Lecture 13: Recursion

Creation Through Self-Referral Dynamics

Wholeness of the Lesson

Computation of a function by recursion involves repeated self-calls of the function. Recursion is implicit also at the design level when a reflexive association is present. Recursion mirrors the self-referral dynamics of consciousness, the unified field, on the basis of which all creation emerges.

Outline of Topics

- Recursion Defined and two examples: Factorial and Fibonacci
- How to solve problem with recursion: Recursive Thinking
- Examples of Recursive Functions
  - Reversing characters in a string
  - Finding the minimum character in a string
  - Sum of elements in a list of integers
  - Binary Search
  - Anagrams

Introduction to Recursion

- Theoretically, any problem that can be solved using iteration (while and for loops) can be solved using recursion.
- A ***recursive method*** is a method that contains a statement (or statements) that makes a call to itself.
  - Solving a problem using recursion depends on solving smaller occurrences of the same problem.

First Example: factorial

- The *factorial of N* is the product of the first N positive integers:

  $N * (N - 1) * (N - 2) * \cdots * 2 * 1$

- Another way of defining factorial using recursion.

  N! = N x (N-1)!

  1! = 1.

- *The factorial of N* can be defined *recursively* as

$$\text{factorial(N) = } \begin{cases} 1 & \text{if N = 1} \\ \text{N * factorial(N-1)} & \text{otherwise} \end{cases}$$

First Example: factorial

- Implementing the factorial of N recursively will result in the following method.

```
public int factorial( int N ) {

    if ( N == 1 ) {
        return 1;
    }
    else {
        return N * factorial( N-1 );
    }
}
```

The Call Stack

- Function calls are placed in the area of memory called the *stack*. When a program calls a function, that function goes on top of the call stack. If a function A calls another function B, then B will be placed on top of A, and A cannot proceed until B is done.
- Example of a stack:

The Call Stack

Call stack for `factorial(5)`	Sequence of steps	return for each method call
`factorial(1)`	1	`return 1`
`factorial(2)`	2	`return 2*1`
`factorial(3)`	3	`return 3*2*1`
`factorial(4)`	4	`return 4*3*2*1`
`factorial(5)`	5	`return 5*4*3*2*1`

Two components

- In order to be a *valid recursion* (one that eventually terminates), the following criteria must be met:
  - *Base Case Exists.* The method must have a base case which returns a value without making a self-call.
  - *Self-calls Lead to Base Case.* For every input to the method, the sequence of self-calls must eventually lead to a self-call in which the base case is accessed.

Base case

Recursive calls

```
public int factorial( int N ) {

   if ( N == 1 ) {
      return 1;
   }
   else {
      return N * factorial( N-1 );
   }
}
```

Another Example: Fibonacci Numbers

The Fibonacci numbers are defined as follows:

$$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, \ldots,$$

$$F_n = F_{n-1} + F_{n-2}, \ldots$$

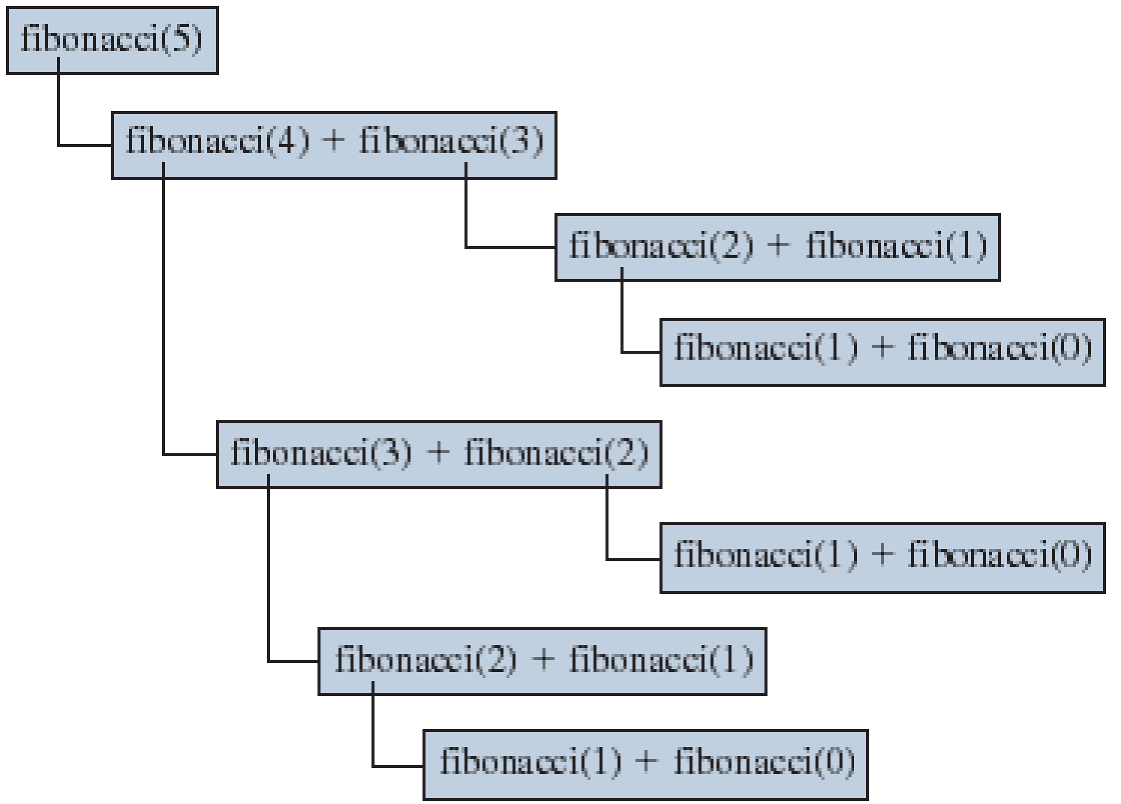
The nth Fibonacci number can be computed by:

```
int fib(int n) {
      if(n == 0 || n == 1) {
            return n;
      }
      return fib(n-1) + fib(n-2);
}
```

Exercise: draw the self calls for `fib(3)`

Excessive Repetition

- Recursive Fibonacci ends up repeating the same computation numerous times.



Exercise: implement Fibonacci in a non-recursive way.

Design Guideline

No recursion should involve a large amount of redundant computation.

Usually, if a recursion *does* involve redundant computations, it can be rewritten as a loop or by using a more efficient recursive strategy.

Exercise

Question: What does this method do?

```
public int sum ( int N ) { //assume N >= 1
     if (N == 1)
        return 1;
     else
        return N + sum( N-1 );
}
```

computes the sum of the first N positive integers 1, 2, . . . , N

Outline of Topics

- Recursion Defined and two examples: Factorial and Fibonacci
- How to solve problem with recursion: Recursive Thinking
- Examples of Recursive Functions
  - Reversing characters in a string
  - Finding the minimum character in a string
  - Sum of elements in a list of integers
  - Binary Search
  - Anagrams

Recursive Thinking

Think declaratively

1. Define the base cases
   - Instance(s) that can be calculated without using recursive calls
2. Decompose the problem into simpler or smaller instances of the original problem
   - A smaller/simpler instance must be moving toward one of the base cases (so the function terminates)
3. Determine what to do in addition to the recursive calls

Recursive Thinking – the cheat sheet

To implement a method someMethod(. . .) recursively, follow these steps
[could be: someMethod(int N), someMethod(String s), someMethod(int[] arr)]

1. Determine the base case

   typical base cases: N -> 0, 1; String -> "" or string of length 1; array -> empty array or array of length 1

   May have to adjust the base case based on the problem – do this by testing with a few small values.
2. Make the recursive/self call(s)
   someMethod(reduced input)
   typically: N -> N-1; string -> subString(1); array -> sub array
3. "Trust" the recursive call - what is the expected output from the recursive call?
4. Use the expected output from the recursive call to construct a solution to the original call.
   (There is no general formula for doing this -- it depends on the problem.)

Outline of Topics

- Recursion Defined and two examples: Factorial and Fibonacci
- How to solve problem with recursion: Recursive Thinking
- Examples of Recursive Functions
  - Reversing characters in a string
  - Finding the minimum character in a string
  - Sum of elements in a list of integers
  - Binary Search
  - Anagrams

Using Recursive Thinking: Reversing a String

Attempt to reverse the order of the characters in an input String by using the following strategy:

- Remove the 0th character `ch` from the input string and name the modified string `t`.
- Reverse `t` and append `ch`.

```
static String reverse(String s) {
  if(s == null || s.length() == 0)
        return s;
  char first = s.charAt(0);
  return reverse(s.substring(1)) + first;
}
```

Exercises

1. Write pseudocode function, *isEven*(n) to recursively determine whether a natural number, n, is an even number.
2. Write pseudocode function, *findMin*(str), to recursively find the minimum letter in the input string str.
3. Write pseudocode function, *sum*(list), to recursively calculate the sum of the integers in the given list of integers

Another famous Recursive Algorithm

Problem: Given an array of integers in sorted order, is it possible to perform a search for an element in such a way that no more than half the elements of the array are examined? (Assume the array has 8 or more elements.)

Binary Search

Algorithm search(A,x)
Input: An already sorted array A with n elements and search value x
Output: true or false
return binSearch(A, x, 0, A.length-1)

Algorithm binSearch(A, x, lower, upper)
Input: Already sorted array A of size n, value x to be
searched for in array section A[lower]..A[upper]
Output: true or false

```
if lower > upper then  return false
mid ← (upper + lower)/2
if x = A[mid] then  return true
if x < A[mid]  then
     return binSearch(A, x, lower, mid – 1)
else
     return binSearch(A, x, mid + 1, upper)
```

Example

Search key x = 7

(1 2 5 7 12 **14** 21 24 25 38 52)

search left

(1 2 **5** 7 12)

search right

(**7** 12)

A[mid] = 7 → return true

Example

Search key x = 20

(1 2 5 7 **12** 14 21 24 25 38)

search right

(14 21 **24** 25 38)

search left

(**14** 21)

search right

(**21**)

search left

lower > upper → return false

Recursive Implementations of a Utility Method

- Sorting, searching, and other manipulations of characters in a string or elements in arrays or lists are often done recursively. Sometimes (but not always), an implementation of such a utility provides a public method

```
public <return-value-type> thePublicMethod(params)
```

whose signature and return type make sense to potential users, and a private recursive method

```
private <ret-value-type> privateRecurMethod(otherParams)
```

which does the real work and is designed to call itself.

The Divide and Conquer Strategy

- The Binary Search algorithm is an example of a "Divide And Conquer" algorithm, which is typical strategy when recursion is used.

- The method:
  - **Divide** the problem into subproblems (divide input array into left and right halves)
  - **Conquer** the subproblems by solving them recursively (search recursively in whichever half could potentially contain target element)
  - **Combine** the solutions to the subproblems into a solution to the problem (return value found or indicate not found)

Last Example: Anagram

- List all anagrams of a given word. An ***anagram*** is a word formed by reordering the letters of the given word.

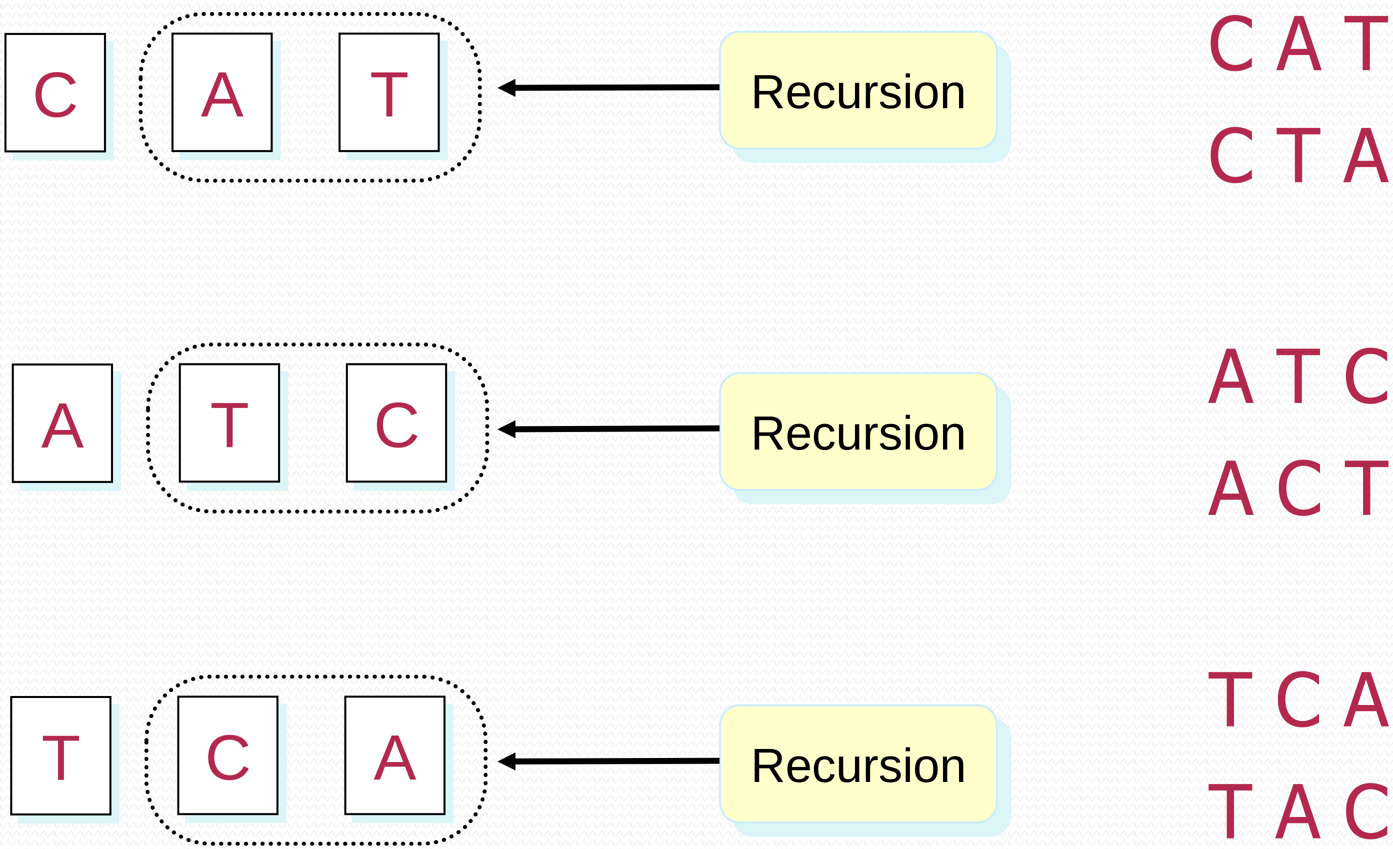
Word → C A T

C T A
A T C
A C T
T C A
T A C

Anagrams

Anagram Solution

- The basic idea is to make recursive calls on each sub-word. Here's how:



Anagram Method

```
public static String[] anagram(String word) {
    int n = word.length();
    if(n == 1)
         return new String[]{word};
    String[] anagrams = new String[Factorial.factorial(n)];
    int j = 0;
    for(int i = 0; i < n; i++) {
         char current = word.charAt(i);
         String sub = word.substring(0,i) +
                                     word.substring(i+1, n);
         String[] temp = anagram(sub);
         for(int k = 0; k < temp.length; k++) {
            anagrams[j] = current + temp[k];
            j++;
         }
    }
    return anagrams;
}
```

Main Point

Java supports the creation of recursive methods, characterized by the fact that they call themselves in their method body. A self-calling method is a *valid* recursive function if it contains a *base case* – a branch of code that exits the method under certain conditions but does not involve a self-call – and if the sequence of self-calls, on any input to the method, always converges to the base case. Likewise, a quest for self-knowledge not based in the direct experience of the "Self" is endless (and baseless).

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Recursion creates from self-referral activity

1. In Java, it is possible for a method to call itself.
2. For a self-calling method to be a legitimate recursion, it must have a base case, and whenever the method is called, the sequence of self-calls must converge to the base case.

---

3. **Transcendental Consciousness:** TC is the self-referral field of existence, at the basis of all manifest existence.
4. **Wholeness moving within itself**: In Unity Consciousness, one sees that all activity in the universe springs from the self-referral dynamics of wholeness. The "base case" – the reference point – is always the Self, realized as Brahman.