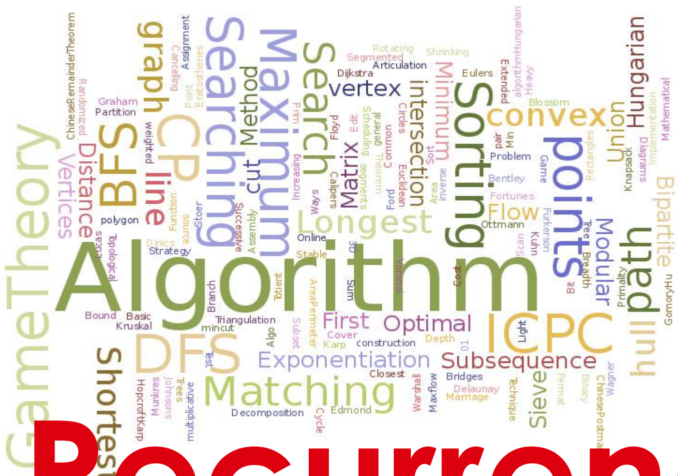
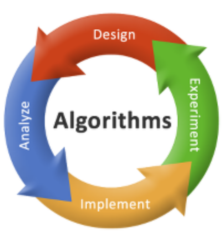


Recurrence Relations

Course: Algorithms



Faculty: Dr. Rajendra Prasath

Autumn 2018

Recurrence Relations

This class covers many **Computational Methods** to solve a recurrence relations. This lecture illustrates the derivation of the **Closed Forms** of a few selected problems and their analysis

Recap: Complexity Analysis

- **Computational Complexity of the Algorithms**
  - **Best Case Analysis**
  - **Average Case Analysis and**
  - **Worst Case Analysis**

  - **Data Structures – Their Effects on the Algorithm Design**
- How do we efficiently handle the running time and space needed to hold the data?
  - Optimization problems
    - Running Time - **Minimize** or Maximize?
    - Space Needed - **Minimize** or Maximize?

Recap: Efficient Algorithms

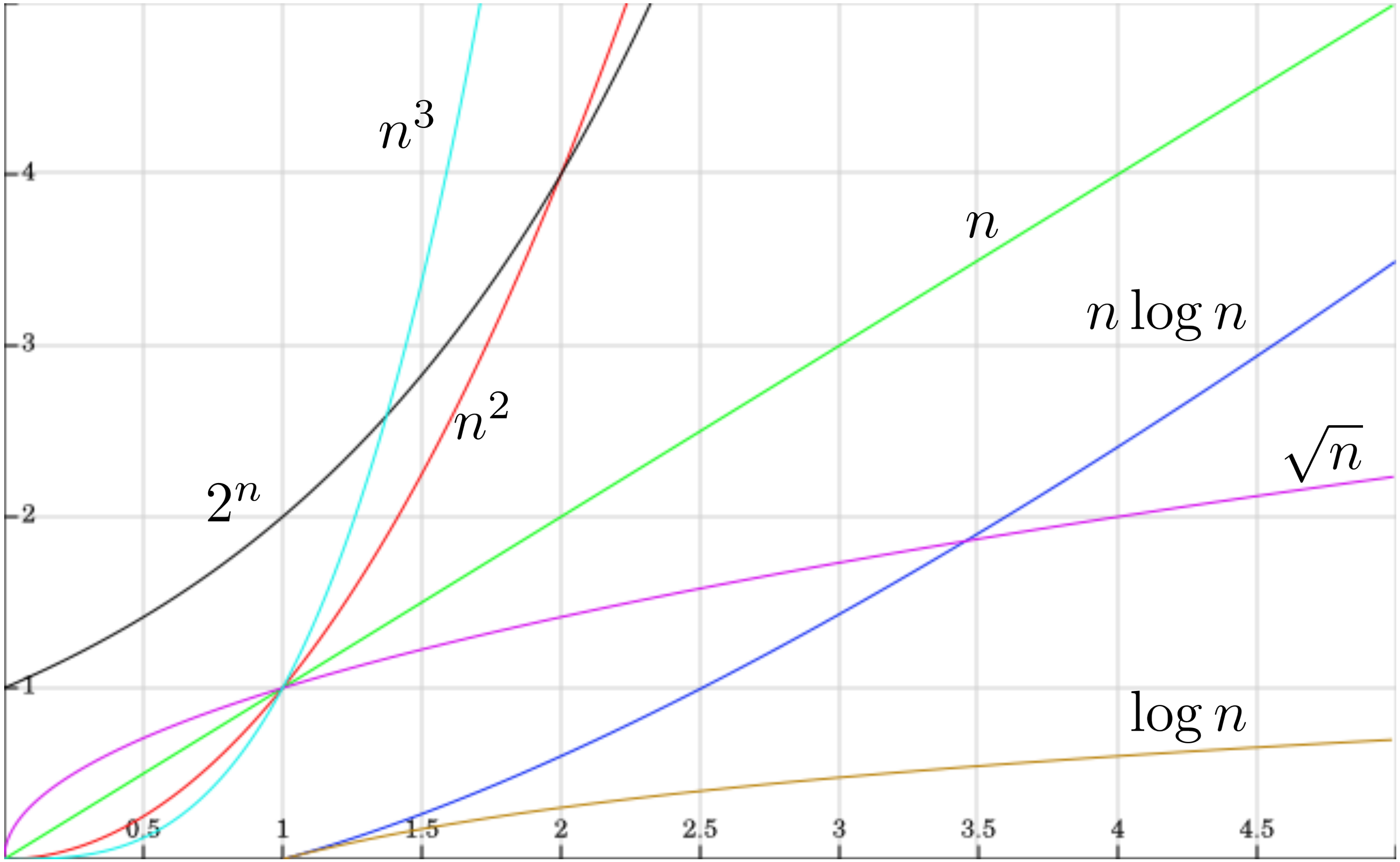
- Running Times of the algorithms may vary based on n

- Estimating runtime
  - Considering the input size
  - Growth in the order of magnitude of the input
  - Perform tight / upper bound analysis

  - Compute the space requirements and choose appropriate data structures
  - Efficient implementations with the right choice of the programming language for the given task

Asymptotic Complexity

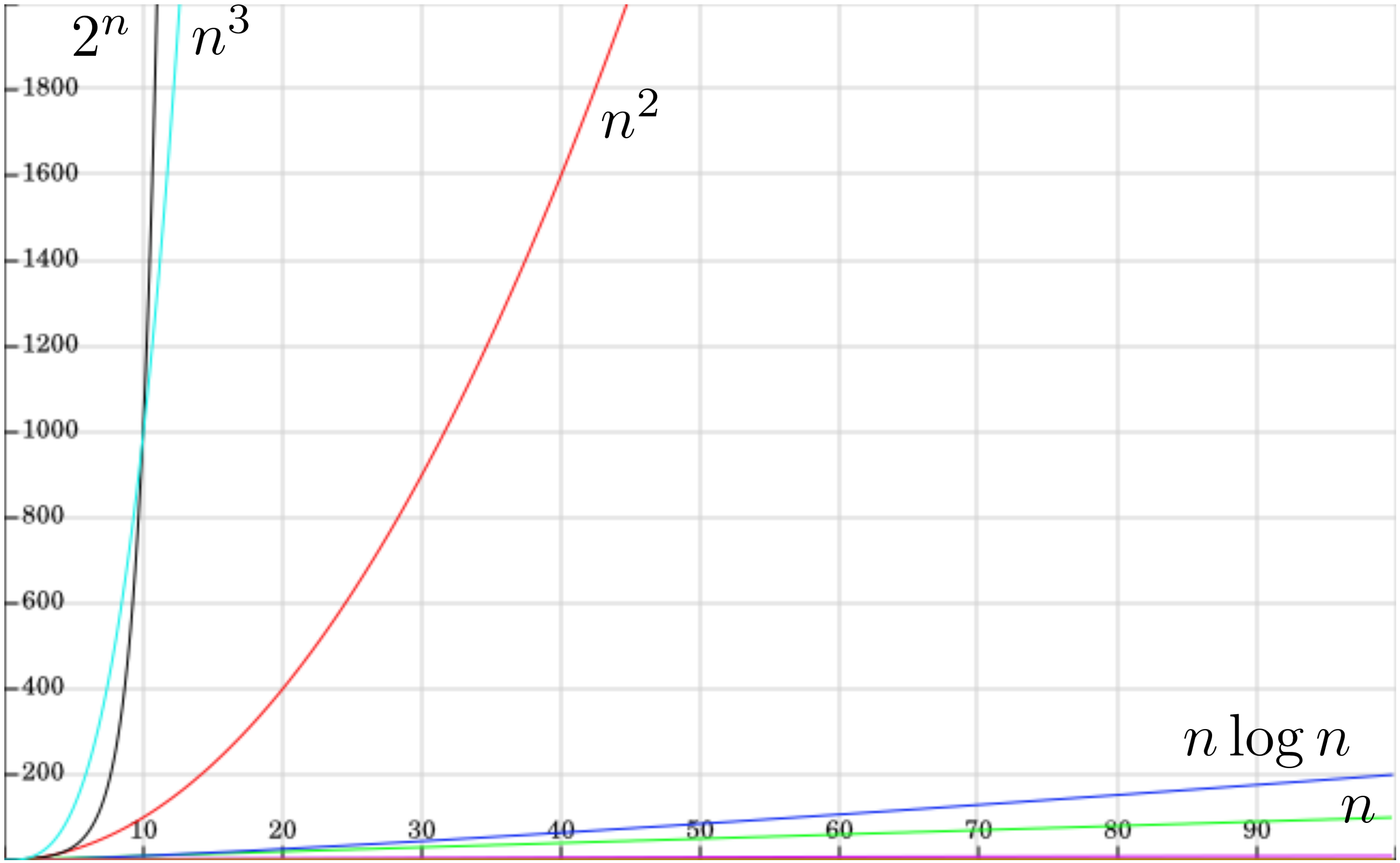
- For smaller input values



5

Asymptotic Complexity

- For larger input values



Floating Point Computations

- Computers are finite state machines
- Efficiently handle integer based and floating point–based arithmetic computations
  - Which one is a hard problem?

Look at the Olympic 100 meters race

- **Usain Bolt (JAM)** - World record (9.58s, Berlin, 2009) and Olympic record (9.63s, London 2012)
- 100 meters – Rounds 1 (2016 Summer Olympics)

Rank	Lane	Name	Nationality	Reaction	Time
1	3	Kemarley Brown	Bahrain	0.146	10.13
2	5	Chijindu Ujah	Great Britain	0.150	10.13
3	7	Marvin Bracy	United States	0.155	10.16

Floating Point Computations

Rounding Errors:

- Any computation using floating-point values may introduce rounding errors
- Floating Point number - a finite representation designed to approximate a real number

- The Hardest Part of Computation:
  - How do we achieve the greatest precision with floating point computations?

  - How do we compare two floating point numbers: a , b

```
if ( a == b )  {  ...  }
while ( a == b ) {  ...  }
```

- Due to the inherent nature of approximation, floating point operations become suspicious

Recursions

Definition:

A program is recursive when it contains a call to itself.

- Recursion can substitute iteration in program design:
- Generally, recursive solutions are simpler than iterative solutions.
- Generally, recursive solutions are slightly less efficient than the iterative ones unless the recursive calls are optimized
- There are natural recursive solutions that can be extremely inefficient … Be aware of such recursions !

How to handle Recursions?

- How to we define a recursive approach?
  - In terms of the input size n
  - The rate of change of n

- Two Steps:
  - Basic Steps
  - Recursive Steps

- Example 1: gcd (a, b)

```
int gcd(int a, int b) {
    if ( b == 0 ) return a;
    return gcd(b, a%b);
}
```

Recursions – Example 2

- Example 2:

```
// Infinite Recursion
int printNum(int n) {
    printf (" %d ", n);
    return printNum(n+1);
}

OR

// Print until MAX is reached
int printNum(int n, int MAX) {
      printf (" %d ", n);
      if ( n > MAX - 1 ) return 0;
      return printNum(n+1, MAX);
}
```

Recurrence Relations

- How to get a closed form of a recurrence relation
  - In terms of the input size n
  - The rate of change of n

- Compute Factorial of n:

```
/* Pre: n >= 0; returns n! */

int factorial(int n) { // iterative solution
    int f = 1, i= 0;

    while(i < n) {
        i= i+ 1;
        f = f*i;
    }
    return f;
}
```

Factorial

- Definition:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$$

- Recursive Definition:

$$n! = \begin{cases} n \cdot (n-1)!, & \text{if } n > 0; \\ 1, & \text{if } n = 0 \end{cases}$$

- Code:

```
int factorial(int n) { //recursive soln.
    if(n == 0) return 1;
    else return n * factorial(n -1);
}
```

Factorial - An Example

- How to work out Factorial (5)?

```
factorial (5)
    = 5 * factorial (4)
    = 5 * 4 * factorial (3)
    = 5 * 4 * 3* factorial (2)
    = 5 * 4 * 3 * 2 * factorial (1)
    = 5 * 4 * 3 * 2 * 1 * factorial (0)
    = 5 * 4 * 3 * 2 * 1 * 1
    = 120
```

Recursion: Important Points

- Each time a function is called, a new instance of the function is created
- Each time a function “returns”, its instance is destroyed
- The creation of a new instance only requires the allocation of memory space for data
- The instances of a function are destroyed in reverse order to their creation, i.e. the first instance to be created will be the last to be destroyed.

Try Another Example

- Design an Algorithm:
  - Given an integer n, write its binary representation

- Steps:
  - Base case (n = 1)
    - prints "1" as the output

  - Recursive Case (n > 1)
    - Repeat with n / 2 and then print n%2 as the output

- Example: n = **11**
  - It should print **1011** (base 2)

Binary Representation

```
Algorithm binaryDigits(int n) {

// input: n > 0
// output: binary representation(n)

if (n = 1 ) print n;
else {
        binaryDigits(n/2);
        print n%2;
    }
}
```

- The procedure always terminates since n/2 is closer to 1 than n. Note that n/2 is never 0 when n > 1. Therefore, the case n = 1 will always be found at the end of the sequence call.

Fibonacci numbers

- Basic case:

  $n = 0 \Rightarrow$ return 1

  $n = 1 \Rightarrow$ return 1
- Recursive case:

  $n > 1 \Rightarrow$ return fib(n -1) + fib(n -2)

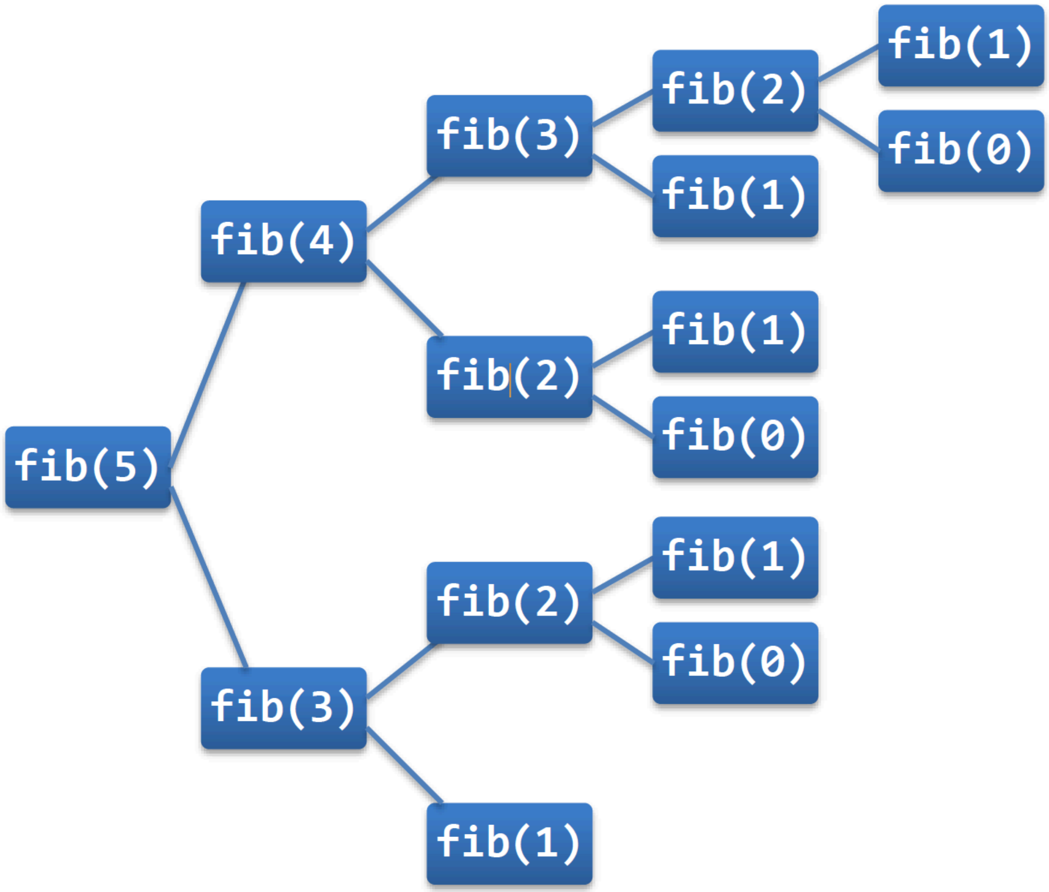
Code:

```
int fib(int n) {
    if(n <= 1) return 1;
    return fib(n -2) + fib(n -1);
}
```

The function always terminates since the parameters of the recursive call (n-2 and n-1) are closer to 0 and 1 than n.

Fibonacci numbers

- fib(5) is illustrated below:



Fibonacci numbers - Analysis

- When fib(5) is calculated:
  - fib(5) is called once
  - fib(4) is called once
  - fib(3) is called twice
  - fib(2) is called 3 times
  - fib(1) is called 5 times
  - fib(0) is called 3 times

- When fib(n) is calculated, how many times will fib(1) and fib(0) be called?

- Example: fib(50) calls fib(1) and fib(0) about $2.4 \cdot 10^{10}$times

Fibonacci – Iterative Method

```
// Input: n >= 0; returns the Fibonacci number of order n.
    int fib(int n) { // iterative solution
        int i = 1;
        int f_i= 1;
        int f_i1 = 1;
        // Inv: f_i is the Fibonacci number of order i
        // f_i1 is the Fibonacci number of order i - 1

        while(i < n) {
            int f = f_i+ f_i1;
            f_i1 = f_i;
            f_i= f;
            i = i + 1;
        }
        return f_i;
    }
```

Fibonacci – Analysis

- With the iterative solution, if we calculate fib(5), we have that:
  - fib(5) is calculated once
  - fib(4) is calculated once
  - fib(3) is calculated once
  - fib(2) is calculated once
  - fib(1) is calculated once
  - fib(0) is calculated once

- Which one is efficient?
  - Iterative or Recursive ??

Complexity Analysis - Recursion

```
void f(int n) {
    if(n > 0) {
        DoSomething(n);         // O(n)
        f(n/2);
    }
}
```

$$\begin{aligned} T(n) &= n + T(n/2) \\ T(n) &= n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 2 + 1 \\ 2 \cdot T(n) &= 2n + n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 4 + 2 \end{aligned}$$

$$2 \cdot T(n) - T(n) = T(n) = 2n - 1$$ → O(n)

Complexity Analysis - Recursion

```
void f(int n) {
    if(n > 0) {
        DoSomething(n);          // O(n)
        f(n/2);    f(n/2);
    }
}
```

$$
\begin{aligned}
T(n) &= n + 2 \cdot T(n/2) \\
&= n + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + 8 \cdot \frac{n}{8} + \cdots \\
&= \underbrace{n + n + n + \cdots + n}_{\log_2 n} = n \log_2 n
\end{aligned}
$$

➔ O(n logn)

Complexity Analysis - Recursion

```
void f(int n) {
    if(n > 0) {
        DoSomething(n);        // O(n)
        f(n - 1);
    }
}
```

$$T(n) = n + T(n-1)$$

$$T(n) = n + (n-1) + (n-2) + \cdots + 2 + 1$$

$$T(n) = \frac{n^2 + n}{2}$$

→ $O(n^2)$

Complexity Analysis - Recursion

```
void f(int n) {
    if(n > 0) {
        DoSomething(); // O(1)
        f(n - 1); f(n - 1);
    }
}
```

$$\begin{aligned} T(n) &= 2 \cdot T(n-1) \\ &= 2 \cdot 2 \cdot T(n-2) \\ &= 2 \cdot 2 \cdot 2 \cdot T(n-3) \\ &\vdots \\ &= \underbrace{2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdots 2}_{n} = 2^n \end{aligned}$$

$\rightarrow$ O(2^n)

Recursion - Closed Forms

- How to get a closed form of a recurrence relation?

$$a_0 = 4$$

$$a_n = a_{n-1} - n$$

Find the Closed form solution for T(n)

$$a_n = 4 - n(n+1)/2$$

How to find this closed form?

Recursion - Closed Forms

- How to get a closed form of a recurrence relation?

Consider the following recurrence relation:

$$T(n) = 5, \quad \text{if } n <= 2$$
$$= T(n-1) + n, \quad \text{otherwise}$$

Find the Closed form solution for T(n)

Help among Yourselves?

- **Perspective Students** (having CGPA above 8.5 and above)
- **Promising Students** (having CGPA above 6.5 and less than 8.5)

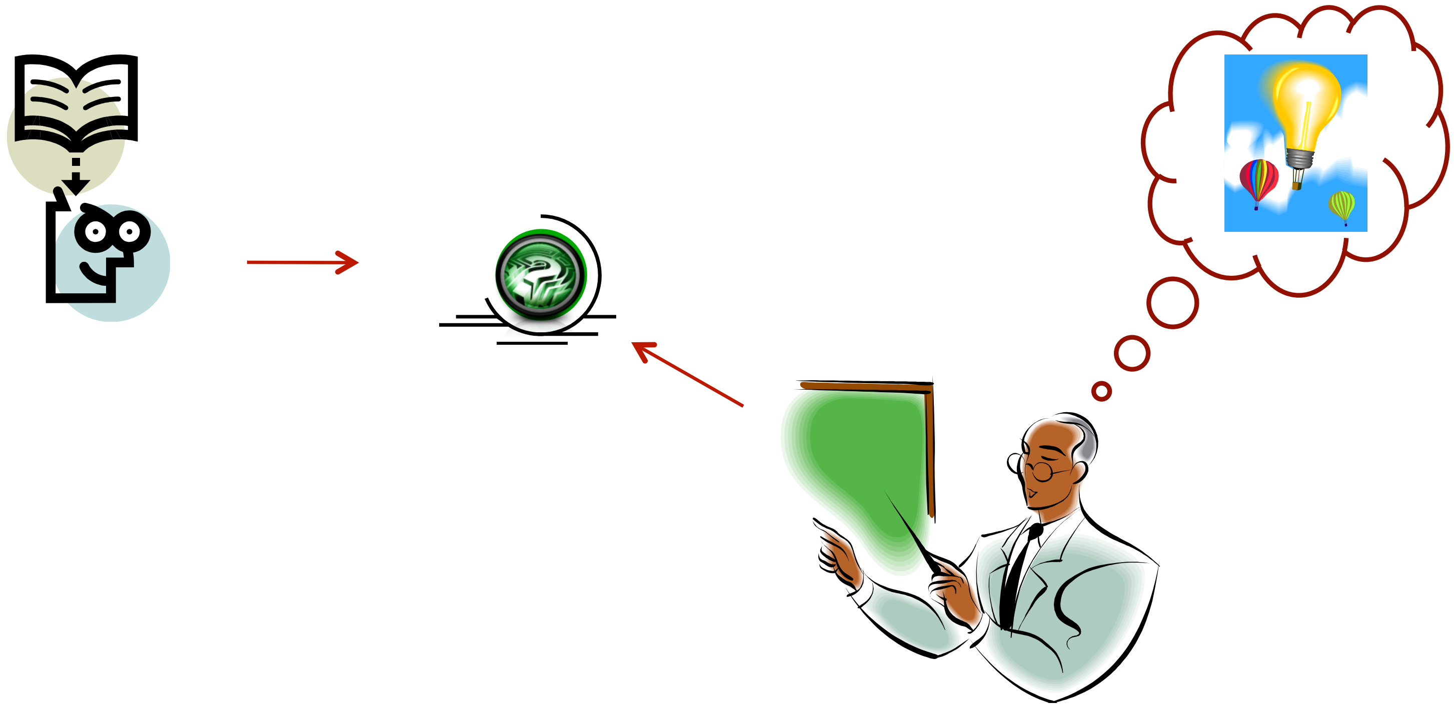
- **Needy Students** (having CGPA less than 6.5)
  - Can the above group help these students? (Your work will also be rewarded)

- You may grow a culture of **collaborative learning** by helping the needy students

Assistance

- You may post your questions to me at any time
- You may meet me in person on available time or with an appointment
- TA s would assist you to clear your doubts.
- You may leave me an email any time (email is the best way to reach me faster)

Thanks …



… Questions ???