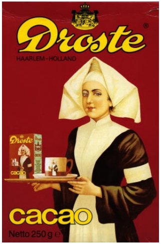
Recursion

CS284



Structure of this week's classes

What is Recursion?

More Examples

Problem Solving with Recursion

Recursion (in Programming)

- The self-referring condition of some datatypes whereby a data element can be decomposed into “smaller” ones of a “similar” nature
- The self-referring condition of some algorithms whereby a programming problem can be decomposed into “smaller” ones of a “similar” nature

Recursive Datatypes

The self-referring condition of some datatypes whereby an element can be decomposed into “smaller” ones of a “similar” nature

- Natural Numbers N:
  - $0 \in N$
  - $1 + n \in N$ if $n \in N$
- Context-free grammar for balanced parentheses:
  - $E \rightarrow EE$
  - $E \rightarrow (E)$
  - $E \rightarrow \oslash$
- Trees: We’ll study them later

Recursive Programs

The self-referring condition of some algorithms whereby a problem can be decomposed into "smaller" ones of a "similar" nature

- Computing the size of a list l
  - If it is empty, return 0
  - If not, compute the size of l without the head element and add 1
- Computing the factorial of a number n
  - If it is zero, return 1
  - If not, compute the factorial of $n - 1$ and multiply by n

Lets take a closer look at the second example

Factorial – Mathematically

$$
\begin{aligned}
0! &\stackrel{def}{=} 1 \\
n! &\stackrel{def}{=} n*!(n-1),\ \ n > 0
\end{aligned}
$$

- The first clause is the base case
- The second clause is the recursive case

$$
\begin{aligned}
5! &= 5*4! \\
&= 5*4*3! \\
&= 5*4*3*2! \\
&= 5*4*3*2*1! \\
&= 5*4*3*2*1*0! \\
&= 5*4*3*2*1*1 \\
&= 120
\end{aligned}
$$

Factorial – Java

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

- Consider `factorial(4)`
- We follow its execution by tracing each recursive call

Stacks and Calls

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

- On the board: `factorial(4)`

Infinite Recursion and Stack Overflow

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

- What happens if we execute `factorial(-2)`?

Infinite Recursion and Stack Overflow

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

- What happens if we execute `factorial(-2)`?
- `Exception in thread "main"java.lang.StackOverflowError`

Some Questions

What's wrong with this program?

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

What about this one?

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

Tail Recursion

- Only one recursive call
- It is the last instruction performed

```
public static int factorialTail(int n, int a) {
  if (n == 0)
     return a;
  else
     return factorialTail(n-1, n*a);
}

public static int factorial(int n) {
     return factorialTail(n,1);
}
```

Computing Factorial Iteratively (i.e. without recursion)

```
public static int factorial_it(int n) {
  int r = 1;
  for (int i=1; i<n+1; i++) {
      r = r * i;
  }
  return r;
}
```

The above code can be obtained automatically from the tail recursive version:

```
public static int factorialTail(int n, int a) {
  if (n == 0)
     return a;
  else
     return factorialTail(n-1, n*a);
}

public static int factorial(int n) {
  return factorialTail(n,1);
}
```

Iteration vs Recursion

- Recursive methods often have slower execution times relative to their iterative counterparts
  - Modern optimizing compilers make this difference often imperceptible
- The overhead for loop repetition is smaller than the overhead for a method call and return
- If it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive method
- The reduction in efficiency does not outweigh the advantage of readable code that is easy to debug

What is Recursion?

More Examples

Problem Solving with Recursion

Fibonacci - In Maths

The Fibonacci numbers are a sequence defined as follows

$$
\begin{aligned}
fib(0) &\stackrel{def}{=} 1 \\
fib(1) &\stackrel{def}{=} 1 \\
fib(n) &\stackrel{def}{=} fib(n-1) + fib(n-2), n > 1
\end{aligned}
$$

$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$

Fibonacci - Implemented as a Recursive Program

```
public static int fibonacci(int n)
{
  if (n<=1)
      return 1;
  else
      return fibonacci(n-1) + fibonacci(n-2);
}
```

Efficiency of `fibonacci`

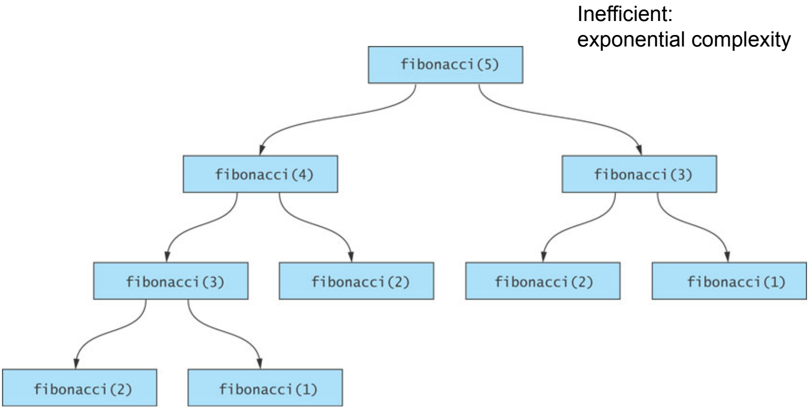
What is the complexity of `fibonacci(n)`?

- Let's draw a picture of the trace of execution of `fibonacci(5)`

Efficiency of `fibonacci`

What is the complexity of `fibonacci(n)`?

- Let's draw a picture of the trace of execution of `fibonacci(5)`



- How can we do better?

Efficient fibonacci

```
private static int ffib(int prevFibo, int currentFibo, int n)
{
  if (n==0)
    return currentFibo;
   else
    return ffib(currentFibo, prevFibo+currentFibo, n-1);
}

public static int ffibonacciStart(int n) {
     return ffib(0, 1, n);
}
```

What is the complexity of `ffibonacciStart(n)`?

- Let's draw a picture of the trace of execution of `ffibonacciStart(5)`

Efficient fibonacci

- Method fibo is an example of tail recursion or last-line recursion
- When recursive call is the last line of the method, arguments and local variables do not need to be saved in the activation frame
- They can be easily implemented using iteration

Functional Programming

- Object-oriented programming languages:
  - Java, Python, etc.
- Functional programming language
  - Haskell, JavaScript, Scala
  - Immutability: does not have the concept of variables, which can be re-assigned a different value;

```
# mutable, OOP style
def function(a):
    a = a + 1
    function_2(a)
```

```
# immutable, FP style
def function(a):
    function_2(a + 1)
```

Why Do We Care about Tail Recursion?

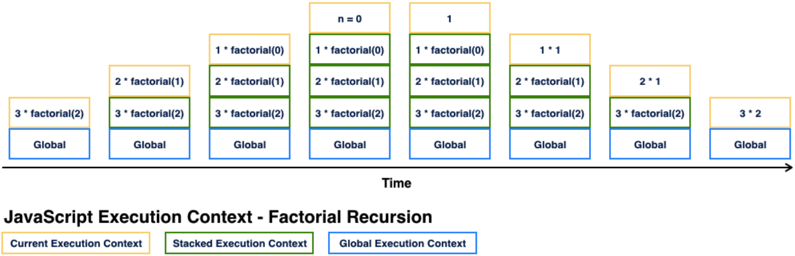
- In functional programming languages such as Haskell, Javascript, there are no imperative loops;
- So any iteration needs to be replaced by recursions;
- That could easily leads to a stack overflow! e.g., sum(10000);
- The key for solving the stack overflow problem is tail recursive elimination (TRE)

Replacing a Recursion with a While Loop

Trampoline - A trampoline function basically wraps our recursive function in a loop

```
const trampoline = fn => (...args) => {
  let result = fn(...args)
  while (typeof result === 'function') {
    result = result()
  }
  return result
}
```

The Stack of Non-Tail Recursion



The Non-Stack of Tail Recursion

- No stack is needed because recursion is now equivalent to iteratively executing the same function;
- Function call in place
  - Calling `result` returns exactly the same result as calling the next `result` except the difference in input parameters;
  - In other words, the same result can be returned by two function call bindings: (`factorialTail`, args1), (`factorialTail`, args2) where the two `factorialTail` are consecutive calls;
  - There is no way (`factorial`, args1), (`factorial`, args2) can return the same result, no matter what args1 and args2 are;

What is Recursion?

More Examples

Problem Solving with Recursion

Towers of Hanoi

- Move the n disks (size 1 through n) to a different peg;
- Disks can be moved only one at a time;
- A larger disk cannot be placed on top of a smaller disk;

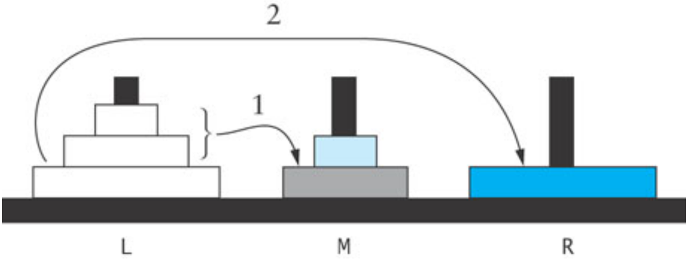
Towers of Hanoi

- Problem input:
  - Number of disks
  - Starting peg
  - Destination peg
  - Temporary peg
- Problem output:
  - List of moves

Algorithm for Towers of Hanoi

Solution to Three-Disk Problem: Move Three Disks from Peg L to Peg R

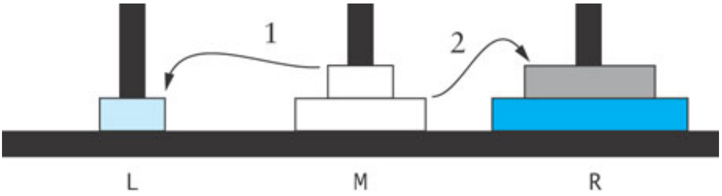
1. Move the top two disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top two disks from peg M to peg R.



Algorithm for Towers of Hanoi

Solution to Two-Disk Problem: Move Top Two Disks from Peg M to Peg R

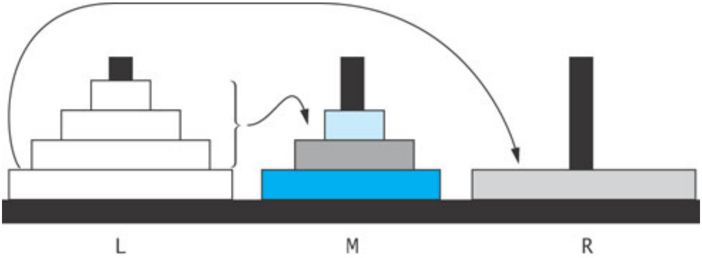
1. Move the top disk from peg M to peg L.
2. Move the bottom disk from peg M to peg R.
3. Move the top disk from peg L to peg R.



Algorithm for Towers of Hanoi

Solution to Four-Disk Problem: Move Four Disks from Peg L to Peg R

1. Move the top three disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top three disks from peg M to peg R.



Recursive Algorithm for Towers of Hanoi – n-Disk Problem

Move n Disks from the Starting Peg to the Destination Peg

- if n is 1
  1. move disk 1 (the smallest disk) from the starting peg to the destination peg
- else
  1. move the top $n-1$ disks from the starting peg to the temporary peg (neither starting nor destination peg)
  2. move disk n (the disk at the bottom) from the starting peg to the destination peg
  3. move the top $n-1$ disks from the temporary peg to the destination peg

Java Code

```java
public class TowersOfHanoi {
   public static String showMoves(int n, char startPeg, char destPeg,
   tempPeg) {

     if (n==1) { // Base case
         return "Move disk 1 from peg " + startPeg
              + " to peg " + destPeg + "\n";
    } else {        // Recursive case
         return showMoves(n-1,startPeg,tempPeg, destPeg)
           + "Move peg " + n + " from peg " + startPeg
           + " to peg " + destPeg + "\n "
           + showMoves(n-1, tempPeg, destPeg, startPeg);
 }
}
}
```

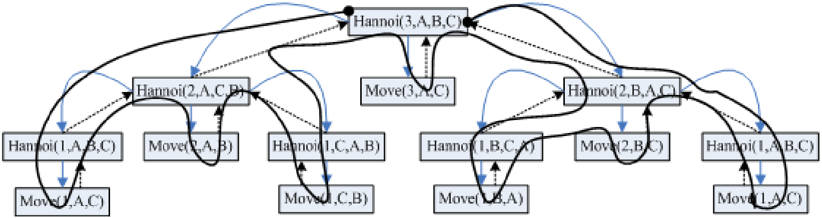
4 disks, (S)ource, (D)estination, (T)emporary

```
Move disk 1 from peg S to peg T
Move peg 2 from peg S to peg D
 Move disk 1 from peg T to peg D
Move peg 3 from peg S to peg T
 Move disk 1 from peg D to peg S
Move peg 2 from peg D to peg T
 Move disk 1 from peg S to peg T
Move peg 4 from peg S to peg D
 Move disk 1 from peg T to peg D
Move peg 2 from peg T to peg S
 Move disk 1 from peg D to peg S
Move peg 3 from peg T to peg D
 Move disk 1 from peg S to peg T
Move peg 2 from peg S to peg D
 Move disk 1 from peg T to peg D
```

Why Recursive Algorithm Preserve the Order?

- i.e., Why are the larger disks always under the smaller ones?
- Every sub-tree at depth i moves disk $1, \cdots, n-i$ from one disk to another, while disk $n-i+1, \cdots, n$ stay still;
  - Sub-tree at depth 1 moves disk $1, \cdots, 2$;
  - Sub-tree at depth 2 moves disk 1;
- Larger disks are always under smaller ones because:
  - Disks $n-i+1, \cdots, n$ are never under disks $1, \cdots, n-i$;
  - Disks $1, \cdots, k$ preserves the order because:
    - Suppose this is true for $1, \cdots, k-1$;
    - Disk k is always under disks $1, \cdots, k-1$;
    - By math induction, disk $1, \cdots, k$ also preserves the order;

Recursive Algorithm



Design of a Binary Search Algorithm

- A binary search can be performed only on an array that has been sorted
- Rather than looking at the first element, a binary search compares the middle element for a match with the target
- A binary search excludes the half of the array within which the target cannot lie
- Base cases?

Design of a Binary Search Algorithm

- A binary search can be performed only on an array that has been sorted
- Rather than looking at the first element, a binary search compares the middle element for a match with the target
- A binary search excludes the half of the array within which the target cannot lie
- Base cases?
  - The array is empty
  - Element being examined matches the target

Design of a Binary Search Algorithm

- if the array is empty
  - return -1 as the search result
- else if the middle element matches the target
  - return the subscript of the middle element as the result
- else if the target is less than the middle element
  - recursively search the array elements before the middle element and return the result
- else
  - recursively search the array elements after the middle element and return the result

Binary Search in an Ordered List – An Example

- Target: Dustin

Caryn	Debbie	Dustin	Elliot	Jacquie	Jonathan	Rich
0	1	2	3	4	5	6

- Initial boundaries of "subarray" to search:
  - The "interval" [first=0,last=6]
  - That is, the entire array

Efficiency of Binary Search

- At each recursive call we eliminate half the array elements from consideration, making a binary search $\mathcal{O}(\log n)$
- An array of 16 would search arrays of length 16, 8, 4, 2, and 1; 5 probes in the worst case
  - $16 = 24$
  - $5 = \log_2 16 + 1$
- A doubled array size would only require 6 probes in the worst case
  - $32 = 25$
  - $6 = \log_2 32 + 1$
- An array with 32,768 elements requires only 16 probes! $(\log_2 32768 = 15)$

Implementation of a Binary Search Algorithm

- Classes that implement the `Comparable` interface must define a `compareTo` method
- Method `obj1.compareTo(obj2)` returns an integer with the following values
  - negative: obj1 $<$ obj2
  - zero: obj1 $==$ obj2
  - positive: obj1 $>$ obj2
- Implementing the `Comparable` interface is an efficient way to compare objects during a search

Implementation of a Binary Search Algorithm

```
private static int binSearch(E[] items, Comparable<E> target, int firs
 if (first > last) {
    return -1; // Base case for unsuccessful search.
 } else {
    int middle = (first+last)/2; // Next probe index
    int compResult = target.compareTo(items[middle]);
    if (compResult == 0) {
        return middle; // Base case for succ. search
     } else if (compResult < 0) {
        return binSearch(items, target, first,middle-1);
    } else {
        return binSearch(items, target, middle+1,last);
    }
 }}}

public static int binSearch(E[] items, Comparable<E> target) {
  return binSearch(items, target, 0, items.length - 1); }
```