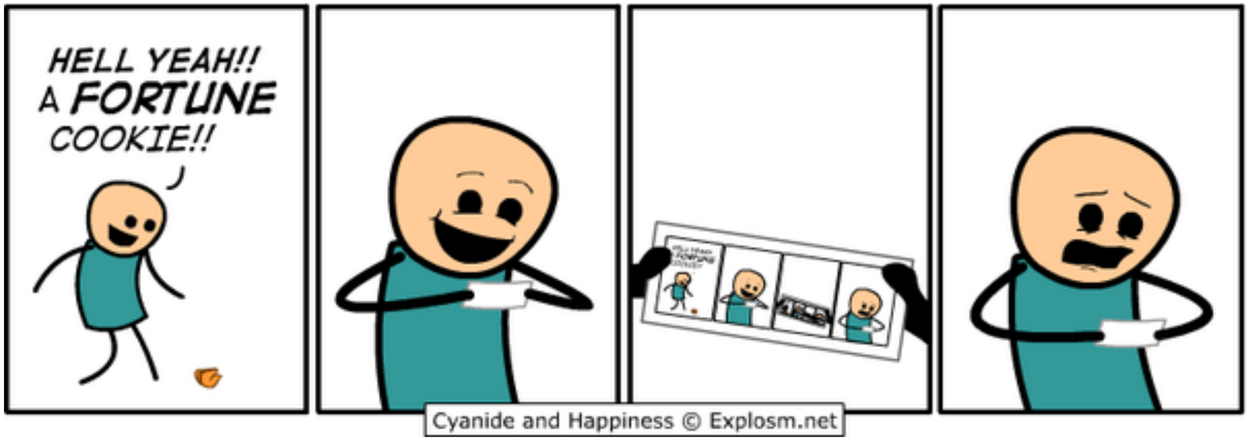
CS 106X, Lecture 7
Introduction to Recursion

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

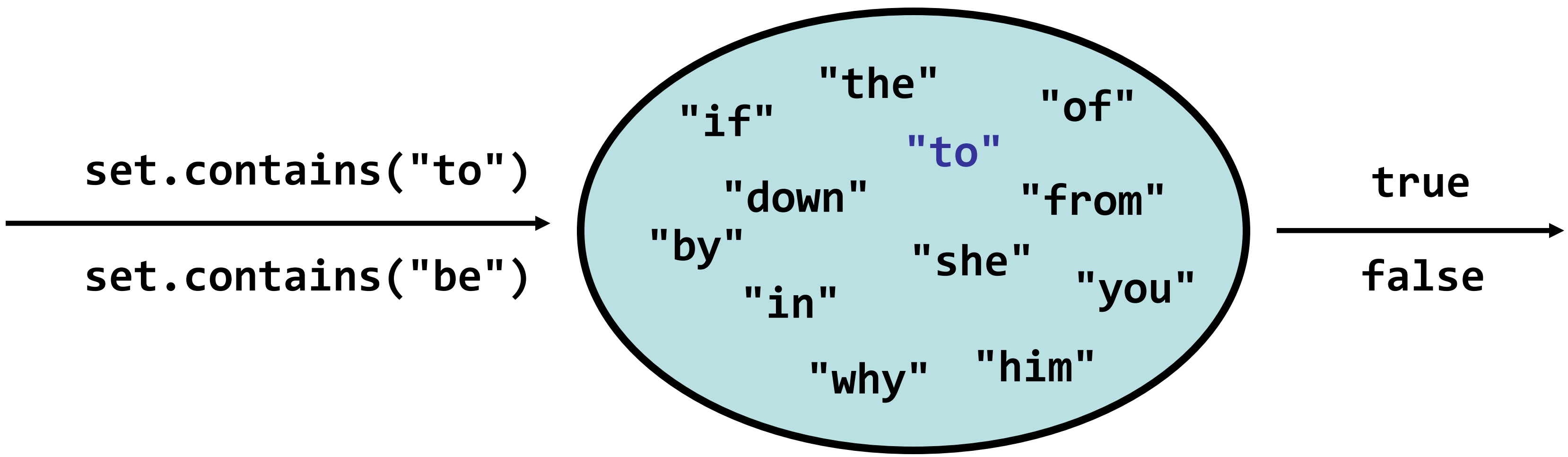
Plan For Today

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

Sets (5.5)

- **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

Stanford C++ sets (5.5)

- **`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 <u>need</u> the elements to be in sorted order?

- If so: Use `Set`.
- If not: Use `HashSet` for the performance boost.

Set members

```
#include "set.h"
#include "hashset.h"
```

Member	Set	HashSet	Description
`s.add(value);`	O(log *N*)	**O(1)**	adds given value to set
`s.clear();`	O(*N*)	O(*N*)	removes all elements of set
`s.contains(value)`	O(log *N*)	**O(1)**	`true` if given value is found
`s.isEmpty()`	O(1)	O(1)	`true` if set contains no elements
`s.isSubsetOf(set)`	O(*N* log *N*)	O(*N*)	`true` if *set* contains all of this one
`s.remove(value);`	O(log *N*)	**O(1)**	removes given value from set
`s.size()`	O(1)	O(1)	number of elements in set
`s.toString()`	O(*N*)	O(*N*)	e.g `"{3, 42, -7, 15}"`
`ostr << s`	O(*N*)	O(*N*)	print set to stream

Set operators

`s1 == s2`	`true` if the sets contain exactly the same elements
`s1 != s2`	`true` if the sets don't contain the same elements
`s1 + s2`	returns the **union** of s1 and s2 (elements from either)
`s1 += s2;`	sets s1 to the union of s1 and s2 (or adds a value to s1)
`s1 * s2`	returns **intersection** of s1 and s2 (elements in both)
`s1 *= s2;`	sets s1 to the intersection of s1 and s2
`s1 - s2`	returns **difference** of s1, s2 (elements in s1 but not s2)
`s1 -= s2;`	sets s1 to the difference of s1 and s2 (or removes a value from s1)

```
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
}
```

Stanford Lexicon (5.6)

```
#include "lexicon.h"
```

- A set of words optimized for dictionary and prefix lookups

Member	Big-Oh	Description
`Lexicon` ***`name`***`;` `Lexicon name("`***`file`***`");`	O(*N*len*)	create empty lexicon or read from file
`L``.add(`***`word`***`);`	O(*len*)	adds the given word to lexicon
`L``.addWordsFromFile("`***`f`***`");`	O(*N*len*)	adds all words from input file (one per line)
`L``.clear();`	O(*N*len*)	removes all elements of lexicon
`L``.contains("`***`word`***`")`	O(*len*)	`true` if word is found in lexicon
`L``.containsPrefix("`***`str`***`")`	O(*len*)	`true` if s is the start of any word in lexicon
`L``.isEmpty()`	O(1)	`true` if lexicon contains no words
`L``.remove("`***`word`***`");`	O(*len*)	removes word from lexicon, if present
`L``.removePrefix("`***`str`***`");`	O(*len*)	removes all words that start with prefix
`L``.size()`	O(1)	number of elements in lexicon
`L``.toString()`	O(*N*)	e.g. `{"arm", "cot", "zebra"}`

Maps (5.4)

- **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

key		value
"Marty"	→	"685-2181"
"Eric"	→	"123-4567"
"Yana"	→	"685-2181"
"Alisha"	→	"947-2176"

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.

<u>key</u>		<u>value</u>
"Marty"	→	"685-2181"
"Eric"	→	"123-4567"
"Yana"	→	"685-2181"
"Alisha"	→	"947-2176"

Map implementation

- 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.

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

Map members

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

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

Recursion

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 * (n-1) * (n-2) * (n-3) * … * 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
   ...
}
```

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 * (n-1) * (n-2) * (n-3) * … * 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 * (n-1) * (n-2) * (n-3) * … * 1`

`n! = n * (n-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?

Factorial

```
// Returns n!, or 1 * 2 * 3 * 4 * ... * n.
// Assumes n >= 0.
int factorial(int n) {
    if (n == 0) {                      // base case
        return 1;
    } else {
        return n * factorial(n - 1);   // recursive case
    }
}
```

Recursive stack trace

```
int factorialFour = factorial(4); // 24
```

```
int factorial(int n) { // 4
}
int factorial(int n) { // 3
}
int factorial(int n) { // 2
}
int factorial(int n) { // 1
}
int factorial(int n) { // 0
    if (n == 0) {                     // base case
        return 1;
    } else {
        return n * factorial(n - 1);  // recursive case
    }
}
```

Recursive Program Structure

```
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
    Reassamble 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! = <u>4 * 3 * 2 * 1</u>

  5! = 5 * <u>4 * 3 * 2 * 1</u>

  = 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. **`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
    ...
}
```

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: Base Case(s)

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

```
fibonacci(0) = 0;
fibonacci(1) = 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?

Fibonacci: Recursive Step

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

```
fibonacci(x) = fibonacci(x-1) + 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?

Fibonacci: Input Check

1. The 0^{th} Fibonacci number is 0
2. The 1^{st} Fibonacci number is 1
3. The 2^{nd}, 3^{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?

Fibonacci

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

Recursive stack trace

```
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

```
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

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

```
int fibonacci(int i) { // i = 3
}
```

```
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);
    }
}
```

Recursive stack trace

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

```
int fibonacci(int i) { // i = 3
}
```

```
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);
    }
}
```

Recursive stack trace

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

```
int fibonacci(int i) { // i = 3
}
```

```
int fibonacci(int i) { // 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

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

```
int fibonacci(int i) { // i = 3
}
```

```
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

Recursive stack trace

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

```
int fibonacci(int i) { // i = 3
}
```

```
int fibonacci(int i) { // 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

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

```
int fibonacci(int i) { // i = 3
}
```

```
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 0

Recursive stack trace

```
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

Recursive stack trace

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

```
int fibonacci(int i) { // i = 3
}
```

```
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);
    }
}
```

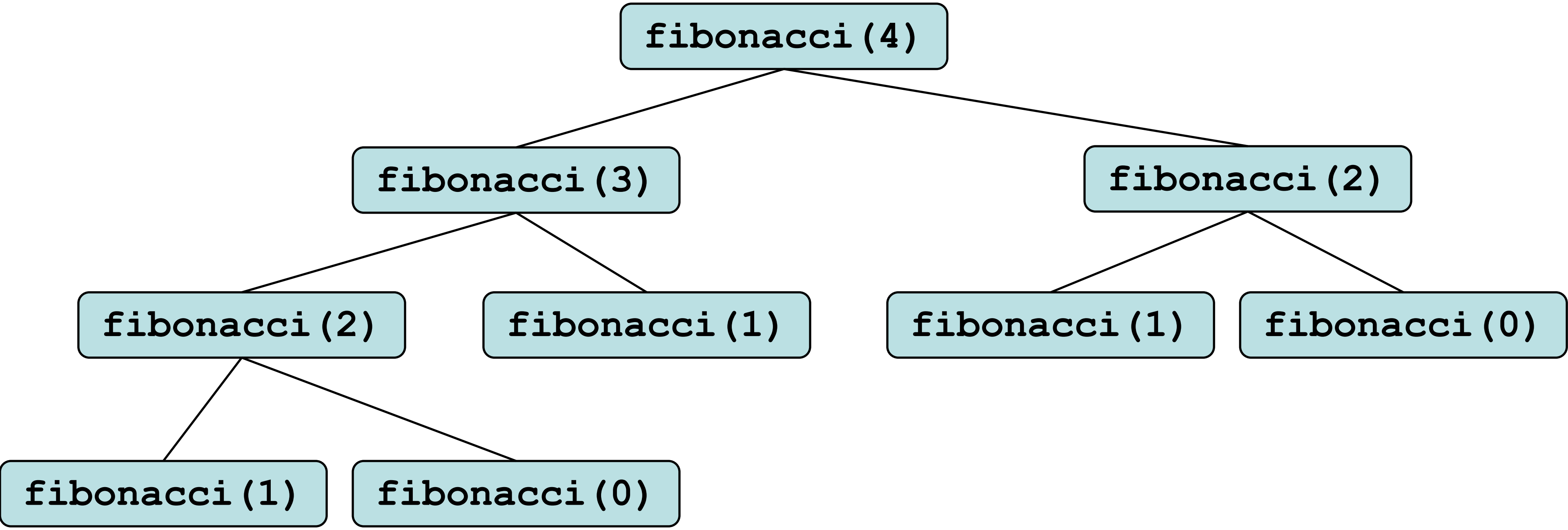
Recursive stack trace

```
int fourthFibonacci = fibonacci(3); // 2
```

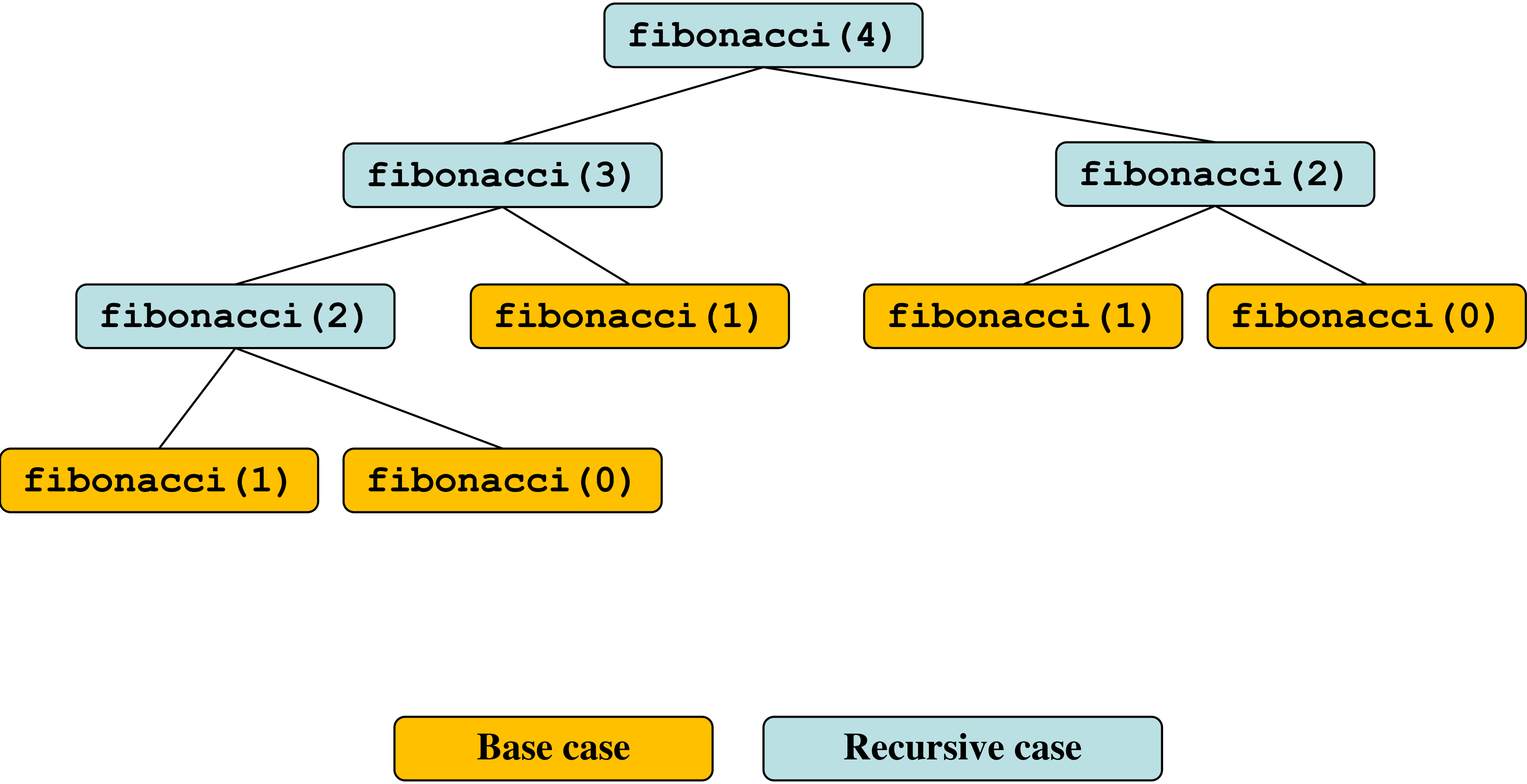
```
int fibonacci(int i) { // i = 3
    if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        return fibonacci(i-1) + fibonacci(i-2);
    }
}
```

1 1

Recursive Tree



Recursive Tree



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?

Throwing exceptions

`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 `Exception`s; 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?

Fibonacci Solution 2

```
// 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

isPalindrome

- 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?

isPalindrome solution

```
// 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

printBinary exercise printBinary

- 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`

place	10	1
value	**4**	**2**

32	16	8	4	2	1
1	**0**	**1**	**0**	**1**	**0**

  - 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?

printBinary solution

```
// 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?

printBinary solution 2

```
// 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);
    }
}
```