

CSCI 136:  
Data Structures  
and  
Advanced Programming

Lecture 7

Recursion, part 1

Instructor: Dan Barowy

**Williams**

## Announcements

- Lab 1 feedback coming soon.
- Lab 2 due today.
- Lab 3: quasi-random partners.

## Outline

1. Study tip
2. Quiz
3. Pre/post conditions
4. Recursion
5. Recursion activity
6. Recursion tradeoffs

Study tip #3: **growth mindset**

Have you ever thought:  
"I'm not good at [x]"



### Study tip #3: growth mindset

If you are motivated and study effectively, there is nothing you cannot learn.

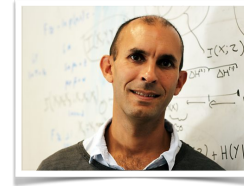
In fact, **you learn whether you want to or not.**

**Proof** (demo). Again. Ungarbled. One more time.

Notice that you can understand the garbled sound!

### Study tip #3: growth mindset

**Every brain** is an **amazing learning machine.**



**Anil Seth.**  
Professor of Cognitive and  
Computational Neuroscience,  
University of Sussex

Your brain is capable of rewiring itself in **milliseconds.**

Learning how to use your brain is a **skill** that requires **practice!**

Quiz

Pre/post conditions

Example

```
x + 1
```

What does this operation do?  
(i.e., what is our desired post-condition?)

Example

```
x + 1
```

Are you sure?

(code)

Example

These examples may seem  
contrived but trust me, they are not.

What **should have been true** about `x`?

1. `x` is an `int`
2. `x < Integer.MAX_VALUE`

## Pre-condition

A **pre-condition** is a **true/false statement** (a "predicate") that must always be true **prior to a code segment (e.g., a function) being called**. If a pre-condition is false, the result of executing the code is **undefined**.

## Post-condition

A **post-condition** is a **true/false statement** (a "predicate") that must always be true after a code segment (e.g., a function) is called **assuming that the pre-condition was true**.

## Post-condition implications

If a **pre-condition is false**, there is **no guarantee that the post-condition will be true**.

Conversely, if a **post-condition is false**, then if the pre-condition is valid, **the pre-condition must have been false**.

## Example, with conditions

```
public static int addOne(int x) {
    Assert.pre(
        x < Integer.MAX_VALUE,
        "x must be an integer less than MAX_VALUE.");
    int z = x + 1;
    Assert.post(z > x, "z must be greater than x.");
    return z;
}
```

## Other methods...

- Recall `charAt(int index)` in Java `String` class
- What are the pre-conditions for `charAt`?
  - `0 <= index < length()`
- What are the post-conditions?
  - Method returns char at position index in string
- It's a good idea to put pre- and post-conditions in comments before your methods

```
/* pre: 0 ≤ index < length
 * post: returns char at position index
 */
public char charAt(int index) { ... }
```

## Pre/post conditions

- Pre and post conditions **form a contract**.
- *Principle: post-condition is satisfied if pre-condition is satisfied.*
- Examples:
  - `s.charAt(s.length() - 1)`: index < length, so valid
  - `s.charAt(s.length() + 1)`: index > length, so not valid
- These conditions document requirements that user of method should satisfy.
- As comments, they are not enforced.

## Assert class

- Pre- and post-condition comments are useful as a programmer, but it would be really helpful to know as soon as a pre-condition is violated (and return an error)
- The **Assert** class (in `structure5` package) allows us to programmatically check for pre- and post-conditions

Remember: "Assume your code will fail."

## Assert class

The **Assert** class contains the **static** methods

```
public static void pre(boolean test, String message);
public static void post(boolean test, String message);
public static void condition(boolean test, String message);
public static void fail(String message);
```

If the boolean test is NOT satisfied, an exception is raised, the message is printed and the program halts.

## General guidelines

1. State pre/post conditions in comments
  2. Check conditions in code using **Assert**
  3. Use **Fail** in unexpected cases (such as the default block of a switch statement)
- Any questions?
  - You should use **Assert** in Lab 3

## Recursion



## Recursion

- General problem solving strategy
  - Split **big problem** into **smaller sub-problems**.
  - Sub-problems may look a lot like original; are often **smaller versions of same problem!**

## Recursion

**Recursion** is when a thing is **defined in terms of itself**. The most concrete application of recursion in computer science is **when a function is called within its own definition**.

```
public static int fibonacci(int n){
    if (n == 0){
        return 0;
    }
    if (n == 1){
        return 1;
    }

    return fibonacci(n - 1) +
           fibonacci(n - 2);
}
```

## Recursion

- **Many** algorithms are recursive
  - Often **easier to understand** (and prove correctness/state efficiency of) than non-recursive versions!

## Recursion

- $n! = n \times (n-1) \times (n-2) \times \dots \times 1$
- How can we implement this?
  - We could use a `for` loop...

```
int product = 1;
for(int i = 1; i <= n; i++)
    product *= i;
```
  - But we could also write it recursively...

## Activity: Factorial

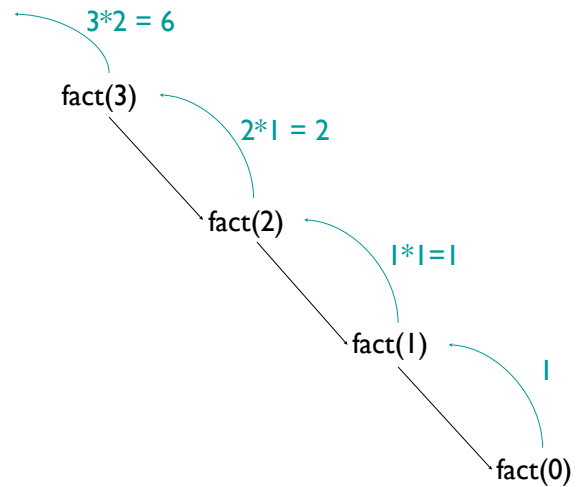
- $n! = n \times (n-1) \times (n-2) \times \dots \times 1$
- Work with a partner and see if you can come up with a recursive solution.

## Recursion: formal structure

- Recursion is a good solution when a problem fits a **basic pattern**:
- It has at least one "terminating" rule that **does not** use recursion, called the **base case**.
- It has at least one rule that **does** use recursion, called the **recursive case**. The recursive case should **reduce the problem toward the base case**.

We will talk about formal (i.e., "inductive") proofs for recursion this week.

Graphically...



Passing a note (recursively)



What are our base/recursive cases?

Recap & Next Class

Today we learned:

- Pre/post conditions
- Recursion
- Recursion activity
- Recursion tradeoffs

Next class:

- Lists