

CS1101S Studio Session Week 3: *Abstraction, Recursion & Order of Growth*

Niu Yunpeng

niuyunpeng@u.nus.edu

August 28, 2018

Studio name

- Hopeless group
- A+ Participation
- Recycle_Bin
- CloudRiders
- Computeers
- ...

How is everything so far?

- Lectures
- Piazza
- Source Academy (mission + quest + path)
- Studios
- ...

Before We Start

How is everything so far?

- Lectures
- Piazza
- Source Academy (mission + quest + path)
- Studios
- ...

And ...

- Have you told *really funny* jokes?

1 Review, abstraction & functions

- From last week
- A good abstraction
- Function execution

2 Recursion

- To understand recursion
- To use recursion
- Examples
- Exercises

3 Order of growth

- To understand order of growth
- To use order of growth
- Exercises

Components of a programming language

- Primitives:
The smallest constituent unit of a programming language.
- Combination:
Ways to put primitives together.
- Abstraction:
The method to simplify the messy combinations.

Means of abstraction

- To abstract data: use naming;
- To abstract procedures: use functions.
- Usually, naming and functions are combined together.

Define a function

```
const pi = 3.1415926535;

function square(x) {
    return x * x;
}

function circle(x) {
    return pi * square(x);
}
```

Apply a function

- The area of a circle with a radius of 3: `circle(3)`;
- The area of a circle with a radius of 5.6: `circle(5.6)`;

What makes a good abstraction?

- Modularity
- Readability
- Reusability
- Maintainability

What makes a good abstraction?

- Modularity:
Separate multiple steps (and sub-steps).
- Readability:
Easy for others to read and understand.
- Reusability:
Provide a generic interface to be used commonly.
- Maintainability:
Convenient to debug, refactor and deploy.

Basic terminologies about function

- In general, there are two steps to use a function:
 - Function definition (or declaration)
 - Function application (or calling)
- A function definition consists of
 - Function name
 - Formal parameter list
 - Function body
- A function application consists of
 - Function name
 - Actual argument list

Function Execution

Example

```
function add(x, y) {  
    return x + y;  
}  
  
add(1, 2);
```

Explanation

- add is the function name
- x and y are formal parameters
- 1 and 2 are actual arguments

Order of reduction

To determine when the value of operands will be evaluated:

- Applicative order reduction
 - Evaluate the operands whenever being applied to an operator
 - *Evaluate the arguments and then apply*
- Normal order reduction
 - Evaluate the operands only if their values are needed
 - *Fully expand and then reduce*
 - Follow *Principle of Last Commitment*

Order of reduction (*continued ...*)

- Source & JavaScript implements applicative order reduction
- The two ways of reduction are equivalent as long as substitution model does not break.
 - Unfortunately, substitution model will breaker later in the semester.
- Eventually, *environment model* will come up on stage.

Substitution model - what happens when a function is called?

- Evaluate each actual argument from left to right
- Bind the value of each actual argument to the corresponding formal parameter
- Execute each statement in the function body in a top-down manner
- Go back to the caller when meeting `return` statement

Overview

1 Review, abstraction & functions

- From last week
- A good abstraction
- Function execution

2 Recursion

- To understand recursion
- To use recursion
- Examples
- Exercises

3 Order of growth

- To understand order of growth
- To use order of growth
- Exercises

Recursion & iteration

When we need to solve a very large problem, in general, we will have two approaches:

- Bottom-up approach
- Top-down approach

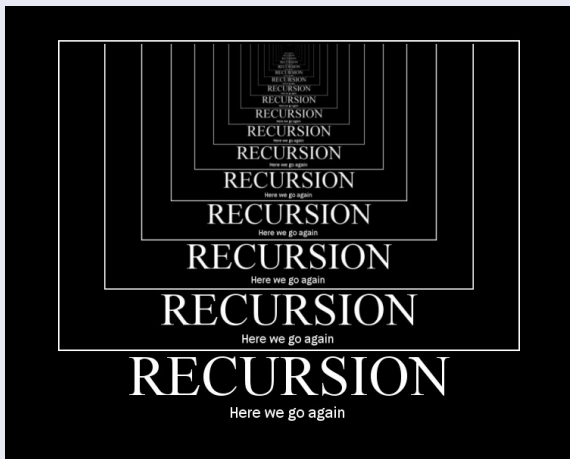
Recursion & iteration

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

Notice

- We will start with and mainly focus on *recursion*
 - The top-down approach.

Recursion is beautiful



Recursion is beautiful



Recursion

Recursion is beautiful



How to understand recursion?

- Use ***substitution model***.
- “Substitution” means two “replace”:
 - Replace a function call by its function body;
 - Replace formal parameters by actual arguments.

Recursive function

- Any function that calls itself (directly or indirectly) is called a recursive function.

To write recursive functions correctly

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

To write recursive functions correctly

- Base case(s):
the largest small problems that can be solved without recursion;
- Scale:
the measurement of the size of the problem;
- Sub-problem(s):
the relationship between one larger problem and all of its smaller sub-problems.

Example of a recursive function

```
function stackn(n, pic) {  
    return n === 1 ? pic  
        : sf(1 / n, pic, stackn(n - 1, pic));  
}
```

To write this recursive function correctly

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

Example of a recursive function

```
function stackn(n, pic) {  
    return n === 1 ? pic  
        : sf(1 / n, pic, stackn(n - 1, pic));  
}
```

To write this recursive function correctly

- Base case(s): $n = 1$;
- Scale: n ;
- Sub-problem(s): Divide the area into n pieces, where the current level takes the top one piece, the rest takes $n - 1$ pieces.

Another example

```
// Calculates the factorial of a non-negative integer n.  
// Pre-condition: The input of n is a non-negative integer.  
function fact(n) {  
    // By definition, the factorial of 0 is 1.  
    return n === 0 ? 1 : fact(n - 1) * n;  
}
```

To write this recursive function correctly

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

Another example

```
// Calculates the factorial of a non-negative integer n.  
// Pre-condition: The input of n is a non-negative integer.  
function fact(n) {  
    // By definition, the factorial of 0 is 1.  
    return n === 0 ? 1 : fact(n - 1) * n;  
}
```

To write this recursive function correctly

- Base case(s): $n = 0$;
- Scale: n ;
- Sub-problem(s): The factorial of n is the product of the factorial of $n - 1$ and itself.

How to understand

- Use substitution model!

Interpretation

```
fact(5) = fact(4) * 5
        = fact(3) * 4 * 5
        = fact(2) * 3 * 4 * 5
        = fact(1) * 2 * 3 * 4 * 5
        = fact(0) * 1 * 2 * 3 * 4 * 5
        = 1 * 1 * 2 * 3 * 4 * 5
        = 1 * 2 * 3 * 4 * 5
        = 2 * 3 * 4 * 5
        = 6 * 4 * 5
        = 24 * 5
        = 120
```

Deferred operation

- The operations that have to be suspended because they need to wait for some other operations to finish first.
- In order to suspend them, we need to remember them in the memory, which is a waste of space.

Why do they occur?

- For recursive functions, if the execution of the recursive function call is not the only and last step, all of the other steps have to wait for it, then they will become deferred operations.

Recursive & iterative process

- Execution of a recursive function may give rise to either:
 - Recursive process: those with deferred operations.
 - Iterative process: those without deferred operations.

Task today

- ***Turn every recursive process into an iterative one!***

To turn a recursive process into an iterative one

- Use a variable to remember the operation that we have to wait for;
- Add a function parameter so that we can keep track of that variable;
- Wrap with an outer function so that the interface does not change (the user does not see any additional parameter).

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 that we are going to cover today...

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

Factorial

- In mathematics, the factorial of a non-negative integer n , denoted by $n!$, is the product of all positive integers less than or equal to n .
- According to the definition of empty product, the factorial of 0 is 1.
- Symbolically, we have

$$n! = \prod_{k=1}^n k, \forall n \geq 0 \text{ and } 0! = 1$$

Factorial 1

```
// This version gives rise to a recursive process.  
function fact(n) {  
    // By definition, the factorial of 0 is 1.  
    return n === 0 ? 1 : fact(n - 1) * n;  
}
```

Think about it...

- Correctness?
- Time/space complexity?

Factorial 2

```
// This version gives rise to an iterative process.  
function fact(n) {  
  function iter(x, result) {  
    return x === 0 ? result : iter(x - 1, result * x);  
  }  
  
  return iter(n, 1);  
}
```

Think about it...

- Correctness?
- Time/space complexity?

Factorial 3

```
// This version gives rise to an iterative process.  
function fact(n) {  
  function iter(x, result) {  
    return x === n ? result * x  
                  : iter(x + 1, result * x);  
  }  
  
  return n === 0 ? 1 : iter(1, 1);  
}
```

Think about it...

- Correctness?
- Time/space complexity?

Factorial 4

```
// This version gives rise to an iterative process.  
function fact(n) {  
  function iter(x, result) {  
    return x > n ? result : iter(x + 1, result * x);  
  }  
  
  return iter(1, 1);  
}
```

Think about it...

- Correctness?
- Time/space complexity?

Square root - Newton's method

In order to find an approximation of \sqrt{x} ,

- Make a guess of y ;
- Calculate the average of y and x/y ;
- Keep improving the guess until it is good enough.

Hint

- How to make the initial guess?
- How to improve the guess?
- What is “good enough”?

Hint

- The initial guess: 1;
- To improve the guess: calculate the average of y and x/y ;
- “Good enough”: set a threshold value, like $\frac{1}{10000}$.

Square root

```
// Calculates the square root of an integer.  
function sqrt(x) {  
  function iter(guess) {  
    var improved = (guess + x / guess) / 2;  
    var diff = math_abs(improved - guess);  
  
    return diff < 1 / 10000 ? guess : iter(improved);  
  }  
  
  return iter(1);  
}
```

Think about it...

- Correctness?

Power function - exponentiation

- Exponentiation is a mathematical operation, written as b^n , involving two numbers, the base b and the exponent n .
- Here, at first, we only consider the case that the exponent is a natural number and the base is a real number.
- Symbolically, we have

$$b^n = \underbrace{b \times \cdots \times b}_n, \forall b \in \mathbb{R} \text{ and } \forall n \in \mathbb{N}$$

- Notice that 0^0 is not defined mathematically.

Power function 1

```
// This version gives rise to a recursive process.  
function power(b, n) {  
    return n === 0 ? 1 : b * power(b, n - 1);  
}
```

Think about it...

- Correctness?
- Time/space complexity?

Power function 2

```
// This version gives rise to an iterative process.  
function power(b, n) {  
  function iter(k, result) {  
    return k === 0 ? result : iter(k - 1, result * b);  
  }  
  
  return iter(n, 1);  
}
```

Think about it...

- Correctness?
- Time/space complexity?

Power function 3

```
// This version also gives rise to an iterative process.  
function power(b, n) {  
  function iter(k, result) {  
    return k === n ? result : iter(k + 1, result * b);  
  }  
  
  return iter(0, 1);  
}
```

Think about it...

- Correctness?
- Time/space complexity?

Fast power 1

```
// This version gives rise to a recursive process.  
function fast_power(b, n) {  
    return n === 0 ? 1  
           : (n % 2 === 0 ? fast_power(b * b, n / 2)  
                    : b * fast_power(b, n - 1));  
}
```

Think about it...

- Correctness?
- Time/space complexity?

(Not really) fast power 2

```
// This version gives rise to an iterative process.  
function fast_power(b, n) {  
  function iter(k, res) {  
    return k === 0 ? res  
           : (k % 2 === 0 ? iter(k / 2, res * res)  
                  : iter(k - 1, res * b));  
  }  
  return iter(n, 1);  
}
```

Think about it...

- What's wrong with it? Speed or correctness or ...?

(Not really) fast power 3

```
// This version gives rise to an iterative process.  
function fast_power(b, n) {  
  function iter(k, res) {  
    return k === 0 ? res  
           : (k % 2 === 0 ? iter(k / 2, res * res)  
                  : iter(k - 1, res * b));  
  }  
  return iter(n, b);  
}
```

Think about it...

- What's wrong with it? Speed or correctness or ...?

Fast power 4

```
// This version also gives rise to an iterative process.  
function fast_power(b, n) {  
  function iter(b, k, res) {  
    return k === 0 ? res  
      : (k % 2 === 0 ? iter(b * b, k / 2, res)  
        : iter(b, k - 1, res * b));  
  }  
  return iter(b, n, 1);  
}
```

Think about it...

- Time/space complexity?

Why is “*fast_power*” fast?

- In normal power function, we iterate through $1\dots n$. Thus, we have to result in a linear order of growth.
- In fast power function, we make use of the relationship $b^n = (b^2)^{n/2}$. Thus, we can skip some of $1\dots n$ and achieve a logarithmic order of growth.

To implement the “**skip**” in “*fast_power*”

- *Binary search approach*: use the sequence $n, \frac{n}{2}, \frac{n}{4}, \dots, 2, 1$.

What about the other direction?

- *Aggressive cow approach*: use the sequence $1, 2, \dots, \frac{n}{4}, \frac{n}{2}, n$.

(Not so) fast power 1

```
// This version gives rise to a recursive process.
function not_so_fast_power(b, n) {
  function part_iter(unit, k) {
    if (k === n) {
      return unit;
    } else {
      return k * 2 <= n ? part_iter(unit * unit, k * 2)
        : b * part_iter(unit, k + 1);
    }
  }
  return part_iter(b, 1);
}
```

Think about it...

- Problem?

(Not so) fast power 2

```
// This version gives rise to an iterative process.
function not_so_fast_power(b, n) {
  function iter(unit, k, res) {
    if (k === n) {
      return unit * res;
    } else {
      return k * 2 <= n ? iter(unit * unit, k * 2, res)
        : iter(unit, k + 1, res * b);
    }
  }
  return iter(b, 1, 1);
}
```

Think about it...

- Time/space complexity?

Fibonacci

- In mathematics, the Fibonacci numbers are the numbers in the following integer sequence, called the *Fibonacci sequence*, and characterized by the fact that every number after the first two is the sum of the two preceding ones:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

- It is a modern convention that the Fibonacci sequence starts from 0 (rather than 1). So do not be confused by some external resources.
- Symbolically, we have

$$fibo(n) = \begin{cases} n, & \text{for } n \leq 1 \\ fibo(n-1) + fibo(n-2), & \text{for } n \geq 2 \end{cases}$$

Fibonacci 1

```
// This version gives rise to a recursive process.  
function fibo(n) {  
    return n <= 1 ? n : fibo(n - 1) + fibo(n - 2);  
}
```

Think about it...

- Correctness?
- Time/space complexity?

Fibonacci 2

```
// This version gives rise to an iterative process.  
function fibo(n) {  
  function iter(x, last1, last2) {  
    return x > n ? last1  
              : iter(x + 1, last1 + last2, last1);  
  }  
  
  return n <= 1 ? n : iter(2, 1, 0);  
}
```

Think about it...

- Correctness?
- Time/space complexity?

Fibonacci 3

```
// You will learn this formula in later chapters of CS1231.  
function fibo(n) {  
  var sqrt5 = Math.sqrt(5);  
  var ratio = (sqrt5 + 1) / 2;  
  var term = Math.pow(ratio, n) - Math.pow(ratio, -n);  
  
  return term / sqrt5;  
}
```

Think about it...

- Correctness?
- Time/space complexity?
- Tradeoff? Is this meaningful?

Fibonacci 4

According to Linear Algebra, Fibonacci relationship is:

$$\begin{bmatrix} f(n) & f(n-1) \\ f(n-1) & f(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

Therefore, in order to find out the solution of Fibonacci sequence, we only need to deal with the power of a matrix.

Caution

- You need solid background knowledge in linear algebra to understand this solution.

A few steps to consider

- How to represent a matrix
- How to implement matrix multiplication
- How to compute the power of matrix

How to do these 3 steps

- Let's consider them one by one.

Step 1

- How to represent a matrix

Answer

- The normal way to represent a matrix is to use a 2^{nd} dimensional array. However, you will learn array in CS1101S later, not now.
- Since the matrix for Fibonacci is only 2×2 , we can just simply use four variables.
- Like this:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \Leftrightarrow \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Step 2

- How to implement matrix multiplication

Hint

- Matrix multiplication is defined mathematically like this:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

- We should follow this definition.

Step 3

- How to compute the power of matrix

Hint

- Last week, we said that “Combination should provide a generic interface (a convention) so that we can (theoretically) combine everything as long as the convention is not broken.”
- Therefore, as long as we can compute the power of a number, we can compute the power of a matrix in a similar way.

The situation now

- We know how to compute the power of a number just now.
- What if we need to compute the power of a matrix?

Fibonacci 4

```
// This version uses the normal power method.
function fibo(n) {
  function iter(k, a, b, c, d) {
    if (k === n) {
      return a;
    } else {
      return iter(k + 1, a + b, a, c + d, c);
    }
  }

  return n <= 1 ? n : iter(2, 1, 1, 1, 0);
}
```

Think about it...

- Correctness?

Fibonacci 5

According to Linear Algebra, Fibonacci relationship is:

$$\begin{bmatrix} f(n) & f(n-1) \\ f(n-1) & f(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

- We know the fast method to compute the power of a number.
- Your task today: How to adapt “fast_power” on matrix?
- Time/space complexity?

Greatest common divisor (GCD)

- In mathematics, the greatest common divisor (GCD) of two or more integers, which are not all zero, is the *largest positive integer* that divides each of the integers.
- Here, we only consider the case of GCD of two integers.
- Symbolically, we have:

$$\forall a, b \in \mathbb{Z}, d = \gcd(a, b) \Leftrightarrow \begin{cases} d \mid a \wedge d \mid b \\ \forall c \in \mathbb{Z}, c \mid a \wedge c \mid b \Leftrightarrow c \leq d \end{cases}$$

Notice

- GCD is also known as the greatest common factor (GCF), highest common factor (HCF), greatest common measure (GCM), or highest common divisor (HCD).

GCD - the ancient Chinese algorithm

- Described in the Chapter 1 of *Nine Chapters on the Mathematical Art*. Also called “*geng xiang jian sun shu*”.
- Based on the following relationship

$$\gcd(a, b) = \gcd(a - b, b), \text{ assuming that } a > b$$

Self reading

- The concept of primes and the algorithm for counting the greatest common divisor in Ancient China. *Shaohua Zhang*. Click [here](#) to read.

GCD 1

```
function gcd(a, b) {  
    if (a === b) {  
        return a;  
    } else {  
        return a > b ? gcd(a - b, b)  
                    : gcd(a, b - a);  
    }  
}
```

Think about it...

- Correctness?
- Time/space complexity?

GCD - the Euclidean algorithm

- Described in Book 7 and 10 of *Euclid's Elements*, also discovered independently in ancient China and India.
- Based on the following relationship

$$\text{gcd}(a, b) = \text{gcd}(b, r), \text{ where } r \text{ is the remainder of } a/b$$

Significance

"[The Euclidean algorithm] is the granddaddy of all algorithms, because it is the oldest non-trivial algorithm that has survived to the present day."

Donald Knuth, *The Art of Computer Programming*, 2nd edition (1981), Vol. 2: Seminumerical Algorithms, p. 318.

GCD 2

```
// Pre-condition: a > b.  
function gcd(a, b) {  
    return b === 0 ? a  
        : gcd(b, a % b);  
}
```

Think about it...

- Correctness?
- Time/space complexity?

Which one is faster?

```
function gcd(a, b) {  
    if (a === b) {  
        return a;  
    } else {  
        return a > b ? gcd(a - b, b)  
                    : gcd(a, b - a);  
    }  
}
```

```
// Pre-condition: a > b.  
function gcd2(a, b) {  
    return b === 0 ? a  
                : gcd(b, a % b);  
}
```

Hint - Which one is faster?

- Compare the actual performance of the computer when doing division and subtraction.

GCD - the Stein's algorithm

- First published by the Israeli physicist and programmer Josef Stein in 1967, also known as the binary GCD algorithm.
- Based on the following relationship

$$\text{gcd}(k \cdot a, k \cdot b) = k \cdot \text{gcd}(a, b)$$

Significance

- The Stein's algorithm fixes the vital fallback of the Euclidean algorithm. It avoids the inefficiency of integer division and remainder.
- Has significant use in cryptography.
- According to Donald Knuth, however, it may have been known in 1st century ancient China.

GCD 3

```
// Assume we do not know the relationship between a and b.
function gcd(a, b) {
    if (a === 0 || b === 0 || a === b) {
        return a;
    } else if (a % 2 === 0 && b % 2 === 0) {
        return 2 * gcd(a / 2, b / 2);
    } else if (a % 2 === 0) {
        return gcd(a / 2, b);
    } else if (b % 2 === 0) {
        return gcd(a, b / 2);
    } else {
        return gcd(Math.abs(a - b), Math.min(a, b));
    }
}
```

Is this really faster?

- The Stein's algorithm seems to combine GCD 1 and GCD 2.
- It still uses division and remainder, why is it faster?

Patch to Stein's algorithm

- Look at the next version, *GCD 4*.

GCD 4

```
// Assume we do not know the relationship between a and b.
function gcd(a, b) {
    if (a === 0 || b === 0 || a === b) {
        return a;
    } else if (a % 2 === 0 && b % 2 === 0) {
        return gcd(a >> 1, b >> 1) << 1;
    } else if (a % 2 === 0) {
        return gcd(a >> 1, b);
    } else if (b % 2 === 0) {
        return gcd(a, b >> 1);
    } else {
        return gcd(Math.abs(a - b), Math.min(a, b));
    }
}
```


Reason

- Bit operator (*right-shift*) achieves the same result as division by 2.
- However, it is less expensive than division.
 - Due to native support in CPU instruction sets.

Caution

- This is just a demonstration. It may not be true in modern CPUs.
- You need CS2100/EE2024 knowledge to understand this.

More about GCD 4

- Some older versions of Javascript may not support bit operator.
- If you really want to have a try, go to your browser's console.
- Akhavi and Vallee proved that, in theory, binary GCD can be about 60% more efficient (in terms of the number of bit operations) on average than the Euclidean algorithm.

Least common multiple (LCM)

- In mathematics, the least common multiple (LCM) of two or more integers, which are all not zero, is the *smallest positive integer* that is divisible by each of the integers.
- Here, we only consider the case of LCM of two integers.
- Symbolically, we have:

$$\forall a, b \in \mathbb{Z}, d = \text{lcm}(a, b) \Leftrightarrow \begin{cases} a \mid d \wedge b \mid d \\ \forall c \in \mathbb{Z}, a \mid c \wedge b \mid c \Leftrightarrow c \geq d \end{cases}$$

Notice

- LCM is also known as the lowest common multiple (also LCM), or smallest common multiple (SCM).

LCM 1

```
// Assume we do not know the relationship between a and b.
function lcm(a, b) {
  function iter(x, y) {
    if (x === y) {
      return x;
    } else {
      return x > y ? iter(x, y + b)
                  : iter(x + a, y);
    }
  }
  return iter(a, b);
}
```

Think about it...

- Correctness?

LCM 2

```
function lcm(a, b) {  
    return a * b / gcd(a, b);  
}
```

To understand...

$$lcm(a, b) = \frac{|a \cdot b|}{gcd(a, b)}$$

Think about it...

- Time/sapce complexity?

Exercises of recursion

- Digit sum
- Multiple of 9
- Palindrome
- Super-fibonacci
- Staircases

Your task today

- Try to answer all of these problems.
- Try to give both the recursive and iterative version.

1. Digit sum

Given a non-negative integer, find the sum of all its digits. Your function name should be `sum_of_digits`.

Examples

- `sum_of_digits(0)` returns 0.
- `sum_of_digits(12965)` returns 23.
- `sum_of_digits(70263)` returns 18.

2. Multiple of 9

A number is a multiple of 9 if and only if its sum_of_digits is a multiple of 9. Using this fact, create a function to check whether a non-negative number is a multiple of 9. Notice that you **MUST NOT** use `% 9`. Your function name should be `is_multiple_of_9`.

Examples

- `is_multiple_of_9(0)` returns true.
- `is_multiple_of_9(12965)` returns false.
- `is_multiple_of_9(70263)` returns true.

3. Palindrome 1

Given a non-negative integer, when the order of all its digits is reversed, we get its palindrome. Create a function to find the palindrome. Your function name should be `palindrome`.

Notice: the return value of your function must be integer (rather than string), you are also not allowed to use explicit data type conversion.

Examples

- `palindrome(0)` returns 0.
- `palindrome(15687)` returns 78651.
- `palindrome(32523)` returns 32523.

4. Palindrome 2

Given a non-negative integer, it is palindromic if its palindrome and itself is equal. Create a function to check whether a number is palindromic. Your function name should be `is_palindromic`.

Examples

- `is_palindromic(0)` returns `true`.
- `is_palindromic(15687)` returns `false`.
- `is_palindromic(32523)` returns `true`.

5. Super-fibonacci

Given the following recurrence relationship,

$$f(n) = \begin{cases} 2 \cdot n + 1, & \text{for } n \leq 2 \\ 3 \cdot f(n-1) + 2 \cdot f(n-2) + f(n-3), & \text{for } n > 3 \end{cases}$$

create a function to find the n^{th} term. Your function name should be `calculate_f`.

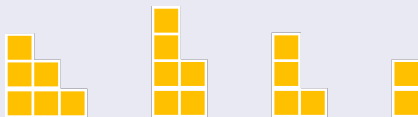
Examples

- `calculate_f(0)` returns 1.
- `calculate_f(1)` returns 3.
- `calculate_f(3)` returns 22.

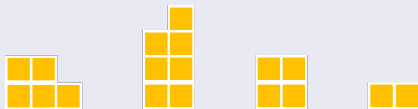
6. Staircase

We can use blocks to create a staircase. However, not every combination of blocks can become a staircase. To build a staircase, the height of each column should be **strictly** descending (from left to right).

Following are some examples of valid staircases:



Following are some examples of invalid staircases:



6. Staircase

Create a function to count the number of possible valid staircases using a given number of blocks. Notice that *all* of the blocks *have to be used*. You can assume the input is positive. Your function name should be `staircase`.

Examples

- `staircase(1)` returns 1.
- `staircase(2)` returns 1.
- `staircase(3)` returns 2.
- `staircase(4)` returns 2.
- `staircase(5)` returns 3.
- `staircase(6)` returns 4.

Overview

- 1 Review, abstraction & functions
 - From last week
 - A good abstraction
 - Function execution
- 2 Recursion
 - To understand recursion
 - To use recursion
 - Examples
 - Exercises
- 3 Order of growth
 - To understand order of growth
 - To use order of growth
 - Exercises

What is order of growth?

- *Purpose*: we are trying to find a rough measure of the resources (time and/or space) used by a computational process (a program).
- *Approach*: Use a mathematical function to describe how the amount of resources consumed grows along with the scale of the problem.
- *Abstraction*: We do not need a precise (but complex) expression of that function. Instead, we want a simple (but useful) expression to describe its limiting behavior.
In other words, we want to find an *approximation* of that function.

Big theta, oh, omega

- Big theta Θ : tight bound (both sides);
- Big oh O : upper bound;
- Big omega Ω : lower bound.

Formal definition

- The function r has an order of growth $\Theta(g(n))$ if there exists positive constants k_1 and k_2 and a number n_0 such that

$$0 \leq k_1 \cdot g(n) \leq r(n) \leq k_2 \cdot g(n), \forall n > n_0$$

- The function r has an order of growth $O(g(n))$ if there exists a positive constant k and a number n_0 such that

$$0 \leq r(n) \leq k \cdot g(n), \forall n > n_0$$

- The function r has an order of growth $\Omega(g(n))$ if there exists a positive constant k and a number n_0 such that

$$0 \leq k \cdot g(n) \leq r(n), \forall n > n_0$$

Order of Growth

Small oh, omega (*optional ...*)

- Small oh o : non-tight upper bound;
- Small omega ω : non-tight lower bound.

Caution

- Please ignore o and ω if they make you confused.
 - Additional materials, non-examinable.
- For more, see CLRS 3rd edition, page 50 - 51.

More formal definition (*optional ...*)

- The function r has an order of growth $o(g(n))$ if for any positive constant k , there exists a number n_0 such that

$$0 \leq r(n) < k \cdot g(n), \forall n > n_0$$

- The function r has an order of growth $\omega(g(n))$ if for any positive constant k , there exists a number n_0 such that

$$0 \leq k \cdot g(n) < r(n), \forall n > n_0$$

How to find order of growth

You only need to follow two steps:

- Analyse the recurrence relationship.
- Calculate the asymptotic notation of that relationship.

Order of Growth

Order of growth in CS1101S...

For the interest of examination-oriented or grade-oriented, you need enough exercises on such questions.

How to tackle this kind of problems easily

- Remember a few commonly-used asymptotic notation:

$$1, \log n, n, n \cdot \log n, n^k, 2^n, \dots$$

- For polynomials, only consider the term with the highest order (ignore minor terms).
- Always neglect constants and set the coefficient as 1.

Exercises

- In the following slides, you are going to see a few programs.
- Use whatever method you have learnt (or guess), find out their order of growth in time and space.

Exercise 1

```
function a(n) {  
    if (n < 0) {  
        return 0;  
    } else {  
        return a(n - 1);  
    }  
}
```

Think about it...

- Order of growth in time/space

Exercise 2

```
function b(n) {  
    if (n < 0) {  
        return 0;  
    } else {  
        return b(n - 1) + 2;  
    }  
}
```

Think about it...

- Order of growth in time/space

Exercise 3

```
function c(n) {  
    if (n < 1) {  
        return 0;  
    } else {  
        return c(n / 2);  
    }  
}
```

Think about it...

- Order of growth in time/space

Exercise 4

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

Think about it...

- Order of growth in time/space

Order of Growth

Exercise 5

```
function e(n) {  
    var k = n / 3;  
  
    function iter(n) {  
        return n < 0 ? 0 : iter(n - k);  
    }  
  
    return iter(n);  
}
```

Think about it...

- Order of growth in time/space

Exercise 6

```
function f(n) {  
    if(n < 0) {  
        return 0;  
    } else {  
        return f(n - 1) + f(n - 1);  
    }  
}
```

Think about it...

- Order of growth in time/space

Exercise 7

```
function g(n) {  
    if(n < 0) {  
        return 0;  
    } else {  
        return g(n - 1) * 2;  
    }  
}
```

Think about it...

- Order of growth in time/space

Exercise 8

```
function h(n) {  
    if(n < 0) {  
        return 0;  
    } else {  
        return h(n / 2) + h(n / 2);  
    }  
}
```

Think about it...

- Order of growth in time/space

Exercise 9

```
function i(n) {  
    if(n < 0) {  
        return 0;  
    } else {  
        return i(n / 2) * 2;  
    }  
}
```

Think about it...

- Order of growth in time/space

Order of Growth

Exercise 10

```
function j(n) {  
    var k = Math.sqrt(n);  
  
    function iter(n) {  
        return n < 0 ? 0 : iter(n - k);  
    }  
  
    return iter(n);  
}
```

Think about it...

- Order of growth in time/space

Order of Growth

Exercise 11

```
function k(n) {  
    var k = Math.log(n);  
  
    function iter(n) {  
        return n < 0 ? 0 : iter(n - k);  
    }  
  
    return iter(n);  
}
```

Think about it...

- Order of growth in time/space

Order of Growth

Exercise 12

```
function l(n) {  
    function fibo(x) {  
        return x < 2 ? x : fibo(x - 1) + fibo(x - 2);  
    }  
  
    return n === 0 ? 0 : fibo(n) + l(n - 1);  
}
```

Think about it...

- Order of growth in time/space

Order of Growth

Recap - Two steps to find order of growth

You only need to follow two steps:

- Analyse the recurrence relationship.
- Calculate the asymptotic notation of that relationship.

But...

- How to execute these two steps?

When the question is simple...

- Have you seen the question before?
 - Usually, it is similar to one of Exercise 1 - 12.
- If not, can you “guess” the answer?

Most questions in CS1101S are considered to be “simple”.

When the question is too complicated...

- Use the formal approach described as follows.

Step 1 - analyse the recurrence relationship

- This is an easy but crucial step.
- “Easy”: because this relationship is the same as the relationship of sub-problems when you write the recursion.
Therefore, you do not need to do any extra work here.
- “Crucial”: if you cannot find the relationship here, that means you also cannot find the relationship of sub-problems in recursion.
That means, you have to leave the whole question blank in an exam.

Step 2 - calculate the asymptotic notation

- This step is only about some mathematical tricks.
- In fact, no CS knowledge involved.

Two mathematical tricks

- Recurrence tree: use a tree diagram to help you analyse;
- Master theorem: a useful theorem to help determine the asymptotic notation.

Order of Growth

Recurrence tree

Idea: *visualize* the recurrence relationship by drawing a *tree diagram*.

- Expand the tree big enough;
- Find the sum *on each level*;
- Find the number of levels.

A few mathematical series

- $1 + 2 + \cdots + n = O(n^2)$
- $a + a \cdot q + \cdots + a \cdot q^n = a \cdot \frac{1-q^{n+1}}{1-q}$
 $= \frac{a}{1-q} = O(1)$ or $O(a)$, when $|q| < 1$
- $1 + \frac{1}{2} + \cdots + \frac{1}{n} = O(\log n)$

Master Theorem

Idea: an empirical theorem which summarizes 3 common scenarios of recurrence relationship and their order of growth.

Content

If the recurrence relationship can be expressed as $T(n) = a \cdot T(\frac{n}{b}) + f(n)$, let $x = n^{\log_b a}$, then:

- If $x > f(n)$, then $T(n) = O(n^{\log_b a})$;
- If $x = f(n)$, then $T(n) = O(f(n) \cdot \log n)$;
- If $x < f(n)$ and $a \cdot f(\frac{n}{b}) < f(n)$, then $T(n) = O(f(n))$;

Master Theorem (*continued ..*)

Master Theorem may fail in a few scenarios, such as

- The function $f(n)$ oscillates, such as $\sin(n)$ and $\cos(n)$
- The derivative of $f(n)$ is smaller than any polynomial, such as $\log(n)$
- ...

Order of Growth

Self reading

- MIT has an interesting course on order of growth by Prof. Erik Demaine.
 - 6.890 Algorithmic Lower Bounds: Fun with Hardness Proofs

Resources

- *Fall 2014 version on MIT OCW*

Exercise

In the next page, you are going to see a few pairs of $f(n)$ and $g(n)$, find out their relationship from one of the following three:

- $f(n) \in O(g(n))$
- $f(n) \in \Theta(g(n))$
- $f(n) \in \Omega(g(n))$

Please use Θ when possible.


Exercise (*continued ...*)

- $f(n) = 0$ and $g(n) = 1$
- $f(n) = 2^n$ and $g(n) = 2^{n+1}$
- $f(n) = 2^n$ and $g(n) = 2^{2n}$
- $f(n) = \log(n!)$ and $g(n) = n \cdot \log n$
- $f(n) = n^{\frac{1}{\log n}}$ and $g(n) = 2 \log n$
- $f(n) = n^{k+1}$ and $g(n) = \sum_{i=1}^n i^k$, where k is a constant

Let's discuss them now.

End

The End

Niu Yunpeng © 2017 - 2018. Under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. 

Appropriate credits **MUST** be given when sharing, copying or redistributing this material in any medium or format. No use for commercial purposes is allowed.

This work is mostly an original by Niu Yunpeng. It may either directly or indirectly benefit from the previous work of Martin Henz, Cai Deshun. For illustration purposes, some pictures in the public domain are used. Upon request, detailed acknowledgments will be provided.