

# CSE 143

## Computer Programming II

## Recursive Programming

```
public static void solveProblem() {
    solveProblem();
}
```

### Outline

- 1 Writing Recursive Functions

### hasX

1

#### hasX

Implement a function

```
public static boolean hasX(String s)
```

which returns true if there is an 'x' in s, and false otherwise. You may not use contains or loops.

For example:

- hasX("hello") → false
- hasX("xylophone") → true
- hasX("abcdx") → true
- hasX("qrst") → false

### Solving Recursion Problems



#### Procedure For Solving Recursion Problems

- 1 Figure out what the base case is. To do this, look at the type of the argument and think of the simplest thing of that type.
- 2 Now that you know the base case(s), figure out what the answer for the base case is.
- 3 Now, we have to figure out the recursive step. First, write down what the function we are writing does.
- 4 Then, ask the question: "What is the smallest piece of the problem I can break off?"
- 5 Figure out the answer to the question for the tiny problem.
- 6 Use a recursive call to solve whatever is left.

### Solving hasX

3

- 1 Figure out what the base case is. To do this, look at the type of the argument and think of the simplest thing of that type:  
**The argument is a String. The simplest String is the empty String.**

- 2 Now that you know the base case(s), figure out what the answer for the base case is:

```
1 if (s.length() == 0) {
2     // The empty string doesn't have any x's
3     return false;
4 }
```

- 3 Now, we have to figure out the recursive step. First, write down what the function we are writing does:

**hasX(s) returns true when s contains an 'x'**

## Solving hasX (continued)

4

- 5 Ask the question: "What is the smallest piece of the problem I can break off?"

**A String is made up of chars. We can break off a single char:**

```
1 else {
2   // Break off the first character
3   char c = s.charAt(0);
4   ...
```

- 6 Figure out the answer to the question for the tiny problem:

```
1   // Answer the question for that character
2   if (c == 'x') {
3     return true;
4   }
5   ...
```

- 7 Use a recursive call to solve whatever is left.

```
1   // Ask someone else to solve the rest of the problem
2   return hasX(s.substring(1));
3 }
```

## Putting hasX Together

5

### hasX Solution

```
1 public static boolean hasX(String s) {
2   if (s.length() == 0) {
3     // The empty string doesn't have any x's
4     return false;
5   }
6   else {
7     // Break off the first character
8     char c = s.charAt(0);
9
10    // Answer the question for that character
11    return c == 'x' || hasX(s.substring(1));
12  }
13 }
```

## hasX

6

### hasX

Implement a function

```
public static boolean hasX(String s)
```

which returns true if there is an 'x' in s, and false otherwise. You may not use contains or loops.

For example:

- hasX("hello") → false
- hasX("xylophone") → true
- hasX("abcdx") → true
- hasX("qrst") → false

## Implementing the "Paint Bucket" Tool

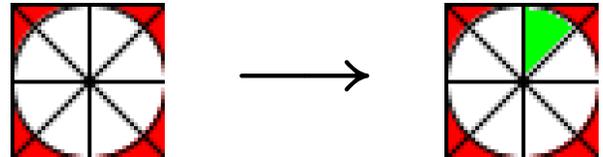
7

### paintbucket

Implement a function

```
public static void paintbucket(int x, int y, Color empty)
```

which fills an area with Color.GREEN when the DrawingPanel is clicked.



## Floodfill

8

The algorithm to fill in the space is called "floodfill." To see it working with DrawingPanel, download the code. Here it is more generally:

### floodfill

```
1 public static final ThingToFill drawing;
2 public static final int SIZE = ...;
3 public static void floodfill(int x, int y, Color empty) {
4   if (x >= 0 && y >= 0 && x < SIZE && y < SIZE &&
5     drawing.getColor(x, y).equals(empty)) {
6     drawing.setColor(x, y, Color.GREEN);
7     floodfill(x - 1, y, empty);
8     floodfill(x, y - 1, empty);
9     floodfill(x + 1, y, empty);
10    floodfill(x, y + 1, empty);
11  }
12 }
```

## Checking For Palindromes

9

### isPalindrome

Implement a function

```
public static boolean isPalindrome(int[] arr, int begin, int end)
```

which returns true if the elements between begin and end (inclusive) of arr are the same forwards and backwards.

For example:

- isPalindrome({1, 1, 1, 1}, 0, 2) → true
- isPalindrome({1, 1, 1, 1}, 0, 3) → true
- isPalindrome({1, 1, 1, 1}, 1, 2) → true
- isPalindrome({1, 2, 3, 4}, 1, 1) → true
- isPalindrome({1, 2, 3, 4}, 1, 2) → false
- isPalindrome({1, 2, 3, 2, 1}, 1, 3) → true
- isPalindrome({1, 2, 3, 2, 1}, 0, 1) → false

## Checking For Palindromes Solution

10

```
isPalindrome
1 public static boolean isPalindrome(int[] arr, int begin, int end) {
2     if (begin >= end) {
3         return true;
4     }
5     else {
6         return arr[begin] == arr[end] &&
7             isPalindrome(arr, begin + 1, end - 1);
8     }
9 }
```

Okay, but wait. Are we ever going to ask for partial arrays being palindromes? We'd rather a function:

```
public static boolean isPalindrome(int[] arr)
```

How can we write this using our previous function?

```
1 public static boolean isPalindrome(int[] arr) {
2     return isPalindrome(arr, 0, arr.length - 1);
3 }
```

Note that, now, we should make our original function **PRIVATE**, because we don't want a user to ever actually call it.

## Public-Private Recursive Pairs

11

In general, to solve some recursive problems, we will need to make **public-private** pairs. The private method will have extra arguments we can use and the public method will call the private one.

```
1 public static boolean isPalindrome(int[] arr) {
2     return isPalindrome(0, arr.length - 1);
3 }
4
5 private static boolean isPalindrome(int[] arr, int begin, int end) {
6     if (begin >= end) {
7         return true;
8     }
9     else {
10        return arr[begin] == arr[end] &&
11            isPalindrome(arr, begin + 1, end - 1);
12    }
13 }
```

## Crawling The Filesystem

12

crawl

Write a method

```
public static void crawl(File f)
```

that prints out the names of the files we reach by looking inside any folders starting at *f*. The names should be indented as many times as the number of folders it is inside.

So, for instance, an output might look like:

```
OUTPUT
>> folder1
>>     file.txt
>>     IAmInsideFolder1
>>         insideinside.html
>>         lecture.pdf
>>     oops.jpg
```

## Crawling The Filesystem: public/private pair again!

13

Because we need to keep track of how far we are supposed to indent, our recursive function will need to have a second argument *indent*:

```
public static void crawl(File f, String indent)
```

crawl Solution

```
1 public static void crawl(File f) {
2     return crawl(f, "");
3 }
4 private static void crawl(File f, String indent) {
5     System.out.println(indent + f.getName());
6     if (f.isDirectory()) {
7         List<File> filesInDir = f.listFiles();
8         for (int i = 0; i < filesInDir.size(); i++) {
9             crawl(filesInDir.get(i), indent + " ");
10        }
11    }
12 }
```

## Some Recursion Tips!

14

- See earlier starred slide with approach to recursion problems.
- Practice writing recursive functions **a lot**. Looking at an answer does not count as practicing.
- Always identify **how many** base cases (e.g. the special, weird ones) you will need.

Non-negative numbers	→	0
int	→	negative, 0
String	→	""
File System	→	non-folder

- If the problem doesn't seem like you can break it down easily, think about what arguments you could add to help (and use a public-private pair). Generally, you need arguments to **"keep track of"** something.
  - Saving the start and end bounds of an int []
  - Saving the number of times we've recursed into a folder (to print them indented)