CS 106X, Lecture 7
Introduction to Recursion

reading:
*Programming Abstractions in C++, Chapter 7*
Plan For Today

- **Recap:** Maps, Sets and Lexicons
- Thinking Recursively
- **Examples:** Factorial and Fibonacci
- Announcements
- **Coding Together:** Palindromes
- **Bonus:** Binary
Plan For Today

- **Recap:** Maps, Sets and Lexicons
- Thinking Recursively
- **Examples:** Factorial and Fibonacci
- Announcements
- **Coding Together:** Palindromes
- **Bonus:** Binary
• **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  – add, remove, search (contains)
  – We don't think of a set as having any indexes; we just add things to the set in general and don't worry about order

```
set.contains("to")  # true
set.contains("be")  # false
```
– **Set**: implemented using a linked structure called a *binary tree*.
  • pretty fast; elements are stored in **sorted order**
  • values must have a `<` operation

– **HashSet**: implemented using a special array called a *hash table*.
  • *very* fast; elements are stored in **unpredictable order**
  • values must have a `hashCode` function (*provided for most standard types*)
    – variant: `LinkedHashSet` (*slightly slower, but remembers insertion order*)

**How to choose**: Do you **need** the elements to be in sorted order?
  • If so: Use `Set`.  
  • If not: Use `HashSet` for the performance boost.
#include "set.h"
#include "hashset.h"

<table>
<thead>
<tr>
<th>Member</th>
<th>Set</th>
<th>HashSet</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.add(value);</td>
<td>O(log N)</td>
<td>O(1)</td>
<td>adds given value to set</td>
</tr>
<tr>
<td>s.clear();</td>
<td>O(N)</td>
<td>O(N)</td>
<td>removes all elements of set</td>
</tr>
<tr>
<td>s.contains(value)</td>
<td>O(log N)</td>
<td>O(1)</td>
<td>true if given value is found</td>
</tr>
<tr>
<td>s.isEmpty()</td>
<td>O(1)</td>
<td>O(1)</td>
<td>true if set contains no elements</td>
</tr>
<tr>
<td>s.isSubsetOf(set)</td>
<td>O(N log N)</td>
<td>O(N)</td>
<td>true if set contains all of this one</td>
</tr>
<tr>
<td>s.remove(value);</td>
<td>O(log N)</td>
<td>O(1)</td>
<td>removes given value from set</td>
</tr>
<tr>
<td>s.size()</td>
<td>O(1)</td>
<td>O(1)</td>
<td>number of elements in set</td>
</tr>
<tr>
<td>s.toString()</td>
<td>O(N)</td>
<td>O(N)</td>
<td>e.g &quot;{3, 42, -7, 15}&quot;</td>
</tr>
<tr>
<td>ostr &lt;&lt; s</td>
<td>O(N)</td>
<td>O(N)</td>
<td>print set to stream</td>
</tr>
</tbody>
</table>
## Set operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s1 == s2</code></td>
<td>true if the sets contain exactly the same elements</td>
</tr>
<tr>
<td><code>s1 != s2</code></td>
<td>true if the sets don't contain the same elements</td>
</tr>
<tr>
<td><code>s1 + s2</code></td>
<td>returns the <strong>union</strong> of <code>s1</code> and <code>s2</code> (elements from either)</td>
</tr>
<tr>
<td><code>s1 += s2;</code></td>
<td>sets <code>s1</code> to the union of <code>s1</code> and <code>s2</code> (or adds a value to <code>s1</code>)</td>
</tr>
<tr>
<td><code>s1 * s2</code></td>
<td>returns <strong>intersection</strong> of <code>s1</code> and <code>s2</code> (elements in both)</td>
</tr>
<tr>
<td><code>s1 *= s2;</code></td>
<td>sets <code>s1</code> to the intersection of <code>s1</code> and <code>s2</code></td>
</tr>
<tr>
<td><code>s1 - s2</code></td>
<td>returns <strong>difference</strong> of <code>s1</code>, <code>s2</code> (elements in <code>s1</code> but not <code>s2</code>)</td>
</tr>
<tr>
<td><code>s1 -= s2;</code></td>
<td>sets <code>s1</code> to the difference of <code>s1</code> and <code>s2</code> (or removes a value from <code>s1</code>)</td>
</tr>
</tbody>
</table>

```cpp
Set<string> set;
set += "Jess";
set += "Alex";
Set<string> set2 {"a", "b", "c"};  // initializer list
...```
Looping over a set

// forward iteration with for-each loop (read-only)
for (type name : collection) {
    statements;
}

- sets have no indexes; can't use normal for loop with index [i]
- Set iterates in sorted order; HashSet in unpredictable order

for (int i = 0; i < set.size(); i++) {
    do something with set[i];   // does not compile
}
#include "lexicon.h"

- A set of words optimized for dictionary and prefix lookups

<table>
<thead>
<tr>
<th>Member</th>
<th>Big-Oh</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexicon name; Lexicon name(&quot;file&quot;);</td>
<td>O(N*len)</td>
<td>create empty lexicon or read from file</td>
</tr>
<tr>
<td>L.add(word);</td>
<td>O(len)</td>
<td>adds the given word to lexicon</td>
</tr>
<tr>
<td>L.addWordsFromFile(&quot;f&quot;);</td>
<td>O(N*len)</td>
<td>adds all words from input file (one per line)</td>
</tr>
<tr>
<td>L.clear();</td>
<td>O(N*len)</td>
<td>removes all elements of lexicon</td>
</tr>
<tr>
<td>L.contains(&quot;word&quot;)</td>
<td>O(len)</td>
<td>true if word is found in lexicon</td>
</tr>
<tr>
<td>L.containsPrefix(&quot;str&quot;)</td>
<td>O(len)</td>
<td>true if s is the start of any word in lexicon</td>
</tr>
<tr>
<td>L.isEmpty()</td>
<td>O(1)</td>
<td>true if lexicon contains no words</td>
</tr>
<tr>
<td>L.remove(&quot;word&quot;);</td>
<td>O(len)</td>
<td>removes word from lexicon, if present</td>
</tr>
<tr>
<td>L.removePrefix(&quot;str&quot;);</td>
<td>O(len)</td>
<td>removes all words that start with prefix</td>
</tr>
<tr>
<td>L.size()</td>
<td>O(1)</td>
<td>number of elements in lexicon</td>
</tr>
<tr>
<td>L.toString()</td>
<td>O(N)</td>
<td>e.g. {&quot;arm&quot;, &quot;cot&quot;, &quot;zebra&quot;}</td>
</tr>
</tbody>
</table>
• **map**: A collection that stores **pairs**, where each pair consists of a first half called a *key* and a second half called a *value*.
  – sometimes called a "dictionary", "associative array", or "hash"
  – usage: add *(key, value)* pairs; look up a value by supplying a key.

• real-world examples:
  – dictionary of words and definitions
  – phone book
  – social buddy list

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Marty&quot;</td>
<td>&quot;685-2181&quot;</td>
</tr>
<tr>
<td>&quot;Eric&quot;</td>
<td>&quot;123-4567&quot;</td>
</tr>
<tr>
<td>&quot;Yana&quot;</td>
<td>&quot;685-2181&quot;</td>
</tr>
<tr>
<td>&quot;Alisha&quot;</td>
<td>&quot;947-2176&quot;</td>
</tr>
</tbody>
</table>
Map operations

• **m.put(key, value);** Adds a key/value pair to the map.

  m.put("Eric", "650-123-4567");  // or,
  m["Eric"] = "650-123-4567";
  • Replaces any previous value for that key.

• **m.get(key)** Returns the value paired with the given key.

  string phoneNum = m.get("Yana");  // "685-2181", or,
  string phoneNum = m["Yana"];  
  • Returns a default value (0, 0.0, "", etc.) if the key is not found.

• **m.remove(key);** Removes the given key and its paired value.

  m.remove("Marty");
  • Has no effect if the key is not in the map.

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Marty&quot; → &quot;685-2181&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Eric&quot; → &quot;123-4567&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Yana&quot; → &quot;685-2181&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Alisha&quot; → &quot;947-2176&quot;</td>
<td></td>
</tr>
</tbody>
</table>
in the Stanford C++ library, there are two map classes:

- **Map**: implemented using a linked structure called a *binary search tree*.
  - pretty fast for all operations; keys are stored in **sorted order**
  - both kinds of maps implement exactly the same operations
    - the keys' type must be a comparable type with a `<` operation

- **HashMap**: implemented using a special array called a *hash table*.
  - *very* fast, but keys are stored in unpredictable order
    - the keys' type must have a hashCode function (but most types have one)

Requires 2 type parameters: one for keys, one for values.

```cpp
// maps from string keys to integer values
Map<string, int> votes;
```
# Map members

<table>
<thead>
<tr>
<th>Member</th>
<th>Map</th>
<th>HashMap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m.clear();</code></td>
<td>O(N)</td>
<td>O(N)</td>
<td>removes all key/value pairs</td>
</tr>
<tr>
<td><code>m.containsKey(key)</code></td>
<td>O(log N)</td>
<td>O(1)</td>
<td>true if map has a pair with given key</td>
</tr>
<tr>
<td><code>m[key]</code> or <code>m.get(key)</code></td>
<td>O(log N)</td>
<td>O(1)</td>
<td>returns value mapped to given key; if not found, adds it with a default value</td>
</tr>
<tr>
<td><code>m.isEmpty()</code></td>
<td>O(1)</td>
<td>O(1)</td>
<td>true if the map contains no pairs</td>
</tr>
<tr>
<td><code>m.keys()</code></td>
<td>O(N)</td>
<td>O(N)</td>
<td>a Vector copy of all keys in map</td>
</tr>
<tr>
<td><code>m[key] = value;</code> or <code>m.put(key, value);</code></td>
<td>O(log N)</td>
<td>O(1)</td>
<td>adds a key/value pair; if key already exists, replaces its value</td>
</tr>
<tr>
<td><code>m.remove(key);</code></td>
<td>O(log N)</td>
<td>O(1)</td>
<td>removes any pair for given key</td>
</tr>
<tr>
<td><code>m.size()</code></td>
<td>O(1)</td>
<td>O(1)</td>
<td>returns number of pairs in map</td>
</tr>
<tr>
<td><code>m.toString()</code></td>
<td>O(N)</td>
<td>O(N)</td>
<td>e.g. &quot;{a:90, d:60, c:70}&quot;</td>
</tr>
<tr>
<td><code>m.values()</code></td>
<td>O(N)</td>
<td>O(N)</td>
<td>a Vector copy of all values in map</td>
</tr>
<tr>
<td><code>ostr &lt;&lt; m</code></td>
<td>O(N)</td>
<td>O(N)</td>
<td>prints map to stream</td>
</tr>
</tbody>
</table>
Looping over a map

• On a map, a for-each loop processes the keys.
  – Sorted order in a Map; unpredictable order in a HashMap.
  – If you want the values, just look up map[k] for each key k.

Map<string, double> gpa;
gpa.put("Victoria", 3.98);
gpa.put("Marty", 2.7);
gpa.put("BerkeleyStudent", 0.0);
...
for (string name : gpa) {
    cout << name << "'s GPA is " << gpa[name] << endl;
}
Plan For Today

- **Recap:** Maps, Sets and Lexicons
- **Thinking Recursively**
- **Examples:** Factorial and Fibonacci
- **Announcements**
- **Coding Together:** Palindromes
- **Bonus:** Binary
How many people are sitting in the column behind you?
How Many Behind Me?

1. If there is no one behind me, I will answer 0.
2. If there is someone behind me:
   – Ask them how many people are behind *them*
   – My answer is their answer plus 1

1. **Base case:** the simplest possible instance of this question. One that requires no additional recursion.

2. **Recursive case:** describe the problem using smaller occurrences of the same problem.
Recursive Thinking

• **In code**, recursion is when a function in your program calls itself as part of its execution.

• **Conceptually**, a recursive problem is one that is *self-similar*; it can be solved via smaller occurrences of the same problem.
Plan For Today

- Recap: Maps, Sets and Lexicons
- Thinking Recursively
- **Examples**: Factorial and Fibonacci
- Announcements
- Coding Together: Palindromes
- Bonus: Binary
The Recursion Checklist

☐ **Find what information we need to keep track of.**
  
  What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?

☐ **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?

☐ **Find our recursive step.** How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?

☐ **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?
Example 1: Factorial

\( n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 1 \)

- Write a function that computes and returns the factorial of a provided number, recursively (no loops).
  - e.g. `factorial(4)` should return 24
  - You should be able to compute the value of any non-negative number. (0! = 1).
The Recursion Checklist

- **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?

- **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?

- **Find our recursive step.** How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?

- **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?
Factorial: Function

n! = n * (n-1) * (n-2) * (n-3) * ... * 1

// Takes in n as parameter
int factorial(int n) {
    // returns factorial
    ...
}

// Function body
The Recursion Checklist

✓ Find what information we need to keep track of.
   What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?

☐ Find our base case(s). What are the simplest (non-recursive) instance(s) of this problem?

☐ Find our recursive step. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?

☐ Ensure every input is handled. Do we cover all possible cases? Do we need to handle errors?
Factorial: Base Case

\[ n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 1 \]

\[ 0! = 1 \]
The Recursion Checklist

✓ Find what information we need to keep track of. What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?

✓ Find our base case(s). What are the simplest (non-recursive) instance(s) of this problem?

✗ Find our recursive step. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?

✗ Ensure every input is handled. Do we cover all possible cases? Do we need to handle errors?
Factorial: Recursive Step

\[ n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 1 \]

We solve part of the problem.

We tackle a smaller instance of the factorial problem that leads us towards 0!. 
The Recursion Checklist

☑ Find what information we need to keep track of. What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?

☑ Find our base case(s). What are the simplest (non-recursive) instance(s) of this problem?

☑ Find our recursive step. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?

☑ Ensure every input is handled. Do we cover all possible cases? Do we need to handle errors?
Factorial: Input Check

1. If n is 0, the factorial is 1

2. If n is greater than 0:
   1. Calculate (n-1)!
   2. The factorial of n is that result times n
The Recursion Checklist

✓ Find what information we need to keep track of. What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?

✓ Find our base case(s). What are the simplest (non-recursive) instance(s) of this problem?

✓ Find our recursive step. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?

✓ Ensure every input is handled. Do we cover all possible cases? Do we need to handle errors?
// Returns n!, or 1 * 2 * 3 * 4 * ... * n.  
// Assumes n >= 0.  
int factorial(int n) {
    if (n == 0) {
        return 1;  // base case
    } else {
        return n * factorial(n - 1);  // recursive case
    }
}
int factorialFour = factorial(4); // 24

int factorial(int n) { // 4
    if (n == 0) { // base case
        return 1;
    } else { // recursive case
        return n * factorial(n - 1);
    }
}

int factorial(int n) { // 3
    if (n == 0) { // base case
        return 1;
    } else { // recursive case
        return n * factorial(n - 1);
    }
}

int factorial(int n) { // 2
    if (n == 0) { // base case
        return 1;
    } else { // recursive case
        return n * factorial(n - 1);
    }
}

int factorial(int n) { // 1
    if (n == 0) { // base case
        return 1;
    } else { // recursive case
        return n * factorial(n - 1);
    }
}

int factorial(int n) { // 0
    if (n == 0) { // base case
        return 1;
    } else { // recursive case
        return n * factorial(n - 1);
    }
}
recursiveFunc() {
    if (test for simple case) { // base case
        Compute the solution without recursion
    } else { // recursive case
        Break the problem into subproblems of the same form
        Call recursiveFunc() on each self-similar subproblem
        Reassemble the results of the subproblems
    }
}
Non-recursive factorial

// Returns n!, or 1 * 2 * 3 * 4 * ... * n.
// Assumes n >= 1.
int factorial(int n) {
    int total = 1;
    for (int i = 1; i <= n; i++) {
        total *= i;
    }
    return total;
}

• Important observations:

0! = 1! = 1

4! = 4 * 3 * 2 * 1

5! = 5 * 4 * 3 * 2 * 1
    = 5 * 4!
Example 2: Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... 

- The Fibonacci sequence starts with 0 and 1, and each subsequent number is the sum of the two previous numbers.
- Write a function that computes and returns the nth Fibonacci number, recursively (no loops).
  - e.g. \texttt{fibonacci(6)} should return 8
The Recursion Checklist

- **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?

- **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?

- **Find our recursive step.** How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?

- **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?
The Recursion Checklist

- **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?

- **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?

- **Find our recursive step.** How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?

- **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?
Fibonacci: Function

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

// Takes in index
int fibonacci(int i) {
    // returns i’th fibonacci number
    ...
}

✓ Find what information we need to keep track of. What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?

☐ Find our base case(s). What are the simplest (non-recursive) instance(s) of this problem?

☐ Find our recursive step. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?

☐ Ensure every input is handled. Do we cover all possible cases? Do we need to handle errors?
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

\[
\begin{align*}
\text{fibonacci}(0) &= 0; \\
\text{fibonacci}(1) &= 1;
\end{align*}
\]
The Recursion Checklist

- Find what information we need to keep track of. What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?

- Find our base case(s). What are the simplest (non-recursive) instance(s) of this problem?

- Find our recursive step. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?

- Ensure every input is handled. Do we cover all possible cases? Do we need to handle errors?
Fibonacci: Recursive Step

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

\[ \text{fibonacci}(x) = \text{fibonacci}(x-1) + \text{fibonacci}(x-2); \]

We tackle two smaller instances of the Fibonacci problem that lead us towards the first and second Fibonacci numbers.
The Recursion Checklist

✓ Find what information we need to keep track of. What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?

✓ Find our base case(s). What are the simplest (non-recursive) instance(s) of this problem?

✓ Find our recursive step. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?

☐ Ensure every input is handled. Do we cover all possible cases? Do we need to handle errors?
1. The 0\textsuperscript{th} Fibonacci number is 0
2. The 1\textsuperscript{st} Fibonacci number is 1
3. The 2\textsuperscript{nd}, 3\textsuperscript{rd}, etc. Fibonacci number is the sum of the previous two Fibonacci numbers
The Recursion Checklist

- **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?

- **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?

- **Find our recursive step.** How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?

- **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?
// Returns the i’th Fibonacci number in the sequence
// (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …)
// Assumes i >= 0.
int fibonacci(int i) {
    if (i == 0) {
        return 0; // base case 1
    } else if (i == 1) {
        return 1; // base case 2
    } else {
        // recursive case
        return fibonacci(i-1) + fibonacci(i-2);
    }
}
Recursive stack trace

int fourthFibonacci = fibonacci(3);

```c
int fibonacci(int i) { // i = 3
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i-1) + fibonacci(i-2);
    }
}
```
Recursive stack trace

```java
int fourthFibonacci = fibonacci(3);

int fibonacci(int i) {
    // i = 3
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i-1) + fibonacci(i-2);
    }
}
```
int fourthFibonacci = fibonacci(3);

```c
int fibonacci(int i) { // i = 3
    if (i == 0) {
        if (i == 0) {
            return 0;
        } else if (i == 1) {
            return 1;
        } else {
            return fibonacci(i-1) + fibonacci(i-2);
        }
    }
}
```
Recursive stack trace

```c
int fourthFibonacci = fibonacci(3);

int fibonacci(int i) { // i = 3
  if (i == 0) {
    return 0;
  } else if (i == 1) {
    return 1;
  } else {
    return fibonacci(i-1) + fibonacci(i-2);
  }
}
```

Recursive stack trace

```java
int fourthFibonacci = fibonacci(3);

int fibonacci(int i) {
    // i = 3
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i - 1) + fibonacci(i - 2);
    }
}

int fibonacci(int i) {
    // i = 2
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i - 1) + fibonacci(i - 2);
    }
}

int fibonacci(int i) {
    // i = 1
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i - 1) + fibonacci(i - 2);
    }
}
```
Recursive stack trace

```c
int fourthFibonacci = fibonacci(3);

int fibonacci(int i) { // i = 3
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i-1) + fibonacci(i-2);
    }
}

int fibonacci(int i) { // i = 2
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i-1) + fibonacci(i-2);
    }
}

1
```
int fourthFibonacci = fibonacci(3);

```c
int fibonacci(int i) { // i = 3
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i-1) + fibonacci(i-2);
    }
}

int fibonacci(int i) { // i = 2
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i-1) + fibonacci(i-2);
    }
}

int fibonacci(int i) { // i = 0
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i-1) + fibonacci(i-2);
    }
}
```

Recursive stack trace
Recursive stack trace

```c
int fourthFibonacci = fibonacci(3);

int fibonacci(int i) { // i = 3
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i-1) + fibonacci(i-2);
    }
}
```

1 0
int fourthFibonacci = fibonacci(3);

int fibonacci(int i) { // i = 3
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i-1) + fibonacci(i-2);
    }
}
Recursive stack trace

```java
int fibonacci(int i) {  // i = 3
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i-1) + fibonacci(i-2);
    }
}

int fourthFibonacci = fibonacci(3);
```

```java
int fibonacci(int i) {  // i = 1
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i-1) + fibonacci(i-2);
    }
}
```
Recursive stack trace

int fourthFibonacci = fibonacci(3); // 2

```c
int fibonacci(int i) { // i = 3
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i-1) + fibonacci(i-2);
    }
}
```
Recursive Tree

- fibonacci(4)
  - fibonacci(3)
    - fibonacci(2)
      - fibonacci(1)
    - fibonacci(1)
  - fibonacci(2)
    - fibonacci(1)
    - fibonacci(0)
  - fibonacci(0)
Recursive Tree

- Base case
- Recursive case
Preconditions

• **precondition**: Something your code *assumes is true* when called.
  – Often documented as a comment on the function's header:

    ```
    // Returns the ith Fibonacci number
    // Precondition: i >= 0
    int fibonacci(int i) {
    ```

  – Stating a precondition doesn't really "solve" the problem, but it at least documents our decision and warns the client what not to do.

  – What if the caller doesn't listen and passes a negative power anyway? What if we want to actually *enforce* the precondition?
throw expression;

- Generates an exception that will crash the program, unless it has code to handle ("catch") the exception.
- In Java, you can only throw objects that are Exceptions; in C++ you can throw any type of value (int, string, etc.)

- There is a class std::exception that you can use.
  - Stanford C++ lib's "error.h" also has an error(string) function.

• Why would anyone ever want a program to crash?
// Returns the ith Fibonacci number
// Precondition: i >= 0
int fibonacci(int i) {
    if (i < 0) {
        throw "illegal negative index";
    } else ... 
    ... 
}
Plan For Today

- **Recap**: Maps, Sets and Lexicons
- Thinking Recursively
- **Examples**: Factorial and Fibonacci
- **Announcements**
- **Coding Together**: Palindromes
- **Bonus**: Binary
Announcements

• Section swap/change deadline is **tomorrow (10/9) @ 5PM**
• Zach’s Office Hours Change (this week only): Thurs. 2:30-4:30PM
• Qt Creator Warnings (Piazza)
• VPTL Tutoring Resources (Piazza)
Plan For Today

- **Recap:** Maps, Sets and Lexicons
- Thinking Recursively
- **Examples:** Factorial and Fibonacci
- Announcements
- **Coding Together:** Palindromes
- **Bonus:** Binary
isPalindrome exercise

- Write a recursive function isPalindrome accepts a string and returns true if it reads the same forwards as backwards.

  - isPalindrome("madam") → true
  - isPalindrome("racecar") → true
  - isPalindrome("step on no pets") → true
  - isPalindrome("able was I ere I saw elba") → true
  - isPalindrome("Q") → true
  - isPalindrome("Java") → false
  - isPalindrome("rotater") → false
  - isPalindrome("byebye") → false
  - isPalindrome("notion") → false
The Recursion Checklist

- **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?

- **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?

- **Find our recursive step.** How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?

- **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?
// Returns true if the given string reads the same forwards as backwards.
// By default, true for empty or 1-letter strings.
bool isPalindrome(string s) {
    if (s.length() < 2) {  // base case
        return true;
    } else {  // recursive case
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        string middle = s.substr(1, s.length() - 2);
        return isPalindrome(middle);
    }
}
isPalindrome solution 2

// Returns true if the given string reads the same forwards as backwards.
// By default, true for empty or 1-letter strings.  
// This version is also case-insensitive.

bool isPalindrome(string s) {
    if (s.length() < 2) {
        // base case
        return true;
    } else {  
        // recursive case
        return tolower(s[0]) == tolower(s[s.length() - 1])
        && isPalindrome(s.substr(1, s.length() - 2));
    }
}
Plan For Today

• Recap: Maps, Sets and Lexicons
• Thinking Recursively
• Examples: Factorial and Fibonacci
• Announcements
• Coding Together: Palindromes
• Bonus: Binary

Next time: More recursion
Overflow Slides
Plan For Today

- **Recap**: Maps, Sets and Lexicons
- Thinking Recursively
- **Examples**: Factorial and Fibonacci
- Announcements
- **Coding Together**: Palindromes
- **Bonus**: Binary
• Write a recursive function `printBinary` that accepts an integer and prints that number's representation in binary (base 2).

  – Example: `printBinary(7)` prints `111`
  – Example: `printBinary(12)` prints `1100`
  – Example: `printBinary(42)` prints `101010`

<table>
<thead>
<tr>
<th>place</th>
<th>10</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>place</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

  – Write the function recursively and without using any loops.
Case analysis

- Recursion is about solving a small piece of a large problem.
  - What is 69743 in binary?
    - Do we know *anything* about its representation in binary?

- Case analysis:
  - What is/are easy numbers to print in binary?
  - Can we express a larger number in terms of a smaller number(s)?
Seeing the pattern

• Suppose we are examining some arbitrary integer N.
  – if N's binary representation is 10010101011
  – \((N / 2)\)'s binary representation is 1001010101
  – \((N \% 2)\)'s binary representation is 1

  – What can we infer from this relationship?
// Prints the given integer's binary representation.
// Precondition: n >= 0
void printBinary(int n) {
    if (n < 2) {
        // base case; same as base 10
        cout << n;
    } else {
        // recursive case; break number apart
        printBinary(n / 2);
        printBinary(n % 2);
    }
}

– Can we eliminate the precondition and deal with negatives?
// Prints the given integer's binary representation.
void printBinary(int n) {
    if (n < 0) {
        // recursive case for negative numbers
        cout << "-";
        printBinary(-n);
    } else if (n < 2) {
        // base case; same as base 10
        cout << n << endl;
    } else {
        // recursive case; break number apart
        printBinary(n / 2);
        printBinary(n % 2);
    }
}