# CS1101S Discussion Group Week 4: *Recursion & Higher-order Programming*

Niu Yunpeng

*niuyunpeng@u.nus.edu*

September 5, 2017

# Overview

1. More about recursion
   - From last week
   - Examples
2. Higher-order programming
   - Before we start
   - To understand higher-order programming
   - To use higher-order programming
   - Exercises

# More about Recursion

## A few terms so far

- Primitives/combination/abstraction
- Recursive/iterative function
- Recursive/iterative process

# More about Recursion

## Two approaches

We have two general approaches to solve a really large problem:

- Bottom-up approach: begin with all the smallest units of this problem and combine them together.
- Top-down approach: repeatedly divide a larger problem into several smaller problems and "**wish**" these sub-problems could be solved.

## Two programming styles

- Iteration: the bottom-up approach;
- Recursion: the top-down approach.

# More about Recursion

## To understand recursion

- Use ***substitution model***.

## Substitution model

To use substitution model on understanding a function:

- Evaluate all actual arguments;
- Replace all formal parameters with their actual arguments;
- Apply each statement in the function body (and get the return value);
- Repeat the first 3 steps until done.

# Recursion

## Classical examples of recursion

- Factorial
- Square root
- Power function
- Fibonacci
- Greatest common divisor (GCD)
- Least common multiple (LCM)
- Hanoi tower
- Coin change
- Permutation / combination
- ...

## Examples in Week 3 slides

- Factorial
- Square root
- Power function
- Fibonacci
- Greatest common divisor (GCD)
- Least common multiple (LCM)

# Recursion

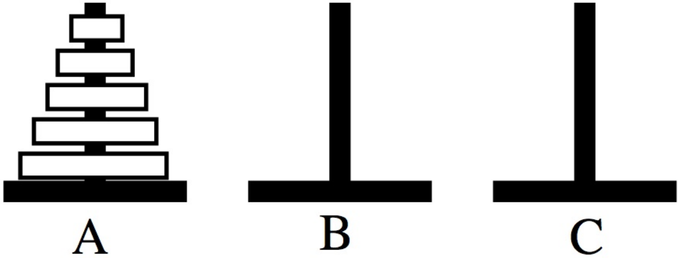
### Examples in Week 4 slides

- Hanoi tower
- Coin change

# Recursion

## Hanoi tower

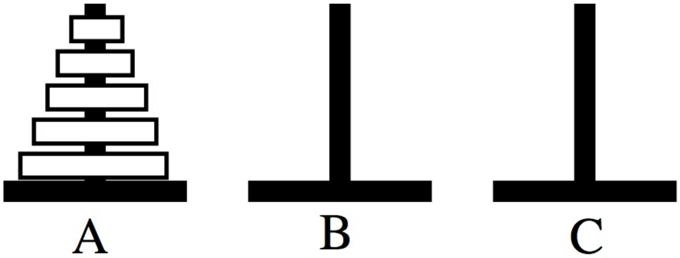
- Given: a tower consisting of disks in increasing size;
- Goal: move all disks from A to B with the help of C;
- Constraint: never put a larger disk on top of a smaller one.



# Recursion

## Recursion for Hanoi tower

- Base case: move 2 disks from A to B with the help of C.
- Scale: $n$ disks.
- Sub-problem: how to solve the problems of $n-1$ disks.



# Recursion

## Hanoi tower

```
function hanoi(size, from, to, extra) {
    if (size === 0) {
        ;
    } else {
        hanoi(size - 1, from, extra, to);
        move_disk(from, to);
        hanoi(size - 1, extra, to, from);
    }
}
```

## An interesting concern

- When I used to be a student in CS1101S, I am confused by

```
display("move from " + from + " to " + to);
```

- Why do we need it?

# Recursion

## Answer

- It is used to print the solution of the hanoi tower in the Source Playground.
- In the online demo for Hanoi tower, they are replaced by the graphic animation.
- Anyway, it is just a way to tell you that, the top disk will be moved from somewhere to elsewhere. Therefore, I make the abstraction

```
move_disk(from, to);
```

# Recursion

## Coin change

- Given: a set of unlimited coins (however limited number of kinds);
- Given also: a specific amount of money in cents;
- Goal: find the number of ways to change this amount into coins.

# Recursion

## Recursion for coin change

- Base case: the amount of money left is 0, which means a valid way to make the changes.
- Scale: the amount of money left *in cents*.
- Sub-problem: to use the same kind or a new kind.

# Recursion

## Recursion for coin change

- Base case: the amount of money left is 0, which means a valid way to make the changes.
- Scale: the amount of money left *in cents*.
- Sub-problem: to use the same kind or a new kind.

# Recursion

## Coin change

```
function coin_change(amount, kind) {
    if (amount === 0) {
        return 1;
    } else if (amount < 0 || kind === 0) {
        return 0;
    } else {
        return coin_change(amount, kind - 1) +
               coin_change(amount - value(kind), kind);
    }
}
```

# Recursion

## Coin change

```
function value(kind) {
    if (kind === 1) {
        return 5;
    } else if (kind === 2) {
        return 10;
    } else if (kind === 3) {
        return 20;
    } else if (kind === 4) {
        return 50;
    } else if (kind === 5) {
        return 100;
    } else {
        display("invalid coin");
    }
}
```

**What is coin change really about?**

- It is to count the number of ways we can solve a problem.
- In fact, it is to count the number of leaves in a *decision tree*.

# Recursion

## What is coin change really about?

- It is to count the number of ways we can solve a problem.
- In fact, it is to count the number of leaves in a *decision tree*.

## What?

- Unbelievable! We are learning part of the simplest form of *machine learning* or so-called *artificial intelligence (AI)*.

## AlphaGo vs Lee Sedol last year

# Recursion

## Recommended modules at SoC

- CS3243(R) Introduction to Artificial Intelligence
- CS3244 Machine Learning
- CS5339 Theory and Algorithms for Machine Learning
- CS5340 Uncertainty Modelling in AI

## Caution

- Very hard modules;
- Need strong mathematical foundations.

# Recursion

## Examples we have learn so far...

- Factorial
- Square root
- Power function
- Fibonacci
- Greatest common divisor (GCD)
- Least common multiple (LCM)
- Hanoi tower
- Coin change

## One thing left...

- Permutation / combination

# Overview

1. More about recursion
   - From last week
   - Examples
2. Higher-order programming
   - Before we start
   - To understand higher-order programming
   - To use higher-order programming
   - Exercises

# Higher-order Programming

## Before we start...

*We need to mention a few things before we start:*

- How to check the correctness of a program;
- Revisit of variable scoping;
- Why we can do higher-order programming in JavaScript?

# Higher-order Programming

**How to check the correctness of a program**

- Invariant
- Termination
  - Base case(s)
  - Finite time/space complexity

## Order of growth exercise from last week

```
function d(n) {
    if (n < 0) {
        return 0;
    } else {
        return d(n / 3);
    }
}

d(10);
```

# Higher-order Programming

## Revisit of variable scoping

- **System** functions or variables are visible everywhere.
- A function or variable is visible within the closest surrounding curly braces where it is declared. Or it will be visible in the whole program if none (top-level varaibles, or **global variables**).
- **Formal parameters** are visible within the function body to which it belongs.

# Higher-order Programming

## Exercises of variable scoping

- Find out the output of each program, and
- Explain the reason.

## Importance

- Friday Test - Analytical Reading 1

## Exercise 1

```
var x = 5;

function f(x) {
    return x;
}

f(3);
```

## Exercise 2

```
var x = 5;

function f(x) {
    function g() {
        return x;
    }

    return g();
}

f(x);
```

# Higher-order Programming

**Before we move on...**

- We claimed that "Pre-declared built-in functions or variables are visible everywhere."
- So, what are "*Pre-declared built-in functions or variables*"?

# Higher-order Programming

## Core built-in functions

- `display`
- `alert`
- `prompt`
- `parseInt`

## A few keywords

- `undefined`
- `Infinity`
- `-Infinity`
- `NaN`

# Higher-order Programming

## Mathematical library - functions

- math_abs(x)
- math_sin(x) math_cos(x) math_tan(x)
- math_asin(x) math_acos(x) math_atan(x) math_atan2(y, x)
- math_floor(x) math_ceil(x) math_round(x)
- math_max(a, b, ...)  math_min(a, b, c, ...)
- math_pow(x, y) math_exp(x)
- math_sqrt(x)
- math_log(x) math_log10(x) math_log2(x)

# Higher-order Programming

## Mathematical library - constants

- math_E
- math_PI
- math_SQRT2
- math_SQRT1_2
- math_LN10
- math_LN2

# Higher-order Programming

## Things...

- Variables can be functions.
- Parameters can be functions.
- Return values can be functions.

## Result...

- That's all about higher-order programming.

## Original version

```
function fact(n) {
    // By definition, the factorial of 0 is 1.
    return n === 0 ? 1 : fact(n - 1) * n;
}
```

## Notice

- This version gives rise to a recursive process.

# Higher-order Programming

## Abstract the multiplication

```
function make_multiplier(x) {
    return function(y) {
        return x * y;
    };
}

var multiply_by_4 = make_multiplier(4);
multiply_by_4(5);
```

## Using the abstraction of multiplication

```
function fact(n) {
    if (n === 0) {
        return 1;
    } else {
        return (make_multiplier(n))(fact(n - 1));
    }
}
```

# Higher-order Programming

## Abstract the sub-problem relationship

```
function product(term, next, upper, lower) {
    if (upper <= lower) {
        return 1;
    } else {
        return term(upper) *
               product(term, next, next(upper), lower);
    }
}
```

# Higher-order Programming

## Abstract the relationship again

```
function product(term, next, terminate, now) {
    if (terminate(now)) {
        return 1;
    } else {
        return term(now) *
               product(term, next, terminate, next(now));
    }
}
```

# Higher-order Programming

## Think about it carefully...

Three key aspects for a recursive function:

- Base case(s)
- Scale
- Sub-problem(s)

Three functions as parameters for `product`:

- `terminate`
- `term`
- `next`

# Higher-order Programming

## Using the abstraction for sub-problem relationship

```
function fact(n) {
    return product(function(x) { return x; },
                   function(x) { return x - 1; },
                   function(x) { return x <= 0; },
                   n);
}
```

# Higher-order Programming

## What about this?

- $1 + 2 + \cdots + n$
- $1 \times 2 \times \cdots \times n$
- For these two different series, what is in common?

# Higher-order Programming

## Abstract the multiplication and sub-problem relationship

```
function accum(term, next, terminate, operation, now) {
    if (terminate(now)) {
        return 1;
    } else {
        return operation(term(now),
                         accum(term, next, terminate,
                               operation, next(now)));
    }
}
```

# Higher-order Programming

## Once again

```
function accum(term, next, terminate, oper, base, now) {
    if (terminate(now)) {
        return base();
    } else {
        return oper(term(now),
                    accum(term, next, terminate, oper,
                          base, next(now)));
    }
}
```

## Think about it...

- What changes?

# Higher-order Programming

## Using everything together

```
function fact(n) {
    return accum(function(x) { return x; },
                 function(x) { return x - 1; },
                 function(x) { return x <= 0; },
                 function(x, y) { return x * y; },
                 function() { return 1; },
                 n);
}
```

## Think about it...

- What changes?

# Higher-order Programming

### Your task today...

- Does this function gives rise to a recursive or iterative process?
- If it gives rise to a recursive process, can you change it into an iterative process?

# Higher-order Programming

## Notice

- In the following slides, you are going to see a few problems.
- They are selected from past year papers.

# Higher-order Programming

## Exercise 1

You are given the function below called `strict`. Consider a restricted version of Source, in which each function is only allowed to have at most 1 parameter. Find out how to define `strict` under this constraint.

```
function strict(a, b, c) {
    return a * b + c;
}
```

# Higher-order Programming

## Exercise 2

```
function plus_one(x) {
    return x + 1;
}

function trans(func) {
    return function(x) {
        return 2 * func(x * 2);
    };
}

function twice(func) {
    return function(x) {
        return func(func(x));
    };
}
```

# Higher-order Programming

## Exercise 2

Given the three functions in the last slide, try to find out the output of the following programs:

- `((twice(trans))(plus_one))(1);`
- `((twice(trans(plus_one))))(1);`

## Exercise 3

- According to the substitution model of execution, a process can be said to *exhaust all time resources* if it keeps evaluating and never reaches any result value.
- Also, a process can be said to exhaust all space resources if it keeps growing while it evaluates sub-expressions, i.e. the number of sub-expressions and deferred operations will keep growing.

### Exercise 3

For the following programs, find out whether they will exhause time or space resources (or both):

1) Will it exhaust time/space resources or both?

```
function loop(x) {
    return loop(x);
}
loop(0);
```

# Higher-order Programming

## Exercise 3

For the following programs, find out whether they will exhause time or space resources (or both):

2) Will it exhaust time/space resources or both?

```
function loop2(x) {
    return loop2(loop2(x));
}
loop2(0);
```

## Exercise 3

For the following programs, find out whether they will exhause time or space resources (or both):

3) Will it exhaust time/space resources or both?

```
function recur(x) {
    return x(x);
}
recur(function(x) { return x(x(x)); });
```

Let's discuss them now.

# The End