Data structures

TREES AND RECURRENCE

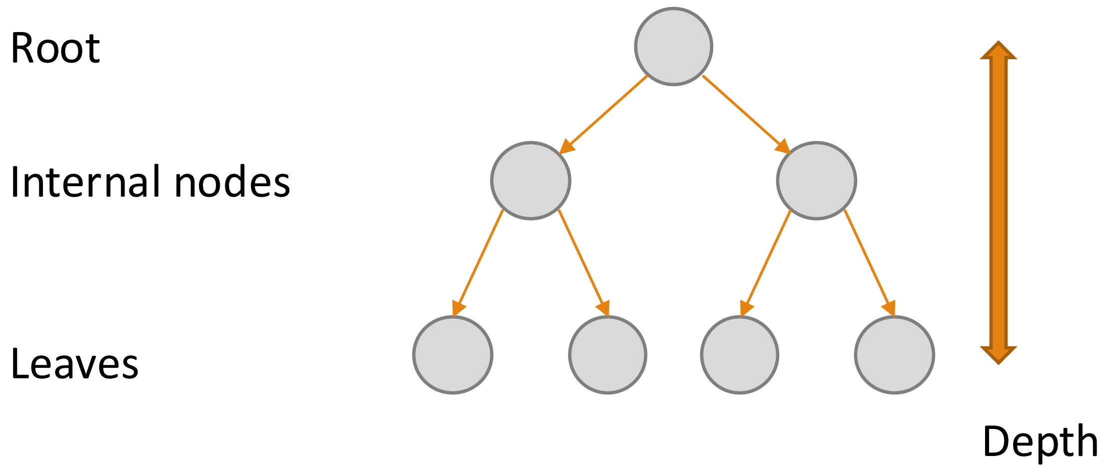
Today

- Trees
- Complexity analysis of recursive functions

Trees - Reminder

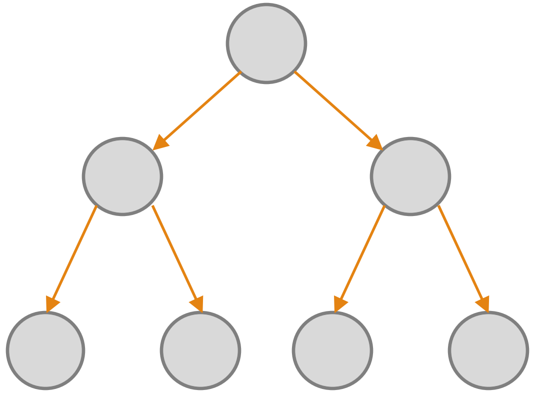
Trees - Reminder

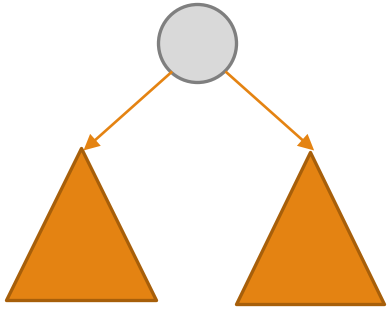
A tree is a way to represent hierarchical information:



Trees - Exercise

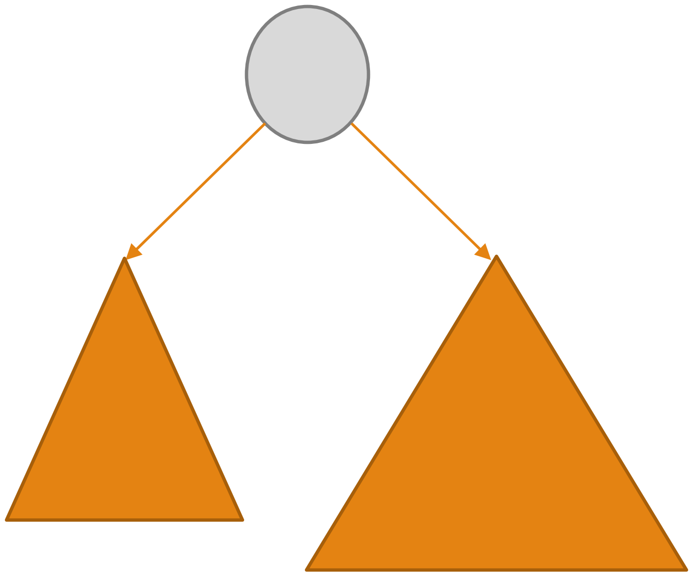
Describe an Algorithm that returns the sum of all values in the tree





Trees - Exercise

Describe an Algorithm that returns the sum of all values in the tree



Trees - Exercise

Describe an Algorithm that returns the sum of all values in the tree

GetTreeSum(Node):

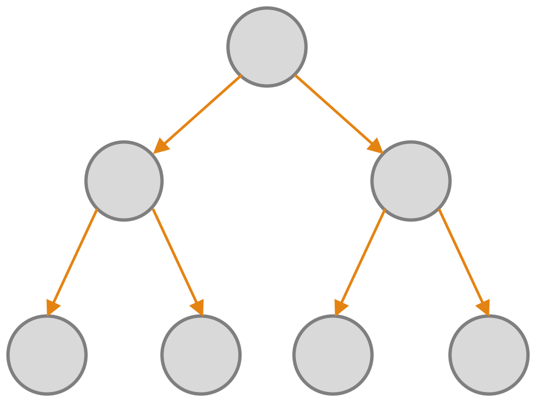
1. If Node = null
    1. Return 0
2. MyValue ←Node.Value
3. LeftSum ← GetTreeSum(Node.Left)
4. RightSum ← GetTreeSum(Node.Right)
5. Return MyValue + LeftSum + RightSum

Trees - Exercise

Describe an Algorithm that returns the sum of all values in the tree

GetTreeSum(Node):

1. If Node = null
   1. Return 0
2. MyValue ←Node.Value
3. LeftSum ← GetTreeSum(Node.Left)
4. RightSum ← GetTreeSum(Node.Right)
5. Return MyValue + LeftSum + RightSum



What kind of Traversal did we use in this algorithm?

Can we use another kind of traversal?

Recurrence Complexity Analysis

Recurrence Complexity Analysis

Recursion defines the solution to a problem using the solution to a smaller instance of the problem.

Recurrence Complexity Analysis

Recursion defines the solution to a problem using the solution to a smaller instance of the problem.

Examples:

Recurrence Complexity Analysis

Recursion defines the solution to a problem using the solution to a smaller instance of the problem.

Examples:

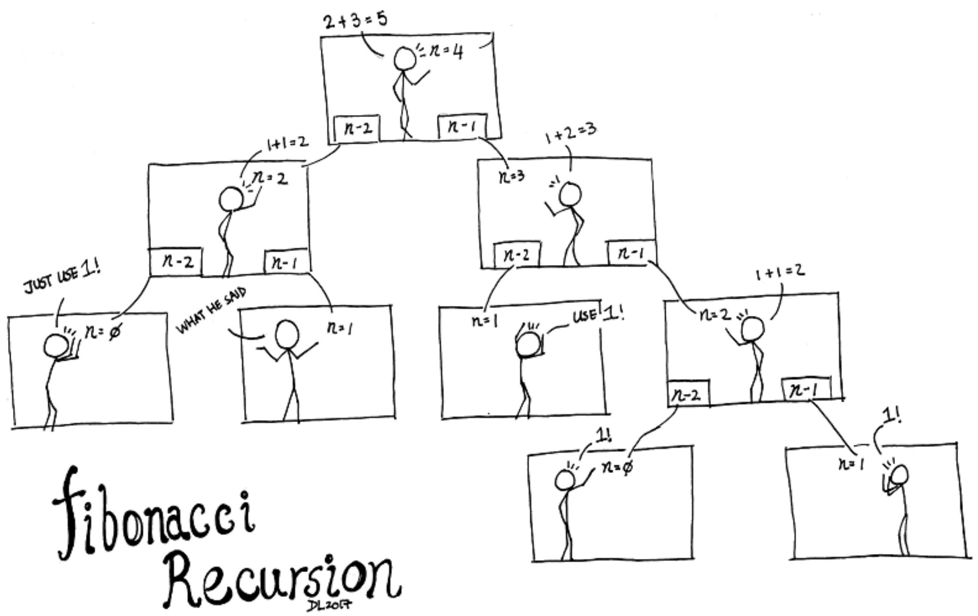
- Computing factorial
  - n! = (n-1)! * n

Recurrence Complexity Analysis

Recursion defines the solution to a problem using the solution to a smaller instance of the problem.

Examples:

- Computing factorial
  - n! = (n-1)! * n
- Fibonacci series
  - f(n) = f(n-1) + f(n-2)

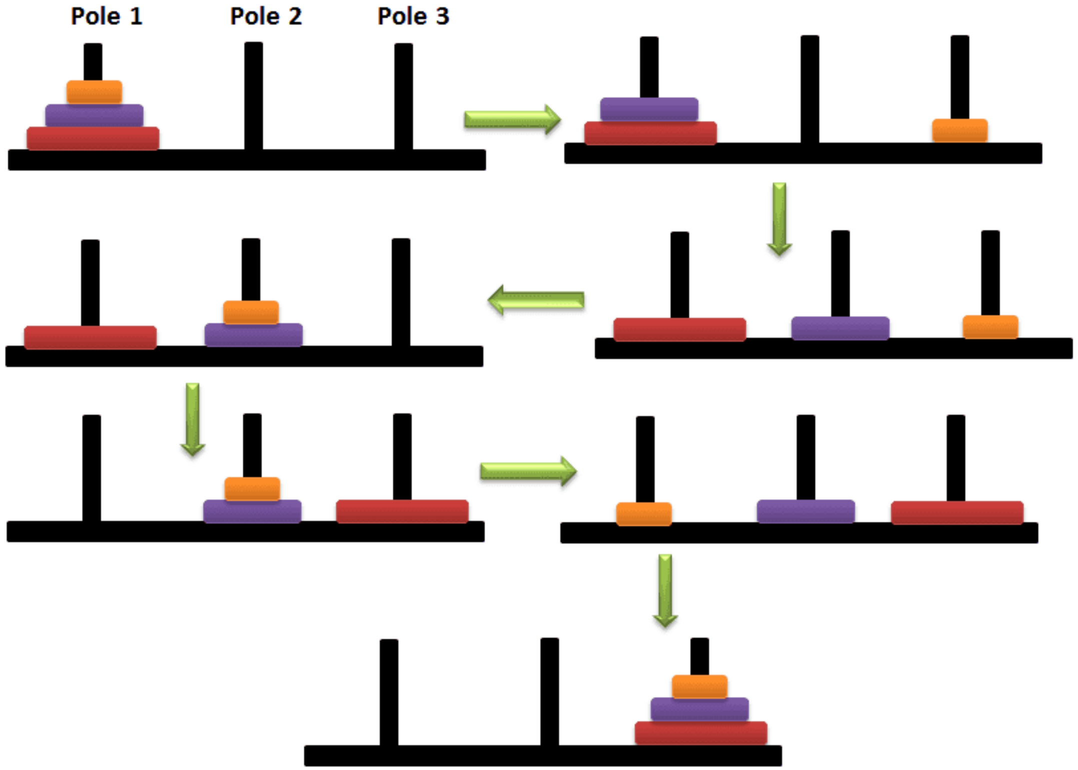


Recurrence Complexity Analysis

Recursion defines the solution to a problem using the solution to a smaller instance of the problem.

<u>Examples:</u>

- Computing factorial
  - n! = (n-1)! * n
- Fibonacci series
  - f(n) = f(n-1) + f(n-2)
- Solving Towers of Hanoi
  - To move n rings: move n-1 rings to aux pole, move 1 ring, then move n-1 rings from aux pole



Recurrence Complexity Analysis

Recursion defines the solution to a problem using the solution to a smaller instance of the problem.

<u>Examples:</u>

- Computing factorial
  - n! = (n-1)! * n
- Fibonacci series
  - f(n) = f(n-1) + f(n-2)
- Solving Towers of Hanoi
  - To move n rings: move n-1 rings to aux pole, move 1 ring, then move n-1 rings from aux pole
- Binary Search

How to Solve Recurrence Complexity?

<u>The iteration/substitution method</u>

- Assign the formula into itself several times
- Make an educated "guess"
- Assign an initial condition and extract a closed form solution
- Prove your guess by induction

Exercise

Given the following code:

```
def my_algo(n):
  if n > 1:
    for i in range(n):
        do_something();
    m = n // 2
    my_algo(m)
```

Assume we are calling *my_algo(n)* and that the runtime of *do_something* is O(1).

1 – Find a recursive formula T(n) for the runtime complexity of the given code.

2 – Find a closed form solution of T(n) using the iteration method.

Exercise

Given the following code:

```
def my_algo(n):
  if n > 1:
    for i in range(n):
        do_something();
    m = n // 2
    my_algo(m)
```

T(n)=
O(1)
O(n)
· O(1)
O(1)
T(m) = T(n/2)

$T(n) =$

Exercise

Given the following code:

```python
def my_algo(n):
  if n > 1:
    for i in range(n):
        do_something();
    m = n // 2
    my_algo(m)
```

T(n)=
O(1)
O(n)
· O(1)
O(1)
T(m) = T(n/2)

$$T(n) = O(1) + O(n \cdot 1) + O(1) + T\left(\frac{n}{2}\right)$$

Exercise

Given the following code:

```
def my_algo(n):
  if n > 1:
    for i in range(n):
        do_something();
    m = n // 2
    my_algo(m)
```

T(n)=
O(1)
O(n)
· O(1)
O(1)
T(m) = T(n/2)

$$T(n) = O(1) + O(n \cdot 1) + O(1) + T\left(\frac{n}{2}\right) = T\left(\frac{n}{2}\right) + O(n)$$

Exercise

Given the following code:

```
def my_algo(n):
  if n > 1:
    for i in range(n):
        do_something();
    m = n // 2
    my_algo(m)
```

T(n)=
O(1)
O(n)
· O(1)
O(1)
T(m) = T(n/2)

$$T(n) = O(1) + O(n \cdot 1) + O(1) + T\left(\frac{n}{2}\right) = T\left(\frac{n}{2}\right) + O(n)$$

$$T(1) = O(1)$$

Exercise

Given the following code:

```
def my_algo(n):
  if n > 1:
    for i in range(n):
        do_something();
    m = n // 2
    my_algo(m)
```

T(n)=
O(1)
O(n)
· O(1)
O(1)
T(m) = T(n/2)

Recursive formula: $T(n) = O(1) + O(n \cdot 1) + O(1) + T\left(\frac{n}{2}\right) = T\left(\frac{n}{2}\right) + O(n)$

Base case: $T(1) = O(1)$

Exercise

- How do we find a closed formula for $T(n) = T\left(\frac{n}{2}\right) + O(n)$?
- Break it down and see what we get!

Exercise

- How do we find a closed formula for $T(n) = T\left(\frac{n}{2}\right) + O(n)$?
- Break it down and see what we get!

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

Exercise

- How do we find a closed formula for $T(n) = T\left(\frac{n}{2}\right) + O(n)$?
- Break it down and see what we get!

$$T(n) = T\left(\frac{n}{2}\right) + n = T\left(\frac{n}{4}\right) + \frac{n}{2} + n$$

Exercise

- How do we find a closed formula for $T(n) = T\left(\frac{n}{2}\right) + O(n)$?
- Break it down and see what we get!

$$T(n) = T\left(\frac{n}{2}\right) + n = T\left(\frac{n}{4}\right) + \frac{n}{2} + n = T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n$$

Exercise

- How do we find a closed formula for $T(n) = T\left(\frac{n}{2}\right) + O(n)$?
- Break it down and see what we get!

$$T(n) = T\left(\frac{n}{2}\right) + n = T\left(\frac{n}{4}\right) + \frac{n}{2} + n = T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n$$

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

Exercise

- How do we find a closed formula for $T(n) = T\left(\frac{n}{2}\right) + O(n)$?
- Break it down and see what we get!

$$T(n) = T\left(\frac{n}{2}\right) + n = T\left(\frac{n}{4}\right) + \frac{n}{2} + n = T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n$$

$$= \sum_{i=0}^{\log(n)} \frac{n}{2^i} = \sum_{i=0}^{\log(n)-1} \left(\frac{n}{2^i}\right) + \frac{n}{2^{\log(n)}}$$

Exercise

- How do we find a closed formula for $T(n) = T\left(\frac{n}{2}\right) + O(n)$?
- Break it down and see what we get!

$$T(n) = T\left(\frac{n}{2}\right) + n = T\left(\frac{n}{4}\right) + \frac{n}{2} + n = T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n$$

$$= \sum_{i=0}^{\log(n)} \frac{n}{2^i} = \sum_{i=0}^{\log(n)-1} \left(\frac{n}{2^i}\right) + \frac{n}{2^{\log(n)}} = \sum_{i=0}^{\log(n)-1} \left(\frac{n}{2^i}\right) + 1$$

Exercise

- How do we find a closed formula for $T(n) = T\left(\frac{n}{2}\right) + O(n)$?
- Break it down and see what we get!

$$T(n) = T\left(\frac{n}{2}\right) + n = T\left(\frac{n}{4}\right) + \frac{n}{2} + n = T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n$$

$$= \sum_{i=0}^{\log(n)} \frac{n}{2^i} = \sum_{i=0}^{\log(n)-1} \left(\frac{n}{2^i}\right) + \frac{n}{2^{\log(n)}} = \sum_{i=0}^{\log(n)-1} \left(\frac{n}{2^i}\right) + 1 = n\frac{1 - \frac{1}{2^{\log(n)}}}{1 - \frac{1}{2}} + 1$$

Exercise

- How do we find a closed formula for $T(n) = T\left(\frac{n}{2}\right) + O(n)$?
- Break it down and see what we get!

$$T(n) = T\left(\frac{n}{2}\right) + n = T\left(\frac{n}{4}\right) + \frac{n}{2} + n = T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n$$

$$= \sum_{i=0}^{\log(n)} \frac{n}{2^i} = \sum_{i=0}^{\log(n)-1} \left(\frac{n}{2^i}\right) + \frac{n}{2^{\log(n)}} = \sum_{i=0}^{\log(n)-1} \left(\frac{n}{2^i}\right) + 1 = n\frac{1-\frac{1}{n}}{1-\frac{1}{2}} + 1$$

Exercise

- How do we find a closed formula for $T(n) = T\left(\frac{n}{2}\right) + O(n)$?
- Break it down and see what we get!

$$T(n) = T\left(\frac{n}{2}\right) + n = T\left(\frac{n}{4}\right) + \frac{n}{2} + n = T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n$$

$$= \sum_{i=0}^{\log(n)} \frac{n}{2^i} = \sum_{i=0}^{\log(n)-1} \left(\frac{n}{2^i}\right) + \frac{n}{2^{\log(n)}} = \sum_{i=0}^{\log(n)-1} \left(\frac{n}{2^i}\right) + 1 = n\frac{1-\frac{1}{n}}{1-\frac{1}{2}} + 1 = 2n - 1$$

Exercise

- How do we find a closed formula for $T(n) = T\left(\frac{n}{2}\right) + O(n)$?
- Break it down and see what we get!

$$T(n) = T\left(\frac{n}{2}\right) + n = T\left(\frac{n}{4}\right) + \frac{n}{2} + n = T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n$$

$$= \sum_{i=0}^{\log(n)} \frac{n}{2^i} = \sum_{i=0}^{\log(n)-1} \left(\frac{n}{2^i}\right) + \frac{n}{2^{\log(n)}} = \sum_{i=0}^{\log(n)-1} \left(\frac{n}{2^i}\right) + 1 = n\frac{1-\frac{1}{n}}{1-\frac{1}{2}} + 1 = 2n - 1$$

- Our closed formula guess will be $\mathrm{T(n)} = 2n - 1$

How to solve recurrence complexity?

- We are led to the educated guess that $\boldsymbol{T(n) = 2n - 1}$
- We prove this formula by induction on n:
- **Base:** For $\mathrm{n} = 1, T(1) = 1 = 2 - 1.$
- **Assumption**: Assume the formula is true for any $k < n$ and show for n:
- **Step:** $T(n) = T\left(\frac{n}{2}\right) + n = \left(\frac{2n}{2} - 1\right) + n = 2\mathrm{n} - 1\,.\,\blacksquare$

What about recursion and big-O?

Back to the previous example:

$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$
$$T(1) = O(1)$$

- Using the intuition from the previous case, we show $T(n) = \mathrm{O}(n)$.
- We will prove by induction, that there exists a constant $c > 0$, such that for any $n \geq 1$:

$$T(n) \leq cn$$

- How can we find that constant?

Big O recurrence

<u>Solution:</u>

- Note that the definition of $\mathrm{T}(n)$ hides two constants.

$$T(1) = c_1 \text{ and } T(n) \leq T\left(\frac{n}{2}\right) + c_2 n$$

Big O recurrence

Solution:

- Note that the definition of $\mathrm{T}(n)$ hides two constants.

$$T(1) = c_1 \text{ and } T(n) \leq T\left(\frac{n}{2}\right) + c_2 n$$

- To satisfy both $T(1)$ and $T(n)$, we'll use $\mathrm{c} = \max(c_1, 2c_2)$.

Big O recurrence

Solution:

- Note that the definition of $\mathrm{T}(n)$ hides two constants.

$$T(1) = c_1 \text{ and } T(n) \leq T\left(\frac{n}{2}\right) + c_2 n$$

- To satisfy both $T(1)$ and $T(n)$, we'll use $\mathrm{c} = \max(c_1, 2c_2)$.
- We now show $T(n) \leq cn$, for $n \geq 1$.

Big O recurrence

Solution:

- Note that the definition of $\mathrm{T}(n)$ hides two constants.

$$T(1) = c_1 \text{ and } T(n) \leq T\left(\frac{n}{2}\right) + c_2 n$$

- To satisfy both $T(1)$ and $T(n)$, we'll use $\mathrm{c} = \max(c_1, 2c_2)$.
- We now show $T(n) \leq cn$, for $n \geq 1$.
- **Basis**: $n = 1$**.** In this case, $T(1) = c_1 \leq \max(c_1, 2c_2) = c$.

Big O recurrence

Solution:

- Note that the definition of $\mathrm{T}(n)$ hides two constants.

$$T(1) = c_1 \text{ and } T(n) \leq T\left(\frac{n}{2}\right) + c_2 n$$

- To satisfy both $T(1)$ and $T(n)$, we'll use $\mathrm{c} = \max(c_1, 2c_2)$.
- We now show $T(n) \leq cn$, for $n \geq 1$.
- **Basis**: $n = 1$**.** In this case, $T(1) = c_1 \leq \max(c_1, 2c_2) = c$.
- **Assumption:** Assume that for $k < n$, $T(k) \leq ck$ and prove $T(n) \leq cn$.

Big O recurrence

Solution:

- Note that the definition of $\mathrm{T}(n)$ hides two constants.

$$T(1) = c_1 \text{ and } T(n) \leq T\left(\frac{n}{2}\right) + c_2 n$$

- To satisfy both $T(1)$ and $T(n)$, we'll use $\mathrm{c} = \max(c_1, 2c_2)$.
- We now show $T(n) \leq cn$, for $n \geq 1$.
- **Basis**: $n = 1$. In this case, $T(1) = c_1 \leq \max(c_1, 2c_2) = c$.
- **Assumption:** Assume that for $k < n$, $T(k) \leq ck$ and prove $T(n) \leq cn$.
- **Step**: Using the induction assumption:

$$T(n) \leq T\left(\frac{n}{2}\right) + c_2 n \leq_1 c\frac{n}{2} + c_2 n = \frac{cn + 2c_2 n}{2} \leq_2 \frac{cn + \max(c_1, 2c_2)\, n}{2} = \frac{cn + cn}{2} = cn$$

Trees – Run Time?

Describe an Algorithm that returns the sum of all values in the tree

GetTreeSum(Node):

1. If Node = null
    1. Return 0
2. MyValue ←Node.Value
3. LeftSum ← GetTreeSum(Node.Left)
4. RightSum ← GetTreeSum(Node.Right)
5. Return MyValue + LeftSum + RightSum

Trees – Run Time?

Describe an Algorithm that returns the sum of all values in the tree

GetTreeSum(Node):	$T(n)$
1. If Node = null	$O(1)$
1. Return 0	$O(1)$
2. MyValue ←Node.Value	$O(1)$
3. LeftSum ← GetTreeSum(Node.Left)	$T(n_L)$
4. RightSum ← GetTreeSum(Node.Right)	$T(n_R) = T(n - 1 - n_L)$
5. Return MyValue + LeftSum + RightSum	$O(1)$

Trees – Run Time?

Describe an Algorithm that returns the sum of all values in the tree

GetTreeSum(Node):	$T(n)$
1. If Node = null	$O(1)$
1. Return 0	$O(1)$
2. MyValue ←Node.Value	$O(1)$
3. LeftSum ← GetTreeSum(Node.Left)	$T(n_L)$
4. RightSum ← GetTreeSum(Node.Right)	$T(n_R) = T(n – 1 – n_L)$
5. Return MyValue + LeftSum + RightSum	$O(1)$

$$T(0) = O(1)$$

$$T(n) = O(1) + O(1) + T(n_L) + T(n - 1 - n_L) + O(1)$$

Trees – Run Time?

Describe an Algorithm that returns the sum of all values in the tree

GetTreeSum(Node):	$T(n)$
1. If Node = null	$O(1)$
1. Return 0	$O(1)$
2. MyValue ←Node.Value	$O(1)$
3. LeftSum ← GetTreeSum(Node.Left)	$T(n_L)$
4. RightSum ← GetTreeSum(Node.Right)	$T(n_R) = T(n - 1 - n_L)$
5. Return MyValue + LeftSum + RightSum	$O(1)$

$$T(0) = O(1)$$

$$T(n) = O(1) + O(1) + T(n_L) + T(n - 1 - n_L) + O(1) = O(1) + T(n_L) + T(n - 1 - n_L)$$

Trees – Run Time?

Describe an Algorithm that returns the sum of all values in the tree

GetTreeSum(Node):	$T(n)$
1. If Node = null	$O(1)$
1. Return 0	$O(1)$
2. MyValue ← Node.Value	$O(1)$
3. LeftSum ← GetTreeSum(Node.Left)	$T(n_L)$
4. RightSum ← GetTreeSum(Node.Right)	$T(n_R) = T(n - 1 - n_L)$
5. Return MyValue + LeftSum + RightSum	$O(1)$

$$T(0) = O(1)$$

$$T(n) = O(1) + O(1) + T(n_L) + T(n - 1 - n_L) + O(1) = O(1) + T(n_L) + T(n - 1 - n_L) = O(1) + O(1) + T(n_L^1) + T(n_L - 1 - n_L^1) + O(1) + T(n_L^2) + T(n - 1 - n_L - 1 - n_L^2)$$

Trees – Run Time?

Describe an Algorithm that returns the sum of all values in the tree

GetTreeSum(Node): $T(n)$
1. If Node = null $O(1)$
    1. Return 0 $O(1)$
2. MyValue ←Node.Value $O(1)$
3. LeftSum ← GetTreeSum(Node.Left) $T(n_L)$
4. RightSum ← GetTreeSum(Node.Right) $T(n_R) = T(n - 1 - n_L)$
5. Return MyValue + LeftSum + RightSum $O(1)$

$$T(0) = O(1)$$

$$T(n) = O(1) + O(1) + T(n_L) + T(n - 1 - n_L) + O(1) = O(1) + T(n_L) + T(n - 1 - n_L) = O(1) + O(1) + T(n_L^1) + T(n_L - 1 - n_L^1) + O(1) + T(n_L^2) + T(n - 1 - n_L - 1 - n_L^2) = n \cdot O(1)$$

Trees – Run Time?

Describe an Algorithm that returns the sum of all values in the tree

GetTreeSum(Node):	T(n)
1. If Node = null	O(1)
1. Return 0	O(1)
2. MyValue ←Node.Value	O(1)
3. LeftSum ← GetTreeSum(Node.Left)	$T(n_L)$
4. RightSum ← GetTreeSum(Node.Right)	$T(n_R) = T(n - 1 - n_L)$
5. Return MyValue + LeftSum + RightSum	O(1)

$$T(0) = O(1)$$

$$T(n) = O(1) + O(1) + T(n_L) + T(n - 1 - n_L) + O(1) = O(1) + T(n_L) + T(n - 1 - n_L) = O(1) + O(1) + T(n_L^1) + T(n_L - 1 - n_L^1) + O(1) + T(n_L^2) + T(n - 1 - n_L - 1 - n_L^2) = n \cdot O(1) = O(n)$$

Exercise 2

Exercise:

- Give an upper bound for the runtime of the following function.

```
define func(n: number)
    if (n is 1 or n is 2)
        return 2 * n
    if (n modulo 2 is 0)
        num = func(⌊n / 2⌋)
        return num * num
    else
        return func(n-1) * 2
```

```
define func(n: number)
  if (n is 1 or n is 2)
    return 2 * n
  if (n modulo 2 is 0)
    num = func(⌊n / 2⌋)
    return num * num
  else
    return func(n-1) * 2
```

Exercise 2 - Solution

- We begin by writing the recursive formula
- $T(n) = \begin{cases} T\left(\frac{n}{2}\right) + O(1), & \text{if } n \text{ is even} \\ T(n-1) + O(1), & \text{if } n \text{ is odd} \end{cases}$, $T(2) = O(1), T(1) = O(1)$
- By applying the formula twice for odd n we may rewrite it as
- $T(n) = \begin{cases} T\left(\frac{n}{2}\right) + O(1) & \text{if } n \text{ is even} \\ T\left(\frac{n-1}{2}\right) + O(1) & \text{if } n \text{ is odd} \end{cases}$ $T(3) = O(1), T(2) = O(1), T(1) = O(1)$
- $T(n)$ is an ugly formula. We have no way for finding an exact closed form.
- Instead we will prove by induction that $T(n) = \mathrm{O}(\log(n))$.
- We need to find constants $\mathrm{n}_0, c > 0$ such that $T(n) \leq c \log(n)$ whenever $n \geq n_0$.

```
define func(n: number)
   if (n is 1 or n is 2)
      return 2 * n
   if (n modulo 2 is 0)
      num = func(⌊n / 2⌋)
      return num * num
   else
      return func(n-1) * 2
```

Exercise 2 - Solution

- We show that $T(n) = O(\log(n))$.
- Since $T(1), T(2), T(3) = O(1)$ there exists $c > 0$ such that $\mathrm{T}(3), T(2) \leq c$ and
- $T(n) = \begin{cases} T\left(\frac{n}{2}\right) + O(1) & \text{if } n \text{ is even} \\ T\left(\frac{n-1}{2}\right) + O(1) & \text{if } n \text{ is odd} \end{cases} \rightarrow T(n) \leq \begin{cases} T\left(\frac{n}{2}\right) + c, & n \text{ } is \text{ } even \\ T\left(\frac{n-1}{2}\right) + c, & n \text{ } is \text{ } odd \end{cases}$
- c seems like a natural candidate for the constant we are looking for.
- We prove by induction $T(n) \leq c\log(n)$ for every $n \geq 2$.

Exercise 2 - Solution

```
define func(n: number)
   if (n is 1 or n is 2)
      return 2 * n
   if (n modulo 2 is 0)
      num = func(⌊n / 2⌋)
      return num * num
   else
      return func(n-1) * 2
```

- **Base:**

  For $n = 2$ –

  $T(2) \leq c = c * log_2(2)$ by definition

  For n=3 –

  $T(3) \leq c \leq c * log_2(3)$

- **Assumption:** we assume that for every $k < n, T(k) \leq \text{clog}(k)$ and prove for n.

Exercise 2 - Solution

```
define func(n: number)
   if (n is 1 or n is 2)
      return 2 * n
   if (n modulo 2 is 0)
      num = func(⌊n / 2⌋)
      return num * num
   else
      return func(n-1) * 2
```

- **Step:** We have two cases,

(1) When n is even then using the induction hypothesis,

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

$$= c\log(n) - c\log(2) + c\log(2) = c\log(n).$$

(2) When n is odd, then by the same arguments

- $T(n) \leq T\left(\frac{n-1}{2}\right) + c \leq c\log\left(\frac{n-1}{2}\right) + c = c\log\left(\frac{n-1}{2}\right) + c\log(2) =$

$$= c\log(n-1) \leq c\log(n).$$