CS1101S Discussion Group Week 3: *Abstraction, Recursion & Order of Growth*

Niu Yunpeng

niuyunpeng@u.nus.edu

August 29, 2017

Before we start

DG Participation Marks

- In class?
- In WhatsApp Group?
- Ask and/or answer?

Overview

1. Review of abstraction
    - From last week
    - A good abstraction
    - Scoping
2. Recursion
    - To understand recursion
    - To use recursion
    - Examples
    - Exercises
3. Order of growth
    - To understand order of growth
    - Exercises

Review of Abstraction

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.

Review of Abstraction

Means of abstraction

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

Review of Abstraction

Define a function

```
var 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);`

Review of Abstraction

What makes a good abstraction?

- Modularity
- Readability
- Reusability
- Maintainability

Review of Abstraction

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.

Scope of Variables

Scoping rules

- Pre-declared built-in functions or variables?
- Formal parameters?
- Global variables?
- Local variables?

Overview

Recursion

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

Recursion & iteration

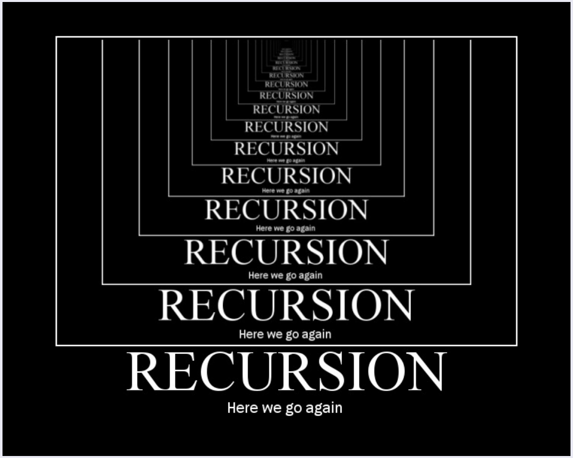
- *Iteration*: the buttom-up approach;
- *Recursion*: the top-down approach.

Notice

- We will start with and mainly focus on *recursion* (the top-down approach).

Recursion

Recursion is beautiful



Recursion

Recursion is beautiful

Recursion

Recursion is beautiful

Recursion

How to understand recursion?

- Use ***substitution model***.
- In details, it means two "replace":
    - Replace a function call by its function body;
    - Replace formal parameters by actual arguments.

Recursion

Recursive function

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

Recursion

To write recursive functions correctly

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

Recursion

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.

Recursion

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):

Recursion

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.

Recursion

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):

Recursion

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 fatorial of $n - 1$ and itself.

Recursion

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.

Recursion

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.

Recursion

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!***

Recursion

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).

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
- ...

Recursion

Examples that we are going to cover today...

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

Recursion

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

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

Recursion

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?

Recursion

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?

Recursion

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?

Recursion

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?

Recursion

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.

Recursion

Hint

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

Recursion

Hint

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

Recursion

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?

Recursion

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.

Recursion

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?

Recursion

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?

Recursion

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?

Recursion

Fast power 1

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

Think about it...

- Correctness?
- Time/space complexity?

Recursion

(Not really) fast power 2

```
// This version gives rise to an iterative process.
function fast_power(b, n) {
    function iter(k, res) {
        if (k === 0) {
            return res;
        } else {
            return 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 ...?

Recursion

(Not really) fast power 3

```
// This version gives rise to an iterative process.
function fast_power(b, n) {
    function iter(k, res) {
        if (k === 0) {
            return res;
        } else {
            return 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 ...?

Recursion

Fast power 4

```
// This version also gives rise to an iterative process.
function fast_power(b, n) {
    function iter(b, k, res) {
        if (k === 0) {
            return res;
        } else {
            return 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?

Recursion

Why is "*fast_power*" fast?

- In normal power function, we iterate through 1...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...n and achieve a logarithmic order of growth.

Recursion

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

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

What about the other direction?

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

Recursion

(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?

Recursion

(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?

Recursion

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:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

- 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

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

Recursion

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?

Recursion

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?

Recursion

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?

Recursion

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:

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

Notice

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

Recursion

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.

Recursion

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?

Recursion

GCD - the Euclidean algorithm

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

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

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.

Recursion

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?

Is this faster or slower?

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

Recursion

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:

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

Notice

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

Recursion

LCM 1

```
// Assume we do not 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?

Recursion

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?

Recursion

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.

Recursion

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.

Recursion

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`.

Recursion

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.

Recursion

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`.

Recursion

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

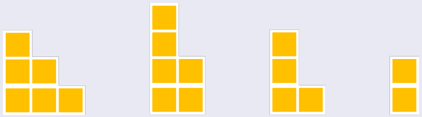
- `calculate_f(0)` returns 1.
- `calculate_f(1)` returns 3.
- `calculate_f(3)` returns 22.

Recursion

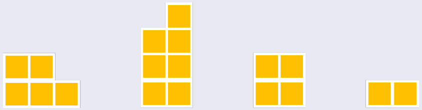
6. Staircase

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:



Recursion

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

Order of Growth

Big theta, oh, omega

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

Order of Growth

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

Order of Growth

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 do not need to use either recurrence tree method or Master Theorem method.

How to tackle this kind of problems easily

- Remember a few commonly-used asymptotic notation:

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

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

Order of Growth

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.

Order of Growth

Exercise 1

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

Think about it...

- Order of growth in time/space

Order of Growth

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

Order of Growth

Exercise 3

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

Think about it...

- Order of growth in time/space

Order of Growth

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

Order of Growth

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

Order of Growth

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

Order of Growth

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

Order of Growth

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

Discussion Group Problems

Let's discuss them now.

End

The End