IS2020 COMP 2540: Data Structures and Algorithms

Lecture 05: Recursion

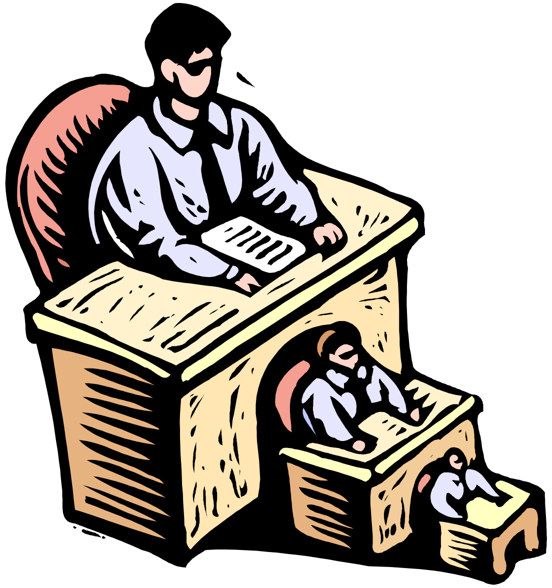
Dr. Kalyani Selvarajah
kalyanis@uwindsor.ca

School of Computer Science
University of Windsor
Windsor, Ontario, Canada

June 10, 2020

Outline

1. Background
2. Classic examples
3. Structure of Recursive Implementations
4. Helper Methods



Background

- Recursion is a technique that allows us to break down a problem into one or more subproblems that are similar in form to the original problem.
- Recursion is not appropriate for every problem, but it's an important tool in your software development toolbox, and one that many people scratch their heads over.

- A recursive function is defined in terms of ***base cases*** and ***recursive steps***.
  - In a base case, we compute the result immediately given the inputs to the function call (there should be at least one base case). Every possible chain of recursive calls must eventually reach a base case.
  - In a recursive step, we compute the result with the help of one or more *recursive calls* to this same function, but with the inputs somehow reduced in size or complexity, closer to a base case.

Classic Example

- The factorial function:

  $$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$$

- Recursive definition:

  $$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

- which leads to two different implementations:

Solution 1

```
public static long factorial(int n)
{
    long fact = 1;
    for (int i = 1; i <= n; i++)
    {
        fact = fact * i;
    }
    return fact;
}
```

Solution 2

```
public static long recursiveFactorial(int
n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

Visualizing Recursive

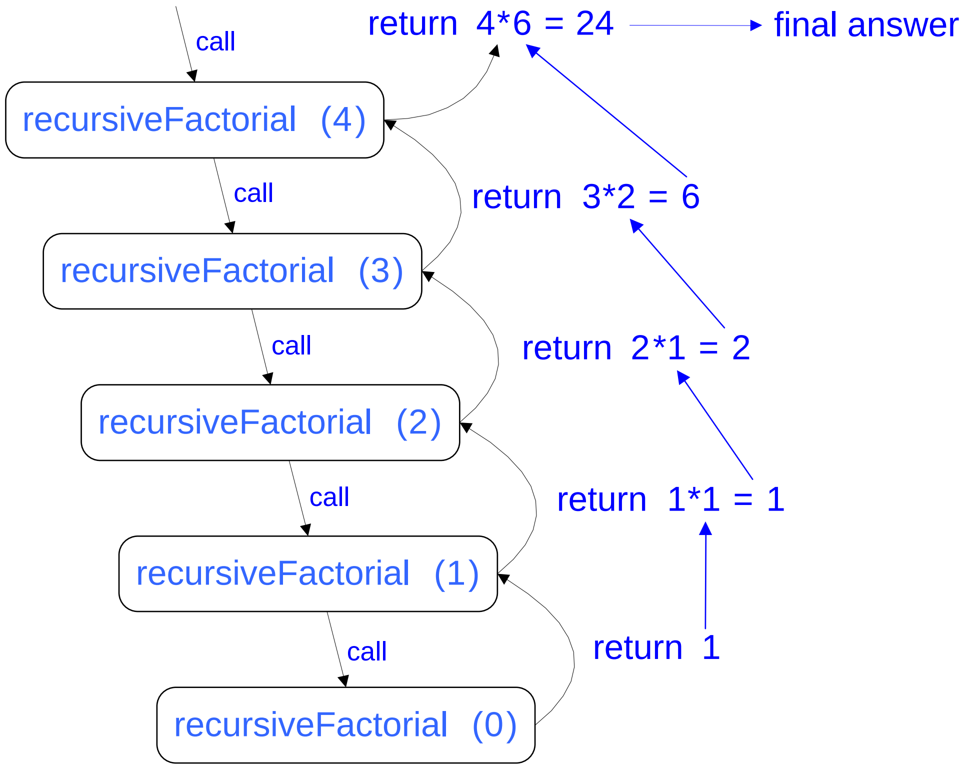
- To visualize the execution of a recursive function, it is helpful to diagram the call stack of currently-executing functions as the computation proceeds.

- Let’s run the recursive implementation of factorial in a main method:

```
public static void main(String[] args) {
    long x = recursiveFactorial(4);
}
```

Recursion trace

A box for each recursive call
An arrow from each caller to callee
An arrow from each callee to caller showing return value



Example

Fibonacci series

```
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return 1; // base cases
    } else {
        return fibonacci(n-1) + fibonacci(n-2); // recursive step
    }
}
```

Visualization:

Self Assessment!

- Consider this recursive implementation of the Fibonacci sequence.

For fibonacci(3), how many times will the base case return 1 be executed?

a. 0 times
b. 1 time
c. 2 times
d. 3 times
e. more than 3 times

- Mystery Recursion: Trace this function

```
int mystery(int n) {
    if (n < 10) {
        return n;
    } else {
        int a = n/10;
        int b = n % 10;
        return mystery(a + b);
    }
}
```

What is the result of mystery(648)?

a. 8
b. 9
c. 54
d. 72
e. 648

More Examples! isPalendrome(string s)

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("Java") → false
- isPalindrome("byebye") →false

Write a recursive function for Reversing an Array?

Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.

- Such methods can be easily converted to non-recursive methods (which saves on some resources)

```java
public static void rvereseArray(int arr[], int start, int end)
{
    int temp;
    if (start >= end)
        return;
    temp = arr[start];
    arr[start] = arr[end];
    arr[end] = temp;
    rvereseArray(arr, start+1, end-1);
}
```

Binary Recursion

- Binary recursion occurs whenever there are two recursive calls for each non-base case.

- The Fibonacci sequence is usually defined as follows:

```
fib(1) = fib(2) = 1
fib(n) = fib(n-1)+fib(n-2),  if n>2
```

- There are two base cases. The recursive step uses fib twice. This leads directly to a binary-recursive coding:

```
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return 1; // base cases
    } else {
        return fibonacci(n-1) + fibonacci(n-2); // recursive step
    }
}
```

Try Yourself!

- Use linear recursion for Fibonacci sequence.

Multiple Recursion

- Multiple recursion:
  - makes potentially many recursive calls
  - not just one or two

- Motivating example:
  - summation puzzles
    - $pot + pan = bib$
    - $dog + cat = pig$
    - $boy + girl = baby$

Algorithm for Multiple Recursion

```
Algorithm PuzzleSolve(k,S,U):
 Input: Integer k, sequence S, and set U (universe of elements to test)
 Output:  Enumeration of all k-length extensions to S using elements in U without repetitions
 for all e  in U do
    Remove e from U       {e is now being used}
    Add e to the end of S
    if k  =  1 then
          Test whether S is a configuration that solves the puzzle
          if S solves the puzzle then
                return "Solution found: " S
    else
          PuzzleSolve(k - 1, S,U)
    Add e back to U       {e is now unused}
    Remove e from the end of S
```

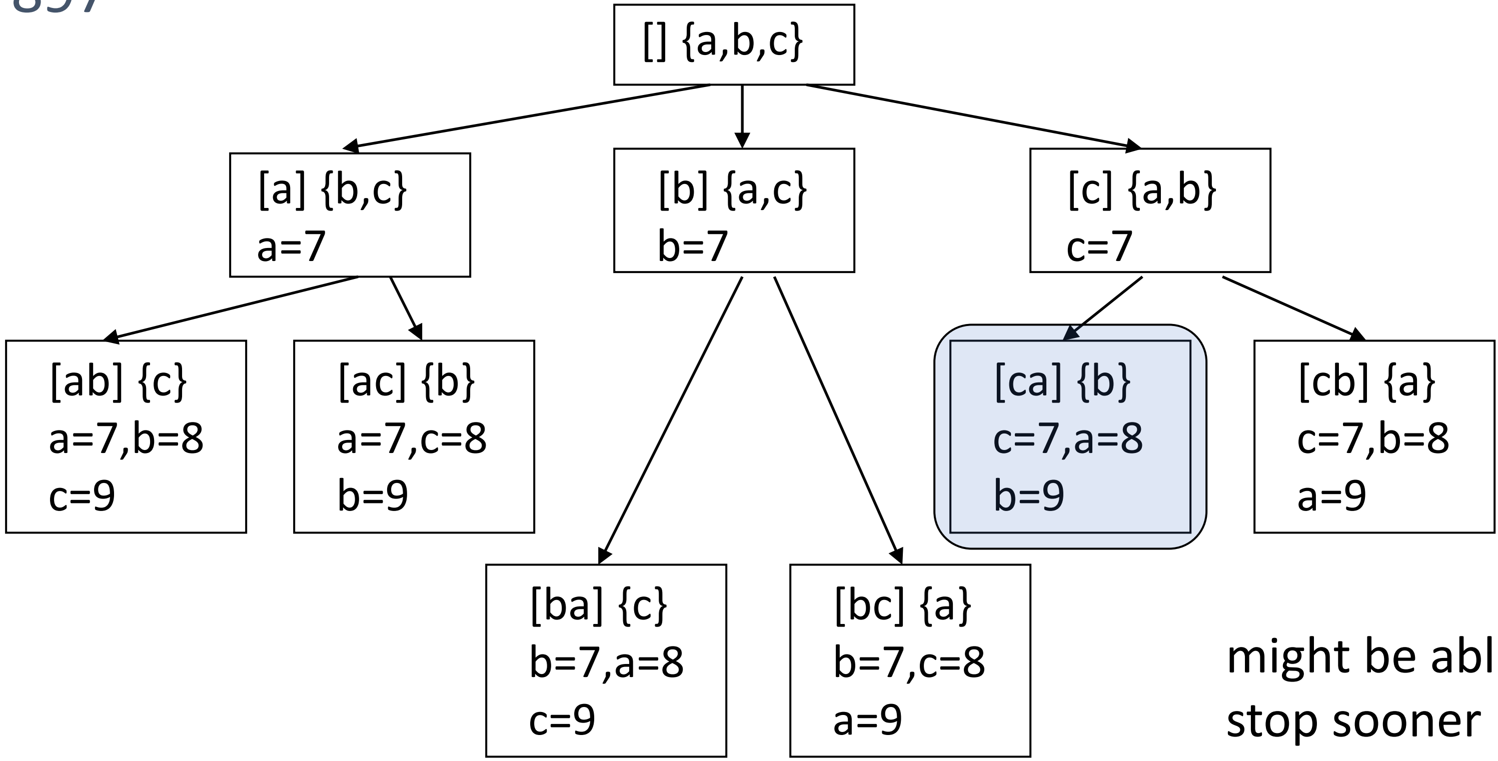
Example

Slide by Matt Stallmann included with permission.

cbb + ba = abc

799 + 98 = 897

a,b,c stand for 7,8,9; not necessarily in that order



[] {a,b,c}

[a] {b,c}
a=7

[b] {a,c}
b=7

[c] {a,b}
c=7

[ab] {c}
a=7,b=8
c=9

[ac] {b}
a=7,c=8
b=9

[ba] {c}
b=7,a=8
c=9

[bc] {a}
b=7,c=8
a=9

[ca] {b}
c=7,a=8
b=9

[cb] {a}
c=7,b=8
a=9

might be able to stop sooner

Choosing the Right Decomposition for a Problem (1/2)

- Finding the right way to decompose a problem, such as a method implementation, is important.
- Good decompositions are simple, short, easy to understand, safe from bugs, and ready for change.

- Recursion is an elegant and simple decomposition for some problems. Suppose we want to implement this specification:
  - A word consisting only of letters A-Z or a-z, write a function to return all subsequences of word, separated by commas, where a subsequence is a string of letters found in word in the same order that they appear in word.

Choosing the Right Decomposition for a Problem (2/2)

```java
public static String subsequences(String word) {
    if (word.isEmpty()) {
        return ""; // base case
    } else {
        char firstLetter = word.charAt(0);
        String restOfWord = word.substring(1);

        String subsequencesOfRest = subsequences(restOfWord);
        String result = "";
        for (String subsequence : subsequencesOfRest.split(",", -1)) {
            result += "," + subsequence;
            result += "," + firstLetter + subsequence;
        }
        result = result.substring(1); // remove extra leading comma
        return result;
    }
}
```

What does subsequences("c") return?

a. “c”
b. “”
c. “,c”
d. “c,”

What does subsequences(“gc") return?

a. “g,c”
b. “,g,c,gc”
c. “,gc,g,c”
d. “g,c,gc”

Structure of Recursive Implementations

- A recursive implementation always has two parts:
  - **base case**, which is the simplest, smallest instance of the problem, that can't be decomposed any further. Base cases often correspond to emptiness – the empty string, the empty list, the empty set, the empty tree, zero, etc.
  - **recursive step**, which **decomposes** a larger instance of the problem into one or more simpler or smaller instances that can be solved by recursive calls, and then **recombines** the results of those subproblems to produce the solution to the original problem.

- It's important for the recursive step to transform the problem instance into something smaller, otherwise the recursion may never end. If every recursive step shrinks the problem, and the base case lies at the bottom, then the recursion is guaranteed to be finite.
- A recursive implementation may have more than one base case, or more than one recursive step.

? Recursive methods have a base case and a recursive step. What other concepts from computer science also have (the equivalent of) a base case and a recursive step?

Helper Methods

- In some cases, it's useful to require a stronger (or different) specification for the recursive steps, to make the recursive decomposition simpler or more elegant.

- In the example of `subsequences(),` what if we built up a partial subsequence using the initial letters of the word, and used the recursive calls to complete that partial subsequence using the remaining letters of the word?

Helper Methods

- In some cases, it’s useful to require a stronger (or different) specification for the recursive steps, to make the recursive decomposition simpler or more elegant.

- In the example of `subsequences(),` what if we built up a partial subsequence using the initial letters of the word, and used the recursive calls to complete that partial subsequence using the remaining letters of the word?

helper method

```
private static String subsequencesAfter(String partialSubsequence, String word) {
    if (word.isEmpty()) {
        // base case
        return partialSubsequence;
    } else {
        // recursive step
        return subsequencesAfter(partialSubsequence, word.substring(1))
                + ","
                + subsequencesAfter(partialSubsequence + word.charAt(0), word.substring(1));
    }
}
```

Helper Methods

- To finish the implementation, we need to implement the original subsequences spec, which gets the ball rolling by calling the helper method with an initial value for the partial subsequence parameter:

```java
public static String subsequences(String word) {
    return subsequencesAfter("", word);
}
```

Common Mistakes in Recursive Implementations

- Here are two common ways that a recursive implementation can go wrong:
  - The base case is missing entirely, or the problem needs more than one base case but not all the base cases are covered.
  - The recursive step doesn't reduce to a smaller subproblem, so the recursion doesn't converge.
- Look for these when you're debugging.

- On the bright side, what would be an infinite loop in an iterative implementation usually becomes a StackOverflowError in a recursive implementation. A buggy recursive program fails faster.