# Analysis of Algorithms

## Divide and Conquer

Andres Mendez-Vazquez

January 10, 2018

# Outline



Cinvestav

# Outline

# Divide an Conquer

## Divide et impera

A classic technique based on the multi-based recursion.

- Until you reach a base case!!!

# Divide an Conquer

## Divide et impera

A classic technique based on the multi-based recursion.

## Thus, we have

That Divide and Conquer works by recursively breaking down the problem into subproblems and solving those subproblems recursively.

- Until you reach a base case!!!

# Divide an Conquer

## Divide et impera

A classic technique based on the multi-based recursion.

## Thus, we have

That Divide and Conquer works by recursively breaking down the problem into subproblems and solving those subproblems recursively.

- Until you reach a base case!!!

## Remark

Given the fact of the following equivalence:

$$Recursion \equiv Iteration \quad (1)$$

Cinvestav

# Divide an Conquer

## Divide et impera

A classic technique based on the multi-based recursion.

## Thus, we have

That Divide and Conquer works by recursively breaking down the problem into subproblems and solving those subproblems recursively.

- Until you reach a base case!!!

## Remark

Given the fact of the following equivalence:

$$Recursion \equiv Iteration \tag{1}$$

# Outline

# Gauss and the Beginning

## Carl Friedrich Gauss (1777–1855)

He devised a way to multiply two imaginary numbers as

$$(a + bi)(c + di) = ac + (ad + bc)i - bd \quad (2)$$

## By realizing that

$$bc + ad = (a + b)(c + d) - ac - bd \quad (3)$$

Thus minimizing the number of multiplications from four to three.

## Actually

We can represent binary numbers like 1001 as $1000 + 01 = 2^2 \times 10 + 01$

Cinvestav

# Gauss and the Beginning

**Carl Friedrich Gauss (1777–1855)**

He devised a way to multiply two imaginary numbers as

$$(a+bi)(c+di) = ac + (ad+bc)\,i - bd \tag{2}$$

**By realizing that**

$$bc + ad = (a+b)(c+d) - ac - bd \tag{3}$$

Thus minimizing the number of multiplications from four to three.

**Actually**

We can represent binary numbers like $1001$ as $1000 + 01 = 2^2 \times 10 + 01$

# Gauss and the Beginning

## Carl Friedrich Gauss (1777–1855)

He devised a way to multiply two imaginary numbers as

$$(a+bi)(c+di) = ac + (ad+bc)\,i - bd \tag{2}$$

## By realizing that

$$bc + ad = (a+b)(c+d) - ac - bd \tag{3}$$

Thus minimizing the number of multiplications from four to three.

## Actually

We can represent binary numbers like 1001 as $1000 + 01 = 2^2 \times 10 + 01$

# Thus

## We can represent numbers $x, y$ as

- $x = x_L \circ x_R = 2^{n/2} x_L + x_R$
- $y = y_L \circ y_R = 2^{n/2} y_L + y_R$

# Thus

**We can represent numbers $x, y$ as**

- $x = x_L \circ x_R = 2^{n/2} x_L + x_R$
- $y = y_L \circ y_R = 2^{n/2} y_L + y_R$

**Thus, the multiplication can be found by using**

$$xy = \left(2^{n/2} x_L + x_R\right)\left(2^{n/2} y_L + y_R\right) = 2^n x_L y_L + 2^{n/2}\left(x_L y_R + x_R y_L\right) + x_R y_R \quad (4)$$

# Thus

**We can represent numbers $x, y$ as**

- $x = x_L \circ x_R = 2^{n/2} x_L + x_R$
- $y = y_L \circ y_R = 2^{n/2} y_L + y_R$

**Thus, the multiplication can be found by using**

$$xy = \left(2^{n/2} x_L + x_R\right)\left(2^{n/2} y_L + y_R\right) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R \quad (4)$$

**However**

if we use the Gauss's trick, we only need $x_L y_L$, $x_R y_R$, $(x_L + x_R)(y_L + y_R)$ to calculate the multiplication:

- $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

# Thus

## We can represent numbers $x, y$ as

- $x = x_L \circ x_R = 2^{n/2} x_L + x_R$
- $y = y_L \circ y_R = 2^{n/2} y_L + y_R$

## Thus, the multiplication can be found by using

$$xy = \left(2^{n/2} x_L + x_R\right)\left(2^{n/2} y_L + y_R\right) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R \quad (4)$$

## However

if we use the Gauss's trick, we only need $x_L y_L$, $x_R y_R$, $(x_L + x_R)(y_L + y_R)$ to calculate the multiplication:

- $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

# Thus

## We can represent numbers $x, y$ as

- $x = x_L \circ x_R = 2^{n/2} x_L + x_R$
- $y = y_L \circ y_R = 2^{n/2} y_L + y_R$

## Thus, the multiplication can be found by using

$$xy = \left(2^{n/2} x_L + x_R\right)\left(2^{n/2} y_L + y_R\right) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R \tag{4}$$

## However

if we use the Gauss's trick, we only need $x_L y_L$, $x_R y_R$, $(x_L + x_R)(y_L + y_R)$ to calculate the multiplication:

- $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

# Now, You have this...

**We have that**

$xy$ can be calculated by using the two parts, Left and Right.

**Then**

Thus, each $x_L x_L$, $x_L y_R$, $x_R y_L$ and $x_R y_R$ can be calculated in a similar way

**Recursion**

This is know as a Recursive Procedure!!!

# Now, You have this...

**We have that**

$xy$ can be calculated by using the two parts, Left and Right.

**Then**

Thus, each $x_L x_L$, $x_L y_R$, $x_R y_L$ and $x_R y_R$ can be calculated in a similar way

**Recursion**

This is know as a Recursive Procedure!!!

# Now, You have this...

**We have that**

$xy$ can be calculated by using the two parts, Left and Right.

**Then**

Thus, each $x_L x_L$, $x_L y_R$, $x_R y_L$ and $x_R y_R$ can be calculated in a similar way

**Recursion**

This is know as a Recursive Procedure!!!

Cinvestav

# Complexities

## Old Multiplication

$$T(n) = 4T\left(\frac{n}{2}\right) + \text{ Some Work} \tag{5}$$

# Complexities

## Old Multiplication

$$T(n) = 4T\left(\frac{n}{2}\right) + \text{ Some Work} \tag{5}$$

## New Multiplication

$$T(n) = 3T\left(n\right) + \text{ Some Work} \tag{6}$$

- For new style multiplications $O\left(n^{\log_2 3}\right)$

# Complexities

## Old Multiplication

$$T(n) = 4T\left(\frac{n}{2}\right) + \text{ Some Work} \tag{5}$$

## New Multiplication

$$T(n) = 3T\left(n\right) + \text{ Some Work} \tag{6}$$

## We will prove that

- For old style multiplications $O\left(n^2\right)$.
- For new style multiplications $O\left(n^{\log_2 3}\right)$

# Complexities

## Old Multiplication

$$T(n) = 4T\left(\frac{n}{2}\right) + \text{ Some Work} \tag{5}$$

## New Multiplication

$$T(n) = 3T\left(n\right) + \text{ Some Work} \tag{6}$$

## We will prove that

- For old style multiplications $O\left(n^2\right)$.
- For new style multiplications $O\left(n^{\log_2 3}\right)$

# Epitaph

**We can do divide and conquer**

In a really unclever way!!!

# Epitaph

### We can do divide and conquer

In a really unclever way!!!

### Or we can go and design something better

Thus, improving speedup!!!

- Or a crappy job

# Epitaph

**We can do divide and conquer**

In a really unclever way!!!

**Or we can go and design something better**

Thus, improving speedup!!!

**The difference between**

- A great design...
- Or a crappy job...

# Epitaph

**We can do divide and conquer**

In a really unclever way!!!

**Or we can go and design something better**

Thus, improving speedup!!!

**The difference between**

- A great design...
- Or a crappy job...

# Recursion is the base of Divide and Conquer

**This is the natural way we do many things**

We always attack smaller versions first of the large one!!!

**Stephen Cole Kleene**

- He defined the basics about the use of recursion.

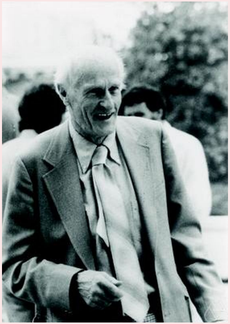
# Recursion is the base of Divide and Conquer

## This is the natural way we do many things

We always attack smaller versions first of the large one!!!

## Stephen Cole Kleene

- He defined the basics about the use of recursion.



# Kleene and Company

## Some facts about him

- Stephen Cole Kleene (January 5, 1909 – January 25, 1994) was an American mathematician.
- One of the students of Alonzo Church!!!
  - Church is best known for the lambda calculus, Church–Turing thesis and proving the undecidability of the use of an algorithm to say Yes(Valid) or No(No Valid) to a first order logic statement on a FOL System (Proposed by David Hilbert).

# Kleene and Company

## Some facts about him

- Stephen Cole Kleene (January 5, 1909 – January 25, 1994) was an American mathematician.
- One of the students of Alonzo Church!!!
  - Church is best known for the lambda calculus, Church–Turing thesis and proving the undecidability of the use of an algorithm to say Yes(Valid) or No(No Valid) to a first order logic statement on a FOL System (Proposed by David Hilbert).

## Recursion Theory

- Kleene, along with Alan Turing, Emil Post, and others, is best known as a founder of the branch of mathematical logic known as recursion theory.
- This theory subsequently helped to provide the foundations of theoretical computer science.

# Kleene and Company

## Some facts about him

- Stephen Cole Kleene (January 5, 1909 – January 25, 1994) was an American mathematician.
- One of the students of Alonzo Church!!!
  - Church is best known for the lambda calculus, Church–Turing thesis and proving the undecidability of the use of an algorithm to say Yes(Valid) or No(No Valid) to a first order logic statement on a FOL System (Proposed by David Hilbert).

## Recursion Theory

- Kleene, along with Alan Turing, Emil Post, and others, is best known as a founder of the branch of mathematical logic known as recursion theory.
- This theory subsequently helped to provide the foundations of theoretical computer science.

# Kleene and Company

## Some facts about him

- Stephen Cole Kleene (January 5, 1909 – January 25, 1994) was an American mathematician.
- One of the students of Alonzo Church!!!
  - Church is best known for the lambda calculus, Church–Turing thesis and proving the undecidability of the use of an algorithm to say Yes(Valid) or No(No Valid) to a first order logic statement on a FOL System (Proposed by David Hilbert).

## Recursion Theory

- Kleene, along with Alan Turing, Emil Post, and others, is best known as a founder of the branch of mathematical logic known as recursion theory.
- This theory subsequently helped to provide the foundations of theoretical computer science.

# Kleene and Company

## Some facts about him

- Stephen Cole Kleene (January 5, 1909 – January 25, 1994) was an American mathematician.
- One of the students of Alonzo Church!!!
  - Church is best known for the lambda calculus, Church–Turing thesis and proving the undecidability of the use of an algorithm to say Yes(Valid) or No(No Valid) to a first order logic statement on a FOL System (Proposed by David Hilbert).

## Recursion Theory

- Kleene, along with Alan Turing, Emil Post, and others, is best known as a founder of the branch of mathematical logic known as recursion theory.
- This theory subsequently helped to provide the foundations of theoretical computer science.

# Recursion

## Something Notable

- Sometimes it is difficult to define an object explicitly.
- It may be easy to define this object in smaller version of itself.
- **This process is called recursion!!!**

Cinvestav

# Recursion

## Something Notable

- Sometimes it is difficult to define an object explicitly.
- It may be easy to define this object in smaller version of itself.
- **This process is called recursion!!!**

## Thus

We can use recursion to define sequences, functions, and sets.

- Thus, the sequence can be defined in a recursive way

# Recursion

## Something Notable

- Sometimes it is difficult to define an object explicitly.
- It may be easy to define this object in smaller version of itself.
- **This process is called recursion!!!**

## Thus

We can use recursion to define sequences, functions, and sets.

## Example

- $a_n = 2^n$ for $n = 0, 1, 2, \ldots \Longrightarrow 1, 2, 4, 8, 16, 32, \ldots$
- Thus, the sequence can be defined in a recursive way:

$$a_{n+1} = 2 \times a_n \quad (7)$$

# Recursion

## Something Notable

- Sometimes it is difficult to define an object explicitly.
- It may be easy to define this object in smaller version of itself.
- **This process is called recursion!!!**

## Thus

We can use recursion to define sequences, functions, and sets.

## Example

- $a_n = 2^n$ for $n = 0, 1, 2, \ldots \Longrightarrow 1, 2, 4, 8, 16, 32, \ldots$
- Thus, the sequence can be defined in a recursive way:

$$a_{n+1} = 2 \times a_n \tag{7}$$

# Recursion

## Something Notable

- Sometimes it is difficult to define an object explicitly.
- It may be easy to define this object in smaller version of itself.
- **This process is called recursion!!!**

## Thus

We can use recursion to define sequences, functions, and sets.

## Example

- $a_n = 2^n$ for $n = 0, 1, 2, \ldots \Longrightarrow 1, 2, 4, 8, 16, 32, \ldots$
- Thus, the sequence can be defined in a recursive way:

$$a_{n+1} = 2 \times a_n \quad (7)$$

# Recursion

## Something Notable

- Sometimes it is difficult to define an object explicitly.
- It may be easy to define this object in smaller version of itself.
- **This process is called recursion!!!**

## Thus

We can use recursion to define sequences, functions, and sets.

## Example

- $a_n = 2^n$ for $n = 0, 1, 2, \ldots \Longrightarrow 1, 2, 4, 8, 16, 32, \ldots$
- Thus, the sequence can be defined in a recursive way:

$$a_{n+1} = 2 \times a_n \tag{7}$$

# Recursion

## Something Notable

- Sometimes it is difficult to define an object explicitly.
- It may be easy to define this object in smaller version of itself.
- **This process is called recursion!!!**

## Thus

We can use recursion to define sequences, functions, and sets.

## Example

- $a_n = 2^n$ for $n = 0, 1, 2, \ldots \Longrightarrow 1, 2, 4, 8, 16, 32, \ldots$
- Thus, the sequence can be defined in a recursive way:

$$a_{n+1} = 2 \times a_n \tag{7}$$

# Outline

# Recursively Defined Functions

### First

Assume $T$ is a function with the set of nonnegative integers as its domain.

# Recursively Defined Functions

## First

Assume $T$ is a function with the set of nonnegative integers as its domain.

## Second

We use two steps to define $T$:

- **Basis step:** Specify the value of $T(0)$.
- **Recursive step:** Give a rule for $T(x)$ using $T(y)$ where $0 \leq y < x$.

# Recursively Defined Functions

## First

Assume $T$ is a function with the set of nonnegative integers as its domain.

## Second

We use two steps to define $T$:

**Basis** step:

Specify the value of $T(0)$.

**Recursive** step:

Give a rule for $T(x)$ using $T(y)$ where $0 \leq y < x$.

## Thus

Such a definition is called a recursive or inductive definition.

Cinvestav

# Recursively Defined Functions

## First

Assume $T$ is a function with the set of nonnegative integers as its domain.

## Second

We use two steps to define $T$:

**Basis** step:
Specify the value of $T(0)$.

**Recursive** step:
Give a rule for $T(x)$ using $T(y)$ where $0 \leq y < x$.

## Thus

Such a definition is called a recursive or inductive definition.

Cinvestav

# Recursively Defined Functions

## First

Assume $T$ is a function with the set of nonnegative integers as its domain.

## Second

We use two steps to define $T$:

**Basis** step:
Specify the value of $T(0)$.

**Recursive** step:
Give a rule for $T(x)$ using $T(y)$ where $0 \leq y < x$.

## Thus

Such a definition is called a recursive or inductive definition.

# Recursively Defined Functions

## First

Assume $T$ is a function with the set of nonnegative integers as its domain.

## Second

We use two steps to define $T$:

**Basis** step:
Specify the value of $T(0)$.

**Recursive** step:
Give a rule for $T(x)$ using $T(y)$ where $0 \leq y < x$.

## Thus

Such a definition is called a recursive or inductive definition.

# Recursively Defined Functions

## First

Assume $T$ is a function with the set of nonnegative integers as its domain.

## Second

We use two steps to define $T$:

**Basis** step:
Specify the value of $T(0)$.

**Recursive** step:
Give a rule for $T(x)$ using $T(y)$ where $0 \leq y < x$.

## Thus

Such a definition is called a recursive or inductive definition.

# Example

**Can you give me the following?**

Give an inductive definition of the factorial function $T(n) = n!$.

**Base case**

Which is the base case?

**Recursive case**

What is the recursive case?

# Example

### Can you give me the following?

Give an inductive definition of the factorial function $T(n) = n!$.

### Base case

Which is the base case?

### Recursive case

What is the recursive case?

# Example

## Can you give me the following?

Give an inductive definition of the factorial function $T(n) = n!$.

## Base case

Which is the base case?

## Recursive case

What is the recursive case?

# We can go further...

## Recursively Defined Sets and Structures

- Assume $S$ is a set.
- We can use two steps to define the elements of $S$.

# We can go further...

## Recursively Defined Sets and Structures

- Assume $S$ is a set.
- We can use two steps to define the elements of S.

## Basis Step

Specify an initial collection of elements.

# We can go further...

## Recursively Defined Sets and Structures

- Assume $S$ is a set.
- We can use two steps to define the elements of S.

## Basis Step

Specify an initial collection of elements.

## Recursive Step

Give a rule for forming new elements from those already known to be in $S$.

# We can go further...

**Recursively Defined Sets and Structures**

- Assume $S$ is a set.
- We can use two steps to define the elements of S.

**Basis Step**

Specify an initial collection of elements.

**Recursive Step**

Give a rule for forming new elements from those already known to be in $S$.

# Example

**Consider**

Consider $S \subseteq \mathbb{Z}$ defined by...

**Basis Step**

$3 \in S$

**Recursive Step**

If $x \in S$ and $y \in S$, then $x + y \in S$.

# Example

## Consider

Consider $S \subseteq \mathbb{Z}$ defined by...

## Basis Step

$3 \in S$

## Recursive Step

If $x \in S$ and $y \in S$, then $x + y \in S$.

# Example

## Consider

Consider $S \subseteq \mathbb{Z}$ defined by...

## Basis Step

$3 \in S$

## Recursive Step

If $x \in S$ and $y \in S$, then $x + y \in S$.

# Example

## Elements

- $3 \in S$
- $3+3=6 \in S$
- $6+3=9 \in S$
- $6+6=12 \in S$
- $\cdots$

# Outline



Cinvestav

# Divide and Conquer

**Divide**

Split problem into a number of subproblems.

**Conquer**

Solve each subproblem recursively.

**Combine**

The solution of the problems into the solution of the original problem.

# Divide and Conquer

**Divide**

Split problem into a number of subproblems.

**Conquer**

Solve each subproblem recursively.

**Combine**

The solution of the problems into the solution of the original problem.

# Divide and Conquer

## Divide

Split problem into a number of subproblems.

## Conquer

Solve each subproblem recursively.

## Combine

The solution of the problems into the solution of the original problem.

# Time Complexities

## Definition

- Given an input as a string where the problem is being encoded using an alphabet $\Sigma$,
  - The **time complexity** quantifies the amount of time taken by an algorithm to run as a function on the length of such string.

# The Divide and Conquer of Merge Sort

**Merge-Sort$(A, p, r)$**

1. if $p < r$ then
2. $\quad q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
3. $\quad$ Merge-Sort$(A, p, q)$
4. $\quad$ Merge-Sort$(A, q+1, r)$
5. $\quad$ MERGE$(A, p, q, r)$

**Explanation**

Divide part into the conquer!!!

# The Divide and Conquer of Merge Sort

**Merge-Sort$(A, p, r)$**

1. if $p < r$ then
2. $\quad q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
3. $\quad$ Merge-Sort$(A, p, q)$
4. $\quad$ Merge-Sort$(A, q+1, r)$
5. $\quad$ MERGE$(A, p, q, r)$

**Explanation**

The combine part!!!

# Merge Sort

- Merge(A, p, q, r )

$n_1 \leftarrow q - p + 1$, $n_2 \leftarrow r - p$
let $L[1, 2, ..., n_1 + 1]$ and
$R[1, 2, ..., n_2 + 1]$ be new arrays.
for $i \leftarrow 1$ to $n_1$
    $L[i] \leftarrow A[p + i - 1]$
for$j \leftarrow 1$ to $n_2$
    $R[i] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$, $j \leftarrow 1$
for $k \leftarrow p$ to $r$
    if $L[i] \leq R[j]$ then
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    else
        $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$

**Explanation**

- Copy all to be merged lists into two containers.

# Merge Sort

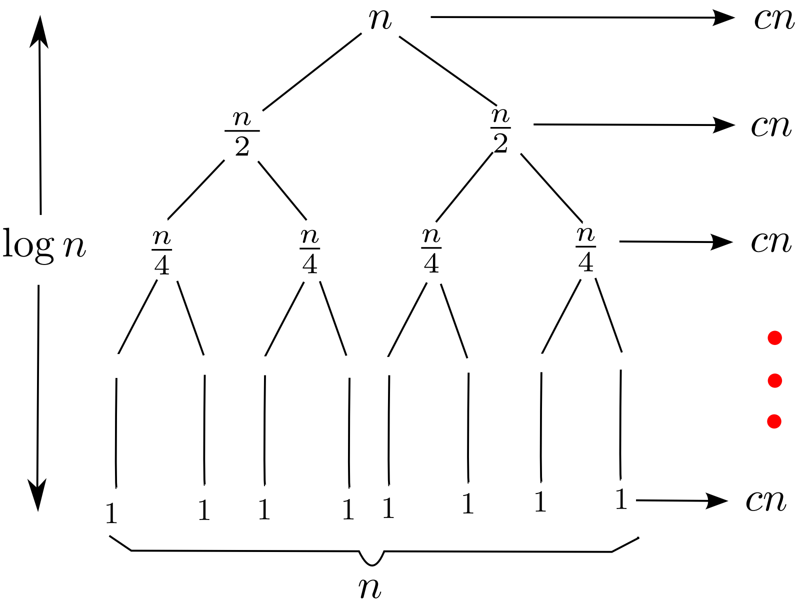
- Merge(A, p, q, r )

```
n1 ← q − p + 1, n2 ← r − p
let L[1, 2, ..., n1 + 1] and
R[1, 2, ..., n2 + 1] be new arrays.
for i ← 1 to n1
      L[i] ← A[p + i − 1]
forj ← 1 to n2
      R[i] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1, j ← 1
for k ← p to r
       if L[i] ≤ R[j] then
             A[k] ← L[i]
             i ← i + 1
      else
             A[k] ← R[j]
             j ← j + 1
```

**Explanation**

- Merging part.

# The Merge Sort Recursion Cost Function



# Outline

# Recursive Functions

## Using Church-Turing Thesis

**Every computable function from natural numbers to natural numbers is recursive and computable.**

## YES!!!

We can use recursive functions to represent the TOTAL number of steps carried when computing an ALGORITHM

# Recursive Functions

## Using Church-Turing Thesis

**Every computable function from natural numbers to natural numbers is recursive and computable.**

## YES!!!

We can use recursive functions to represent the TOTAL number of steps carried when computing an ALGORITHM

# Thus, we have

## Each Step for ONE Merging takes...

A certain constant time $c$!!!

## Thus, if we merge $n$ elements

Total time at level 1 of recursion:

$$cn \tag{8}$$

## In addition...

We have that the recursion split each work by

$$\frac{1}{2^i}, \text{ for } i = 1, ..., \log n \tag{9}$$

# Thus, we have

**Each Step for ONE Merging takes...**

A certain constant time $c$!!!

**Thus, if we merge $n$ elements**

Total time at level 1 of recursion:

$$cn \tag{8}$$

**In addition...**

We have that the recursion split each work by

$$\frac{1}{2^i}, \text{ for } i = 1, ..., \log n \tag{9}$$

# Thus, we have

## Each Step for ONE Merging takes...

A certain constant time $c$!!!

## Thus, if we merge $n$ elements

Total time at level 1 of recursion:

$$cn \tag{8}$$

## In addition...

We have that the recursion split each work by

$$\frac{1}{2^i}, \text{ for } i = 1, ..., \log n \tag{9}$$

# Thus, we have the following Recursion

## Base Case $n = 1$

$$T(n) = c \tag{10}$$

Where $c$ stands for a constant in the number of time units or assembly instructions per line!!!

## Recursive Step $n > 1$

$$2T\left(\frac{n}{2}\right) + cn \tag{11}$$

## Finally

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases} \tag{12}$$

# Thus, we have the following Recursion

## Base Case $n = 1$

$$T(n) = c \tag{10}$$

Where $c$ stands for a constant in the number of time units or assembly instructions per line!!!

## Recursive Step $n > 1$

$$2T\left(\frac{n}{2}\right) + cn \tag{11}$$

## Finally

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases} \tag{12}$$

# Thus, we have the following Recursion

**Base Case $n = 1$**

$$T(n) = c \tag{10}$$

Where $c$ stands for a constant in the number of time units or assembly instructions per line!!!

**Recursive Step $n > 1$**

$$2T\left(\frac{n}{2}\right) + cn \tag{11}$$

**Finally**

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases} \tag{12}$$

# Outline



Cinvestav

# Recursion and Induction

## Something Notable

When a sequence is defined recursively, mathematical induction can be used to prove results about the sequence.

# For Example

**We want**

To show that the set $S$ is the set $A$ of all positive integers that are multiples of 3.

**First $A \subseteq S$**

Show that if $\forall k \geq 1$ $P(k)$ is true, then $P(k+1)$ is true

**We define, first, the inductive hypothesis**

$P(k) : 3k \in S$ is true

# For Example

**We want**

To show that the set $S$ is the set $A$ of all positive integers that are multiples of 3.

**First $A \subseteq S$**

Show that if $\forall k \geq 1 \; P(k)$ is true, then $P(k+1)$ is true

**We define, first, the inductive hypothesis**

$P(k) : 3k \in S$ is true

# For Example

**We want**

To show that the set $S$ is the set $A$ of all positive integers that are multiples of 3.

**First $A \subseteq S$**

Show that if $\forall k \geq 1$ $P(k)$ is true, then $P(k+1)$ is true

**We define, first, the inductive hypothesis**

$P(k) : 3k \in S$ is true

# Thus

**We know the following by definition**

$$3 \in S$$

# Thus

**We know the following by definition**

$$3 \in S$$

**We need to prove that for any $k$**

$$P(k+1) : 3(k+1) = 3k + 3 \in S$$

# Thus

**We know the following by definition**

$$3 \in S$$

**We need to prove that for any $k$**

$$P(k+1) : 3(k+1) = 3k + 3 \in S$$

**We have using the recursive definition**

We know that $3k \in S$ and $3 \in S$ is true, therefore by definition

$$3k + 3 = 3(k+1) \in S$$

# Thus

**We know the following by definition**

$$3 \in S$$

**We need to prove that for any $k$**

$$P(k+1) : 3(k+1) = 3k + 3 \in S$$

**We have using the recursive definition**

We know that $3k \in S$ and $3 \in S$ is true, therefore by definition

$$3k + 3 = 3(k+1) \in S$$

Cinvestav

# Now

## Finally

- We have that for $x \in A$ with $x = 3k$ for $k \geq 1$, then by the previous proof $3k \in S$!!!
- Then, $A \subseteq S$!!!

# Now

## Finally

- We have that for $x \in A$ with $x = 3k$ for $k \geq 1$, then by the previous proof $3k \in S$!!!
- Then, $A \subseteq S$!!!

## Now, show that $S \subseteq A$

Or $\forall x$, $x \in S$ then $x \in A$

# Now

## Finally

- We have that for $x \in A$ with $x = 3k$ for $k \geq 1$, then by the previous proof $3k \in S$!!!
- Then, $A \subseteq S$!!!

## Now, show that $S \subseteq A$

Or $\forall x,\ x \in S$ then $x \in A$

## Given the definition

- Basis Step: $3 \in S$
- Recursive Step: $x \in S$, $y \in S \Rightarrow x + y \in S$

# Now

## Finally

- We have that for $x \in A$ with $x = 3k$ for $k \geq 1$, then by the previous proof $3k \in S$!!!
- Then, $A \subseteq S$!!!

## Now, show that $S \subseteq A$

Or $\forall x, \ x \in S$ then $x \in A$

## Given the definition

- Basis Step: $3 \in S$
- Recursive Step: $x \in S$, $y \in S \Rightarrow x + y \in S$

Cinvestav

# Now

## Finally

- We have that for $x \in A$ with $x = 3k$ for $k \geq 1$, then by the previous proof $3k \in S$!!!
- Then, $A \subseteq S$!!!

## Now, show that $S \subseteq A$

Or $\forall x,\ x \in S$ then $x \in A$

## Given the definition

- Basis Step: $3 \in S$
- Recursive Step: $x \in S$, $y \in S \Rightarrow x + y \in S$

Cinvestav

# Then

**First, $3 \in S$**

It is clear that $3 \in A$

# Then

**First, $3 \in S$**

It is clear that $3 \in A$

**Now, given $x \in S$ and $y \in S$**

- if $x \in A$ and $y \in A$
- Then, $x$ and $y$ are multiples of 3

- Therefore $S \subseteq A$

# Then

**First, $3 \in S$**

It is clear that $3 \in A$

**Now, given $x \in S$ and $y \in S$**

- if $x \in A$ and $y \in A$
- Then, $x$ and $y$ are multiples of 3

**Therefore**

- $x + y$ is a multiple of 3 with $x + y \in S$ by definition, and also $x + y \in A$
- Therefore, $S \subseteq A$

# Then

## First, $3 \in S$

It is clear that $3 \in A$

## Now, given $x \in S$ and $y \in S$

- if $x \in A$ and $y \in A$
- Then, $x$ and $y$ are multiples of 3

## Therefore

- $x + y$ is a multiple of 3 with $x + y \in S$ by definition, and also $x + y \in A$
- Therefore, $S \subseteq A$

# Then

**First, $3 \in S$**

It is clear that $3 \in A$

**Now, given $x \in S$ and $y \in S$**

- if $x \in A$ and $y \in A$
- Then, $x$ and $y$ are multiples of 3

**Therefore**

- $x + y$ is a multiple of 3 with $x + y \in S$ by definition, and also $x + y \in A$
- Therefore, $S \subseteq A$

# Outline



Cinvestav

# Structural induction

**Something Notable**

Instead of mathematical induction to prove a result about a recursively defined sets, we can used more convenient form of induction known as structural induction.

# Structural induction

## Something Notable

Instead of mathematical induction to prove a result about a recursively defined sets, we can used more convenient form of induction known as structural induction.

## First

- Assume we have a recursive definition for a set $S$.
- Given $n \in S$, we must show that $P(n)$ is true using structural induction.

# Structural induction

## Something Notable

Instead of mathematical induction to prove a result about a recursively defined sets, we can used more convenient form of induction known as structural induction.

## First

- Assume we have a recursive definition for a set $S$.
- Given $n \in S$, we must show that $P(n)$ is true using structural induction.

# Definition of Structural induction

## Basis Step

- Assume $j$ is an element specified in the base step of the definition.
- Show that $\forall j$, $P(j)$ is true.

- Assume $k_1, k_2, ..., k_m$ are elements used to construct an element $k$ in the recursive step of the definition.
- Show that $\forall k_1, k_2, ..., k_m, ((P(k_1) \wedge P(k_2) \wedge ... \wedge P(k_m)) \to P(k))$.

# Definition of Structural induction

## Basis Step

- Assume $j$ is an element specified in the base step of the definition.
- Show that $\forall j, \; P(j)$ is true.

## Recursive step

- Let $x$ be a new element constructed in the recursive step of the definition.
- Assume $k_1, k_2, \ldots, k_m$ are elements used to construct an element $x$ in the recursive step of the definition.
- Show that $\forall \; k_1, k_2, \ldots, k_m \;\; ((P(k_1) \land P(k_2) \land \ldots \land P(k_m)) \rightarrow P(x))$.

# Definition of Structural induction

## Basis Step

- Assume $j$ is an element specified in the base step of the definition.
- Show that $\forall j,\ P(j)$ is true.

## Recursive step

- Let $x$ be a new element constructed in the recursive step of the definition.
- Assume $k_1, k_2, \ldots, k_m$ are elements used to construct an element $x$ in the recursive step of the definition.
- Show that $\forall\ k_1, k_2, \ldots, k_m\ \ ((P(k_1) \land P(k_2) \land \ldots \land P(k_m)) \to P(x))$.

# Definition of Structural induction

## Basis Step

- Assume $j$ is an element specified in the base step of the definition.
- Show that $\forall j,\ P(j)$ is true.

## Recursive step

- Let $x$ be a new element constructed in the recursive step of the definition.
- Assume $k_1, k_2, ..., k_m$ are elements used to construct an element $x$ in the recursive step of the definition.
- Show that $\forall\ k_1, k_2, \ldots, k_m\ \left(\left(P\left(k_1\right) \wedge P\left(k_2\right) \wedge \ldots \wedge P\left(k_m\right)\right) \rightarrow P\left(x\right)\right)$.

Cinvestav

# Definition of Structural induction

## Basis Step

- Assume $j$ is an element specified in the base step of the definition.
- Show that $\forall j,\ P(j)$ is true.

## Recursive step

- Let $x$ be a new element constructed in the recursive step of the definition.
- Assume $k_1, k_2, ..., k_m$ are elements used to construct an element $x$ in the recursive step of the definition.
- Show that $\forall\ k_1, k_2, ..., k_m\ \left(\left(P\left(k_1\right) \wedge P\left(k_2\right) \wedge \ldots \wedge P\left(k_m\right)\right) \rightarrow P\left(x\right)\right)$.

Cinvestav

# Therefore

**We can use structural induction**

To prove the correctness of a loop in an algorithm!!!

# Therefore

**We can use structural induction**

To prove the correctness of a loop in an algorithm!!!

**Yes!!! In a loop we have an iteration**

- That goes from 1 to $n$.
- And it has a property $P$ that needs to be maintained!!!

# Therefore

**We can use structural induction**

To prove the correctness of a loop in an algorithm!!!

**Yes!!! In a loop we have an iteration**

- That goes from 1 to $n$.
- And it has a property $P$ that needs to be maintained!!!

**Thus, the new element to be constructed**

It can be our array to be sorted!!!

# Therefore

**We can use structural induction**

To prove the correctness of a loop in an algorithm!!!

**Yes!!! In a loop we have an iteration**

- That goes from 1 to $n$.
- And it has a property $P$ that needs to be maintained!!!

**Thus, the new element to be constructed**

It can be our array to be sorted!!!

# Outline



Cinvestav

# Again Insertion Sort - Proving the Sorting Property

**Data:** Unsorted Sequence $A$

**Result:** Sort Sequence $A$

`Insertion Sort(A)`

**for** $j \leftarrow 2$ **to** *lenght(A)* **do**

  $key \leftarrow A[j]$;

  `// Insert` $A[j]$ `Insert` $A[j]$ `into the sorted sequence` $A[1, ..., j-1]$

  $i \leftarrow j - 1$;

  **while** $i > 0$ *and* $A[i] > key$ **do**

    $A[i+1] \leftarrow A[i]$;

    $i \leftarrow i - 1$;

  **end**

  $A[i+1] \leftarrow key$

**end**

Cinvestav

# Outline



Cinvestav

# The Structure of the Inductive Proof for a Loop

**You have an initial input $n$**

- Input of $n$ elements.
- Always be sure about your input!!!

# The Structure of the Inductive Proof for a Loop

**You have an initial input $n$**

- Input of $n$ elements.
- Always be sure about your input!!!

**Then, we have the following steeps**

1. Initialization - Before the loop.
2. Maintenance - In the loop.
3. Termination - At the end of the loop.

# The Structure of the Inductive Proof for a Loop

## You have an initial input $n$

- Input of $n$ elements.
- Always be sure about your input!!!

## Then, we have the following steeps

1. Initialization - Before the loop.
2. Maintenance - In the loop.
3. Termination - At the end of the loop.

# The Structure of the Inductive Proof for a Loop

## You have an initial input $n$

- Input of $n$ elements.
- Always be sure about your input!!!

## Then, we have the following steeps

1. Initialization - Before the loop.
2. Maintenance - In the loop.
3. Termination - At the end of the loop.

# The Structure of the Inductive Proof for a Loop

**You have an initial input $n$**

- Input of $n$ elements.
- Always be sure about your input!!!

**Then, we have the following steeps**

1. Initialization - Before the loop.
2. Maintenance - In the loop.
3. Termination - At the end of the loop.

# Initialization

**We have the following before the loop**

- That the condition is true for one element!!!
    - For example, in insertion sort $A[1]$ is an already sorted array.

# Initialization

## We have the following before the loop

- That the condition is true for one element!!!
  - For example, in insertion sort $A[1]$ is an already sorted array.

# Maintenance

**First, we must be able to prove that**

- The property holds before entering into the loop.
  - That the array $A[1 \ldots j-1]$ is sorted!!!

# Maintenance

**First, we must be able to prove that**

- The property holds before entering into the loop.
  - That the array $A[1...j-1]$ is sorted!!!

**Then, we need to prove that**

The insertion sort maintains the sorted property during the loop.

# Maintenance

**First, we must be able to prove that**

- The property holds before entering into the loop.
  - That the array $A\left[1...j-1\right]$ is sorted!!!

**Then, we need to prove that**

The insertion sort maintains the sorted property during the loop.

# Termination

## We need

- To prove that the property is TRUE for $n$ elements.
  - At the end of the algorithm $A[1, ..., n]$ is a sorted

# Outline



Cinvestav

# For example, Insertion Sort (Thanks to Luis Rodriguez Oracle Master 2012)

**First, we define the following sets with sorted elements**

- $Less = \langle x_1, ..., x_k | x_i < key, i = 1, .., k \rangle$
- $Greater = \langle x_1, ..., x_m | x_j > key, j = 1, .., m \rangle$
- $I =$ elements still not compared to the key

# For example, Insertion Sort (Thanks to Luis Rodriguez Oracle Master 2012)

**First, we define the following sets with sorted elements**

- $Less = \langle x_1, ..., x_k | x_i < key, i = 1, .., k \rangle$
- $Greater = \langle x_1, ..., x_m | x_j > key, j = 1, .., m \rangle$
- $I =$ elements still not compared to the key

**Initialization**

We have $A[1...1]$ with only one element $\Rightarrow$ it is sorted

- $A[1..j-1]$ an already sorted array
- $Less = \emptyset$
- $Greater = \emptyset$
- $I = A[1..j-1]$

# For example, Insertion Sort (Thanks to Luis Rodriguez Oracle Master 2012)

**First, we define the following sets with sorted elements**

- $Less = \langle x_1, ..., x_k | x_i < key, i = 1, .., k \rangle$
- $Greater = \langle x_1, ..., x_m | x_j > key, j = 1, .., m \rangle$
- $I =$ elements still not compared to the key

**Initialization**

We have $A[1...1]$ with only one element $\Rightarrow$ it is sorted

**Maintanence**

Before we enter to the inner while loop, we have

1. $A[1..j-1]$ an already **sorted array**
2. $Less = \emptyset$
3. $Greater = \emptyset$
4. $I = A[1...j-1]$.

# For example, Insertion Sort (Thanks to Luis Rodriguez Oracle Master 2012)

**First, we define the following sets with sorted elements**

- $Less = \langle x_1, ..., x_k | x_i < key, i = 1, .., k \rangle$
- $Greater = \langle x_1, ..., x_m | x_j > key, j = 1, .., m \rangle$
- $I =$ elements still not compared to the key

**Initialization**

We have $A[1...1]$ with only one element $\Rightarrow$ it is sorted

**Maintanence**

Before we enter to the inner while loop, we have

1. $A[1..j-1]$ an already **sorted array**
2. $Less = \emptyset$
3. $Greater = \emptyset$
4. $I = A[1...j-1]$.

# For example, Insertion Sort (Thanks to Luis Rodriguez Oracle Master 2012)

## First, we define the following sets with sorted elements

- $Less = \langle x_1, ..., x_k | x_i < key, i = 1, .., k \rangle$
- $Greater = \langle x_1, ..., x_m | x_j > key, j = 1, .., m \rangle$
- $I =$ elements still not compared to the key

## Initialization

We have $A[1...1]$ with only one element $\Rightarrow$ it is sorted

## Maintanence

Before we enter to the inner while loop, we have

1. $A[1..j-1]$ an already sorted array
2. $Less = \emptyset$
3. $Greater = \emptyset$
4. $I = A[1...j-1]$.

# For example, Insertion Sort (Thanks to Luis Rodriguez Oracle Master 2012)

## First, we define the following sets with sorted elements

- $Less = \langle x_1, ..., x_k | x_i < key, i = 1, .., k \rangle$
- $Greater = \langle x_1, ..., x_m | x_j > key, j = 1, .., m \rangle$
- $I =$ elements still not compared to the key

## Initialization

We have $A[1...1]$ with only one element $\Rightarrow$ it is sorted

## Maintanence

Before we enter to the inner while loop, we have

1. $A[1..j-1]$ an already **sorted array**
2. $Less = \emptyset$
3. $Greater = \emptyset$
4. $I = A[1...j-1]$.

# For example, Insertion Sort (Thanks to Luis Rodriguez Oracle Master 2012)

## First, we define the following sets with sorted elements

- $Less = \langle x_1, ..., x_k | x_i < key, i = 1, .., k \rangle$
- $Greater = \langle x_1, ..., x_m | x_j > key, j = 1, .., m \rangle$
- $I =$ elements still not compared to the key

## Initialization

We have $A[1...1]$ with only one element $\Rightarrow$ it is sorted

## Maintanence

Before we enter to the inner while loop, we have

1. $A[1..j-1]$ an already **sorted array**
2. $Less = \emptyset$
3. $Greater = \emptyset$
4. $I = A[1...j-1]$.

# For example, Insertion Sort (Thanks to Luis Rodriguez Oracle Master 2012)

## First, we define the following sets with sorted elements

- $Less = \langle x_1, ..., x_k | x_i < key, i = 1, .., k \rangle$
- $Greater = \langle x_1, ..., x_m | x_j > key, j = 1, .., m \rangle$
- $I =$ elements still not compared to the key

## Initialization

We have $A[1...1]$ with only one element $\Rightarrow$ it is sorted

## Maintanence

Before we enter to the inner while loop, we have

1. $A[1..j-1]$ an already **sorted array**
2. $Less = \emptyset$
3. $Greater = \emptyset$
4. $I = A[1...j-1]$.

# For example, Insertion Sort (Thanks to Luis Rodriguez Oracle Master 2012)

## First, we define the following sets with sorted elements

- $Less = \langle x_1, ..., x_k | x_i < key, i = 1, .., k \rangle$
- $Greater = \langle x_1, ..., x_m | x_j > key, j = 1, .., m \rangle$
- $I =$ elements still not compared to the key

## Initialization

We have $A[1...1]$ with only one element $\Rightarrow$ it is sorted

## Maintanence

Before we enter to the inner while loop, we have

1. $A[1..j-1]$ an already **sorted array**
2. $Less = \emptyset$
3. $Greater = \emptyset$
4. $I = A[1...j-1]$.

# Then

## Case I

You never enter in the inner loop, thus $A[j-1] < key \Rightarrow$ $Less = A[1..j-1]$, thus $A[1..j]$ is a sorted array.

# Then

## Case I

You never enter in the inner loop, thus $A[j-1] < key \Rightarrow$ $Less = A[1..j-1]$, thus $A[1..j]$ is a sorted array.

## Case II

1. You entered the inner while loop.
2. Thus at each iteration we have the following structure $A[1...j] = \boxed{I} \boxed{A[i]} \boxed{Greater}$
   - where $Greater = \langle A[i], A[i+1], \cdots, A[j-1] \rangle$.

   Note: $I$ and $Greater$ are sorted such that $A[1...j]$ is sorted by itself at this moment in the inner loop

# Then

## Case I

You never enter in the inner loop, thus $A[j-1] < key \Rightarrow Less = A[1..j-1]$, thus $A[1..j]$ is a sorted array.

## Case II

1. You entered the inner while loop.
2. Thus at each iteration we have the following structure
   $A[1...j] = \boxed{I \mid A[i] \mid Greater}$
   - where $Greater = \langle A[i], A[i+1], \cdots, A[j-1] \rangle$.

   Note: $I$ and $Greater$ are sorted such that $A[1...j]$ is sorted by itself at this moment in the inner loop

Cinvestav

# Then

## Case I

You never enter in the inner loop, thus $A[j-1] < key \Rightarrow$ $Less = A[1..j-1]$, thus $A[1..j]$ is a sorted array.

## Case II

1. You entered the inner while loop.
2. Thus at each iteration we have the following structure $A[1...j] = \begin{array}{|c|c|c|} \hline I & A[i] & Greater \\ \hline \end{array}$
   - where $Greater = \langle A[i], A[i+1], \cdots, A[j-1] \rangle$.

Note: $I$ and $Greater$ are sorted such that $A[1...j]$ is sorted by itself at this moment in the inner loop

# Then

## Case I

You never enter in the inner loop, thus $A[j-1] < key \Rightarrow$ $Less = A[1..j-1]$, thus $A[1..j]$ is a sorted array.

## Case II

1. You entered the inner while loop.
2. Thus at each iteration we have the following structure $A[1...j] = \boxed{I \mid A[i] \mid Greater}$
   - where $Greater = \langle A[i], A[i+1], \cdots, A[j-1] \rangle$.

Note: $I$ and $Greater$ are sorted such that $A[1...j]$ is sorted by itself at this moment in the inner loop

Cinvestav

# Now

## Thus, we get out of the inner loop once $I = \emptyset$.

1. We have that $A[1...j] = \boxed{Less}\boxed{A[i+1]}\boxed{Greater}$, where $A[i+2] == A[i+1]$.
2. Thus, $A[1...j]$ is sorted before inserting the key into the position $A[i+1]$.
3. Then, because elements of $A[1...j]$ are sorted,
    1. We have that after inserting the key at position $i+1$ in $A[1...j]$ the array is still sorted after iteration $j$.

# Now

**Thus, we get out of the inner loop once $I = \emptyset$.**

1. We have that $A[1...j] = \begin{array}{|c|c|c|}\hline Less & A[i+1] & Greater \\ \hline \end{array}$, where $A[i+2] == A[i+1]$.
2. Thus, $A[1...j]$ is sorted before inserting the key into the position $A[i+1]$.
3. Then, because elements of $A[1...j]$ are sorted,
   1. We have that after inserting the key at position $i+1$ in $A[1...j]$ the array is still sorted after iteration $j$.

Cinvestav

# Now

**Thus, we get out of the inner loop once $I = \emptyset$.**

1. We have that $A[1...j] = \begin{array}{|c|c|c|}\hline Less & A[i+1] & Greater \\ \hline\end{array}$, where $A[i+2] == A[i+1]$.
2. Thus, $A[1...j]$ is sorted before inserting the key into the position $A[i+1]$.
3. Then, because elements of $A[1...j]$ are sorted,
4. We have that after inserting the key at position $i+1$ in $A[1...j]$ the array is still sorted after iteration $j$.

# Now

**Thus, we get out of the inner loop once $I = \emptyset$.**

1. We have that $A[1...j] = \boxed{Less} \boxed{A[i+1]} \boxed{Greater}$, where $A[i+2] == A[i+1]$.
2. Thus, $A[1...j]$ is sorted before inserting the key into the position $A[i+1]$.
3. Then, because elements of $A[1...j]$ are sorted,
   1. We have that after inserting the key at position $i+1$ in $A[1...j]$ **the array is still sorted after iteration $j$.**

# Finally, Termination

## Termination

- Once $j > length(A)$, we get out of the outer loop and $j = n + 1$.
- Then, using the maintenance procedure we have that the sub-array $A[1...n]$ is sorted as we wanted.

Cinvestav

# Finally, Termination

## Termination

- Once $j > length(A)$, we get out of the outer loop and $j = n + 1$.
- Then, using the maintenance procedure we have that the sub-array $A\left[1...n\right]$ is sorted as we wanted.

Cinvestav

# Actually

**This is known as**

**Loop Invariance!!!**

# Actually

**This is known as**

**Loop Invariance!!!**

**Why is this important? Recursion $\equiv$ Iteration**

- How?
  - A computational system that can compute every Turing Computable function is called Turing complete (or Turing powerful).

- It computes precisely the same class of functions as do Turing machines.

# Actually

**This is known as**

**Loop Invariance!!!**

**Why is this important? Recursion $\equiv$ Iteration**

- How?
  - A computational system that can compute every Turing Computable function is called Turing complete (or Turing powerful).

**Properties**

A Turing-complete system is called Turing equivalent if every function it can compute is also Turing Computable.

- It computes precisely the same class of functions as do Turing machines.

# Actually

**This is known as**

**Loop Invariance!!!**

**Why is this important? Recursion $\equiv$ Iteration**

- How?
  - A computational system that can compute every Turing Computable function is called Turing complete (or Turing powerful).

**Properties**

A Turing-complete system is called Turing equivalent if every function it can compute is also Turing Computable.

- It computes precisely the same class of functions as do Turing machines.

# Actually

## This is known as

**Loop Invariance!!!**

## Why is this important? Recursion $\equiv$ Iteration

- How?
  - A computational system that can compute every Turing Computable function is called Turing complete (or Turing powerful).

## Properties

A Turing-complete system is called Turing equivalent if every function it can compute is also Turing Computable.

- It computes precisely the same class of functions as do Turing machines.

# Recursion $\equiv$ Iteration

**Then**

Since you can build a Turing complete language using strictly iterative structures and a Turning complete language using only recursive structures, then the two are therefore equivalent.

Cinvestav

# Recursion $\equiv$ Iteration

## Then

Since you can build a Turing complete language using strictly iterative structures and a Turning complete language using only recursive structures, then the two are therefore equivalent.

## Proof From Lambda Calculus

- Assume languages IT (with Iterative constructs only) and REC (with Recursive constructs only).
- Simulate a universal Turing machine using IT, then simulate a universal Turing machine using REC.
- The existence of the simulator programs guarantees that both IT and REC can calculate all the computable functions.

# Recursion ≡ Iteration

## Then

Since you can build a Turing complete language using strictly iterative structures and a Turning complete language using only recursive structures, then the two are therefore equivalent.

## Proof From Lambda Calculus

- Assume languages IT (with Iterative constructs only) and REC (with Recursive constructs only).
- Simulate a universal Turing machine using IT, then simulate a universal Turing machine using REC.
- The existence of the simulator programs guarantees that both IT and REC can calculate all the computable functions.

Cinvestav

# Recursion $\equiv$ Iteration

## Then

Since you can build a Turing complete language using strictly iterative structures and a Turning complete language using only recursive structures, then the two are therefore equivalent.

## Proof From Lambda Calculus

- Assume languages IT (with Iterative constructs only) and REC (with Recursive constructs only).
- Simulate a universal Turing machine using IT, then simulate a universal Turing machine using REC.
- The existence of the simulator programs guarantees that both IT and REC can calculate all the computable functions.

# Nevertheless

**Important**

- We use **RECURSIVE** procedures, when we begin to solve new problems so we can understand them.
- Then, we move everything to **ITERATIVE** procedures for speed!!!

# Nevertheless

**Important**

- We use **RECURSIVE** procedures, when we begin to solve new problems so we can understand them.
- Then, we move everything to **ITERATIVE** procedures for speed!!!

# Outline



Cinvestav

# Introduction

**Let's go back to first principles**

- We can look at our problem of complexities as bounding functions for approximation.

**Can we do better?**

Asymptotic Approximation... We will see a little bit more as the course goes...

# Introduction

## Let's go back to first principles

- We can look at our problem of complexities as bounding functions for approximation.

## Can we do better?

Asymptotic Approximation... We will see a little bit more as the course goes...

# Big $O$

**Definition (Big $O$ - Upper Bound)**

For a given function $g(n)$:

$$O(g(n)) = \{f(n) \mid \text{There exists } c > 0 \text{ and } n_0 > 0 \text{ s.t. } 0 \leq f(n) \leq cg(n) \ \forall n \geq n_0\}$$

**Example**

Cinvestav

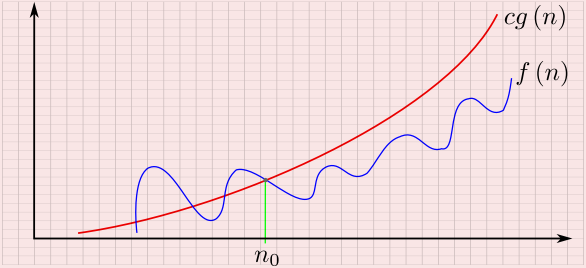
# Big $O$

## Definition (Big $O$ - Upper Bound)

For a given function $g(n)$:

$$O(g(n)) = \{f(n) | \text{ There exists } c > 0 \text{ and } n_0 > 0 \text{ s.t. } 0 \leq f(n) \leq cg(n)\ \forall n \geq n_0\}$$

## Example



# Big $\Omega$

## Definition (Big $\Omega$ - Lower Bound)

For a given function $g(n)$:

$$\Omega(g(n)) = \{f(n) | \text{ There exists } c > 0 \text{ and } n_0 > 0 \text{ s.t. } 0 \leq cg(n) \leq f(n) \; \forall n \geq n_0\}$$

Example

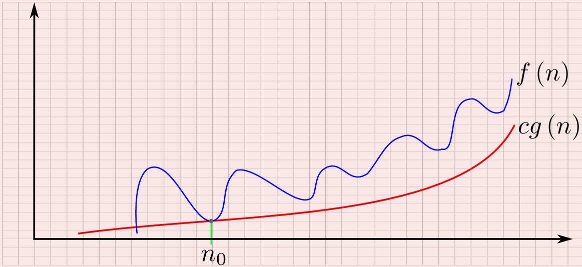
# Big $\Omega$

## Definition (Big $\Omega$ - Lower Bound)

For a given function $g(n)$:

$$\Omega(g(n)) = \{f(n) | \text{ There exists } c > 0 \text{ and } n_0 > 0 \text{ s.t. } 0 \leq cg(n) \leq f(n)\ \forall n \geq n_0\}$$

## Example



# Big $\Theta$

**Definition (Big $\Theta$ - Tight Bound)**

For a given function $g(n)$:

$$\Theta(g(n)) = \{f(n) |\text{ There exists } c_1 > 0, c_2 > 0 \text{ and } n_0 > 0 \\ \text{s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\}$$

**Example**

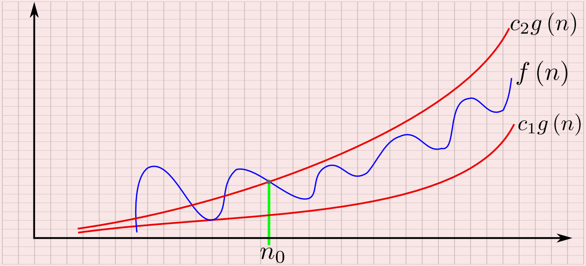
# Big $\Theta$

## Definition (Big $\Theta$ - Tight Bound)

For a given function $g(n)$:

$$\Theta(g(n)) = \{f(n) | \text{ There exists } c_1 > 0, c_2 > 0 \text{ and } n_0 > 0 \\ \text{s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \ \forall n \geq n_0\}$$

## Example



# Outline



Cinvestav

# Can we relate this with practical examples?

**You could say**

This is too theoretical!

**However, this is not the case!!**

Look at this java code...

# Can we relate this with practical examples?

**You could say**

This is too theoretical!

**However, this is not the case!!**

Look at this java code...

# Example: Step count of Insertion Sort in Java

## Counting when $A.length = n$

```
// Sort A assume is full
public int[] InsertionSort(int[] A){          Step
 // Initial Variables                          0
 int B[] = new int[A.length];                  1
 int size = 1;                                 1
 int i, j, t;                                  1
 // Initialize the Array B                     0
 B[0]=A[0];                                    1
 for(i = 1; i < A.length; i++){                n
   t = A[i];                                   n-1
   for(j=size-1;
       j>=0&&t<B[j];j--)                       i+1
    {
     //shift to the right                      0
       B[j+1]=B[j];}                           i
    B[j+1]=t;                                  n-1
    size++;                                    n-1
  }
 return B;                                     1
}
```

# The Result

**Step count for body of for loop is**

$$6 + 3(n-1) + n + \sum_{i=1}^{n-1}(i+1) + \sum_{j=1}^{n-1}(i) \qquad (13)$$

**The summation**

They have the quadratic terms $n^2$.

**Complexity**

Insertion sort complexity is $O\left(n^2\right)$

Cinvestav

# The Result

## Step count for body of for loop is

$$6 + 3\left(n-1\right) + n + \sum_{i=1}^{n-1}\left(i+1\right) + \sum_{j=1}^{n-1}\left(i\right) \tag{13}$$

## The summation

They have the quadratic terms $n^2$.

## Complexity

Insertion sort complexity is $O\left(n^2\right)$

# The Result

**Step count for body of for loop is**

$$6+3\left(n-1\right)+n+\sum_{i=1}^{n-1}\left(i+1\right)+\sum_{j=1}^{n-1}\left(i\right) \tag{13}$$

**The summation**

They have the quadratic terms $n^2$.

**Complexity**

Insertion sort complexity is $O\left(n^2\right)$

# What does this means for insertion sort?

## We have

$$6 + 3(n-1) + n + \sum_{i=1}^{n-1} (i+1) + \sum_{j=1}^{n-1} (i) = \ldots$$

$$3 + 4n + \frac{n(n-1)}{2} + n - 1 + \frac{n(n-1)}{2} = \ldots$$

$$2 + 5n + n(n-1) = \ldots$$

$$n^2 + 4n + 2 \leq n^2 + 4n^2 + 2n^2$$

# What does this means for insertion sort?

## We have

$$6 + 3(n-1) + n + \sum_{i=1}^{n-1}(i+1) + \sum_{j=1}^{n-1}(i) = \ldots$$

$$3 + 4n + \frac{n(n-1)}{2} + n - 1 + \frac{n(n-1)}{2} = \ldots$$

$$2 + 5n + n(n-1) = \ldots$$

$$n^2 + 4n + 2 \leq n^2 + 4n^2 + 2n^2$$

## Thus

$$n^2 + 4n + 2 \leq 7n^2 \tag{14}$$

With $T_{insertion}(n) = n^2 + 4n + 2$ describing the number of steps for insertion when we have $n$ numbers.

# What does this means for insertion sort?

## We have

$$
\begin{aligned}
6+3\,(n-1)+n+\sum_{i=1}^{n-1}(i+1)+\sum_{j=1}^{n-1}(i) &= \ldots\\
3+4n+\frac{n\,(n-1)}{2}+n-1+\frac{n\,(n-1)}{2} &= \ldots\\
2+5n+n(n-1) &= \ldots\\
n^2+4n+2 &\leq n^2+4n^2+2n^2
\end{aligned}
$$

## Thus

$$
n^2+4n+2 \leq 7n^2 \tag{14}
$$

With $T_{insertion}(n) = n^2+4n+2$ describing the number of steps for insertion when we have $n$ numbers.

# What does this means for insertion sort?

## We have

$$
\begin{aligned}
6+3(n-1)+n+\sum_{i=1}^{n-1}(i+1)+\sum_{j=1}^{n-1}(i) &= \ldots \\
3+4n+\frac{n(n-1)}{2}+n-1+\frac{n(n-1)}{2} &= \ldots \\
2+5n+n(n-1) &= \ldots \\
n^2+4n+2 &\leq n^2+4n^2+2n^2
\end{aligned}
$$

## Thus

$$n^2+4n+2 \leq 7n^2 \tag{14}$$

With $T_{insertion}(n) = n^2+4n+2$ describing the number of steps for insertion when we have $n$ numbers.

# What does this means for insertion sort?

## We have

$$
\begin{aligned}
6+3(n-1)+n+\sum_{i=1}^{n-1}(i+1)+\sum_{j=1}^{n-1}(i) &= \ldots \\
3+4n+\frac{n(n-1)}{2}+n-1+\frac{n(n-1)}{2} &= \ldots \\
2+5n+n(n-1) &= \ldots \\
n^2+4n+2 &\leq n^2+4n^2+2n^2
\end{aligned}
$$

## Thus

$$n^2+4n+2 \leq 7n^2 \tag{14}$$

With $T_{insertion}(n)=n^2+4n+2$ describing the number of steps for insertion when we have $n$ numbers.

# Actually

**For $n_0 = 2$**

$$2^2 + 4 \times 2 + 2 = 14 < 7 \times 2^2 = 28 \tag{15}$$

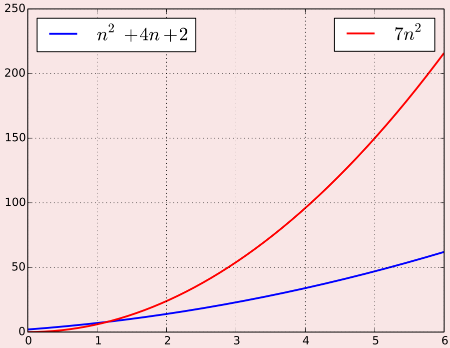
**Graphically**

# Actually

**For $n_0 = 2$**

$$2^2 + 4 \times 2 + 2 = 14 < 7 \times 2^2 = 28 \quad (15)$$

**Graphically**



# Meaning

## First

Time or number of operations does not exceed $cn^2$ for a constant $c$ on any input of size $n$ ($n$ suitably large).

## Questions

- Is $O(n^2)$ too much time?
- Is the algorithm practical?

# Meaning

## First

Time or number of operations does not exceed $cn^2$ for a constant $c$ on any input of size $n$ ($n$ suitably large).

## Questions

- Is $O(n^2)$ too much time?
- Is the algorithm practical?

# Outline



Cinvestav

# Then

## We have the following

| $n$ | $n$ | $n \log n$ | $n^2$ | $n^3$ | $n^4$ |
|---|---|---|---|---|---|
| 1000 | 1 micros | 10 micros | 1 milis | 1 second | 17 minutes |
| 10,000 | 10 micros | 130 micros | 100 milis | 17 minutes | 116 days |
| $10^6$ | 1 milis | 20 milis | 17 minutes | 32 years | $3 \times 10^7$years |

## It is much worse

# Then

## We have the following

| $n$ | $n$ | $n \log n$ | $n^2$ | $n^3$ | $n^4$ |
|---|---|---|---|---|---|
| 1000 | 1 micros | 10 micros | 1 milis | 1 second | 17 minutes |
| 10,000 | 10 micros | 130 micros | 100 milis | 17 minutes | 116 days |
| $10^6$ | 1 milis | 20 milis | 17 minutes | 32 years | $3 \times 10^7$years |

## It is much worse

| $n$ | $n^{10}$ | $2^n$ |
|---|---|---|
| 1000 | $3.2 \times 10^{13}$ years | $3.2 \times 10^{283}$ years |
| 10,000 | ??? | ??? |
| $10^6$ | ????? | ????? |

**The Reign of the Non Polynomial Algorithms**

Cinvestav

# Outline



Cinvestav

# Little $o$ Bound

## Definition

For a given function $g(n)$:

$$o(g(n)) = \{f(n) \mid \text{For any } c > 0 \text{ there exists } n_0 > 0 \text{ s.t. } 0 \leq f(n) < cg(n)\ \forall n \geq n_0\}$$

## Observations

It is not tight.

- For example, We have that $2n = o\left(n^2\right)$, but $2n^2 \neq o(n^2)$.

# Little $o$ Bound

## Definition

For a given function $g(n)$:

$$o(g(n)) = \{f(n) | \text{ For any } c > 0 \text{ there exists } n_0 > 0 \text{ s.t. } 0 \leq f(n) < cg(n)\ \forall n \geq n_0\}$$

## Observations

It is not tight.

- For example, We have that $2n = o\left(n^2\right)$, but $2n^2 \neq o(n^2)$.

Cinvestav

# Little $o$ Bound

## Not only that

Under the definition, we have for any $f(n) \in o(g(n))$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

# Little $\omega$ Bound

## Definition

For a given function $g(n)$:

$$\omega(g(n)) = \{f(n) | \text{ For any } c > 0 \text{ there exists } n_0 > 0 \text{ s.t. } 0 \leq cg(n) < f(n) \ \forall n \geq n_0\}$$

## Observations

It is not tight.

- For example, $\frac{n^2}{2} = \omega(n)$, but $\frac{n^2}{2} \neq \omega(n^2)$.

# Little $\omega$ Bound

## Definition

For a given function $g(n)$:

$$\omega(g(n)) = \{f(n) | \text{ For any } c > 0 \text{ there exists } n_0 > 0 \text{ s.t. } 0 \leq cg(n) < f(n) \ \forall n \geq n_0\}$$

## Observations

It is not tight.

- For example, $\frac{n^2}{2} = \omega(n)$, but $\frac{n^2}{2} \neq \omega(n^2)$.

Cinvestav

# Little $\omega$ Bound

## Not only that

Under the definition, we have for any $f(n) \in \omega(g(n))$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

# Outline



Cinvestav

# Interpretation

**How do you interpret $f(n) = O(n^2)$?**

It means that $f(n)$ belongs to $O(n^2)$

# Interpretation

**How do you interpret $f(n) = O(n^2)$?**

It means that $f(n)$ belongs to $O(n^2)$

**How do you interpret $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$?**

$\exists f(n) \in \Theta(n)$ such that:

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + f(n) \\ &= 2n^2 + \Theta(n) \end{aligned}$$

# Interpretation

**How do you interpret $f(n) = O(n^2)$?**

It means that $f(n)$ belongs to $O(n^2)$

**How do you interpret $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$?**

$\exists f(n) \in \Theta(n)$ such that:

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + f(n) \\ &= 2n^2 + \Theta(n) \end{aligned}$$

Cinvestav

# Interpretation

**How do you interpret $f(n) = O(n^2)$?**

It means that $f(n)$ belongs to $O(n^2)$

**How do you interpret $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$?**

$\exists f(n) \in \Theta(n)$ such that:

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + f(n) \\ &= 2n^2 + \Theta(n) \end{aligned}$$

Cinvestav

# Outline



Cinvestav

# Properties

## Equivalence

For any two functions $f(n)$ and $g(n)$, we have that $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

## Transitivity

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ then $f(n) = \Theta(h(n))$

## Reflexivity

$f(n) = \Theta(f(n))$

## Symmetry

$f(n) = \Theta(g(n)) \Longleftrightarrow g(n) = \Theta(f(n))$

## Transpose Symmetry

$f(n) = O(g(n)) \Longleftrightarrow g(n) = \Omega(f(n))$

# Properties

**Equivalence**

For any two functions $f(n)$ and $g(n)$, we have that $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

**Transitivity**

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ then $f(n) = \Theta(h(n))$

**Reflexivity**

$f(n) = \Theta(f(n))$

**Symmetry**

$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$

**Transpose Symmetry**

$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$

# Properties

## Equivalence

For any two functions $f(n)$ and $g(n)$, we have that $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

## Transitivity

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ then $f(n) = \Theta(h(n))$

## Reflexivity

$f(n) = \Theta(f(n))$

## Symmetry

$f(n) = \Theta(g(n)) \Longleftrightarrow g(n) = \Theta(f(n))$

## Transpose Symmetry

$f(n) = O(g(n)) \Longleftrightarrow g(n) = \Omega(f(n))$

# Properties

### Equivalence

For any two functions $f(n)$ and $g(n)$, we have that $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

### Transitivity

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ then $f(n) = \Theta(h(n))$

### Reflexivity

$f(n) = \Theta(f(n))$

### Symmetry

$f(n) = \Theta(g(n)) \Longleftrightarrow g(n) = \Theta(f(n))$

### Transpose Symmetry

$f(n) = O(g(n)) \Longleftrightarrow g(n) = \Omega(f(n))$

# Properties

## Equivalence

For any two functions $f(n)$ and $g(n)$, we have that $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

## Transitivity

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ then $f(n) = \Theta(h(n))$

## Reflexivity

$f(n) = \Theta(f(n))$

## Symmetry

$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$

## Transpose Symmetry

$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$

# Ok, we have the basics...

**Now...**

What do we do?

# Ok, we have the basics...

**Now...**

What do we do?

**We will look at methods to solve recursions!!!**

1. Substitution Method
2. Recursion-Tree Method
3. Master Method

# Ok, we have the basics...

**Now...**

What do we do?

**We will look at methods to solve recursions!!!**

1. Substitution Method
2. Recursion-Tree Method
3. Master Method

# Ok, we have the basics...

**Now...**

What do we do?

**We will look at methods to solve recursions!!!**

1. Substitution Method
2. Recursion-Tree Method
3. Master Method

# Outline

# The Substitution Method

## The Steps in the Method

- Guess the form of the solution.
- Use mathematical induction to find the constants and show that the solution works.

# The Substitution Method

## The Steps in the Method

- Guess the form of the solution.
- Use mathematical induction to find the constants and show that the solution works.

# Example

**Solve the following recurrence**

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \tag{16}$$

**I decide to do the following GUESS**

Guess that $T(n) = O(n \log n)$!!!

**For this**

We assume that the bound holds for $\left\lfloor \frac{n}{2} \right\rfloor < n$ (Remember Inductive Hypothesis!!!).

Cinvestav

# Example

## Solve the following recurrence

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \tag{16}$$

## I decide to do the following GUESS

Guess that $T(n) = O(n \log n)$!!!

## For this

We assume that the bound holds for $\left\lfloor \frac{n}{2} \right\rfloor < n$ (Remember Inductive Hypothesis!!!).

Cinvestav

# Example

## Solve the following recurrence

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \tag{16}$$

## I decide to do the following GUESS

Guess that $T(n) = O(n \log n)$!!!

## For this

We assume that the bound holds for $\left\lfloor \frac{n}{2} \right\rfloor < n$ (Remember Inductive Hypothesis!!!).

# Therefore

**We have that the following inequality holds**

$$T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) \leq c\left\lfloor\frac{n}{2}\right\rfloor \log_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right) \tag{17}$$

**Thus, we have that**

# Therefore

**We have that the following inequality holds**

$$T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \leq c\left\lfloor \frac{n}{2} \right\rfloor \log_2\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \tag{17}$$

**Thus, we have that**

$$\begin{aligned} T(n) &= 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq 2c\left\lfloor \frac{n}{2} \right\rfloor \log_2\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \end{aligned}$$

# Therefore

**We have that the following inequality holds**

$$T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \leq c \left\lfloor \frac{n}{2} \right\rfloor \log_2 \left(\left\lfloor \frac{n}{2} \right\rfloor\right) \tag{17}$$

**Thus, we have that**

$$\begin{aligned} T(n) &= 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq 2c \left\lfloor \frac{n}{2} \right\rfloor \log_2 \left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \end{aligned}$$

Cinvestav

# Thus

## We have that

$$T(n) \leq 2c \left\lfloor \frac{n}{2} \right\rfloor \log_2 \left( \left\lfloor \frac{n}{2} \right\rfloor \right) + n$$

$$\leq 2c \times \frac{n}{2} \times \log_2 \left( \frac{n}{2} \right) + n$$

$$= cn \log_2 \left( \frac{n}{2} \right) + n$$

# Thus

## We have that

$$\begin{aligned} T(n) &\leq 2c\left\lfloor \frac{n}{2} \right\rfloor \log_2\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq 2c \times \frac{n}{2} \times \log_2\left(\frac{n}{2}\right) + n \\ &= cn\log_2\left(\frac{n}{2}\right) + n \end{aligned}$$

## Remember the following

$$\begin{aligned} \log_2\left(\frac{n}{2}\right) &= \log_2 n - \log_2 2 \\ &= \log_2 n - 1 \end{aligned}$$

# Thus

## We have that

$$
\begin{aligned}
T(n) &\leq 2c\left\lfloor \frac{n}{2} \right\rfloor \log_2\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n\\
&\leq 2c \times \frac{n}{2} \times \log_2\left(\frac{n}{2}\right) + n\\
&= cn\log_2\left(\frac{n}{2}\right) + n
\end{aligned}
$$

## Remember the following

$$
\begin{aligned}
\log_2\left(\frac{n}{2}\right) &= \log_2 n - \log_2 2\\
&= \log_2 n - 1
\end{aligned}
$$

# Thus

## We have that

$$
\begin{aligned}
T(n) &\leq 2c \left\lfloor \frac{n}{2} \right\rfloor \log_2 \left( \left\lfloor \frac{n}{2} \right\rfloor \right) + n \\
&\leq 2c \times \frac{n}{2} \times \log_2 \left( \frac{n}{2} \right) + n \\
&= cn \log_2 \left( \frac{n}{2} \right) + n
\end{aligned}
$$

## Remember the following

$$
\begin{aligned}
\log_2 \left( \frac{n}{2} \right) &= \log_2 n - \log_2 2 \\
&= \log_2 n - 1
\end{aligned}
$$

# Thus

## We have that

$$
\begin{aligned}
T(n) &\leq 2c\left\lfloor\frac{n}{2}\right\rfloor\log_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right)+n\\
&\leq 2c\times\frac{n}{2}\times\log_2\left(\frac{n}{2}\right)+n\\
&= cn\log_2\left(\frac{n}{2}\right)+n
\end{aligned}
$$

## Remember the following

$$
\begin{aligned}
\log_2\left(\frac{n}{2}\right) &= \log_2 n-\log_2 2\\
&= \log_2 n-1
\end{aligned}
$$

# Finally, we have

## We have

$$T(n) \leq cn\log_2 n - cn + n$$

# Finally, we have

## We have

$$T(n) \leq cn \log_2 n - cn + n$$

## Now, we need to have that

$$-cn + n \leq 0$$

$$n \leq cn$$

$$1 \leq n$$

# Finally, we have

**We have**

$$T(n) \leq cn \log_2 n - cn + n$$

**Now, we need to have that**

$$\begin{aligned} -cn + n &\leq 0 \\ n &\leq cn \\ 1 &\leq n \end{aligned}$$

**Then, as long $c \geq 1$, we have that**

$$\begin{aligned} T(n) &\leq cn \log_2 n - cn + n \\ &\leq cn \log_2 n \end{aligned}$$

# Finally, we have

**We have**

$$T(n) \leq cn \log_2 n - cn + n$$

**Now, we need to have that**

$$\begin{aligned} -cn + n &\leq 0 \\ n &\leq cn \\ 1 &\leq n \end{aligned}$$

**Then, as long $c \geq 1$, we have that**

$$\begin{aligned} T(n) &\leq cn \log_2 n - cn + n \\ &\leq cn \log_2 n \end{aligned}$$

# Finally, we have

## We have

$$T(n) \leq cn \log_2 n - cn + n$$

## Now, we need to have that

$$\begin{aligned} -cn + n &\leq 0 \\ n &\leq cn \\ 1 &\leq n \end{aligned}$$

## Then, as long $c \geq 1$, we have that

$$\begin{aligned} T(n) &\leq cn \log_2 n - cn + n \\ &\leq cn \log_2 n \end{aligned}$$

# Finally, we have

## We have

$$T(n) \leq cn\log_2 n - cn + n$$

## Now, we need to have that

$$\begin{aligned} -cn + n &\leq 0 \\ n &\leq cn \\ 1 &\leq n \end{aligned}$$

## Then, as long $c \geq 1$, we have that

$$\begin{aligned} T(n) &\leq cn\log_2 n - cn + n \\ &\leq cn\log_2 n \end{aligned}$$

# Subtleties

**What about ?**

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1$$

**Here, we have a problem**

Look at the Board!!!

# Outline



Cinvestav

# The Recursion-Tree Method

## Surprise

- Sometimes is hard to do a good guess.
- For example $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$

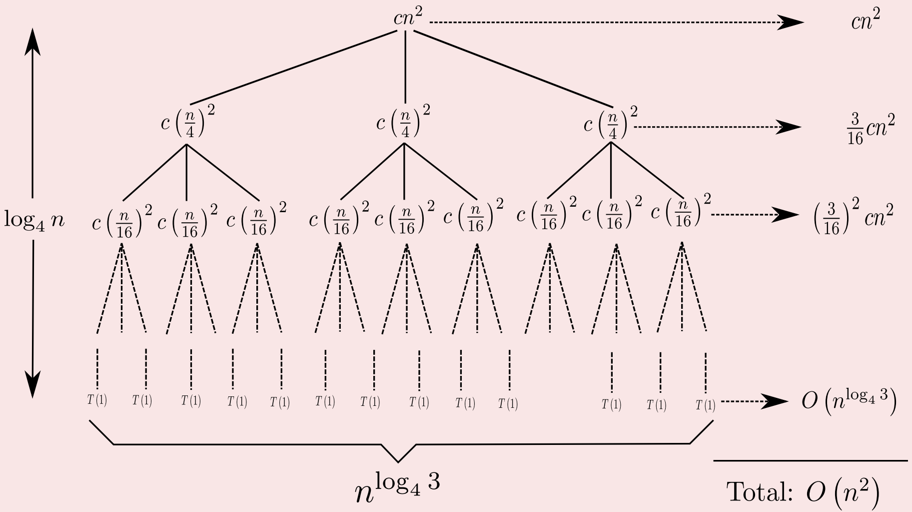
# The Recursion-Tree Method

## Surprise

- Sometimes is hard to do a good guess.
- For example $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$

# The Recursion-Tree Method

## Therefore, we draw the recursion tree



# Using the previous expansion, we count!!!

## Counting Again!!!

- A subproblem for a node at depth $i$ is $n/4^i$, then once

$$n/4^i = 1 \Rightarrow i = \log_4 n \tag{18}$$

- At each level $i = 0, 1, 2, ..., \log_4 n - 1$ the cost of each node is

$$c\left(\frac{n}{4^i}\right)^2 \tag{19}$$

- At each level $i = 0, 1, 2, ..., \log_4 n - 1$ the total cost of the work is

$$3^i c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2 \tag{20}$$

- At depth $\log_4 n$, we have this many nodes

$$3^{\log_4 n} = n^{\log_4 3} \tag{21}$$

# Using the previous expansion, we count!!!

## Counting Again!!!

- A subproblem for a node at depth $i$ is $n/4^i$, then once

$$n/4^i = 1 \Rightarrow i = \log_4 n \tag{18}$$

- At each level $i = 0, 1, 2, ..., \log_4 n - 1$ the cost of each node is

$$c\left(\frac{n}{4^i}\right)^2 \tag{19}$$

- At each level $i = 0, 1, 2, ..., \log_4 n - 1$ the total cost of the work is

$$3^i c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2 \tag{20}$$

- At depth $\log_4 n$, we have this many nodes

$$3^{\log_4 n} = n^{\log_4 3} \tag{21}$$

# Using the previous expansion, we count!!!

## Counting Again!!!

- A subproblem for a node at depth $i$ is $n/4^i$, then once

$$n/4^i = 1 \Rightarrow i = \log_4 n \tag{18}$$

- At each level $i = 0, 1, 2, ..., \log_4 n - 1$ the cost of each node is

$$c\left(\frac{n}{4^i}\right)^2 \tag{19}$$

- At each level $i = 0, 1, 2, ..., \log_4 n - 1$ the total cost of the work is

$$3^i c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2 \tag{20}$$

- At depth $\log_4 n$, we have this many nodes

$$3^{\log_4 n} = n^{\log_4 3} \tag{21}$$

# Using the previous expansion, we count!!!

## Counting Again!!!

- A subproblem for a node at depth $i$ is $n/4^i$, then once

$$n/4^i = 1 \Rightarrow i = \log_4 n \tag{18}$$

- At each level $i = 0, 1, 2, ..., \log_4 n - 1$ the cost of each node is

$$c\left(\frac{n}{4^i}\right)^2 \tag{19}$$

- At each level $i = 0, 1, 2, ..., \log_4 n - 1$ the total cost of the work is

$$3^i c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2 \tag{20}$$

- At depth $\log_4 n$, we have this many nodes

$$3^{\log_4 n} = n^{\log_4 3} \tag{21}$$

# Now, we add all this counts!!!

## Then, we have that

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + n^{\log_4 3} \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + n^{\log_4 3} \\
&= \frac{1}{1-(3/16)} cn^2 + n^{\log_4 3} \\
&= O(n^2)
\end{aligned}
$$

# Now, we add all this counts!!!

## Then, we have that

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + n^{\log_4 3} \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + n^{\log_4 3} \\
&= \frac{1}{1-(3/16)} cn^2 + n^{\log_4 3} \\
&= O(n^2)
\end{aligned}
$$

# Now, we add all this counts!!!

## Then, we have that

$$
\begin{aligned}
T(n) = & \sum_{i=0}^{log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + n^{\log_4 3} \\
& < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + n^{\log_4 3} \\
& = \frac{1}{1-(3/16)} cn^2 + n^{\log_4 3} \\
& = O(n^2)
\end{aligned}
$$

## Now, we add all this counts!!!

### Then, we have that

$$
\begin{aligned}
T(n) = & \sum_{i=0}^{log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + n^{\log_4 3} \\
< & \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + n^{\log_4 3} \\
= & \frac{1}{1-(3/16)} cn^2 + n^{\log_4 3} \\
= & O(n^2)
\end{aligned}
$$

Cinvestav

# Outline

# The Master Theorem

## Theorem - Cookbook for solving $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{22}$$

where we interpret $\frac{n}{b}$ as $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$. Then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$.

# The Master Theorem

## Theorem - Cookbook for solving $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{22}$$

where we interpret $\frac{n}{b}$ as $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$. Then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$.

# The Master Theorem

**Theorem - Cookbook for solving $T(n) = aT\left(\frac{n}{b}\right) + f(n)$**

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{22}$$

where we interpret $\frac{n}{b}$ as $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$. Then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$.

# The Master Theorem

**Theorem - Cookbook for solving $T(n) = aT\left(\frac{n}{b}\right) + f(n)$**

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{22}$$

where we interpret $\frac{n}{b}$ as $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$. Then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$.

Cinvestav

# The Master Theorem

**Theorem - Cookbook for solving $T(n) = aT\left(\frac{n}{b}\right) + f(n)$**

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{22}$$

where we interpret $\frac{n}{b}$ as $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$. Then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$.

Cinvestav

# We will prove a simplified version

## Simplified Master Method

If $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$ for some constants $a > 0$, $b > 1$, and $d \geq 0$ then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

## Proof at the Board

Look at this

# Using the Master Theorem

## Consider the following recursion

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

## We have that

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

## Thus

$n^{\log_3 9} = \Theta(n^2)$ and $f(n) = O(n^{\log_3 9 - \epsilon})$ with $\epsilon = 1$

## Then, we use then the case 1 of the Master Theorem

$$T(n) = O\left(n^2\right) \tag{23}$$

# Using the Master Theorem

## Consider the following recursion

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

## We have that

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

## Thus

$n^{\log_3 9} = \Theta(n^2)$ and $f(n) = O(n^{\log_3 9 - \epsilon})$ with $\epsilon = 1$

## Then, we use then the case 1 of the Master Theorem

$$T(n) = O\left(n^2\right) \tag{23}$$

Cinvestav

# Using the Master Theorem

## Consider the following recursion

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

## We have that

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

## Thus

$n^{\log_3 9} = \Theta(n^2)$ and $f(n) = O(n^{\log_3 9 - \epsilon})$ with $\epsilon = 1$

## Then, we use then the case 1 of the Master Theorem

$$T(n) = O\left(n^2\right) \tag{23}$$

# Using the Master Theorem

## Consider the following recursion

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

## We have that

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

## Thus

$n^{\log_3 9} = \Theta(n^2)$ and $f(n) = O(n^{\log_3 9 - \epsilon})$ with $\epsilon = 1$

## Then, we use then the case 1 of the Master Theorem

$$T(n) = O\left(n^2\right) \tag{23}$$