Lab 06: Recursion

CS 0445: Data Structures

TAs: Jon Rutkauskas
Brian Nixon

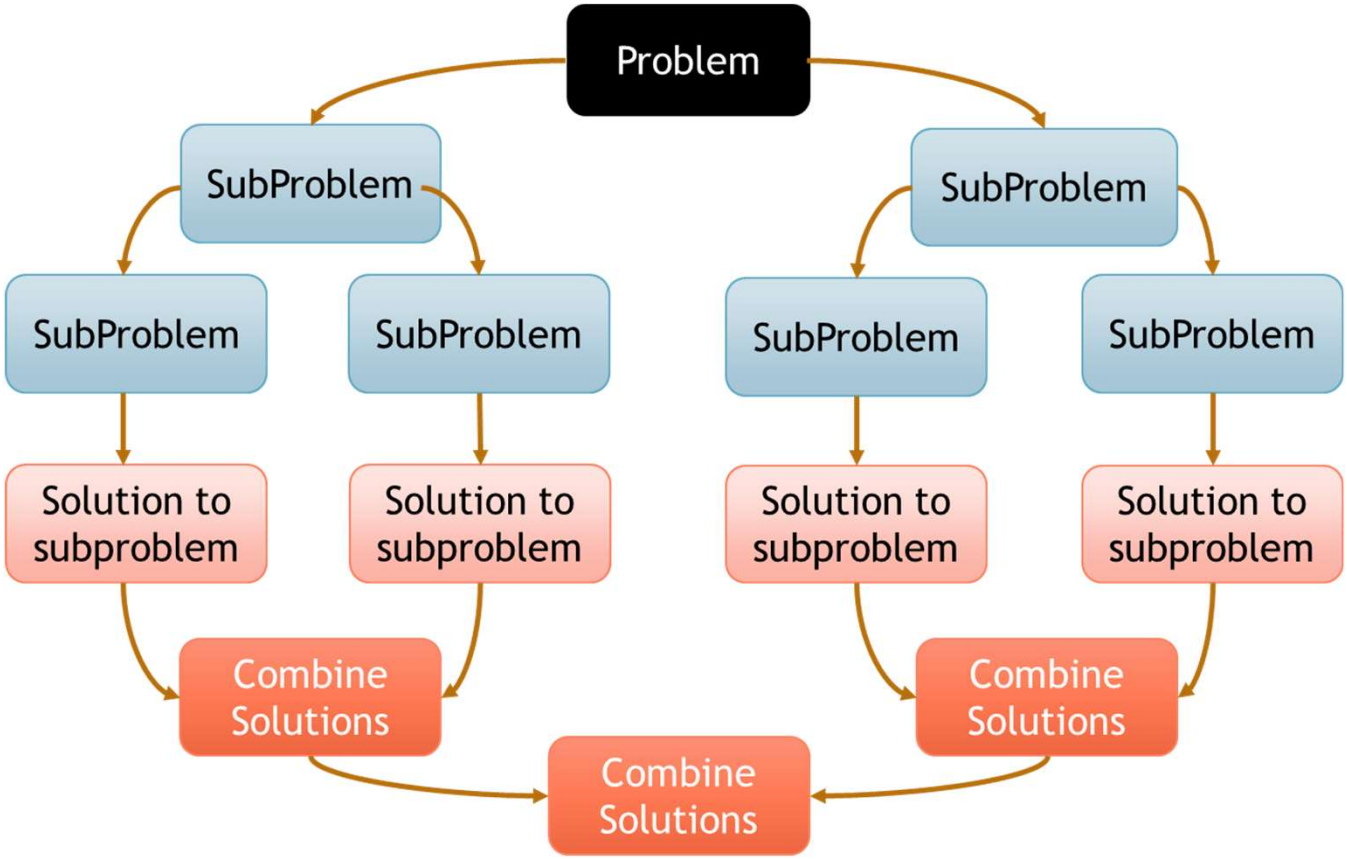
http://db.cs.pitt.edu/courses/cs0445/current.term/

October 21, 2019
University of Pittsburgh, Pittsburgh, PA

Idea of Recursion

- A method defined in terms of itself
- Solves a problem by solving smaller sub-problems and using those solutions to solve the larger problem

Recursive Solutions



Requirements

- Recursive Case
  - Method calls itself by passing a smaller sub-problem as an argument
- Base Case
  - Method solves the problem directly and does not call itself again
- Termination
  - The sequence of recursive calls and base cases eventually leads to the method's termination

Steps to Building a Recursive Algorithm

- Identify the problem
  - What are the name and arguments of the problem to be solved?
- Identify the smaller problems
  - What are the smaller problems that will be used to solve the original one?
- Identify how the answers are composed
  - How do the smaller answers combine to solve the larger problem?
- Identify the base cases
  - What are the smallest problems that must be solved directly?
- Compose the recursive definition
  - Combine all of the parts into one definition

Idea of Recursion

- A method defined in terms of itself
- Solves a problem by solving smaller sub-problems and using those solutions to solve the larger problem
- Example: Factorial
  - n! = n * (n – 1)!
  - 6! = 6 * (5)!

Idea of Recursion

- A method defined in terms of itself
- Solves a problem by solving smaller sub-problems and using those solutions to solve the larger problem
- Example: Factorial
  - n! = n * (n – 1)!
  - 6! = 6 * (5)!
  - 5! = 5 * (4)!

Idea of Recursion

- A method defined in terms of itself
- Solves a problem by solving smaller sub-problems and using those solutions to solve the larger problem
- Example: Factorial
  - n! = n * (n – 1)!
  - 6! = 6 * (5)!
  - 5! = 5 * (4)!
  - 4! = 4 * (3)!

Idea of Recursion

- A method defined in terms of itself
- Solves a problem by solving smaller sub-problems and using those solutions to solve the larger problem
- Example: Factorial
  - n! = n * (n – 1)!
  - 6! = 6 * (5)!
  - 5! = 5 * (4)!
  - 4! = 4 * (3)!
  - 3! = 3 * (2)!

Idea of Recursion

- A method defined in terms of itself
- Solves a problem by solving smaller sub-problems and using those solutions to solve the larger problem
- Example: Factorial
  - n! = n * (n – 1)!
  - 6! = 6 * (5)!
  - 5! = 5 * (4)!
  - 4! = 4 * (3)!
  - 3! = 3 * (2)!
  - 2! = 2 * (1)!
  - 1! = 1

Idea of Recursion

- A method defined in terms of itself
- Solves a problem by solving smaller sub-problems and using those solutions to solve the larger problem
- Example: Factorial
  - n! = n * (n – 1)!
  - 6! = 6 * (5)!
  - 5! = 5 * (4)!
  - 4! = 4 * (3)!
  - 3! = 3 * (2)!
  - 2! = 2 * (1)!
  - 1! = 1 ➔ 1

Idea of Recursion

- A method defined in terms of itself
- Solves a problem by solving smaller sub-problems and using those solutions to solve the larger problem
- Example: Factorial
  - n! = n * (n – 1)!
  - 6! = 6 * (5)!
  - 5! = 5 * (4)!
  - 4! = 4 * (3)!
  - 3! = 3 * (2)!
  - 2! = 2 * (1)! ➔ 2 * (1) = 2
  - 1! = 1 ➔ 1

Idea of Recursion

- A method defined in terms of itself
- Solves a problem by solving smaller sub-problems and using those solutions to solve the larger problem
- Example: Factorial
  - n! = n * (n – 1)!
  - 6! = 6 * (5)! ➔ 6 * (120) = 720
  - 5! = 5 * (4)! ➔ 5 * (24) = 120
  - 4! = 4 * (3)! ➔ 4 * (6) = 24
  - 3! = 3 * (2)! ➔ 3 * (2) = 6
  - 2! = 2 * (1)! ➔ 2 * (1) = 2
  - 1! = 1 ➔ 1

Sum of Array Contents

```
public static int sumOfArray(int[] theArray) {
    return sumOfArray(theArray, theArray.length);
}

private static int sumOfArray(int[] theArray, int howMany) {
    int sum = 0;
    int lastIndex = howMany - 1;
    if (howMany > 0) {
        sum += sumOfArray(theArray, howMany - 1);
        sum += theArray[lastIndex];
    }
    return sum;
    }
```

- Getting the sum of all contents in an array recursively.
- Note the call to sumOfArray() inside of the method sumOfArray()
- Method will halt until recursive call to the same method returns.
- The method will then complete the remaining code.

Sum of Array Contents

Why write a helper method and not just use a single line such as:

sumOfArray(Arrays.copyOf(theArray, theArray.length – 1))

```
private static int sumOfArray(int[] theArray, int howMany) {
    int sum = 0;
    int lastIndex = howMany - 1;
    if (howMany > 0) {
        sum += sumOfArray(theArray, howMany - 1);
        sum += theArray[lastIndex];
    }
    return sum;
    }
```

Sum of Array Contents

```
public static int sumOfArray(int[] theArray) {
    return sumOfArray(theArray, theArray.length);
}

private static int sumOfArray(int[] theArray, int howMany) {
    int sum = 0;
    int lastIndex = howMany - 1;
    if (howMany > 0) {
        sum += sumOfArray(theArray, howMany - 1);
        sum += theArray[lastIndex];
    }
    return sum;
    }
```

- By using a single statement, the array will be copied on each call to recursive cases.
- This will not only increase the runtime but also waste memory to create a new array on each call.
- With helper method, we can pass the number of elements in an array and pass the reference variable to the array rather than a new copy.
- This will have lower runtime and prevent unnecessary copies of the array

Sum of Array Contents

```
private static int sumOfArray(int[] theArray, int howMany) {
    int sum = 0;
    int lastIndex = howMany - 1;
    if (howMany > 0) {
        sum += sumOfArray(theArray, howMany - 1);
        sum += theArray[lastIndex];
    }
    return sum;
    }
```

Howmany = 4

lastIndex = 3

Howmany > 0

Focus on the sum of the smaller subArray

Sum of Array Contents

```
private static int sumOfArray(int[] theArray, int howMany) {
    int sum = 0;
    int lastIndex = howMany - 1;
    if (howMany > 0) {
        sum += sumOfArray(theArray, howMany - 1);
        sum += theArray[lastIndex];
    }
    return sum;
    }
```

Howmany = 3

lastIndex = 2

Howmany > 0

Focus on the sum of the smaller subArray

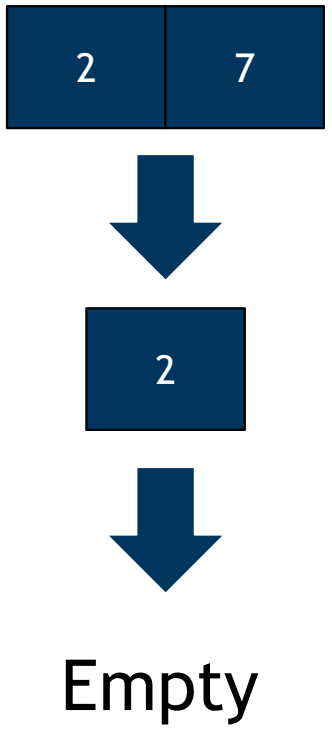
Sum of Array Contents

```
private static int sumOfArray(int[] theArray, int howMany) {
    int sum = 0;
    int lastIndex = howMany - 1;
    if (howMany > 0) {
        sum += sumOfArray(theArray, howMany - 1);
        sum += theArray[lastIndex];
    }
    return sum;
    }
```

Howmany = 2
lastIndex = 1
Howmany > 0

Howmany = 1
lastIndex = 0
Howmany > 0

Howmany = 0
lastIndex = -1
Howmany == 0

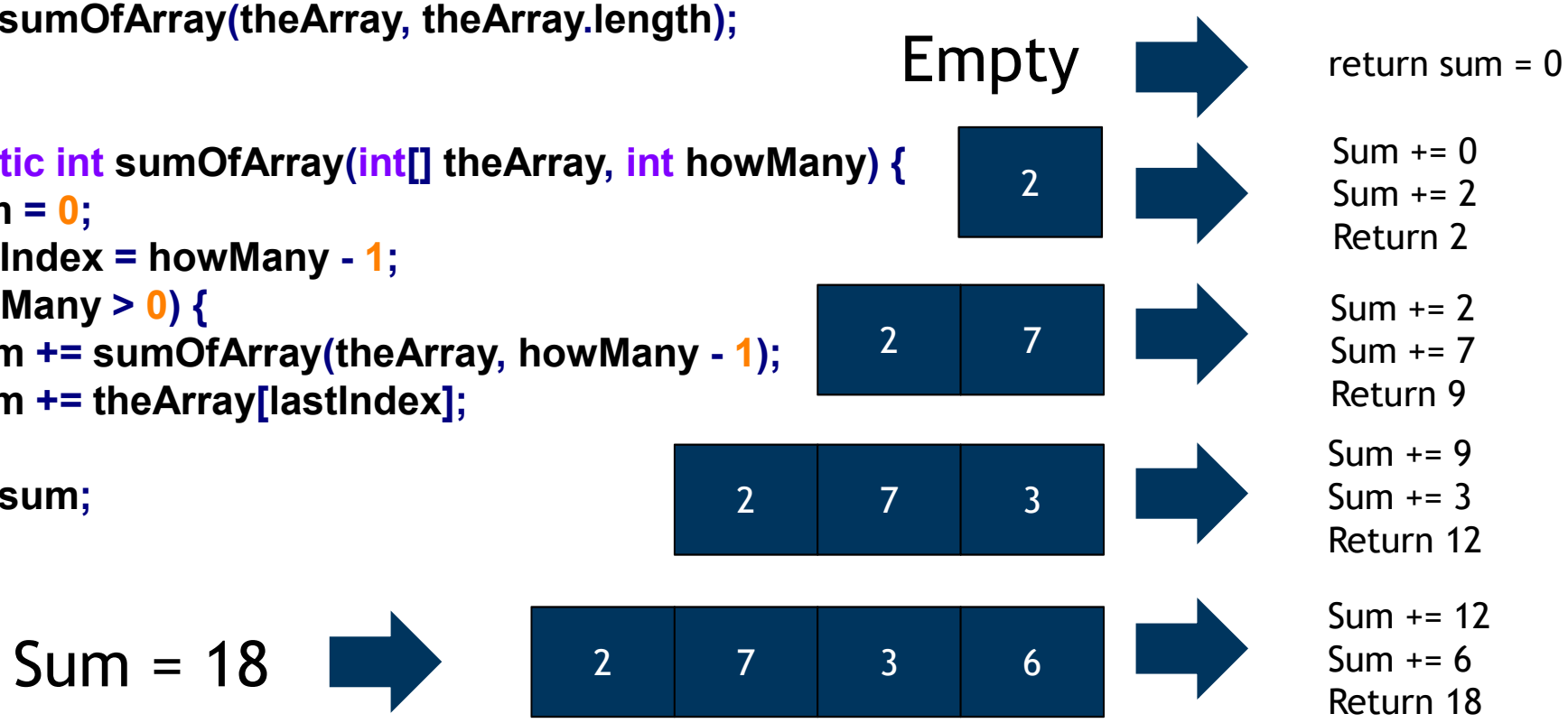


Focus on the sum of the smaller subArray

Sum of Array Contents

```
public static int sumOfArray(int[] theArray) {
    return sumOfArray(theArray, theArray.length);
}

private static int sumOfArray(int[] theArray, int howMany) {
    int sum = 0;
    int lastIndex = howMany - 1;
    if (howMany > 0) {
        sum += sumOfArray(theArray, howMany - 1);
        sum += theArray[lastIndex];
    }
    return sum;
    }
```



Lab Exercises

- Now that we've seen a couple of examples of recursion, let's see how we can apply it to the lab exercises
- We will:
  - Reverse the contents of an array
  - Replace a character in a String

Reversing an Array

- Identify the problem
  - Reverse the array
- Identify the smaller problems
  - Reduce size of problem by reversing smaller portions of the array
- Identify how the answers are composed
  - Reverse(array) = append(Reverse(portion), last_item_of_array)
- Identify the base cases
  - Empty array – return an empty array
  - Array of size 1 – return that array

Int[] array = { 1, 2, 3, 4, 5 };

- Reverse({1, 2, 3, 4, 5}) = append(Reverse{2, 3, 4, 5}, 1)
  - Reverse({2, 3, 4, 5}) = append(Reverse {3, 4, 5}, 2)
    - Reverse({3, 4, 5}) = append(Reverse{4, 5}, 3)
      - Reverse({4, 5}) = append(Reverse{5}, 4)
        - Reverse({5}) = {5}

Replacing Characters in a String

- Identify the problem
  - Replace character *before* with character *after* in the string
- Identify the smaller problems
  - Examine each character individually
- Identify how the answers are composed
  - Replace(string) = concatenate(Replace(sub_portion), current_char)
- Identify the base cases
  - Empty String – return an empty string

Replacing a character in “Hello World”

- Must consider each letter of the string “Hello World”
- Recursive Case will break “Hello World” into smaller substrings
- Concatenate the letter or the letter’s replacement to the already checked substring
- Once all letters or their replacements have been concatenated, the string “Hello World” will have swapped out all replaceable characters

Lab 6

- Write a recursive method that can reverse an array
- Write a recursive method that can perform character replacements in Strings
- All needed files are on the course website:

http://db.cs.pitt.edu/courses/cs0445/current.term/