public static void mystery(int n) {
    if (n <= 4) {
        System.out.println("Hello");
    } else {
        mystery(n - 1);
        for (int i = 0; i < n; i++)
            System.out.println("World");
        mystery(n - 2);
    }
}

With your neighbor, answer the following questions:

1. How much work is done JUST within the base case?
2. Within the recursive case, how much work do we do IGNORING the recursive calls?
3. How much work does each recursive call make, in terms of $T(...)$ and $n$?
public static void mystery(int n) {
    if (n <= 4) {
        System.out.println("Hello");
    } else {
        mystery(n - 1);
        for (int i = 0; i < n; i++)
            System.out.println("World");
        mystery(n - 2);
    }
}

Now, fill in the gaps to construct your recurrence:

\[
T(n) = \begin{cases} 
\text{workDoneInBaseCase} & \text{When } n \text{ is...} \\
\text{nonrecursiveWork} + \text{recursiveWork} & \text{Otherwise}
\end{cases}
\]
public static void mystery(int n) {
    if (n <= 4) {
        System.out.println("Hello");
    } else {
        mystery(n - 1);
        for (int i = 0; i < n; i++)
            System.out.println("World");
        mystery(n - 2);
    }
}

Now, fill in the gaps to construct your recurrence:

\[
T(n) = \begin{cases} 
\text{workDoneInBaseCase} & \text{When } n \text{ is...} \\
\text{nonrecursiveWork} + \text{recursiveWork} & \text{Otherwise}
\end{cases}
\]

Answer:

\[
T(n) = \begin{cases} 
1 & \text{When } n \leq 4 \\
 n + T(n - 1) + T(n - 2) & \text{Otherwise}
\end{cases}
\]
CSE 373 section AD has been changed to THO 125
▶ CSE 373 section AD has been changed to THO 125
▶ Project 2 due tonight
  PSA: After uploading to Canvas, double-check and make sure you’ve submitted the correct files.
Announcements

- CSE 373 section AD has been changed to THO 125
- Project 2 due tonight
  PSA: After uploading to Canvas, double-check and make sure you’ve submitted the correct files.
- Written homework 1 will be released tonight; due in a week. (Reminder: work on this solo)
Announcements

- CSE 373 section AD has been changed to THO 125
- Project 2 due tonight
  PSA: After uploading to Canvas, double-check and make sure you’ve submitted the correct files.
- Written homework 1 will be released tonight; due in a week. (Reminder: work on this solo)
- Everybody gets an extra late day.
Observation: sometimes, keys are comparable and sortable.

Idea: Can we exploit the “sortability” of these keys?
Observation: sometimes, keys are comparable and sortable.

Idea: Can we exploit the “sortability” of these keys?

Suppose we add the following invariant to ArrayDictionary:

```
SortedArrayDictionary invariant
The internal array, at all times, must remain sorted.
```

How do you implement get? What’s the big-$\Theta$ bound?
Core algorithm (in *pseudocode*):

```java
class ArraySearch {
    public V get(K key) {
        return search(key, 0, this.size);
    }

    private K search(K key, int lowIndex, int highIndex) {
        if (lowIndex > highIndex) {
            key not found, throw an exception
        } else {
            middleIndex = average of lowIndex and highIndex
            pair = this.array[middleIndex]

            if (pair.key == key) {
                return pair.value
            } else if (pair.key < key) {
                return search(key, lowIndex, middleIndex)
            } else if (pair.key > key) {
                return search(key, middleIndex + 1, highIndex)
            }
        }
    }
}
```

Let \( n = \text{highIndex} - \text{lowIndex} \). Let \( c \) be the time needed to perform the comparisons. Model the runtime as a recurrence.
The binary search algorithm

Core algorithm (in *pseudocode*):

```java
public V get(K key):
    return search(key, 0, this.size)

private K search(K key, int lowIndex, int highIndex):
    if lowIndex > highIndex:
        key not found, throw an exception
    else:
        middleIndex = average of lowIndex and highIndex
        pair = this.array[middleIndex]
        if pair.key == key:
            return pair.value
        else if pair.key < key:
            return search(key, lowIndex, middleIndex)
        else if pair.key > key:
            return search(key, middleIndex + 1, highIndex)
```

Answer: $T(n) \approx \begin{cases} 1 & \text{When } n \leq 0 \\ c + T(\lfloor \frac{n}{2} \rfloor) & \text{Otherwise} \end{cases}$
Finding a closed form

Our answer: \( T(n) \approx \begin{cases} 
1 & \text{When } n \leq 0 \\
c + T \left( \left\lfloor \frac{n}{2} \right\rfloor \right) & \text{Otherwise}
\end{cases} \)

Question: how do we find a closed form?
Finding a closed form

Our answer: \( T(n) \approx \begin{cases} 
1 & \text{When } n \leq 0 \\
c + T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) & \text{Otherwise}
\end{cases} \)

Question: how do we find a closed form? Try unfolding?
Finding a closed form

Our answer: $T(n) \approx \begin{cases} 1 & \text{When } n \leq 0 \\ c + T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) & \text{Otherwise} \end{cases}$

Question: how do we find a closed form? Try unfolding?

$$T(n) = c + (c + (c + \ldots + (c + 1))) = \underbrace{c + c + \ldots + c + 1}_{t=\text{Num times}}$$
Finding a closed form

Our answer: \( T(n) \approx \begin{cases} 1 & \text{When } n \leq 0 \\ c + T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) & \text{Otherwise} \end{cases} \)

Question: how do we find a closed form? Try unfolding?

\[
T(n) = c + (c + (c + \ldots + (c + 1))) = \underbrace{c + c + \ldots + c}_{t=\text{Num times}} + 1
\]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>16</th>
<th>\ldots</th>
<th>32</th>
<th>\ldots</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>\ldots</td>
<td></td>
<td>\ldots</td>
<td></td>
</tr>
</tbody>
</table>
Finding a closed form

Our answer: \( T(n) \approx \begin{cases} 1 & \text{When } n \leq 0 \\ c + T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) & \text{Otherwise} \end{cases} \)

Question: how do we find a closed form? Try unfolding?

\[
\begin{align*}
T(n) &= c + (c + (c + \ldots + (c + 1))) = c + c + \ldots + c + 1 \\
&= \underbrace{c + c + \ldots + c}_t + 1 \\
&= c + ct + 1
\end{align*}
\]

\[
\begin{array}{c|ccccccccccc}
 n & 0 & 1 & 2 & 4 & 6 & 8 & 10 & 12 & 16 & \ldots & 32 & \ldots & 64 \\
 t & 0 & 1 & 2 & 3 & 3 & 4 & 4 & 4 & 5 & \ldots & 6 & \ldots & 7 \\
\end{array}
\]
Finding a closed form

Our answer: \( T(n) \approx \begin{cases} 
1 & \text{When } n \leq 0 \\
\; c + T(\lfloor \frac{n}{2} \rfloor) & \text{Otherwise}
\end{cases} \)

Question: how do we find a closed form? Try unfolding?

\[
T(n) = c + (c + (c + \ldots + (c + 1))) = \underbrace{c + c + \ldots + c}_{t=\text{Num times}} + 1
\]

\begin{tabular}{c|cccccccccccc}
\( n \) & 0 & 1 & 2 & 4 & 6 & 8 & 10 & 12 & 16 & \ldots & 32 & \ldots & 64 \\
\hline
\( t \) & 0 & 1 & 2 & 3 & 3 & 4 & 4 & 4 & 5 & \ldots & 6 & \ldots & 7
\end{tabular}

What’s the relationship?
Finding a closed form

Our answer: \( T(n) \approx \begin{cases} 1 & \text{When } n \leq 0 \\ c + T \left( \left\lfloor \frac{n}{2} \right\rfloor \right) & \text{Otherwise} \end{cases} \)

Question: how do we find a closed form? Try unfolding?

\[
T(n) = c + (c + (c + \ldots + (c + 1))) = c + c + \ldots + c + 1
\]

\[
t = \text{Num times}
\]

| \( n \) | 0 | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | \ldots | 32 | \ldots | 64 |
|---------|---|---|---|---|---|---|-----|-----|-----|---------|-----|---------|
| \( t \) | 0 | 1 | 2 | 3 | 3 | 4 | 4 | 4 | 5 | \ldots | 6 | \ldots | 7 |

What’s the relationship? \( n \approx 2^{t+1} \)
Finding a closed form

Our answer: \( T(n) \approx \begin{cases} 
1 & \text{When } n \leq 0 \\
 c + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{Otherwise}
\end{cases} \)

Question: how do we find a closed form? Try unfolding?

\[
T(n) = c + (c + (c \ldots + (c + 1))) = c + c + \ldots + c + 1 \\
\text{t=Num times}
\]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0 1 2 4 6 8 10 12 16 ... 32 ... 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>0 1 2 3 3 4 4 4 5 ... 6 ... 7</td>
</tr>
</tbody>
</table>

What’s the relationship? \( n \approx 2^{t+1} \)

Solve for \( t \):
Finding a closed form

Our answer: \( T(n) \approx \begin{cases} 1 & \text{When } n \leq 0 \\ c + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{Otherwise} \end{cases} \)

Question: how do we find a closed form? Try unfolding?

\[
T(n) = c + (c + (c + \ldots + (c + 1))) = c + c + \ldots + c + 1
\]

\( t = \text{Num times} \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>16</th>
<th>\ldots</th>
<th>32</th>
<th>\ldots</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>\ldots</td>
<td>6</td>
<td>\ldots</td>
<td>7</td>
</tr>
</tbody>
</table>

What’s the relationship? \( n \approx 2^{t+1} \)

Solve for \( t \): \( t \approx \log_2(n) - 1 \)
Finding a closed form

Our answer: \( T(n) \approx \begin{cases} 1 & \text{When } n \leq 0 \\ c + T(\lfloor \frac{n}{2} \rfloor) & \text{Otherwise} \end{cases} \)

Question: how do we find a closed form? Try unfolding?

\[
T(n) = c + (c + (c + \ldots + (c + 1))) = c + c + \ldots + c + 1
\]

\( t = \text{Num times} \)

\[
\begin{array}{c|cccccccccccc}
 n & 0 & 1 & 2 & 4 & 6 & 8 & 10 & 12 & 16 & \ldots & 32 & \ldots & 64 \\
 \hline
 t & 0 & 1 & 2 & 3 & 3 & 4 & 4 & 4 & 5 & \ldots & 6 & \ldots & 7
\end{array}
\]

What’s the relationship? \( n \approx 2^{t+1} \)

Solve for \( t \): \( t \approx \log_2(n) - 1 \)

Final model: \( T(n) \approx c(\log_2(n) - 1) + 1 \)
Finding a closed form

Our answer: \( T(n) \approx \begin{cases} 1 & \text{When } n \leq 0 \\ c + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{Otherwise} \end{cases} \)

Question: how do we find a closed form? Try unfolding?

\[
T(n) = c + (c + (c + \ldots + (c + 1))) = c + c + \ldots + c + 1
\]

\[t=\text{Num times}\]

\[
\begin{array}{c|cccccccccc}
 n & 0 & 1 & 2 & 4 & 6 & 8 & 10 & 12 & 16 & \ldots & 32 & \ldots & 64 \\
\hline
t & 0 & 1 & 2 & 3 & 3 & 4 & 4 & 4 & 5 & \ldots & 6 & \ldots & 7
\end{array}
\]

What’s the relationship? \( n \approx 2^{t+1} \)

Solve for \( t \): \( t \approx \log_2(n) - 1 \)

Final model: \( T(n) \approx c(\log_2(n) - 1) + 1 \)
Finding a closed form

Our answer: \( T(n) \approx \begin{cases} 
1 & \text{When } n \leq 0 \\
0.5n + T\left( \left\lfloor \frac{n}{2} \right\rfloor \right) & \text{Otherwise}
\end{cases} \)

Question: how do we find a closed form? Try unfolding?

\[
T(n) = c + (c + (c + \ldots + (c + 1))) = c + c + \ldots + c + 1
\]

\[
t = \text{Num times}
\]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>16</th>
<th>...</th>
<th>32</th>
<th>...</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>...</td>
<td>6</td>
<td>...</td>
<td>7</td>
</tr>
</tbody>
</table>

What’s the relationship? \( n \approx 2^{t+1} \)

Solve for \( t \): \( t \approx \log_2(n) - 1 \)

Final model: \( T(n) \approx c(\log_2(n) - 1) + 1 \)

So, we conclude: \( T(n) \in \Theta (\log_2(n)) \)
The punchline:

Binary search takes about $\Theta(\log(n))$ time, where $n$ is the initial size of the array.

Note: in computer science, we assume $\log(n) = \log_2(n)$. 
Fill in the remainder of this table for SortedArrayDictionary:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description of algorithm</th>
<th>Big-Θ bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>Use binary search.</td>
<td>Θ (log(n))</td>
</tr>
<tr>
<td>put</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td></td>
</tr>
<tr>
<td>containsKey</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### SortedArrayDictionary

Fill in the remainder of this table for SortedArrayDictionary:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description of algorithm</th>
<th>Big-Θ bound</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>get</strong></td>
<td>Use binary search.</td>
<td>Θ (log(n))</td>
</tr>
<tr>
<td><strong>put</strong></td>
<td>Use binary search to find key. If it doesn't exist, insert into array.</td>
<td>Θ (n)</td>
</tr>
<tr>
<td><strong>remove</strong></td>
<td>Use binary search to find key. Once found, remove it and shift over remaining elements.</td>
<td>Θ (n)</td>
</tr>
<tr>
<td><strong>containsKey</strong></td>
<td>Use binary search.</td>
<td>Θ (log(n))</td>
</tr>
</tbody>
</table>
Idea: Moving away from lists

Observation: Changing our array is still difficult
Idea: Moving away from lists

Observation: Changing our array is still difficult

Idea: Use a different data structure optimized for both searching and insertion?
Idea: Moving away from lists

Observation: Changing our array is still difficult

Idea: Use a different data structure optimized for both searching and insertion?

Answer: Use a Binary Search Tree (BST)
A tree consists of nodes where each node has at most one parent and zero or more children. Every single node (except one) must have a parent.
Some definitions:

- **Root node:** The (single) node with no parent – the “top” of the tree
- **Branch node:** A node with one or more children
- **Leaf node:** A node with no children
- **Edge:** A pointer from one node to another
- **A subtree:** A node and all of its descendants
- **Height:** The number of edges contained in the longest path from the root node to some leaf node
Height of a tree

The **height** of a tree is the number of edges contained in the **longest** path from the root node to some leaf node.
Height of a tree

The **height** of a tree is the number of edges contained in the **longest** path from the root node to some leaf node.

What are the heights of these trees?
Example of a binary SEARCH tree (BST):

A binary SEARCH tree contains comparable items such that for every node, all children to the left have smaller keys and all children to the right have larger keys.
Important:

Binary Search Tree (BST) ≠ Binary Tree
Implementing the dictionary interface

Question: how do we implement the dictionary operations?

What are their runtimes with respect to $n$ (number of nodes in the tree) and/or $h$ (height of the tree)?
What is $h$, in terms of $n$?

For "balanced" trees, $h \approx \log_c(n)$, where $c$ is the maximum number of children a node can have.

So for "balanced" trees, our dictionary operations are all in $\Theta(\log(n))$. 
What is $h$, in terms of $n$?

For “balanced” trees, $h \approx \log_c(n)$, where $c$ is the maximum number of children a node can have.
What is $h$, in terms of $n$?

For “balanced” trees, $h \approx \log_c(n)$, where $c$ is the maximum number of children a node can have.

So for “balanced” trees, our dictionary operations are all in $\Theta(\log(n))$. 
Is this a valid binary tree? A valid binary search tree?

Yes. We call this a degenerate tree. What is $h$ now?

For "degenerate" trees, $h \approx n$. 

$18$
Is this a valid binary tree? A valid binary search tree?

Yes. We call this a **degenerate tree**. What is $h$ now?
Is this a valid binary tree? A valid binary search tree?

Yes. We call this a **degenerate tree**. What is $h$ now?

For “degenerate” trees, $h \approx n$. 

- 0
- 1
- 2
- 3
- 4
- 5

- 1
- 2
- 3
- 4
- 5

- 0
Fill in the remainder of this table for BinarySearchTreeDictionary:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description of algorithm</th>
<th>Big-$\Theta$ bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>Recursively traverse down left or right child until we find the correct node.</td>
<td>$\Theta(h)$</td>
</tr>
<tr>
<td>put</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td></td>
</tr>
<tr>
<td>containsKey</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fill in the remainder of this table for BinarySearchTreeDictionary:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description of algorithm</th>
<th>Big-$\Theta$ bound</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>get</strong></td>
<td>Recursively traverse down left or right child until we find the correct node.</td>
<td>$\Theta (h)$</td>
</tr>
<tr>
<td><strong>put</strong></td>
<td>Recursively search for node. If it doesn’t exist, keep recursing until we hit an empty spot and add a new node.</td>
<td>$\Theta (h)$</td>
</tr>
<tr>
<td><strong>remove</strong></td>
<td>Recursively find node to remove. Once found, replace it with the largest node in the left subtree (or the smallest node in the right subtree).</td>
<td>$\Theta (h)$</td>
</tr>
<tr>
<td><strong>containsKey</strong></td>
<td>Do a recursive search.</td>
<td>$\Theta (h)$</td>
</tr>
</tbody>
</table>
Core issue:

All BST operations take $O(h)$ time, where $h$ can be anywhere from $\log(n)$ to $n$, depending on the shape of the tree!
A question

Core issue:

All BST operations take $O(h)$ time, where $h$ can be anywhere from $\log(n)$ to $n$, depending on the shape of the tree!

Question:

Is there some way we can make $h$ always equal about $\log(n)$?

Can we somehow modify a BST so it always stays “balanced”?
**Core idea:** add extra invariant to BSTs that enforce balance.

### AVL Tree Invariants

An AVL tree has the following invariants:

- **The "structure" invariant:** All nodes have 0, 1, or 2 children.
- **The "BST" invariant:** For all nodes, all keys in the left subtree are smaller; all keys in the right subtree are larger.
- **The "balance" invariant:** For all nodes, $|\text{height}(\text{left}) - \text{height}(\text{right})| \leq 1$.

AVL = Adelson-Velsky and Landis
**Core idea:** add extra **invariant** to BSTs that enforce balance.

### AVL Tree Invariants

An AVL tree has the following invariants:

- **The “structure” invariant:**
  All nodes have 0, 1, or 2 children.

AVL = Adelson-Velsky and Landis
Core idea: add extra invariant to BSTs that enforce balance.

AVL Tree Invariants

An AVL tree has the following invariants:

- **The “structure” invariant:**
  All nodes have 0, 1, or 2 children.

- **The “BST” invariant:**
  For all nodes, all keys in the *left* subtree are smaller; all keys in the *right* subtree are larger

AVL = Adelson-Velsky and Landis
### AVL Tree Invariants

An AVL tree has the following invariants:

- **The “structure” invariant:**
  All nodes have 0, 1, or 2 children.

- **The “BST” invariant:**
  For all nodes, all keys in the left subtree are smaller; all keys in the right subtree are larger.

- **The “balance” invariant:**
  For all nodes, \( \text{abs}(\text{height(left)}) - \text{height(right))} \leq 1. \)

AVL = Adelson-Velsky and Landis
Question: why $\text{abs}(\text{height (left)}) - \text{height (height (right))} \leq 1$?

Why not $\text{height (left)} = \text{height (right)}$?
Interlude: Exploring the balance invariant

Question: why $\text{abs}(\text{height (left)}) - \text{height (height (right))} \leq 1$?

Why not $\text{height (left)} = \text{height (right)}$?

What happens if we insert two elements. What happens?
Question: is this a valid AVL tree?