while i < A.length do                     ▷ Invariant: i < n + 1
        if m < A[i] then                      ▷ i < n and i < n + 1
            m ← A[i]
        end if
        i ← i + 1                             ▷ i − 1 < n and i − 1 < n + 1
                                              ▷ i − 1 < n ⇒ i < n + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]                                   ▷ m = A[0]
    i ← 1                                      ▷ i = 1 and m = A[0]

    while i < A.length do                      ▷ Invariant: i < n + 1
        if m < A[i] then                       ▷ i < n and i < n + 1
            m ← A[i]
        end if
        i ← i + 1                              ▷ i − 1 < n and i − 1 < n + 1

    end while                                  ▷ i − 1 < n ⇒ i < n + 1
    return m                                   ▷ i ≥ n and i < n + 1
end procedure
```

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]                                          ▷ m = A[0]
    i ← 1                                             ▷ i = 1 and m = A[0]

    while i < A.length do                             ▷ Invariant: i < n + 1
        if m < A[i] then                              ▷ i < n and i < n + 1
            m ← A[i]
        end if
        i ← i + 1                                     ▷ i − 1 < n and i − 1 < n + 1

    end while                                         ▷ i − 1 < n ⇒ i < n + 1
                                                      ▷ i ≥ n and i < n + 1
    return m                                          ▷ ⇒ i = n
end procedure
```

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]                                   ▷ m = A[0]
    i ← 1                                      ▷ i = 1 and m = A[0]
                                               ▷ Invariant: i < n + 1
    while i < A.length do                      ▷ i < n and i < n + 1
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1                              ▷ i − 1 < n and i − 1 < n + 1
                                               ▷ i − 1 < n ⇒ i < n + 1
    end while                                  ▷ i ≥ n and i < n + 1
                                               ▷ ⇒ i = n
    return m                                   ▷ ????
end procedure
```

What is the problem with this invariant?

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$

$$m = 3$$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 3$$
$$i = 1$$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$

$$m = 3$$

$$i = 1$$

**Invariant:** $m = a_0$?

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 3$$
$$i = 1$$

**Invariant:** $m = a_0$?
Test is **true**; $1 < 4$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 3$$
$$i = 1$$

**Invariant:** $m = a_0$?
Test is **true**; $3 < 5$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$

$$m = 5$$

$$i = 1$$

**Invariant:** $m = a_0$?

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 2$$

**Invariant:** $m = a_0$?

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 2$$

**Invariant:** $m = a_0? \sqcap m = a_1 \geq a_0?$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 2$$

**Invariant:** $m \geq a_0? \sqcap m = a_1 \geq a_0?$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 2$$

**Invariant:** $m \geq a_0$?
AND $m \geq a_0$ and $m \geq a_1$?

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 2$$

**Invariant:** $m \geq a_j$ for $0 \leq j < i$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 2$$

**Invariant:** $m \geq a_j$ for $0 \leq j < i$
Test is **true**; $2 < 4$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 2$$

**Invariant:** $m \geq a_j$ for $0 \leq j < i$
Test is **false**; $5 \not< 2$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 3$$

**Invariant:** $m \geq a_j$ for $0 \leq j < i$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 3$$

**Invariant:** $m \geq a_j$ for $0 \leq j < i$ Still true for $i = 3$?

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 3$$

**Invariant:** $m \geq a_j$ for $0 \leq j < i$
Test is **true**; $3 < 4$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 3$$

**Invariant:** $m \geq a_j$ for $0 \leq j < i$
Test is **false**; $5 \not< 4$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 4$$

**Invariant:** $m \geq a_j$ for $0 \leq j < i$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 4$$

**Invariant:** $m \geq a_j$ for $0 \leq j < i$ Still true for $i = 4$?

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 4$$

**Invariant:** $m \geq a_j$ for $0 \leq j < i$
Test is **false**; $4 \not< 4$

# Example: Maximum Value

Loop Invariant:

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- Inductive — $i$ out of $n$ elements processed after iteration $i$

```
procedure FindMax(A)
    m ← A[0]
    i ← 1
    while i ≠ A.length do
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

$$A = \langle 3, 5, 2, 4 \rangle$$
$$m = 5$$
$$i = 4$$

**Invariant:** $m \geq a_j$ for $0 \leq j < i$
$i = 4 = n$, so $m$ is at least as large as every array element

# Example: Maximum Value

- Property that is **true** before each iteration of a loop
- Describe state during loop execution

```
procedure FindMax(A)
    m ← A[0]
    i ← 1

    while i ≠ A.length do                ▷ Invariant: m ≥ a_j for all 0 ≤ j < i
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

- Property that is **true** before each iteration of a loop
  To prove, show is true:
  - At the start of the loop (first iteration)
  - At the end of the loop (later iterations)

```
procedure FindMax(A)
    m ← A[0]                                   ▷ m = a_0
    i ← 1                                      ▷ i = 1 and m = a_0

    while i ≠ A.length do                      ▷ Invariant: m ≥ a_j for all 0 ≤ j < i
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Example: Maximum Value

- Property that is **true** before each iteration of a loop
  To prove, show is true:
  - At the start of the loop (first iteration)
  - At the end of the loop (later iterations)

```
procedure FindMax(A)
    m ← A[0]
    i ← 1

    while i ≠ A.length do                          ▷ Invariant: m ≥ a_j for all 0 ≤ j < i

        if m < A[i] then
            m ← A[i]
        end if

        i ← i + 1                                  ▷ i ≠ n and (m ≥ a_j for all 0 ≤ j < i)

    end while                                      ▷ m ≥ a_j for all 0 ≤ j ≤ i
    return m
end procedure                                      ▷ m ≥ a_j for all 0 ≤ j < i
```

# Example: Maximum Value

- Describe state during loop execution
- *Use* invariant to help show output postcondition

```
procedure FindMax(A)
    m ← A[0]
    i ← 1

    while i ≠ A.length do                    ▷ Invariant: m ≥ a_j for all 0 ≤ j < i
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while

    return m                                 ▷ i = n and (m ≥ a_j for all 0 ≤ j < i)
end procedure                                ▷ m ≥ a_j for all 0 ≤ j < n
```

# Example: Maximum Value

- Property that is **true** before each iteration of a loop
- Describe state during loop execution
- *Use* invariant to help show output postcondition
- Separately, show that the loop terminates

```
procedure FindMax(A)
    m ← A[0]
    i ← 1

    while i ≠ A.length do                ▷ Invariant: m ≥ a_j for all 0 ≤ j < i
        if m < A[i] then
            m ← A[i]
        end if
        i ← i + 1
    end while
    return m
end procedure
```

# Proving Correctness

- To formally prove correctness, need the step-by-step walkthrough
- For class purposes, a less-formal argument will often suffice
  - Clearly state the loop invariant

> $m \geq a_j$ for all $0 \leq j < i$

  - Argue convincingly that it is true at the start of the loop

> Initially, $i$ is 1, and $m$ is $a_0$. The only $j$ less than 1 is 0, and $a_0 \geq a_0$.

  - Argue convincingly that it is preserved across an arbitrary iteration of the loop.

> We know from the loop invariant that all values up to $a_i$ are no larger than $m$. If the $i^{\text{th}}$ value is also less-than-or-equal-to $m$, then the invariant will hold after incrementing $i$. If the $i^{\text{th}}$ value is larger than, then we update $m$, making it equal to $a_i$ and transitively greater than all elements before $a_i$, so again the invariant holds.

# CSC3310 Algorithms

## Divide and Conquer

William Retert

Milwaukee School of Engineering

# Divide and Conquer

Problem-solving strategy

**Divide** Break problem into smaller subproblems
- Similar structure to original problem

**Conquer** Solve the subproblems
- Generally recursively

**Combine** Combine subproblem solutions into a solution for the original problem

# Example: Find Minimum

| | |
|---|---|
| Name | FINDMINIMUMVALUE |
| Description | Given a sequence of $n$ values, return a minimal value from the sequence |
| Input | A sequence of $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ |
| Output | A value $a_i$ such that $a_i$ is in the sequence, and $a_i \leq a_j$ for all $0 \leq j < n$ |

# Example: Find Minimum

**Name** FINDMINIMUMVALUE

**Description** Given a sequence of $n$ values, return a minimal value from the sequence

Finding the minimum value with divide and conquer:

**Divide** split the array into 2 subarrays

**Conquer** Find the minimum value for each subarray

**Combine** The smallest value in the array will be the smaller of the two results

# Example: Find Minimum

**Name** FINDMINIMUMVALUE

**Description** Given a sequence of $n$ values, return a minimal value from the sequence

Finding the minimum value with divide and conquer:

**Divide** split the array into 2 subarrays

**Conquer** Find the minimum value for each subarray

**Combine** The smallest value in the array will be the smaller of the two results

# Example: Find Minimum

**Name** FINDMINIMUMVALUE

**Description** Given a sequence of $n$ values, return a minimal value from the sequence

Finding the minimum value with divide and conquer:

**Divide** split the array into 2 subarrays

**Conquer** Find the minimum value for each subarray

**Combine** The smallest value in the array will be the smaller of the two results

# Example: Find Minimum

**Name** FINDMINIMUMVALUE

**Description** Given a sequence of $n$ values, return a minimal value from the sequence

Finding the minimum value with divide and conquer:

**Divide** split the array into 2 subarrays

**Conquer** Find the minimum value for each subarray

**Combine** The smallest value in the array will be the smaller of the two results

# Example: Find Minimum

Finding the minimum value with divide and conquer:

- **Divide** split the array into 2 subarrays
- **Conquer** Find the minimum value for each subarray
- **Combine** The smallest value in the array will be the smaller of the two results

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

**Correctness**

$n = 1$ The only element is the smallest

$n > 1$ Either the smallest element will be one of the first $n - 1$ elements, and FindMin will return it, or it will be the last element.

Thus the smaller of *left* and *right* is the smallest element.

Note that we assume FindMin is correct in order to prove it is correct. More precisely, we use induction on $n$.

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

**Time Complexity**

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ 5 + T(n-1) & \text{otherwise} \end{cases}$$

This is a *recurrence* relation.

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

## Time Complexity

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ 5 + T(n-1) & \text{otherwise} \end{cases}$$

This is a *recurrence* relation.

## Substitution

$$T(n) = 5 + T(n-1)$$

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

## Substitution

$$T(n) = 5 + T(n-1)$$
$$T(n-1) = 5 + T(n-2)$$

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

## Substitution

$$T(n) = 5 + 5 + T(n-2)$$
$$T(n-1) = 5 + T(n-2)$$

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

## Substitution

$$T(n) = 5 + 5 + T(n-2)$$
$$T(n-2) = 5 + T(n-3)$$

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

## Substitution

$$T(n) = 5 + 5 + 5 + T(n-3)$$
$$T(n-2) = 5 + T(n-3)$$

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

## Substitution

$$T(n) = 5 + 5 + 5 + \cdots + 5 + T(n - (n - 1))$$

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

## Substitution

$$T(n) = \overbrace{5 + 5 + 5 + \cdots 5}^{n-1 \text{ times}} + T(n - (n - 1))$$

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

## Substitution

$$T(n) = \overbrace{5 + 5 + 5 + \cdots 5}^{n-1 \text{ times}} + T(n - (n - 1))$$

$$T(n - (n - 1)) = T(1) = 3$$

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

## Substitution

$$T(n) = \overbrace{5 + 5 + 5 + \cdots 5}^{n-1 \text{ times}} + T(n - (n - 1))$$

$$T(n - (n - 1)) = T(1) = 3$$

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

## Substitution

$$T(n) = \overbrace{5 + 5 + 5 + \cdots 5}^{n-1 \text{ times}} + 3$$

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n − 1])
        right ← a[n − 1]
        return left < right?left : right
    end if
end procedure
```

## Substitution

$$
\begin{aligned}
T(n) &= \overbrace{5 + 5 + 5 + \cdots 5}^{n-1 \text{ times}} + 3 \\
&= (n-1)5 + 3 \\
T(n) &= 5n - 2
\end{aligned}
$$

# Example: Find Minimum

**procedure** FINDMIN(a)
    $n = a$.length
    **if** $n = 1$ **then**
        **return** $a[0]$
    **else**
        $left \leftarrow$ FINDMIN($a[0 : \frac{n}{2}]$)
        $right \leftarrow$ FINDMIN($a[\frac{n}{2} : n]$)
        **return** $left < right ? left : right$
    **end if**
**end procedure**

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n/2])
        right ← FindMin(a[n/2 : n])
        return left < right?left : right
    end if
end procedure
```

**Correctness**

The smallest element must either be in the first half of the array or the second half. As *left* is the smallest element in the first half, and *right* is the smallest in the second, the smaller of the two must be the smallest overall

(We can use *strong* induction to show this formally.)

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : ⌊n/2⌋])
        right ← FindMin(a[⌊n/2⌋ : n])
        return left < right?left : right
    end if
end procedure
```

**Correctness**

(We can use *strong* induction to show this formally.)

*Hypothesis:* FindMin returns a minimal element in an an array of $n$ elements

*Base Case:* If $n = 1$, the array contains a single element, which is the smallest.

*Inductive Step:* Assume FindMin returns a minimal element of any array whose length is less than $n$

By that assumption

$$\begin{aligned} left &\leq a[i] && (\forall 0 \leq i < \lfloor \tfrac{n}{2} \rfloor) \\ right &\leq a[j] && (\forall \lfloor \tfrac{n}{2} \rfloor \leq j < n) \\ \min(left, right) &\leq left \leq a[i] && (\forall 0 \leq i < \lfloor \tfrac{n}{2} \rfloor) \\ \min(left, right) &\leq right \leq a[j] && (\forall \lfloor \tfrac{n}{2} \rfloor \leq j < n) \\ \min(left, right) &\leq a[k] && (\forall 0 \leq k < n) \end{aligned}$$

$\therefore$ The hypothesis holds for all positive integers $n$.

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n/2])
        right ← FindMin(a[n/2 : n])
        return left < right?left : right
    end if
end procedure
```

**Time Complexity**

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ 5 + 2T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

This is a *recurrence* relation.

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n/2])
        right ← FindMin(a[n/2 : n])
        return left < right?left : right
    end if
end procedure
```

**Time Complexity**

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ 5 + T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) & \text{otherwise} \end{cases}$$

This is a *recurrence* relation.

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n/2])
        right ← FindMin(a[n/2 : n])
        return left < right?left : right
    end if
end procedure
```

**Time Complexity**

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ 5 + 2T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

This is a *recurrence* relation.

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n/2])
        right ← FindMin(a[n/2 : n])
        return left < right?left : right
    end if
end procedure
```

## Time Complexity

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ 5 + 2T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

This is a *recurrence* relation.

## Substitution

$$T(n) = 5 + 2T(\frac{n}{2})$$

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n/2])
        right ← FindMin(a[n/2 : n])
        return left < right?left : right
    end if
end procedure
```

## Time Complexity

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ 5 + 2T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

This is a *recurrence* relation.

## Substitution

$$\begin{aligned} T(n) &= 5 + 2T(\frac{n}{2}) \\ &= 5 + 2(5 + 2T(\frac{n}{4})) \\ &= 5 + 2(5 + 2(5 + 2T(\frac{n}{8}))) \end{aligned}$$

# Example: Find Minimum

```
procedure FindMin(a)
    n = a.length
    if n = 1 then
        return a[0]
    else
        left ← FindMin(a[0 : n/2])
        right ← FindMin(a[n/2 : n])
        return left < right?left : right
    end if
end procedure
```

## Time Complexity

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ 5 + 2T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

This is a *recurrence* relation.

## Substitution

$$\begin{aligned} T(n) &= 5 + 2T(\frac{n}{2}) \\ &= 5 + 2(5 + 2T(\frac{n}{4})) \\ &= 5 + 2(5 + 2(5 + 2T(\frac{n}{8}))) \\ &= \dots \end{aligned}$$

We need a more systematic approach to solving recurrences

# Divide-and-Conquer

**Divide** Break problem into smaller subproblems

**Conquer** Solve the subproblems

**Combine** Combine subproblem solutions into a solution for the original problem

If there are $a$ subproblems of size $\frac{n}{b}$, we can represent the time complexity with the recurrence

$$T(n) = D(n) + aT(\frac{n}{b}) + C(n)$$

$D(n)$ is the time to divide the problem

$C(n)$ is the time to combine the solutions

# Solving Recurrence Relations

Convert the recurrence into a “closed-form” solution

- Closed-form is *not* recursive

Example 1 Recursive $T(n) = \begin{cases} 0 & n = 1 \\ T(\frac{n}{2}) + 1 & \text{otherwise} \end{cases}$

Closed-form $T(n) \in \Theta(\log_2 n)$

Example 2 Recursive $T(n) = \begin{cases} 6 & n = 1 \\ 2T(n-1) & \text{otherwise} \end{cases}$

Closed-form $T(n) = 6 \times 2^n \in \Theta(2^n)$

# Methods for Solving Recurrences

- Repeated substitution
- Guess and check
- Recurrence Trees
- Master method
- Generating Functions

# Methods for Solving Recurrences

- Repeated substitution
- **Guess and check**
- Recurrence Trees
- **Master method**
- Generating Functions

In this class, we will only discuss the *guess-and-check* and *master* methods

# Guess and Check

Used for upper/lower asymptotic bounds
Approach:

1. guess the closed form
2. substitute the guess into the right-hand side
3. simplify the result

If the bound still holds, we know the guess was accurate

# Upper Bound

$$T(n) = 2T(n/2) + 5$$

We will guess big-O values to find an upper bound

$T(n) \in O\left(n^2\right)$ If so, then $T(n) \leq cn^2$, and $T(\frac{n}{2}) \leq c\left(\frac{n}{2}\right)^2$. Then

$$T(n) \leq 2(c\left(\frac{n}{2}\right)^2) + 5$$
$$= \frac{c}{2}n^2 + 5$$

We then show that this is bounded by $cn^2$ (for some $c$, $n_0$)

$$\frac{c}{2}n^2 + 5 \leq cn^2$$
$$5 \leq \frac{c}{2}n^2$$
$$\frac{10}{c} \leq n^2 \text{ which is true for } c = 10, n_0 = 1$$

# Upper Bound

$$T(n) = 2T(n/2) + 5$$

We will guess big-O values to find an upper bound

$T(n) \in O(n^2)$

$T(n) \in O(n \log n)$ Show $T(n) \leq cn \log n$ assuming $T(\frac{n}{2}) \leq c\frac{n}{2} \log \frac{n}{2}$.

$$\begin{aligned} T(n) &\leq 2\left(c\frac{n}{2}\log\frac{n}{2}\right) + 5 \\ &= cn\log\frac{n}{2} + 5 \\ &= cn\log n - cn\log 2 + 5 \end{aligned}$$

---

$$cn\log n - cn\log 2 + 5 \stackrel{?}{\leq} cn\log n$$

$$5 \stackrel{?}{\leq} (c\log 2)n, \text{ true for } c = \frac{1}{\log 2}, n_0 = 5$$

# Upper Bound

$$T(n) = 2T(n/2) + 5$$

We will guess big-O values to find an upper bound

$T(n) \in \mathrm{O}\left(n^2\right)$

$T(n) \in \mathrm{O}\left(n \log n\right)$

$T(n) \in \mathrm{O}\left(n\right)$ Show $T(n) \leq cn$ assuning $T(\frac{n}{2}) \leq c\frac{n}{2}$.

$$\begin{aligned} T(n) &\leq 2\left(c\frac{n}{2}\right) + 5 \\ &= cn + 5, \text{ but} \\ cn + 5 &\not\leq cn \end{aligned}$$

**That didn't work!**

# Upper Bound

$$T(n) = 2T(n/2) + 5$$

We will guess big-O values to find an upper bound

$T(n) \in O\left(n^2\right)$

$T(n) \in O\left(n \log n\right)$

$T(n) \in O\left(n - 5\right)$ We show $T(n) \leq c(n-5)$ assuming $T(\frac{n}{2}) \leq c(\frac{n}{2} - 5)$

$$\begin{aligned} T(n) &\leq 2c\left(\frac{n}{2} - 5\right) + 5 \\ &= cn - 10c + 5 \end{aligned}$$

$$cn - 10c + 5 \overset{?}{\leq} c(n-5) = cn - 5c$$

$$5 \overset{?}{\leq} 5c \text{ true for } c = 1, n_0 = 2$$

# Upper Bound

$$T(n) = 2T(n/2) + 5$$

We will guess big-O values to find an upper bound

$T(n) \in O\left(n^2\right)$

$T(n) \in O\left(n \log n\right)$

$T(n) \in O\left(n\right)$ We show $T(n) \leq c(n-5)$ assuming $T(\frac{n}{2}) \leq c(\frac{n}{2} - 5)$

$$\begin{aligned} T(n) &\leq 2c\left(\frac{n}{2} - 5\right) + 5 \\ &= cn - 10c + 5 \end{aligned}$$

$$cn - 10c + 5 \overset{?}{\leq} c(n-5) = cn - 5c$$

$$5 \overset{?}{\leq} 5c \text{ true for } c = 1, n_0 = 2$$

# Upper Bound

$$T(n) = 2T(n/2) + 5$$

We will guess big-O values to find an upper bound

$T(n) \in O\left(n^2\right)$

$T(n) \in O\left(n \log n\right)$

$T(n) \in O\left(n\right)$

$T(n) \in O\left(\log n\right)$ If so, we can show $T(n) \leq c \log n$ assuming $T(\frac{n}{2}) \leq c \log \frac{n}{2}$

$$\begin{aligned} T(n) &\leq 2\left(c \log \frac{n}{2}\right) + 5 \\ &= 2c \log n - 2c \log 2 + 5 \end{aligned}$$

$$2c \log n - 2c \log 2 + 5 \overset{?}{\leq} c \log n$$

$$c \log n \overset{?}{\leq} 2c \log 2 - 5 \qquad \text{Cannot be true as } n \to \infty$$

# Upper Bound

$$T(n) = 2T(n/2) + 5$$

We will guess big-O values to find an upper bound

$T(n) \in \mathrm{O}\left(n^2\right)$

$T(n) \in \mathrm{O}\left(n \log n\right)$

$T(n) \in \mathrm{O}\left(n\right)$

$T(n) \notin \mathrm{O}\left(\log n\right)$ If so, we can show $T(n) \leq c \log n$ assuming $T(\frac{n}{2}) \leq c \log \frac{n}{2}$

$$\begin{aligned} T(n) &\leq 2\left(c \log \frac{n}{2}\right) + 5 \\ &= 2c \log n - 2c \log 2 + 5 \end{aligned}$$

$$2c \log n - 2c \log 2 + 5 \overset{?}{\leq} c \log n$$

$$c \log n \overset{?}{\leq} 2c \log 2 - 5 \qquad \text{Cannot be true as } n \to \infty$$

# Upper Bound

$$T(n) = 2T(n/2) + 5$$

We will guess big-O values to find an upper bound

$T(n) \in \mathrm{O}\left(n^2\right)$

$T(n) \in \mathrm{O}\left(n \log n\right)$

$T(n) \in \mathrm{O}\left(n\right)$

$T(n) \notin \mathrm{O}\left(\log n\right)$ If so, we can show $T(n) \leq c \log n$ assuming $T(\frac{n}{2}) \leq c \log \frac{n}{2}$

$$\begin{aligned} T(n) &\leq 2\left(c \log \frac{n}{2}\right) + 5 \\ &= 2c \log n - 2c \log 2 + 5 \end{aligned}$$

$$2c \log n - 2c \log 2 + 5 \overset{?}{\leq} c \log n$$

$$c \log n \overset{?}{\leq} 2c \log 2 - 5 \qquad \text{Cannot be true as } n \to \infty$$

$\mathrm{O}\left(n\right)$ is the tightest upper bound we found

# Lower Bound

$$T(n) = 2T(n/2) + 5$$

We will guess big-$\Omega$ values to find an lower bound Since we know that $T(n) \in \mathrm{O}(n)$, we can start at

$T(n) \in \Omega(n)$ We show $cn \leq T(n)$ assuming $c\frac{n}{2} \leq T(\frac{n}{2})$.

$$\begin{aligned} 2(c\frac{n}{2}) + 5 &\leq T(n) \\ cn + 5 &\leq T(n) \\ \hline cn &\overset{?}{\leq} cn + 5 \qquad \text{Always true!} \end{aligned}$$

# Lower Bound

$$T(n) = 2T(n/2) + 5$$

We will guess big-$\Omega$ values to find an lower bound

$T(n) \in \Omega(n)$

$T(n) \in \Omega(n \log n)$ True if we show $cn \log n \leq T(n)$ assuming $c\frac{n}{2} \log \frac{n}{2} \leq T(\frac{n}{2})$. Then

$$\begin{aligned}
2\left(c\frac{n}{2}\log\frac{n}{2}\right) + 5 &\leq T(n) \\
cn\log\frac{n}{2} + 5 &\leq T(n) \\
cn\log n - cn\log 2 + 5 &\leq T(n) \\
\hline
cn\log n &\stackrel{?}{\leq} cn\log n - cn\log 2 + 5 \\
cn\log 2 &\stackrel{?}{\leq} 5 \textbf{ false} \text{ for large } n
\end{aligned}$$

# Lower Bound

$$T(n) = 2T(n/2) + 5$$

We will guess big-$\Omega$ values to find an lower bound

$T(n) \in \Omega(n)$

$T(n) \notin \Omega(n \log n)$

# Guess and-Check Method

- Find upper and/or lower bounds
  - not $\Theta$ directly.
  - but if it is $O(n)$ and $\Omega(n)\ldots$
- Guessing Strategy
  - Start with a high guess (e.g. $O\left(n^3\right)$) and tighten the bound until you cannot.
    - Be careful! A guess failing does not mean a *proportional* guess cannot succeed.

# Master Theorem

Suppose $T(n)$ is a recurrence of the form

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta\left(f(n)\right)$

# Master Theorem

Suppose $T(n)$ is a recurrence of the form

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta\left(f(n)\right)$

**Intuition:** $aT(\frac{n}{b}) \approx n^{\log_b a}$. The first case is when this dominates $f(n)$; the third is when $f(n)$ dominates it. When they are proportional, we get a logrithmic factor.

# Master Theorem

Suppose $T(n)$ is a recurrence of the form

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$

  **N.B.** $k = 0$ is an important special case. If $f(n) \in \Theta\left(n^{\log_b a}\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta\left(f(n)\right)$

**Intuition:** $aT(\frac{n}{b}) \approx n^{\log_b a}$. The first case is when this dominates $f(n)$; the third is when $f(n)$ dominates it. When they are proportional, we get a logrithmic factor.

# Master Theorem

Suppose $T(n)$ is a recurrence of the form

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$

  **N.B.** $k = 0$ is an important special case. If $f(n) \in \Theta\left(n^{\log_b a}\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta\left(f(n)\right)$

Recall that divide-and-conquer algorithms tend to have the form

$$T(n) = D(n) + aT(\frac{n}{b}) + C(n)$$

# Case 1

Let's solve

$$T(n) = 9T(\frac{n}{3}) + n$$

# Case 1

Let's solve

$$T(n) = 9T(\frac{n}{3}) + n$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta\left(f(n)\right)$

# Case 1

Let's solve

$$T(n) = 9T(\frac{n}{3}) + n$$

Then,

$$a = 9$$
$$b = 3$$
$$f(n) = n$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta\left(f(n)\right)$

# Case 1

Let's solve

$$T(n) = 9T(\frac{n}{3}) + n$$

Then,

$$a = 9$$
$$b = 3$$
$$f(n) = n$$
$$n^{\log_3 9} = n^2$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta(f(n))$

# Case 1

Let's solve

$$T(n) = 9T(\frac{n}{3}) + n$$

Then,

$$\begin{aligned} a &= 9 \\ b &= 3 \\ f(n) &= n \\ n^{\log_3 9} &= n^2 \\ n &\in O\left(n^{2-\frac{1}{2}}\right) \end{aligned}$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta(f(n))$

# Case 1

Let's solve

$$T(n) = 9T(\frac{n}{3}) + n$$

Then,

$$\begin{aligned} a &= 9 \\ b &= 3 \\ f(n) &= n \\ n^{\log_3 9} &= n^2 \\ n &\in \mathrm{O}\left(n^{2-\frac{1}{2}}\right) \end{aligned}$$

Therefore, $T(n) \in \Theta\left(n^2\right)$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in \mathrm{O}\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$

# Case 2

Let's solve

$$T(n) = T(\frac{2n}{3}) + 1$$

# Case 2

Let's solve

$$T(n) = T(\frac{2n}{3}) + 1$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta\left(f(n)\right)$

# Case 2

Let's solve

$$T(n) = T(\frac{2n}{3}) + 1$$

Then,

$$a = 1$$
$$b = \frac{3}{2}$$
$$f(n) = 1$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta\left(f(n)\right)$

# Case 2

Let's solve

$$T(n) = T(\frac{2n}{3}) + 1$$

Then,

$$a = 1$$
$$b = \frac{3}{2}$$
$$f(n) = 1$$
$$n^{\log_{\frac{3}{2}} 1} = n^0 = 1$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta\left(f(n)\right)$

# Case 2

Let's solve

$$T(n) = T(\frac{2n}{3}) + 1$$

Then,

$$\begin{aligned} a &= 1 \\ b &= \frac{3}{2} \\ f(n) &= 1 \\ n^{\log_{\frac{3}{2}} 1} &= n^0 = 1 \\ 1 &\in \Theta(1) \end{aligned}$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta(f(n))$

# Case 2

Let's solve

$$T(n) = T(\frac{2n}{3}) + 1$$

Then,

$$\begin{aligned} a &= 1 \\ b &= \frac{3}{2} \\ f(n) &= 1 \\ n^{\log_{\frac{3}{2}} 1} &= n^0 = 1 \\ 1 &\in \Theta(1) \\ k &= 0 \end{aligned}$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$

# Case 2

Let's solve

$$T(n) = T(\frac{2n}{3}) + 1$$

Then,

$$\begin{aligned} a &= 1 \\ b &= \frac{3}{2} \\ f(n) &= 1 \\ n^{\log_{\frac{3}{2}} 1} &= n^0 = 1 \\ 1 &\in \Theta(1) \\ k &= 0 \end{aligned}$$

Therefore, $T(n) \in \Theta(\log n)$

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$

# Case 3

Let's solve

$$T(n) = 3T(\frac{n}{4}) + n \log n$$

# Case 3

Let's solve

$$T(n) = 3T(\frac{n}{4}) + n \log n$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta\left(f(n)\right)$

# Case 3

Let's solve

$$T(n) = 3T(\frac{n}{4}) + n \log n$$

Then,

$$a = 3$$
$$b = 4$$
$$f(n) = n \log n$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta(f(n))$

# Case 3

Let's solve

$$T(n) = 3T(\frac{n}{4}) + n \log n$$

Then,

$$a = 3$$
$$b = 4$$
$$f(n) = n \log n$$
$$n^{\log_4 3} \leq n$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta(f(n))$

# Case 3

Let's solve

$$T(n) = 3T(\frac{n}{4}) + n \log n$$

Then,

$$\begin{aligned} a &= 3 \\ b &= 4 \\ f(n) &= n \log n \\ n^{\log_4 3} &\leq n \\ n \log n \in \Omega(n) &= \Omega\left(n^{\log_4 3 + \epsilon}\right) (\epsilon = 1 - \log_4 3) \end{aligned}$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there is a constant $\epsilon > 0$ such that $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$
- If there is a constant $k \geq 0$ such that $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta(f(n))$

# Case 3

Let's solve

$$T(n) = 3T(\frac{n}{4}) + n \log n$$

Then,

$$\begin{aligned} a &= 3 \\ b &= 4 \\ f(n) &= n \log n \\ n^{\log_4 3} &\leq n \\ n \log n \in \Omega(n) &= \Omega\left(n^{\log_4 3 + \epsilon}\right) (\epsilon = 1 - \log_4 3) \\ 3\frac{n}{4} \log \frac{n}{4} &= \frac{3}{4} n \log n - \frac{3}{4} n \log 4 \leq \frac{3}{4} n \log n \qquad (\delta = \frac{3}{4}) \end{aligned}$$

---

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta(f(n))$

# Case 3

Let's solve

$$T(n) = 3T(\frac{n}{4}) + n \log n$$

Then,

$$\begin{aligned} a &= 3 \\ b &= 4 \\ f(n) &= n \log n \\ n^{\log_4 3} &\leq n \\ n \log n \in \Omega(n) &= \Omega\left(n^{\log_4 3 + \epsilon}\right) (\epsilon = 1 - \log_4 3) \\ 3\frac{n}{4} \log \frac{n}{4} &= \frac{3}{4} n \log n - \frac{3}{4} n \log 4 \leq \frac{3}{4} n \log n \qquad (\delta = \frac{3}{4}) \end{aligned}$$

Therefore, $T(n) \in \Theta(n \log n)$

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- If there are constants $\epsilon > 0$ and $\delta < 1$ such that $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ and $af(n/b) \leq \delta f(n)$ for sufficiently large $n$, then $T(n) \in \Theta(f(n))$

# Example

$$T(n) = 2T(\frac{n}{2}) + 5$$

# Example

$$T(n) = 2T(\frac{n}{2}) + 5$$

$$a = 2$$
$$b = 2$$
$$f(n) = 5$$

# Example

$$T(n) = 2T(\frac{n}{2}) + 5$$

$$a = 2$$
$$b = 2$$
$$f(n) = 5$$

Is this Case 1, Case 2, or Case 3?

# CSC3310 Algorithms

## Proof by Induction

William Retert

Milwaukee School of Engineering

# Induction

**Induction** Proof technique for showing that some predicate is true for all positive integers.

# Induction

**Induction** Proof technique for showing that some predicate is true for all positive integers.

Possible predicates $P(n)$:

- $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$
- $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
- Peasant multiplication generates the correct result when multiplying an $n$-bit integer $x$ by an integer $y$
- etc.

Basic steps:

1. Show $P(n)$ true for $n = 0$
2. Show that if $P(n)$ is true, then $P(n + 1)$ is also true.

# Induction

**Induction** Proof technique for showing that some predicate is true for all positive integers.

Possible predicates $P(n)$:

- $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$
- $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
- Peasant multiplication generates the correct result when multiplying an $n$-bit integer $x$ by an integer $y$
- etc.

Basic steps:

1. Show $P(n)$ true for $n = 0$
2. Show that if $P(n-1)$ is true, then $P(n)$ is also true.

# Induction

**Induction** Proof technique for showing that some predicate is true for all positive integers.

Possible predicates $P(n)$:

- $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$
- $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
- Peasant multiplication generates the correct result when multiplying an $n$-bit integer $x$ by an integer $y$
- etc.

Basic steps:

1. Show $P(n)$ true for $n = 0$
2. Show that if $P(n-1)$ is true, then $P(n)$ is also true. Alternately, (strong induction), show that if $P(i)$ is true $\forall 1 \leq i < n$, then $P(n)$ is true

# $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

$n = 0$

$$\sum_{i=0}^{0} i = 0 = \frac{0(1)}{2}$$

$P(n-1) \vdash P(n)$

$$\begin{aligned}
\sum_{i=0}^{n} i &= n + \sum_{i=0}^{n-1} i \\
&= n + \frac{(n-1)((n-1)+1)}{2} && \text{(Induction!)} \\
&= n + \frac{(n-1)n}{2} \\
&= \frac{2n}{2} + \frac{n^2 - n}{2} \\
&= \frac{n^2 + n}{2} \\
&= \frac{n(n+1)}{2}
\end{aligned}$$

# $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

$n = 0$

$$\sum_{i=0}^{0} 2^i = 2^0 = 1 = 2^1 - 1$$

$P(n-1) \vdash P(n)$

$$\begin{aligned}\sum_{i=0}^{n} 2^i &= 2^n + \sum_{i=0}^{n-1} 2^i \\ &= 2^n + (2^n - 1) \qquad \text{(Induction!)} \\ &= 2 \cdot (2^n) - 1 \\ &= 2^{n+1} - 1\end{aligned}$$

# "Peasant" Multiplication

**Theorem**

*The "peasant" multiplication algorithm is correct. That is, for any n-bit integer x,* $\textsc{Peasant}(x, y) = x \cdot y \quad \forall y \in \mathbb{Z}$

```
procedure PEASANT(x,y)
    res ← 0
    while x > 0 do
        if x is odd then
            res ← res + y
        end if
        x ← ⌊x ÷ 2⌋
        y ← y + y
    end while
end procedure
```

We will rewrite the algorithm to be recursive in order to simplify the proof.

# “Peasant” Multiplication

**Theorem**

*The “peasant” multiplication algorithm is correct. That is, for any n-bit integer x,* $\textsc{Peasant}(x, y) = x \cdot y \ \ \forall y \in \mathbb{Z}$

```
procedure PEASANT(x,y)
    res ← 0
    if x > 0 then
        res ← PEASANT(⌊x ÷ 2⌋, y + y)
        if x is odd then
            res ← res + y
        end if
    end if
    return res
end procedure
```

# "Peasant" Multiplication

**Theorem**

*The "peasant" multiplication algorithm is correct. That is, for any n-bit integer x,* $\textsc{Peasant}(x, y) = x \cdot y \ \ \forall y \in \mathbb{Z}$

```
procedure PEASANT(x,y)
    res ← 0
    if x > 0 then
        res ← PEASANT(⌊x ÷ 2⌋, y + y)
        if x is odd then
            res ← res + y
        end if
    end if
    return res
end procedure
```

$n = 0$

If $x$ has 0 bits, then $x = 0$, and the algorithm correctly computes $0 \times y = 0$

# "Peasant" Multiplication

**Theorem**

*The "peasant" multiplication algorithm is correct. That is, for any n-bit integer x,* $\textsc{Peasant}(x, y) = x \cdot y \ \ \forall y \in \mathbb{Z}$

```
procedure PEASANT(x,y)
    res ← 0
    if x > 0 then
        res ← PEASANT(⌊x ÷ 2⌋, y + y)
        if x is odd then
            res ← res + y
        end if
    end if
    return res
end procedure
```

$P(n-1) \vdash P(n)$

If $x$ is a $n$-bit number, then $\lfloor x \div 2 \rfloor$ is an $(n-1)$ bit number. Therefore, by induction, $\textsc{Peasant}(\lfloor x \div 2 \rfloor, y + y) = (\lfloor x \div 2 \rfloor) \times (y + y)$
If $x$ is even, our return value is then

$$\frac{x}{2} \times (2y) = x \cdot y$$

. If $x$ is odd, the return value is

$$\begin{aligned} \frac{x-1}{2} \times (2y) + y &= (x-1)y + y \\ &= x \cdot y - y + y \\ &= x \cdot y \end{aligned}$$

# CSC3310 Algorithms

## Problems and Algorithms

William Retert

Milwaukee School of Engineering

# Algorithms

**Algorithm** a step-by-step procedure for performing a task in a finite amount of time.

In this class:

- Common algorithms
- Algorithmic problem-solving techniques
- Analysis of algorithms

# Multiplication

Let's multiply two 4-digit numbers!

# Multiplication

Let's multiply two 4-digit numbers!

- "Chalkboard" algorithm
- Lattice algorithm

# Multiplication

Let's multiply two 4-digit numbers!

- "Chalkboard" algorithm
- Lattice algorithm
- Both multiply individual digits and add according to place

$$x = X[0, \ldots, n]$$
$$y = Y[0, \ldots, m]$$
$$x \times y = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} X[i] \times Y[j] \times 10^{i+j}$$

- $O(m \cdot n)$ single-digit multiplications

# Peasant Multiplication

**procedure** PEASANTMULTIPLY(x,y)
    $res \leftarrow 0$
    **while** $x > 0$ **do**
        **if** $x$ is odd **then**
            $res \leftarrow res + y$
        **end if**
        $x \leftarrow \lfloor x \div 2 \rfloor$
        $y \leftarrow y + y$
    **end while**
**end procedure**

- Different basic math operations!
  - double
  - halve

# Peasant Multiplication

**procedure** PEASANTMULTIPLY(x,y)
  $res \leftarrow 0$
  **while** $x > 0$ **do**
    **if** $x$ is odd **then**
      $res \leftarrow res + y$
    **end if**
    $x \leftarrow \lfloor x \div 2 \rfloor$
    $y \leftarrow y + y$
  **end while**
**end procedure**

- Different basic math operations!
- Correctness

$$x \times y = \begin{cases} \lfloor x \div 2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x \div 2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

# Peasant Multiplication

**procedure** PEASANTMULTIPLY(x,y)
    $res \leftarrow 0$
    **while** $x > 0$ **do**
        **if** $x$ is odd **then**
            $res \leftarrow res + y$
        **end if**
        $x \leftarrow \lfloor x \div 2 \rfloor$
        $y \leftarrow y + y$
    **end while**
**end procedure**

- Different basic math operations!
- Loop runs $\lceil \lg x \rceil$ times (logarithm base 2)
- The work needed to add a number to itself depends on representation. If it is standard place representation, then it is proportional to $\log y$.
- Peasant multiplication is also $O(m \cdot n)$

# Describing an Algorithm

**Algorithm** a step-by-step procedure for *solving a problem* in a finite amount of time.

When *describing* an algorithm, we want to provide:

- Description of the problem it solves
- The steps of the algorithm
- Proof that the algorithm is correct
- Efficiency of the algorithm

# Describing an Algorithm

Algorithm a step-by-step procedure for *solving a problem* in a finite amount of time.

When *describing* an algorithm, we want to provide:

- **Description of the problem it solves**
  Target to a *user* of the algorithm. What do they need to know to *apply* the algorithm without knowing how it works?
  - Inputs, formally defined
  - Outputs, in terms of inputs
- The steps of the algorithm
- Proof that the algorithm is correct
- Efficiency of the algorithm

# Describing an Algorithm

Algorithm a step-by-step procedure for *solving a problem* in a finite amount of time.

When *describing* an algorithm, we want to provide:

- Description of the problem it solves
- **The steps of the algorithm**
  - Specify the steps precisely
  - Use pseudocode
  - Use code constructs rather than (ambiguous!) English descriptions
  - But remember–your audience is **other people**. Write steps that you can explain.
- Proof that the algorithm is correct
- Efficiency of the algorithm

# Describing an Algorithm

**Algorithm** a step-by-step procedure for *solving a problem* in a finite amount of time.

When *describing* an algorithm, we want to provide:

- Description of the problem it solves
- The steps of the algorithm
- Proof that the algorithm is correct
  - Loop invariants
  - induction
  - Often just a proof sketch
- Efficiency of the algorithm

# Describing an Algorithm

**Algorithm** a step-by-step procedure for *solving a problem* in a finite amount of time.

When *describing* an algorithm, we want to provide:

- Description of the problem it solves
- The steps of the algorithm
- Proof that the algorithm is correct
- Efficiency of the algorithm
  - Also called “Time Complexity.”
  - Generally written as a function of the size of the input (and possibly other factors)

$$T(n) = 3n^2 + \log n + 6$$

  - Number of primitive operations.
  - Summarized using asymptotic notation

# Describing an Algorithm

**Algorithm** a step-by-step procedure for *solving a problem* in a finite amount of time.

When *describing* an algorithm, we want to provide:

- Description of the problem it solves
- The steps of the algorithm
- Proof that the algorithm is correct
- Efficiency of the algorithm

Target your explanations to a competent novice

- Work through the details. Expect your reader to take your words literally.
- Do not assume the audience shares your intuition. Things that are “obvious” to you still need to be written down and explained coherently.
- Similarly, provide evidence in your arguments for correctness and efficiency.

# Example: Sorting

**Name** SORTING

**Description** Given a sequence of $n$ values, return the same values in sorted order

**Input** A sequence of $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$

**Output** A permutation of $A$ $\langle a'_0, a'_1, \ldots, a'_{n-1} \rangle$ such that $a'_0 \leq a'_1 \leq \cdots \leq a'_{n-1}$

# Example: Find Minimum

| | |
|---|---|
| Name | FINDMINIMUMVALUE |
| Description | Given a sequence of $n$ values, return a minimal value from the sequence |
| Input | A sequence of $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ |
| Output | A value $a_i$ such that $a_i$ is in the sequence, and $a_i \leq a_j$ for all $0 \leq j < n$ |

# Example: Sorted Merge

**Name** SORTEDMERGE

**Description** Given two *sorted* sequences with $m$ and $n$ values respectively, combine them into a single sorted sequence with $m + n$ values.

**Input** Two sequences $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ and $B = \langle b_0, b_1, \ldots, b_{m-1}$ such that $a_0 \leq a_1 \leq \cdots \leq a_{n-1}$ and $b_0 \leq b_1 \leq \cdots \leq b_{m-1}$

**Output** A permutation of $AB$ $\langle c_0, c_1, \ldots, c_{m+n-1} \rangle$ such that $c_0 \leq c_1 \leq \cdots \leq c_{m+n-1}$

# Computational Problems

**Decision** For a given input, answer "yes" or "no"
Given an array $A$ and a value $v$, determine whether the value is stored in the array.

**Search** Compute some answer in relation to the input
Given an array $A$ and a value $v$, find an index at which the value is stored.

**Counting** Determine *how many* answers relate to the input
Given an array $A$ and a value $v$, determine how many times the value appears in the array.

**Optimization** Find a "best possible" solution
Given an array $A[0, \ldots, n-1]$, find the smallest value in $A$

**Functional** Produce an output for each valid input
Given an array $A[0, \ldots, n-1]$, return an array with the same elements in *sorted* order

# Computational Problems

**Decision** For a given input, answer "yes" or "no"
Given an array $A$ and a value $v$, determine whether the value is stored in the array.

**Search** Compute some answer in relation to the input
Given an array $A$ and a value $v$, find an index at which the value is stored.

**Counting** Determine *how many* answers relate to the input
Given an array $A$ and a value $v$, determine how many times the value appears in the array.

**Optimization** Find a "best possible" solution
Given an array $A[0, \ldots, n-1]$, find the smallest value in $A$

**Functional** Produce an output for each valid input
Given an array $A[0, \ldots, n-1]$, return an array with the same elements in *sorted* order

# Computational Problems

**Decision** For a given input, answer "yes" or "no"
Given an array $A$ and a value $v$, determine whether the value is stored in the array.

**Search** Compute some answer in relation to the input
Given an array $A$ and a value $v$, find an index at which the value is stored.

**Counting** Determine *how many* answers relate to the input
Given an array $A$ and a value $v$, determine how many times the value appears in the array.

**Optimization** Find a "best possible" solution
Given an array $A[0, \ldots, n-1]$, find the smallest value in $A$

**Functional** Produce an output for each valid input
Given an array $A[0, \ldots, n-1]$, return an array with the same elements in *sorted* order

# Computational Problems

**Decision** For a given input, answer "yes" or "no"
Given an array $A$ and a value $v$, determine whether the value is stored in the array.

**Search** Compute some answer in relation to the input
Given an array $A$ and a value $v$, find an index at which the value is stored.

**Counting** Determine *how many* answers relate to the input
Given an array $A$ and a value $v$, determine how many times the value appears in the array.

**Optimization** Find a "best possible" solution
Given an array $A[0, \ldots, n-1]$, find the smallest value in $A$

**Functional** Produce an output for each valid input
Given an array $A[0, \ldots, n-1]$, return an array with the same elements in *sorted* order

# Computational Problems

**Decision** For a given input, answer "yes" or "no"
Given an array $A$ and a value $v$, determine whether the value is stored in the array.

**Search** Compute some answer in relation to the input
Given an array $A$ and a value $v$, find an index at which the value is stored.

**Counting** Determine *how many* answers relate to the input
Given an array $A$ and a value $v$, determine how many times the value appears in the array.

**Optimization** Find a "best possible" solution
Given an array $A[0, \ldots, n-1]$, find the smallest value in $A$

**Functional** Produce an output for each valid input
Given an array $A[0, \ldots, n-1]$, return an array with the same elements in *sorted* order

# Master Theorem:
# Practice Problems and Solutions

## Master Theorem

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are 3 cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large $n$.

## Practice Problems

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

1. $T(n) = 3T(n/2) + n^2$

2. $T(n) = 4T(n/2) + n^2$

3. $T(n) = T(n/2) + 2^n$

4. $T(n) = 2^n T(n/2) + n^n$

5. $T(n) = 16T(n/4) + n$

6. $T(n) = 2T(n/2) + n \log n$

[1] most of the time, $k = 0$

7. $T(n) = 2T(n/2) + n/\log n$

8. $T(n) = 2T(n/4) + n^{0.51}$

9. $T(n) = 0.5T(n/2) + 1/n$

10. $T(n) = 16T(n/4) + n!$

11. $T(n) = \sqrt{2}T(n/2) + \log n$

12. $T(n) = 3T(n/2) + n$

13. $T(n) = 3T(n/3) + \sqrt{n}$

14. $T(n) = 4T(n/2) + cn$

15. $T(n) = 3T(n/4) + n\log n$

16. $T(n) = 3T(n/3) + n/2$

17. $T(n) = 6T(n/3) + n^2\log n$

18. $T(n) = 4T(n/2) + n/\log n$

19. $T(n) = 64T(n/8) - n^2\log n$

20. $T(n) = 7T(n/3) + n^2$

21. $T(n) = 4T(n/2) + \log n$

22. $T(n) = T(n/2) + n(2 - \cos n)$

## Solutions

1. $T(n) = 3T(n/2) + n^2 \implies T(n) = \Theta(n^2)$ (Case 3)
2. $T(n) = 4T(n/2) + n^2 \implies T(n) = \Theta(n^2 \log n)$ (Case 2)
3. $T(n) = T(n/2) + 2^n \implies \Theta(2^n)$ (Case 3)
4. $T(n) = 2^n T(n/2) + n^n \implies$ Does not apply ($a$ is not constant)
5. $T(n) = 16T(n/4) + n \implies T(n) = \Theta(n^2)$ (Case 1)
6. $T(n) = 2T(n/2) + n \log n \implies T(n) = n \log^2 n$ (Case 2)
7. $T(n) = 2T(n/2) + n/\log n \implies$ Does not apply (non-polynomial difference between $f(n)$ and $n^{\log_b a}$)
8. $T(n) = 2T(n/4) + n^{0.51} \implies T(n) = \Theta(n^{0.51})$ (Case 3)
9. $T(n) = 0.5T(n/2) + 1/n \implies$ Does not apply ($a < 1$)
10. $T(n) = 16T(n/4) + n! \implies T(n) = \Theta(n!)$ (Case 3)
11. $T(n) = \sqrt{2}T(n/2) + \log n \implies T(n) = \Theta(\sqrt{n})$ (Case 1)
12. $T(n) = 3T(n/2) + n \implies T(n) = \Theta(n^{\lg 3})$ (Case 1)
13. $T(n) = 3T(n/3) + \sqrt{n} \implies T(n) = \Theta(n)$ (Case 1)
14. $T(n) = 4T(n/2) + cn \implies T(n) = \Theta(n^2)$ (Case 1)
15. $T(n) = 3T(n/4) + n \log n \implies T(n) = \Theta(n \log n)$ (Case 3)
16. $T(n) = 3T(n/3) + n/2 \implies T(n) = \Theta(n \log n)$ (Case 2)
17. $T(n) = 6T(n/3) + n^2 \log n \implies T(n) = \Theta(n^2 \log n)$ (Case 3)
18. $T(n) = 4T(n/2) + n/\log n \implies T(n) = \Theta(n^2)$ (Case 1)
19. $T(n) = 64T(n/8) - n^2 \log n \implies$ Does not apply ($f(n)$ is not positive)
20. $T(n) = 7T(n/3) + n^2 \implies T(n) = \Theta(n^2)$ (Case 3)
21. $T(n) = 4T(n/2) + \log n \implies T(n) = \Theta(n^2)$ (Case 1)
22. $T(n) = T(n/2) + n(2 - \cos n) \implies$ Does not apply. We are in Case 3, but the regularity condition is violated. (Consider $n = 2\pi k$, where $k$ is odd and arbitrarily large. For any such choice of $n$, you can show that $c \geq 3/2$, thereby violating the regularity condition.)