SEENG1110/SENG6110

Object Oriented Programming

Lecture 10

Recursion

Outline

- Recursive definitions
- Recursive Problem Solving
- Factorial example
- How recursion works
- Tracing a recursive method
- The Run-Time Stack
- Infinite Recursion and Stack Overflow
- Recursion and Iteration
- Sequential search example
- Fibonacci example

Recursive Definitions

- Recursion
  - Process of solving a problem by reducing it to simpler versions of itself.

Recursive Definitions

- Recursive algorithm:
  - Algorithm that finds the solution to a given problem by reducing the problem to smaller versions of itself.
  - Has one or more base cases.
  - Implemented using recursive methods.
- Recursive method:
  - Method that calls itself.
- Base case:
  - Case in recursive definition in which the solution is obtained directly.
  - Stops the recursion.

Recursive Problem Solving

- Find one or more simple cases of the problem that can be solved directly – base cases
- Find a way to make the problem smaller for a recursive solution
- Find a way to combine the partial solutions

Factorial example

- Factorial:
  - n! = 1, when n = 1
  - n! = n * (n - 1)! otherwise

3! = 3 * 2!
3 * 2 * 1!
3 * 2 * 1

Factorial example

base case

n! = 1, when n = 1
n! = n * (n - 1)! otherwise

```
factorial (n)
   if n == 1
      return 1
   else
      return n * factorial(n - 1)
```


Factorial example

base case

n! = 1, when n = 1
n! = n * (n - 1)! otherwise

Recursive case

```
factorial (n)
   if n == 1
      return 1
   else
      return n * factorial(n - 1)
```

Factorial example

n! = 1, when n = 1

n! = n * (n - 1)! otherwise

base case

Recursive case

Combination step

```
factorial (n)
   if n == 1
      return 1
   else
      return n * factorial(n - 1)
```


How Recursion Works

- Each call of a method generates an instance of that method
- An instance of a method contains
  - memory for each parameter
  - memory for each local variable
  - memory for the return value

Tracing a Recursive Method

- Recursive method:
  - Has unlimited copies of itself.
  - Every recursive call has its own:
    - Code
    - Set of parameters
    - Set of local variables

Tracing a Recursive Method

- After completing a recursive call:
  - Control goes back to the calling environment.
  - Recursive call must execute completely before control goes back to previous call.
  - Execution in previous call begins from point immediately following recursive call.

Recursive Factorial Method


```java
public static int fact(int num)
{
    if (num == 1)
        return 1;
    else
        return num * fact(num - 1);
}
```


Example: `factorial(4)`

n 4
return value

Activation record for **`factorial(4)`**

Activations Are Added Dynamically

n 3
return value

Activation record for **`factorial(3)`**

n 4
return value

Activation record for **`factorial(4)`**

Number of Activations = # Calls

n 2
return value

Activation record for **`factorial(2)`**

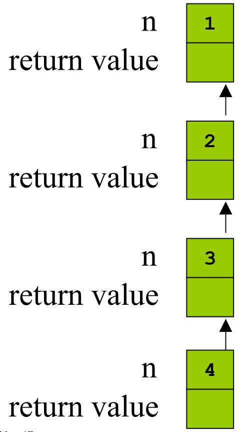
n 3
return value

Activation record for **`factorial(3)`**

n 4
return value

Activation record for **`factorial(4)`**

Recursive Process Bottoms out



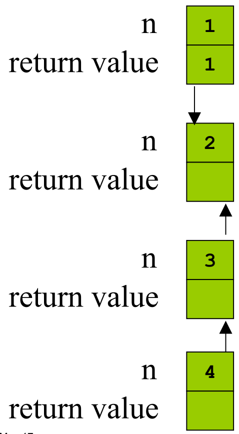
Activation record for **`factorial(1)`**

Activation record for **`factorial(2)`**

Activation record for **`factorial(3)`**

Activation record for **`factorial(4)`**

Recursive Process Unwinds



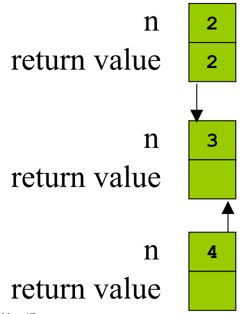
Activation record for **`factorial(1)`**

Activation record for **`factorial(2)`**

Activation record for **`factorial(3)`**

Activation record for **`factorial(4)`**

Activations Are Deallocated

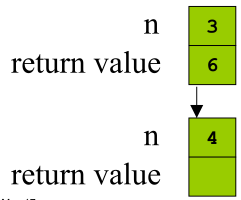


Activation record for **`factorial(2)`**

Activation record for **`factorial(3)`**

Activation record for **`factorial(4)`**

Activations Are Deallocated



Activation record for **`factorial(3)`**

Activation record for **`factorial(4)`**

Value Returned Is in the First Activation

n	4
return value	24

Activation record for **`factorial(4)`**

The Run-Time Stack

- To support recursive method calls, the run-time system treats memory as a *stack* of activation records
- Computing **`factorial(`*`n`*`)`** requires the allocation of *n* activation records on the stack

Infinite Recursion and Stack Overflow


```
int infiniteRecursion (int n) {
   if (n == 0)
      return 1;
   else
      return infiniteRecursion (n);
}
```

The value of n never reaches zero, so the method is called, and records are pushed onto the stack, until the system runs out of memory.

Recursion and Iteration

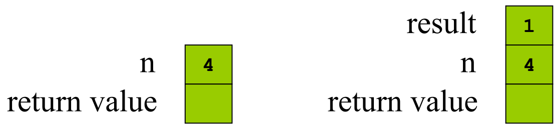

```
int factorial (int n){
   if n == 1
      return 1;
   else
      return n * factorial (n - 1);
}
```

```
int factorial (int n){
   int result = 1;
   while (n > 1){
      result = result * n;
      n--;
   }
   return result;
}
```

Recursive methods can be translated to methods that run loops.

Memory Usage

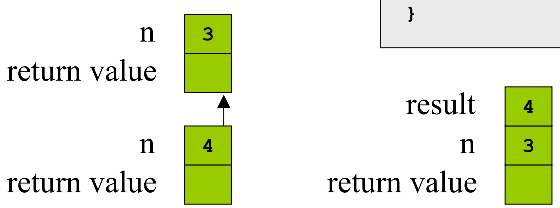

```
int factorial (int n){
   int result = 1;
   while (n > 1){
      result = result * n;
      n--;
   }
   return result;
}
```



Recursive version **Iterative version**

Memory Usage

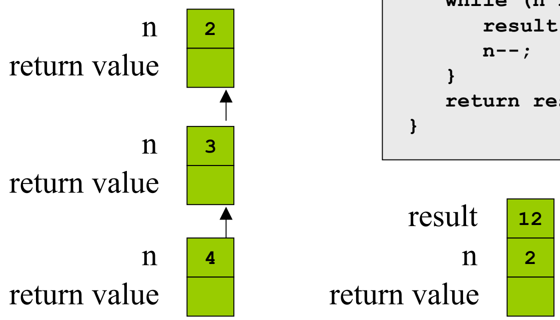

```
int factorial (int n){
   int result = 1;
   while (n > 1){
      result = result * n;
      n--;
   }
   return result;
}
```



Recursive version **Iterative version**

Memory Usage

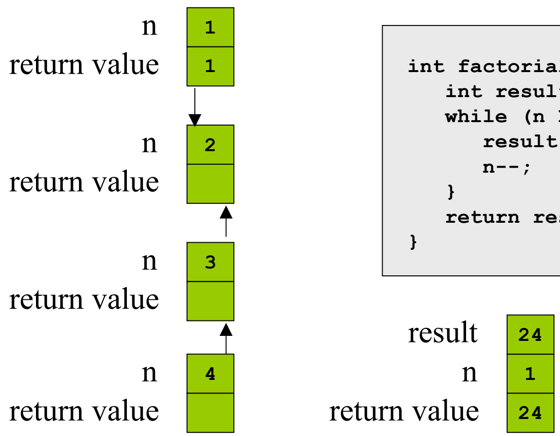

```
int factorial (int n){
   int result = 1;
   while (n > 1){
      result = result * n;
      n--;
   }
   return result;
}
```



Recursive version **Iterative version**

Memory Usage


```
int factorial (int n){
   int result = 1;
   while (n > 1){
      result = result * n;
      n--;
   }
   return result;
}
```



Recursive version **Iterative version**

Sequential Search example

```
int find(int[] a, int target) {
   return recursiveFind(a, target, 0);
}
```

Top-level method maintains interface to clients

```
int recursiveFind(int[] a, int target, int pos) {
   if (pos == a.length)
      return -1;
   else if (a[pos] == target)
      return pos;
   else
      return recursiveFind(a, target, pos + 1);
}
```

Base case 1: not in array

Base case 2: found target

Recursive step: search rest of array

Sequential Search example

34	67	56	41	21	89	13
0	1	2	3	4	5	6

target 41

pos 0 (initially)

Calls

```
recursiveFind(a, 41, 0) ->
   recursiveFind(a, 41, 1) ->
      recursiveFind(a, 41, 2) ->
         recursiveFind(a, 41, 3) ->
         3 <-
      3 <-
   3 <-
3 <-
```

Returns

Fibonacci Example

- The first two numbers in the series are 0 and 1.
- Each remaining number is obtained by taking the
- sum of the previous two numbers in the series.

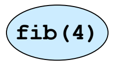
- Example: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

Definition:
fib(n) = 0, when n = 0
fib(n) = 1, when n = 1
fib(n) = fib(n - 1) + fib(n - 2) when n > 1

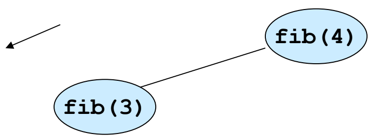
Fibonacci Example

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

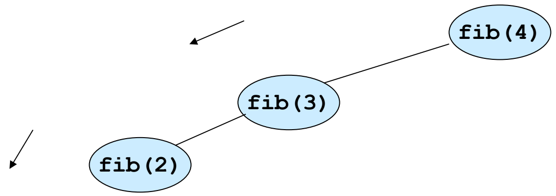
Tracing `fib(4)` with a Call Tree



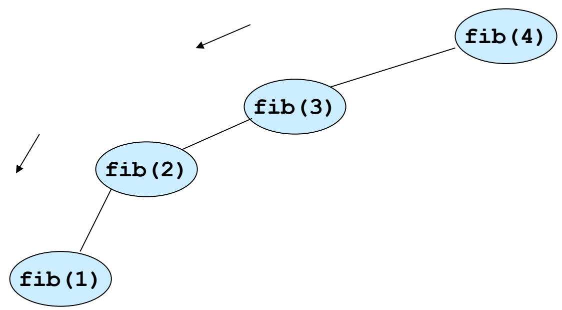
Tracing `fib(4)` with a Call Tree



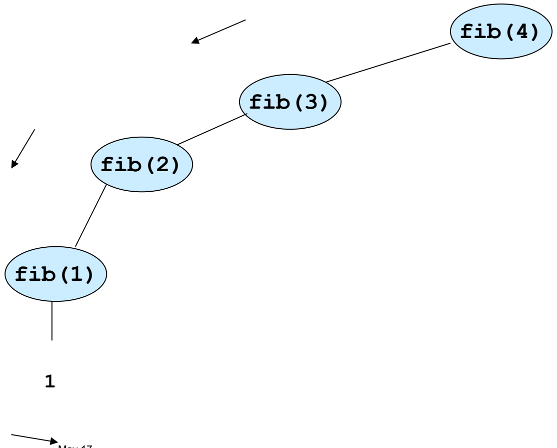
Tracing `fib(4)` with a Call Tree



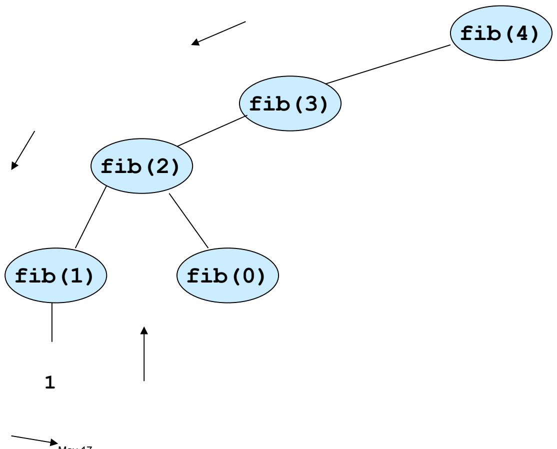
Tracing `fib(4)` with a Call Tree



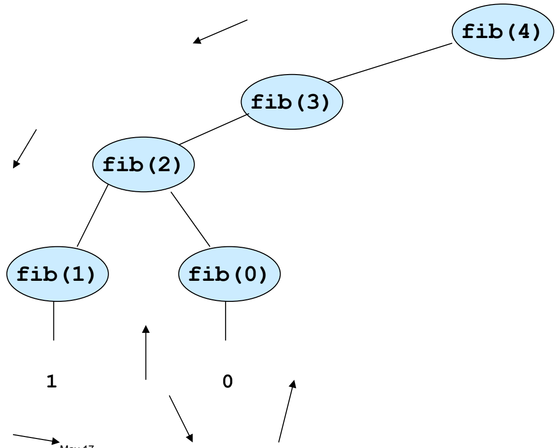
Tracing `fib(4)` with a Call Tree



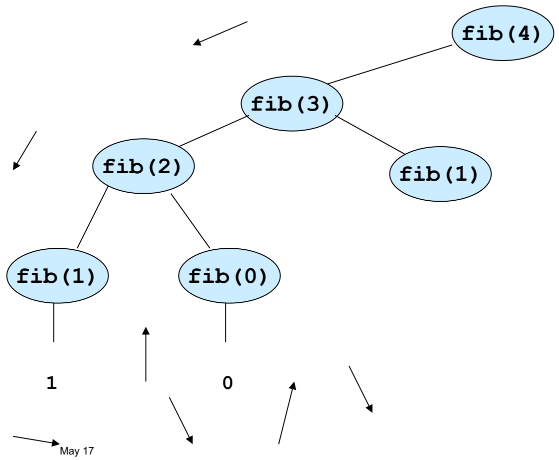
Tracing `fib(4)` with a Call Tree



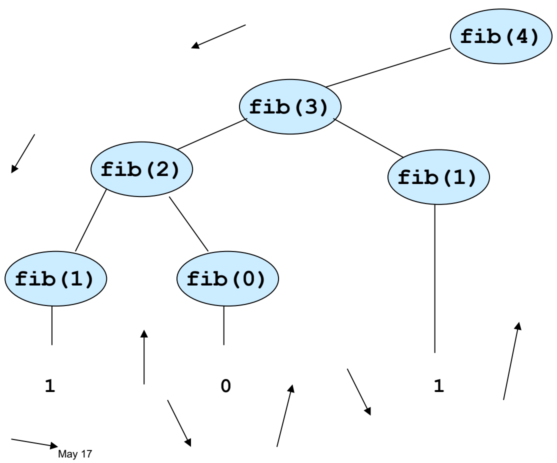
Tracing `fib(4)` with a Call Tree



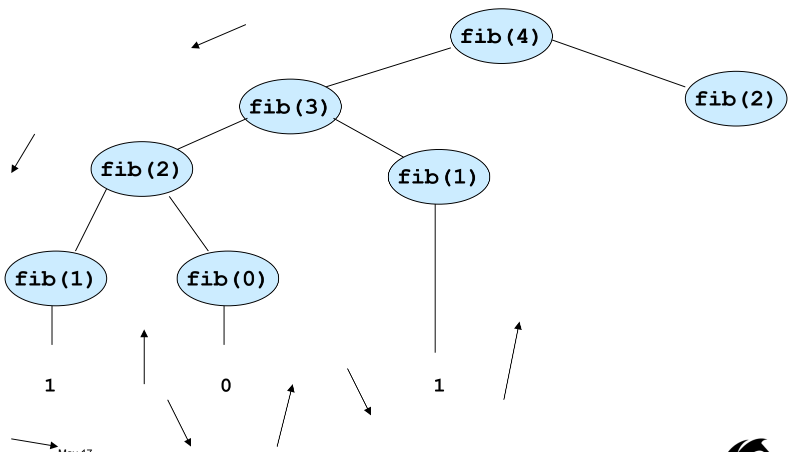
Tracing `fib(4)` with a Call Tree



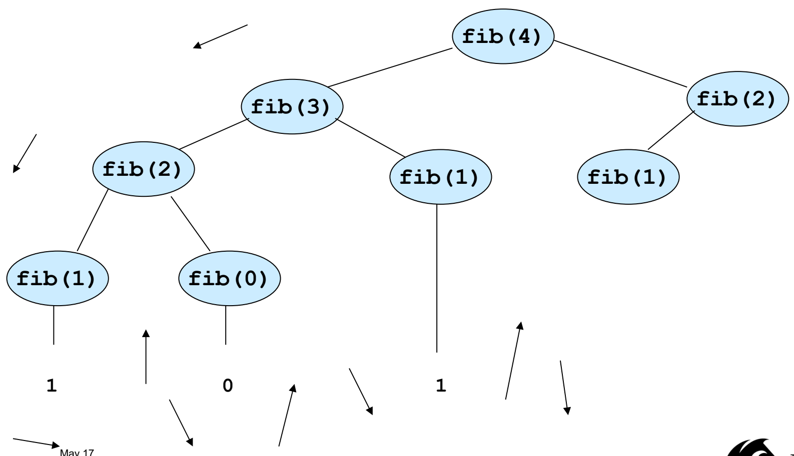
Tracing `fib(4)` with a Call Tree



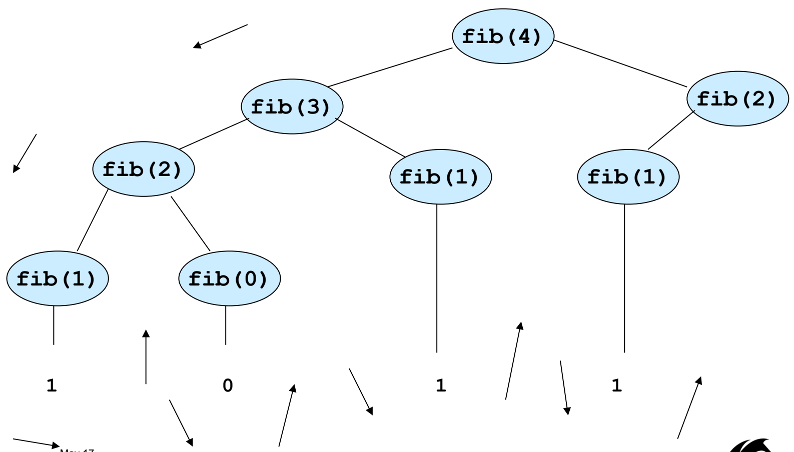
Tracing `fib(4)` with a Call Tree



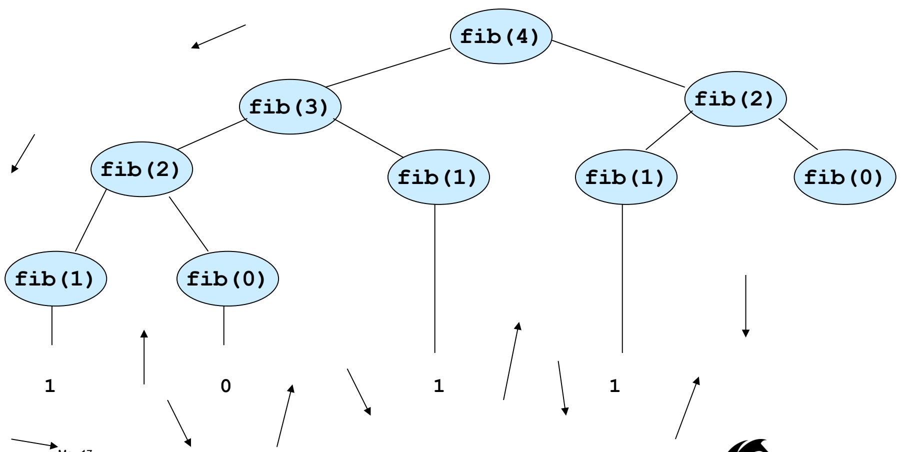
Tracing `fib(4)` with a Call Tree



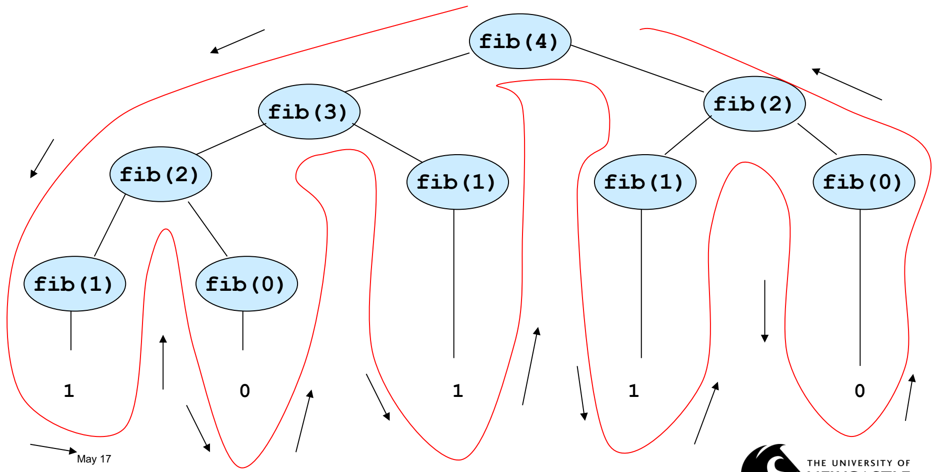
Tracing `fib(4)` with a Call Tree



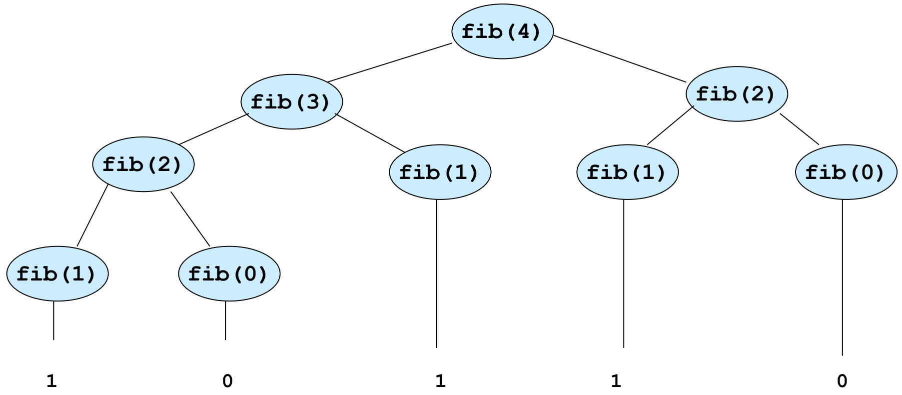
Tracing `fib(4)` with a Call Tree



Tracing `fib(4)` with a Call Tree



Work Done - Stack Memory



Maximum number of activations = The depth of the tree

Fibonacci as Algorithm and Process

- Fibonacci generates a *tree-recursive process* (processing time grows with the size of the call tree, and memory growths with depth of tree)

Fibonacci with a Loop


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

```
int fib (int n){
   int a = 1, b = 0;
   while (n > 0){
      int temp = a;
      a = a + b;
      b = temp;
      n--;
   }
   return b;
}
```


Recursion or Iteration?

- Two ways to solve particular problem:
  - Iteration
  - Recursion
- Iterative control structures use looping to repeat a set of statements.
- Tradeoffs between two options:
  - Sometimes recursive solution is easier.
  - Recursive solution is often slower.

Your task

- Read
  - Lecture slides
  - Chapter 11
- Exercises
  - MyProgrammingLab
  - Computer lab exercises