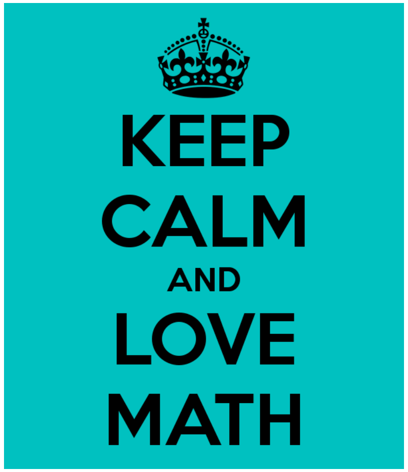
AST20105 DATA STRUCTURES & ALGORITHMS

CHAPTER 2 – MATHEMATICAL AND PROGRAMMATIC PRELIMINARIES

Instructed by Garret Lai

MATHEMATICAL



EXPONENTS

Definition:

- Any expression written as a^n is defined as the variable **a** raised to the power of the number **n**
- **n** is called an exponent of **a**, or a power or an index of **a**

EXPONENTS

Useful properties:

- Zero power: $a^0 = 1$
- Negative power: $a^{-n} = \dfrac{1}{a^n}$
- Fractional power: $a^{\frac{1}{n}} = \sqrt[n]{a}$

EXPONENTS

Other useful properties:

$$a^m \cdot a^n = a^{m+n}$$

$$\frac{a^m}{a^n} = a^{m-n}$$

$$(a^m)^n = a^{m \cdot n}$$

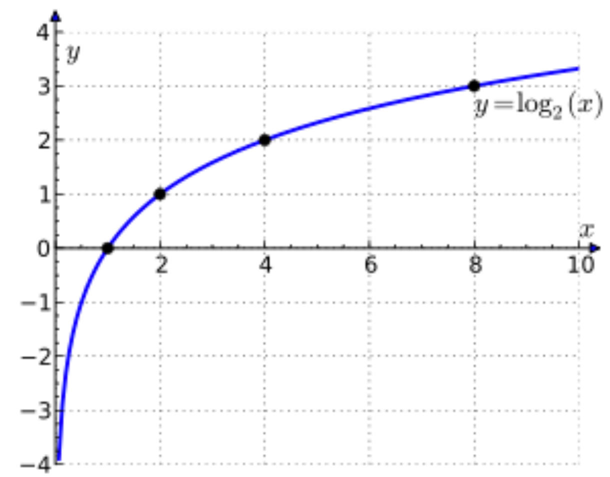
$$a^n \cdot b^n = (ab)^n$$

LOGARITHMS

Definition:

- Logarithms are the “opposite” of exponentials, i.e.

$$a^n = b \text{ if and only if } n = \log_a b$$



LOGARITHMS

Useful properties:

$$\log_a 1 = 0$$

$$\log_a a = 1$$

Special notation: (Natural log), *where e is a constant approximately equal to 2.71828*

$$\log_e n = \ln n$$

$$\log_2 n = \lg n$$

LOGARITHMS

Other useful properties:

$$\log_a(\sqrt[n]{x}) = \frac{1}{n}\log_a x$$

$$\log_a(xy) = \log_a x + \log_a y$$

$$\log_a(\frac{x}{y}) = \log_a x - \log_a y$$

$$\log_a x^m = m\log_a x$$

$$\log_a b = \frac{\log_c b}{\log_c a}$$

FUNCTION SERIES

The following function series are useful for this course.

They are:

- Arithmetic Series
- Geometric Series

ARITHMETIC PROGRESSIONS

An Arithmetic Progression, or AP, is a sequence where each new term after the first is obtained by adding a constant d, called common difference, to the preceding term

If the first term of the sequence is a, then the arithmetic progression is

$$a, a+d, a+2d, a+3d, \ldots$$

where the n-th term is $a + (n-1)d$

ARITHMETIC SERIES

Suppose we would like to add the first n terms of an Arithmetic Progression

$$S_n = a + (a+d) + (a+2d) + \ldots + (a+(n-1)d)$$

This sum is referred as Arithmetic Series and could be computed by

$$S_n = \frac{n}{2}(2a+(n-1)d) \quad \text{OR} \quad S_n = \frac{n(a_1+a_n)}{2}$$

GEOMETRIC PROGRESSIONS

A Geometric Progression, or GP, is a sequence where each new term after the first is obtained by multiplying a constant r, called common ratio, to the preceding term

If the first term of the sequence is a, then the Geometric Progression is

$$a, aR, aR^2, aR^3, \ldots$$

where the n-th term is aR^{n-1}

GEOMETRIC SERIES

Suppose we would like to add the first n terms of an geometric progression

$$S_n = a + (aR) + (aR^2) + \ldots + (aR^{n-1})$$

This sum is referred as Geometric Series and it could be computed by

$$S_n = \begin{cases} \dfrac{a(R^n - 1)}{R - 1} & \text{if } R > 1 \\ \dfrac{a(1 - R^n)}{1 - R} & \text{if } R < 1 \end{cases}$$

SUM TO INFINITY AND SUM OF SQUARES

The ‘sum to infinity’ of a geometric series is

$$S_\infty = \frac{a}{1-R} \quad \text{if } -1 < R < 1$$

The sum of squares could be computed as follows:

$$1 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

INEQUALITIES

Inequality is a relation between two values. It replaces the = sign in an equation with:

$$<, >, \leq, \text{or} \geq$$

Properties: (For any real numbers a, b, c)

- Transitivity:
  - If $a > b$ and $b > c$, then $a > c$
  - If $a < b$ and $b < c$, then $a < c$
  - If $a > b$ and $b = c$, then $a > c$
  - If $a < b$ and $b = c$, then $a < c$

INEQUALITIES

Properties: *(for any real numbers a, b, c)*

- Addition and subtraction
  - If $a < b$, then $a + c < b + c$ and $a - c < b - c$
  - If $a > b$, then $a + c > b + c$ and $a - c > b - c$

- Multiplication and division
  - If c is positive and $a < b$, then $ac < bc$ and $a/c < b/c$
  - If c is negative and $a < b$, then $ac > bc$ and $a/c > b/c$

- Additive inverse
  - If $a < b$, then $-a > -b$
  - If $a > b$, then $-a < -b$

INEQUALITIES

Properties *(for any real numbers a, b, c)*

- Multiplicative inverse
  - For any non-zero real numbers a and b that are both positive or both negative
    - If $a < b$, then $1/a > 1/b$
    - If $a > b$, then $1/a < 1/b$

  - For one of a and b is positive and the other is negative, then
    - If $a < b$, then $1/a < 1/b$
    - If $a > b$, then $1/a > 1/b$

PROGRAMMATIC PRELIMINARIES

BEFORE START

Do you know Mark 6?

BEFORE START

Have you ever won Mark 6?

Prize	Unit Prize	Winning Unit
1st	$31,023,870	1.0
2nd	$1,224,870	3.0
3rd	$40,490	242.0

BEFORE START

Do you know the probability to win the 1st prize?

BEFORE START

The probability is:

$6/49 \times 5/48 \times 4/47 \times 3/46 \times 2/45 \times 1/44$

$\Rightarrow 1 \ / \ {}_{49}C_6$

$= 1 \ / \ (49! \ / \ 6! \times (49-6)!)$

$= 1 \ / \ 13983816$

BEFORE START

So, what is n!?

Factorial of n (denoted *n!*) is a product of integer numbers from 1 to n.

- For instance, 6! = 1 * 2 * 3 * 4 * 5 * 6 = 720.

BEFORE START

Recursion is one of techniques to calculate factorial.

- Indeed, 6! = 5! * 6.
- To calculate factorial of *n*, we should calculate it for *(n-1)*.
- To calculate factorial of *(n-1)* algorithm should find *(n-2)!* and so on.

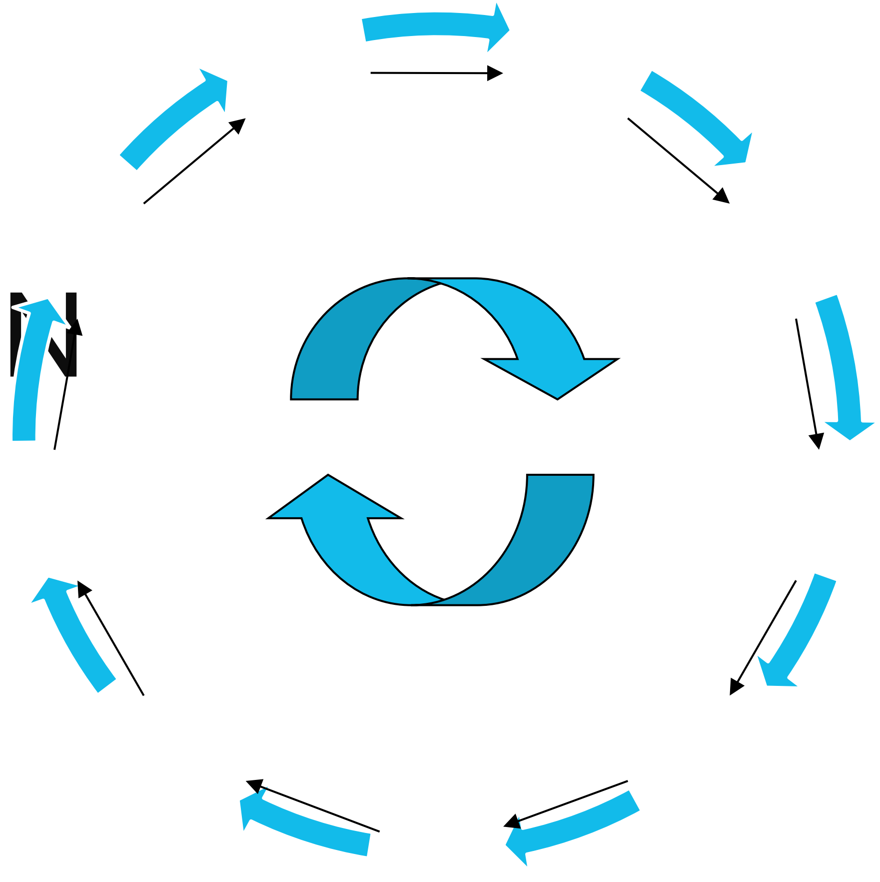
BEFORE START

But described process will last infinitely, because no base case has been defined yet.

Base case is a condition, when recursion should stop.

- In the factorial example, a base case is $n = 1$, for which the result is known.

RECURSION



RECURSIVE FUNCTION (RECURSION)

In some problems, it may be natural to define the problem in terms of the problem itself

Recursion is useful for problems that can be represented by a simpler version of the same problem

Most computer programming languages support recursion by allowing a function to call itself

RECURSIVE FUNCTION (RECURSION)

Many examples of the use of recursion may be found:

- the technique is useful both for the definition of mathematical functions and for the definition of data structures.

Naturally, if a data structure may be defined recursively, it may be processed by a recursive function!

EXAMPLE: FACTORIAL FUNCTION

The factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n

- For example:
  5! = 5 x 4 x 3 x 2 x 1 = 120

In general, factorial function is defined as follows:

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ n \times (n-1)! & \text{if } n > 1 \end{cases}$$

EXAMPLE: FACTORIAL FUNCTION

Now, suppose we would like to write a C++ function to compute n!

- Do you know how to do it?

```
int factorial(int n)
{
   int result = 1;
   for(int i=n; i>=1; i--)
      result *= i;
   return result;
}
```

Iterative Version

- Any other way to do the same thing?
- Yes, use recursion!!!

EXAMPLE: FACTORIAL FUNCTION

Observation:

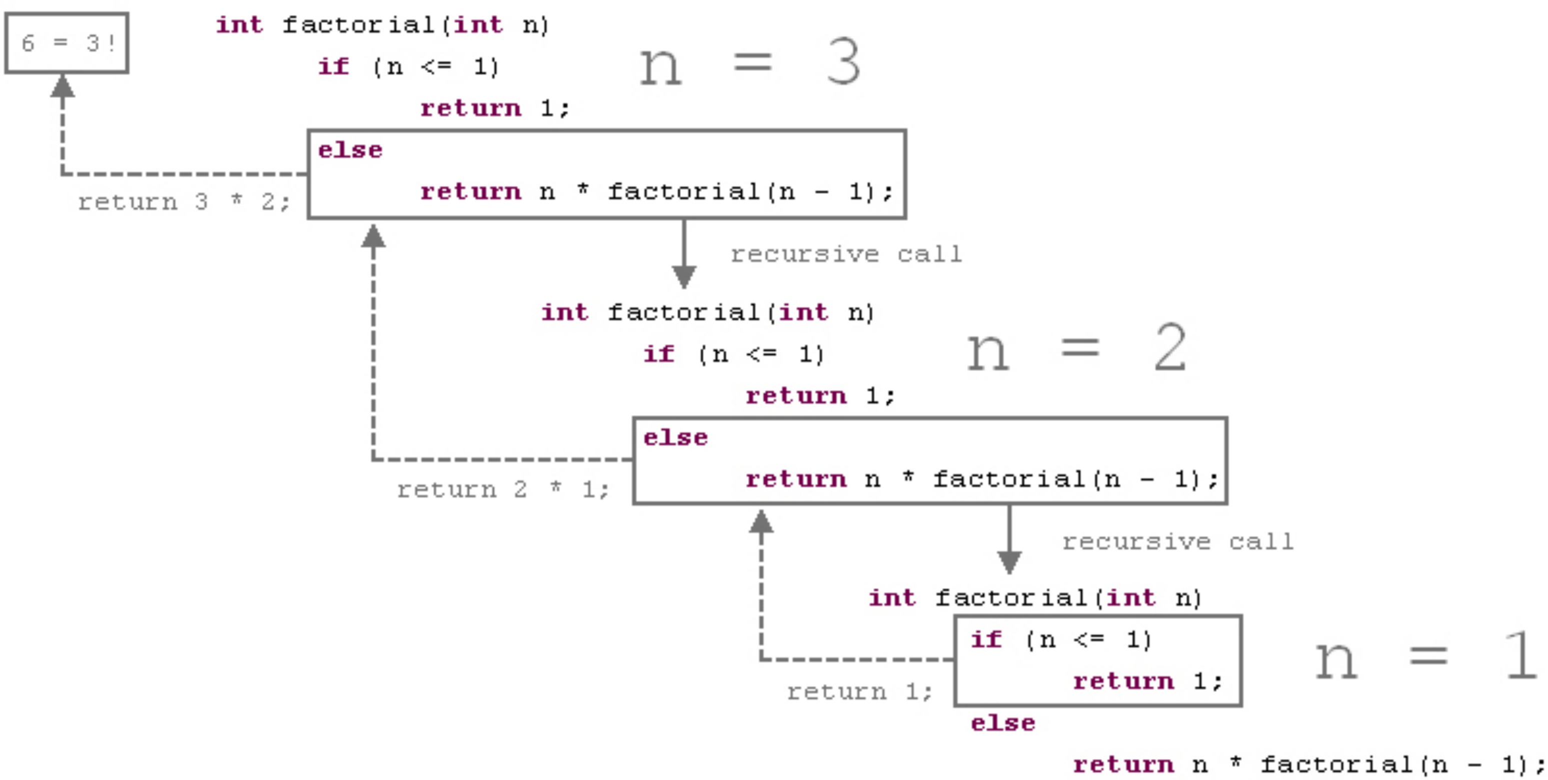
- factorial(n) = n * (n-1) * (n-2) * … * 1 = n * (n-1)!
- If we know the value of factorial(n-1), then we can simply multiply it with n to produce factorial(n)
- However, we don't have factorial(n-1) in hand >.<
- Why don't we call the function itself to compute factorial(n-1)

```
int factorial(int n)
{
    if(n==0 || n==1)            // base case
        return 1;
    else
        return n * factorial(n-1); // recursive case
}
```

Recursive Version

EXAMPLE: FACTORIAL FUNCTION

Calculation of 3! in details



RECURSIVE FUNCTION (RECURSION)

A recursion consists of at least two parts:

- Base case:
  - The problem is simple enough, we can solve it with other help
  - Here, factorial(0) = 1, which is simple enough

- Recursive case:
  - We don't know how to solve the problem, say factorial(2)
  - So call the function itself with a small input and then combine the result to form the solution of the larger input

RECURSIVE FUNCTION GENERAL FORM

```
<type> recursiveFunc(<parameters>)
{
  if(<stopping condition>)
     return <stopping value>;
  return recursiveFunc(<revised parameters>);
}
```

EXAMPLE: EXPONENTIAL FUNCTION

Compute x^y (y is a non-negative integer):

```
double exp(double x, int y)
{
   if(y==0)              // base case
      return 1;
   return x * exp(x, y-1);    // recursive case
}
```

Compute exp(3.2, 3):

```
exp(3.2, 3) : return 3.2 * exp(3.2, 2)
        exp(3.2, 2) : return 3.2 * exp(3.2, 1)
                exp(3.2, 1) : return 3.2 * exp(3.2, 0)
                        Here, exp(3.2, 0) will return 1, so
                exp(3.2, 1) : return 3.2 * 1   // i.e. return 3.2
        exp(3.2, 2) : return 3.2 * 3.2         // i.e. return 10.24
exp(3.2, 3) : return 3.2 * 10.24               // i.e. return 32.768
```

EXAMPLE: FIBONACCI FUNCTION

Fibonacci numbers / Fibonacci series / Fibonacci sequence are numbers in the following integer sequence:

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

- By definition, the first two numbers in the Fibonacci sequence are 1 and 1, and each subsequent number is the sum of the previous two

In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

with seed values $F_0 = 1$, $F_1 = 1$

EXAMPLE: FIBONACCI FUNCTION

Compute the n Fibonacci number:

```
int fib(int n)
{
   if(n == 0 || n == 1)         // base case
      return 1;
   return fib(n-1) + fib(n-2); // recursive case
}
```

Compute fib(4):
fib(4) : return fib(3) + fib(2)
        fib(3) : return fib(2) + fib(1)
        fib(2) : return fib(1) + fib(0)
                Here, fib(0) and fib(1) will return 1, so
        fib(2) : return 1 + 1   // i.e. return 2
        fib(3) : return 2 + 1   // i.e. return 3
fib(4) : return 3 + 2   // i.e. return 5

BINARY SEARCH WITH RECURSION

```
int binarySearchRec(int sortedArr[],
                    int first,
                    int last,
                    int key)
{
   int mid = (first + last) / 2;
   if(sortedArr[mid] == key)
      return mid;
   else if(first >= last)
      return -1;
   else if(key < sortedArr[mid])
      return binarySearchRec(sortedArr, first, mid-1, key);
   else
      return binarySearchRec(sortedArr, mid+1, last, key);
}
```

ADVANTAGES OF RECURSION

Main advantage of recursion is programming simplicity.

- When using recursion, programmer can forget for a while of the whole problem and concentrate on the solution of a current case.

- Then, returning back to the whole problem, base cases (it's possible to have more than one base case) and entry point for recursion are developed.

DRAWBACKS OF RECURSION

Recursion has a serious disadvantage of using large amount of memory.

- Moreover, for most programming languages, recursion use stack to store states of all currently active recursive calls.
- The size of a stack may be quite large, but limited. Therefore too deep recursion can result in *Stack Overflow*.

CHAPTER 2 END