CSCI 136
Data Structures & Advanced Programming

Recursion

Recursion

Recursion

- General problem-solving strategy
  - Decompose problem into sub-problems
    - Sub-problem : Simpler version of same problem
  - Solve sub-problems
    - Either by further decomposition
    - Or directly, if sub-problem is easy
  - Combine sub-problem solutions to build problem solution

Recursion

- Many algorithms are recursive
  - Recursive algorithms are often easier to
    - Understand and implement
    - Prove correct

- In this presentation, we
  - Review recursion
  - Introduce techniques for reasoning about recursive algorithms

A Simple Example : Factorial

- $n!$ (pronounced "n factorial") denotes the product of the first n positive integers
$$n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1$$

- Note: $n!$ is only defined for $n \geq 1$, although by convention we define $0! = 1$

- We could compute $n!$ with a for loop…

```
int product = 1;
for(int i = 1;i <= n; i++)
      product *= i;
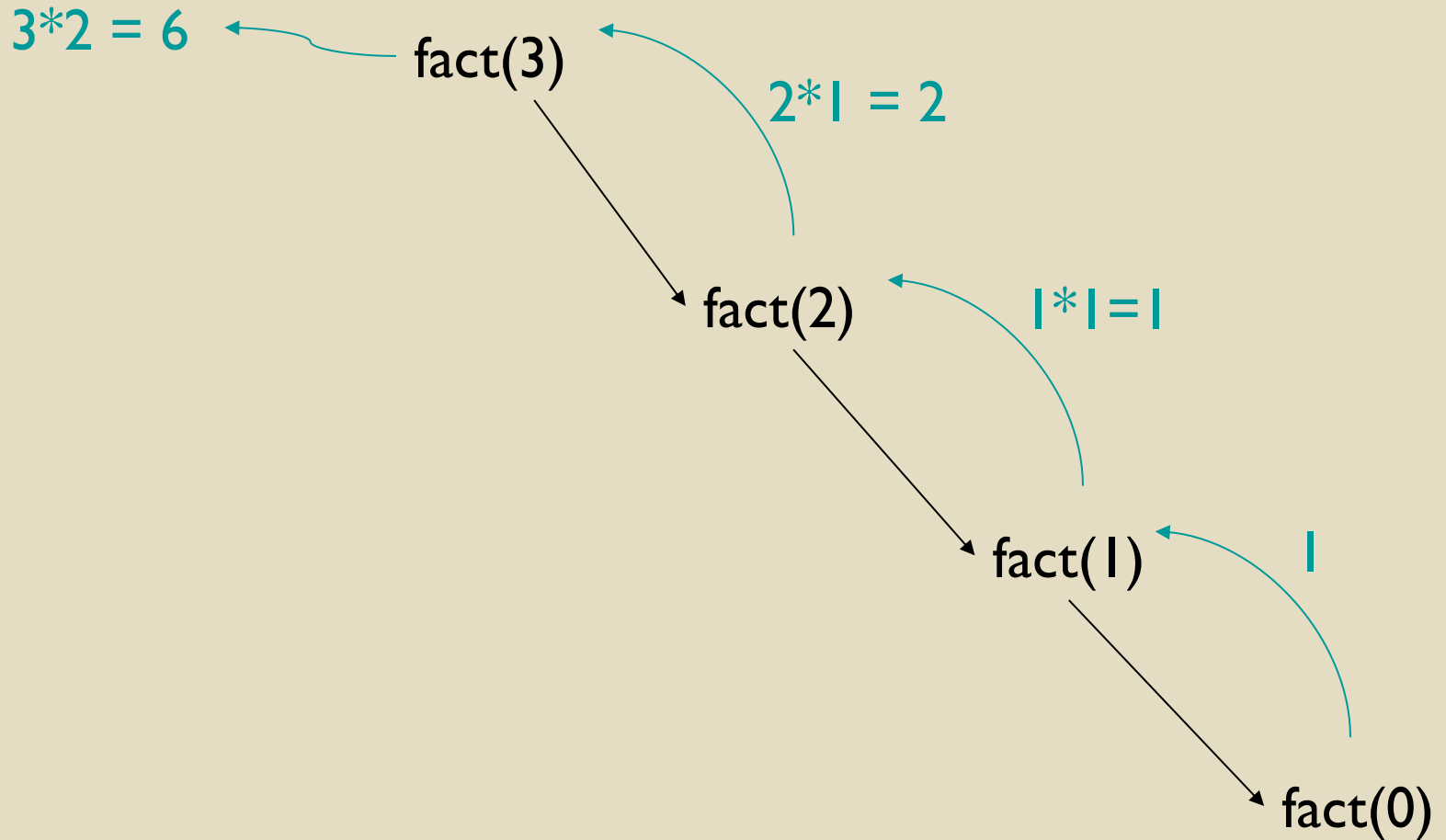```

- But we could also write it recursively….

# Factorial

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

- Recursive definition (what "…" really means!)
  - $n! = n \cdot (n-1)!$
  - 1! = 1 (or 0! = 1)

```
// Pre: n >= 0
public static int fact(int n) {
    if (n==0) return 1;
    else return n*fact(n-1);
}
```

Factorial

```
int k = fact(3);
```

$3*2 = 6$ ← fact(3)

$2*1 = 2$

fact(2)

$1*1=1$

fact(1)

1

fact(0)

Factorial

- In recursion, we always use the same basic approach
- What's our easy case? [Sometimes "cases"]
  - `n=0 : fact(0) = 1`
- What's the recursive relationship?
  - `n>0 : fact(n) = n·fact(n-1)`

Recursive Method Structure
(simple version)

my_recursive_method(data) {

if (data is simple) // typically called "the base case"

   solve directly

else {

   divide data into data$_1$, …, data$_k$

   *call my_recursive_method on each of data$_1$, …, data$_k$
and combine the results as needed*

Example : Fibonacci Numbers

- 1, 1, 2, 3, 5, 8, 13, ...
- Definition

$$F_0 = 1, F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \ (for \ n > 1)$$

- Inherently recursive!

- It appears almost everywhere

  - Growth: Populations, plant features

  - Architecture

  - Data Structures!

fib.java

```java
public class fib{
    // pre: n is non-negative
    public static int fib(int n) {

        if (n==0 || n == 1)                 // Note: Two base cases!
                return 1;
        else
                return fib(n - 1) + fib(n - 2);
    }

    public static void main(String args[]) {
        // This method could be a single line but: readability!
        int n = Integer.valueOf(args[0]);
        int result = fib(n);
        System.out.println(result);
    }
}
```
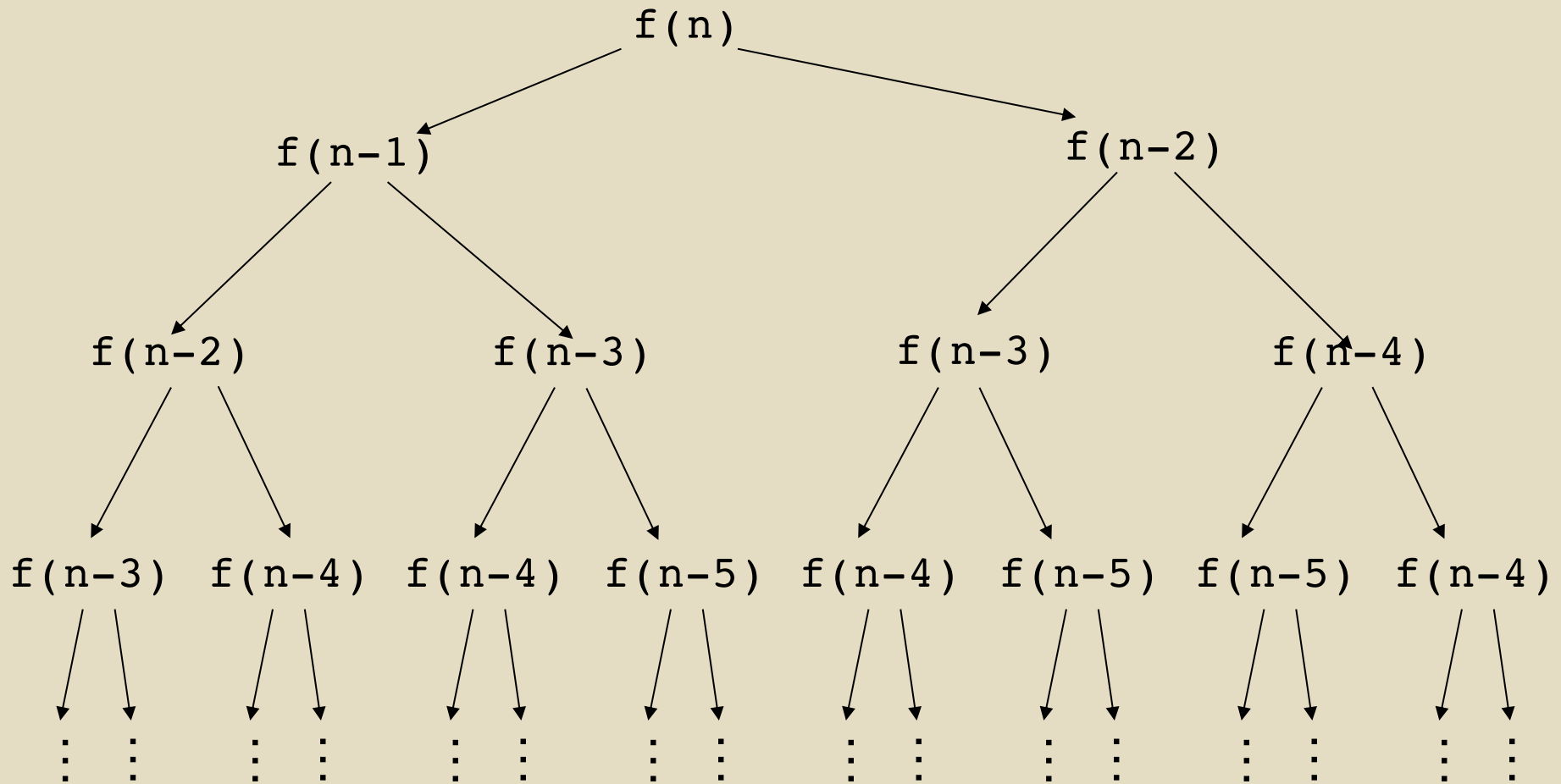
Demo: RecursiveMethods.java....

Question: Why is fib so slow?!

Recursive Fibonacci Method


```
                              f(n)
                    ┌──────────┴──────────┐
                 f(n-1)                 f(n-2)
            ┌───────┴───────┐      ┌───────┴───────┐
         f(n-2)          f(n-3)  f(n-3)          f(n-4)
         ┌──┴──┐         ┌──┴──┐ ┌──┴──┐         ┌──┴──┐
      f(n-3) f(n-4)  f(n-4) f(n-5) f(n-4) f(n-5) f(n-5) f(n-4)
       ⋮  ⋮    ⋮  ⋮    ⋮  ⋮    ⋮  ⋮    ⋮  ⋮    ⋮  ⋮    ⋮  ⋮    ⋮  ⋮
```

Yikes!

Recursion Tradeoffs

- Advantages
  - Often easier to construct recursive solution
  - Code is usually cleaner
  - Some problems do not have obvious non-recursive solutions
- Disadvantages
  - Overhead of recursive calls
  - Can use lots of memory (need to store state for each recursive call until base case is reached)
    - E.g. recursive fibonacci method

Recursive Contains

Consider a slightly rewritten contains method for SinglyLinkedList

```
public static boolean contains(Node<String> n, String v) {
    if( n == null || v == null ) return false;

    while(n != null) {
        if( v.equals(n.value())) return true;
        n = n.next();
    }
    return false;
}
```

Now let's try a recursive approach

```
public static boolean contains(Node<String> n, String v) {
    if( n == null || v == null ) return false;

    return v.equals(n.value()) || contains(n.next(), v);
}
```

Nice!

Recursive Contains for Vector

Replace loop in contains method with recursive helper method
- Helper method will work on any *tail slice* of the array
  - Like python array slice, but only when fromIndex ≤ toIndex
    - and, in this case, toIndex = size() - 1

```
public boolean contains(E elt) {
   return contains(elt, 0); }

// Helper method: returns true if elt is stored in range from..
public boolean contains(E elt, int from) {
   if (from => size)
       return false; // Base case: empty range
   else
       return elt.equals(elementData[from]) ||
              contains(elt, from+1);
}
```
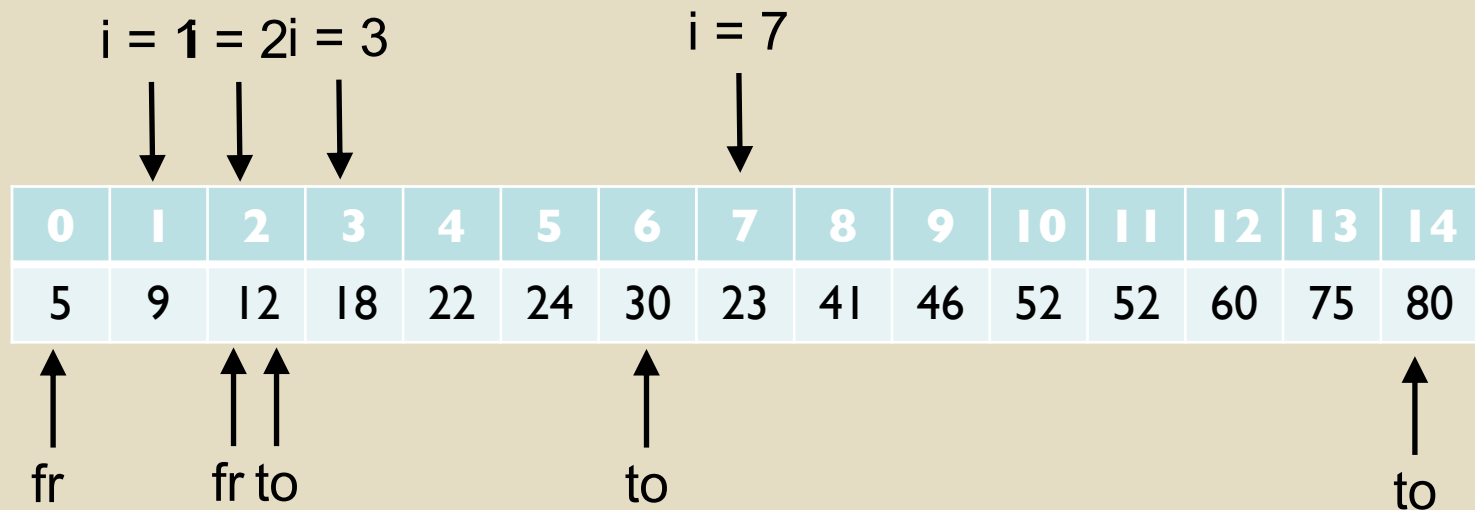
Binary Search in Sorted Array

- Given an array a[] of positive integers in increasing order, and an integer x, find location of x in a[].
  - Take ''indexOf'' approach: return -1 if x is not in a[]

- Idea
  - Compare x to "middle" element of array
    - If they are equal, return index of middle element
  - If x is smaller, recursively check " left half" of array
  - Otherwise, recursively check "right half" of array

Binary Search in Sorted Array

Let's picture the idea: Search for 12

i = 1 i = 2 i = 3   i = 7

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5	9	12	18	22	24	30	23	41	46	52	52	60	75	80

fr   fr to   to   to

Notes
- Need to keep track of current *search range*: `fromLoc..toLoc`
- Need to know when search has failed
  - Search for 11 : Same sequence until failure

Binary Search Pseudo-Code

```
// Search arr[from..to] for value
// Interpret from > to as an empty range
int recBinSearch(int[] arr, int from, int to, value)
    if (from > to) return -1

    int mid = (to + from)/2
    if (value equals arr[mid])
        return mid
    elseif (value < arr[mid])
        return recBinSearch(arr, from, mid-1, value)
    else
        return recBinSearch(arr, mid+1, to, value)
```

Binary Search : Java Version

- Given an array a[] of positive integers in increasing order, and an integer x, find location of x in a[].
  - Take "indexOf" approach: return -1 if x is not in a[]

```java
protected static int recBinarySearch(int a[], int value,
             int low, int high) {
   if (low > high) return -1;
   else {
      int mid = (low + high) / 2;        //find midpoint
      if (a[mid] == value) return mid;  //first comparison
                                        //second comparison
      else if (a[mid] < value)          //search upper half
      return recBinarySearch(a, value, mid + 1, high);
       else                             //search lower half
          return recBinarySearch(a, value, low, mid - 1);
   }
```

Alternative Pseudo-Code

// Pre: from ≤ to
boolean recBinSearch(int[] arr, int from, int to, value)
    if (from == to)
        if (value equals arr[from]) return true
        else return false

    int mid = ⌊(to + from)/2⌋   *// round down*
    if (value <= arr[mid])
        return recBinSearch(arr, from, mid, value)
    else
        return recBinSearch(arr, mid+1, to, value)

Only two tests in all but base case. Modestly better on average!

Next Episode : Mathematical Induction