# Session 02 - Algorithm Efficiency

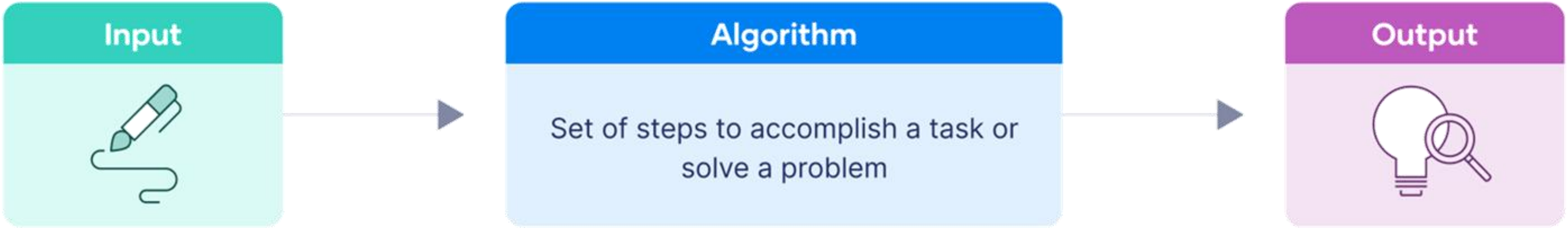
Instructor:

Dr. LE Thanh Tung

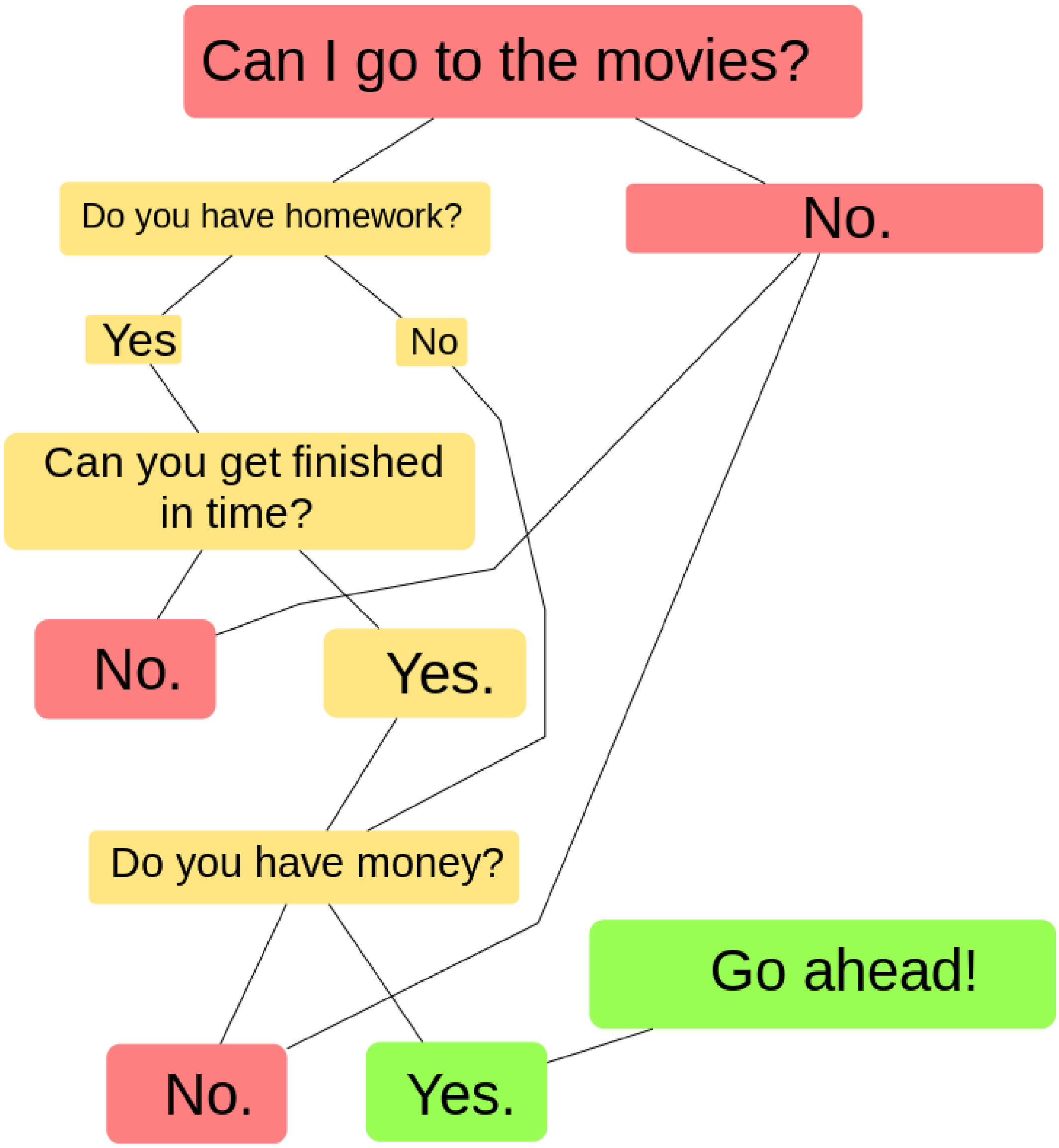
# Content

# A review on Algorithm

- What is Algorithm?
  - A strictly defined **finite** sequence of **well-defined** steps (statements, often called instructions or commands)
  - that provides the solution to a problem.



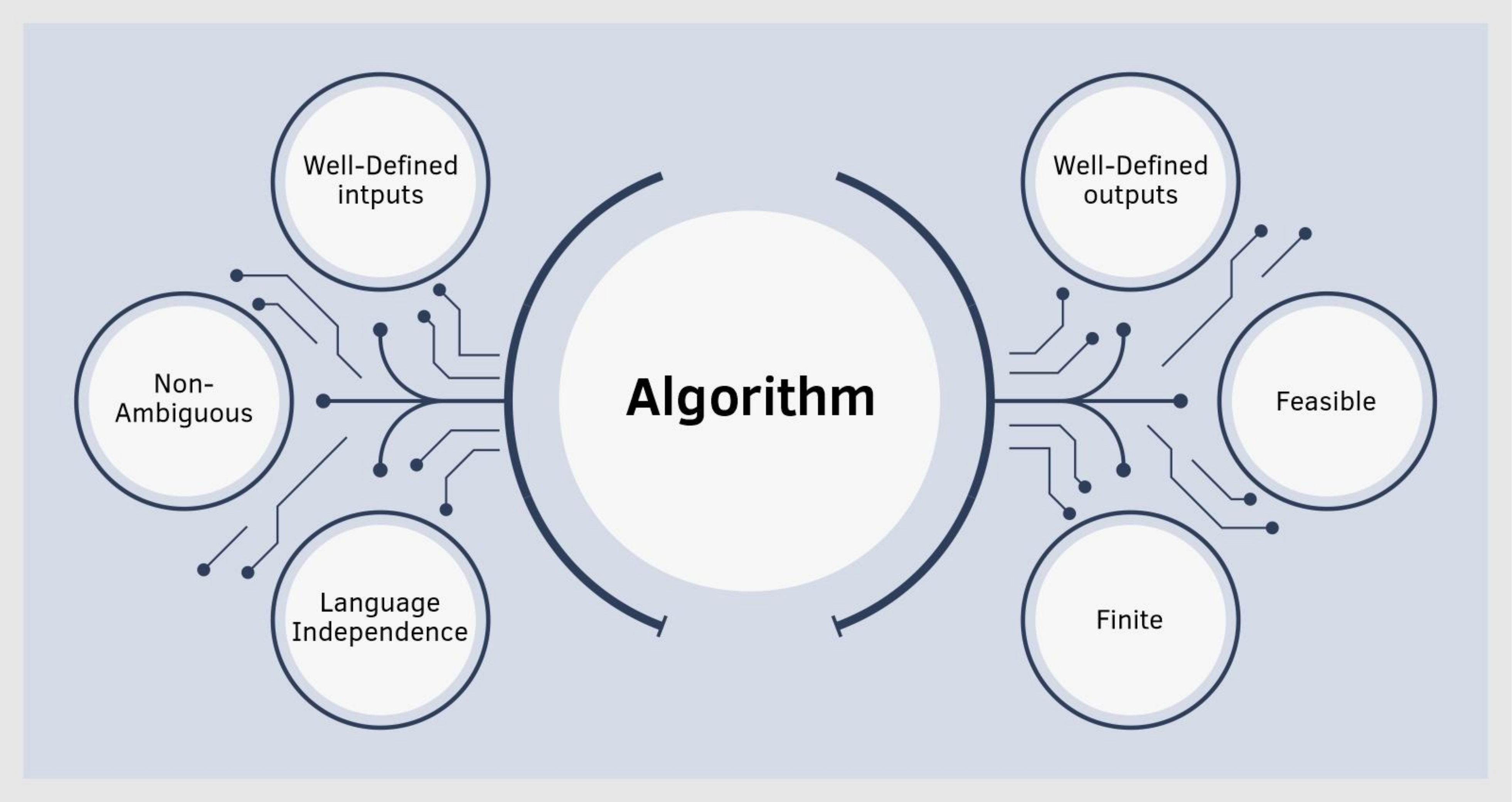
- Example:
  - Problem: a child wonder whether they can go to the movies in case of having a homework today.



# Algorithm

- Why should we study algorithm?
  - To understand the basic idea of the problem.
  - To find an approach to solve the problem.
  - To improve the efficiency of existing techniques.
  - To understand the basic principles of designing the algorithms.
  - To break down problems and conceptualize solutions in terms of discrete steps

- Algorithm's Characteristics:



- Algorithm’s Characteristics:
  - **Finiteness:** for any input, the algorithm must terminate after a finite number of steps.
  - **Correctness:** always correct. Give the same result for different run time.
  - **Definiteness:** all steps of the algorithm must be precisely defined.
  - **Effectiveness:** It must be possible to perform each step of the algorithm correctly and in a finite amount of time.

# Types of Algorithm



# Algorithm



- What kind of problems are solved by Algorithm?
  - Sorting
  - Searching
  - String matching
  - Graph problems
  - Combinatorial problems
  - Geometric problems
  - Numerical problems

- The two factors of **Algorithm Efficiency** are:
  - **Time Factor:** Time is measured by counting the number of key operations.
  - **Space Factor:** Space is measured by counting the maximum memory space required by the algorithm.

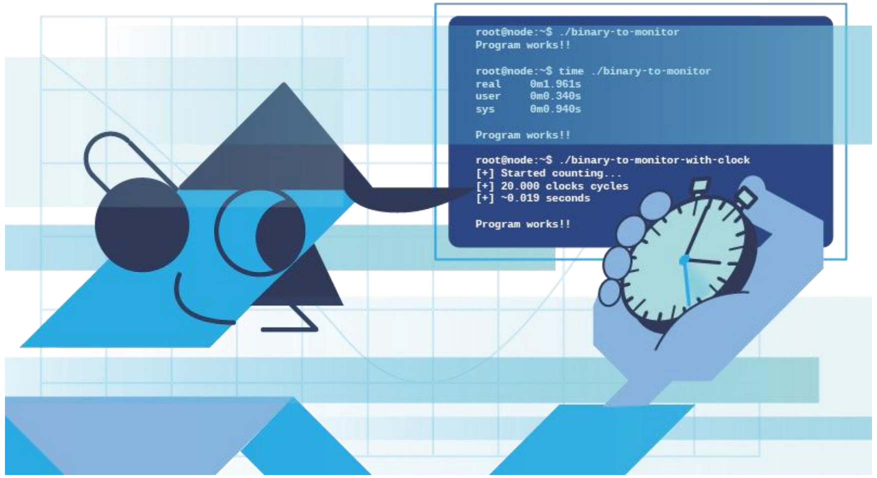
# Algorithm Analysis

- Nowadays, the amount of extra space required by an algorithm is typically not of as much concern
- In most problems, we can achieve much more spectacular progress in speed than in space

→ Concentrate on **time efficiency**, but analytical framework in this course is applicable to analyzing space efficiency as well

- Can we compare two algorithms (in time factor) like this?
  - Implement those algorithms (into programs)
  - Calculate the execution time of those programs
  - Compare those two values of time measurement.

**Is it fair in this measuring process?**



# Measuring Efficiency of Algorithms



- Comparison of algorithms should focus on **significant differences** in efficiency
- Difficulties with comparing programs instead of algorithms
  - How are the algorithms coded?
  - What computer should you use?
  - What data should the programs use?

→ Employ mathematical techniques that **analyze algorithms independently** of specific implementations, computers, or data

- Time complexity is measured by counting the **primitive operations** for the computation that the algorithm needs to perform.
- Key operation is to contribute the most to the total running time of an algorithm
  - Comparisons
  - Assignments
- Derive an algorithm's time requirement as a function of the **problem size**

# Problem size



- Almost all algorithms run **longer on larger inputs**
- For example:
  - Sorting arrays: A1 = {12, 1, 3}
  - Sorting arrays: A2 = {88, 12, 3, 19, 32, 9, 1, 3, 45, 17, 89, 12, 34, 52, 61, 41, 24, 98, 19, 38}
- Algorithm's efficiency is investigated as a function of some parameter **n** indicating the algorithm's **input size**

# Problem size



- **Straightforward**: problems dealing with lists (e.g., sorting, searching, min, max, …)
  - n is the size of the list
- **Not straightforward**:
  - Computing the product of two matrix
  - Checking primality of a positive integer n
  - Spell-checking a document

- Traversal of linked nodes – example:

```
void printList(Node* pHead){
    Node* pCur = pHead;                      ←—— 1        assignment
    while (pCur != nullptr){                 ←—— n + 1    comparisions
        cout << pCur->key << endl;           ←—— n + 1    writes
        pCur = pCur->next;                   ←—— n + 1    assignments
    }
}
```

- Assignment: a time units. Comparison: c time units.
- Write: w time units.
- Displaying data in linked list of **n nodes** requires **time proportional to n**

- Nested loops

```
for (i = 1 through n)
    for (j = 1 through i)
        for (k = 1 through 5)
            Task T
```

- Task *T* requires *t* times units
- How to find the relationship between ***t*** times and the problem sizes **n**

# Example



```
Step 1. Assign sum = 0. Assign i = 0.

Step 2.
   Assign i = i + 1
   Assign sum = sum + i

Step 3. Compare i with 10
   if i < 10, back to step 2.
   otherwise, if i ≥ 10, go to step 4.

Step 4. Return sum
```

How many Assignments? Comparisons?

# Example



**Step 1.** Assign **sum** = 0. Assign **i** = 0.

**Step 2.**

Assign **i** = **i** + 1

Assign **sum** = **sum** + **i**

**Step 3.** Compare ***i*** with n

if **i** < n, back to step 2.

otherwise, if **i** ≥ n, go to step 4.

**Step 4.** Return sum

How many
Assignments?
Comparisons?

# Example

- Sum of n integer S(n) = 0 + 1 + 2 + ... + n - 1

```
int sum = 0;
for (int i = 0; i < n; i++)
      sum = sum + i;
```

Assignment: 2n + 2

```
int sum = 0;
for (int i = 0; i < n ; i++)
      sum = sum + i;
```

Comparison: n + 1

- Running Time: T(n) = 3n + 3

# Algorithm Growth Rates



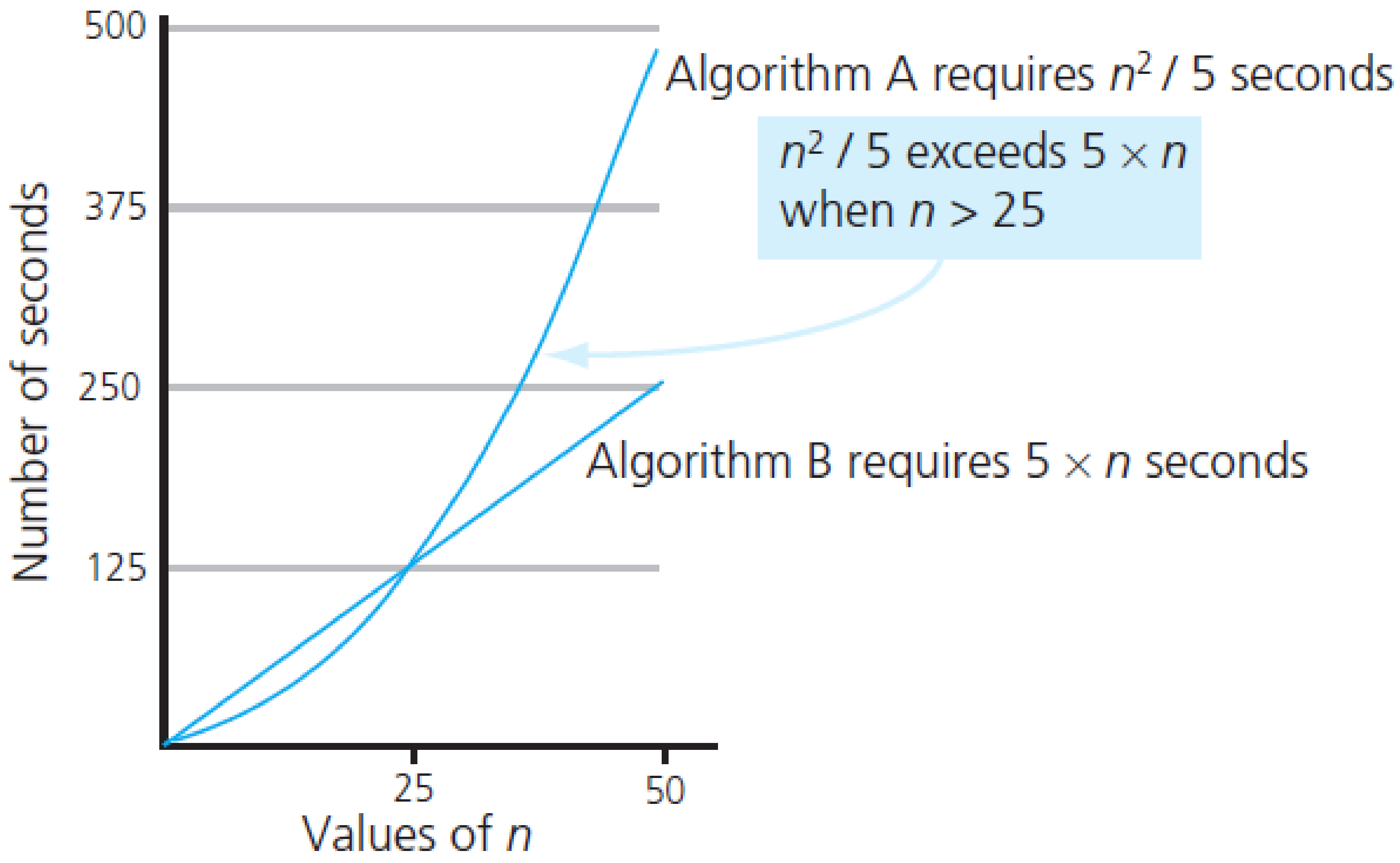
- Measure algorithm's time requirement as a **function of problem size**
- Compare algorithm efficiencies for **large problems**
- Look only at **significant differences**.

# Analysis & Big O Notation

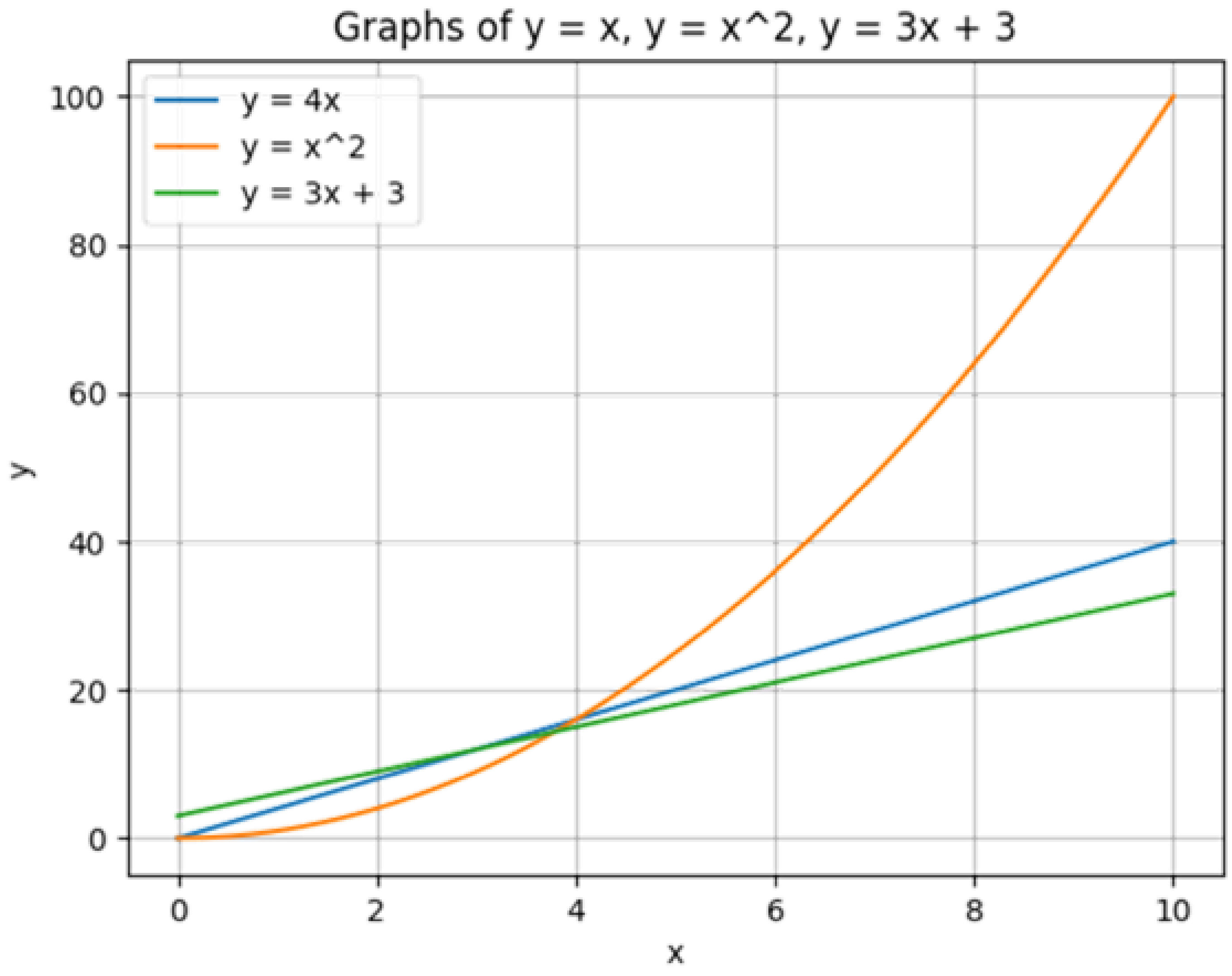
# Algorithm Growth Rates



- Time requirements as a function of the problem size n

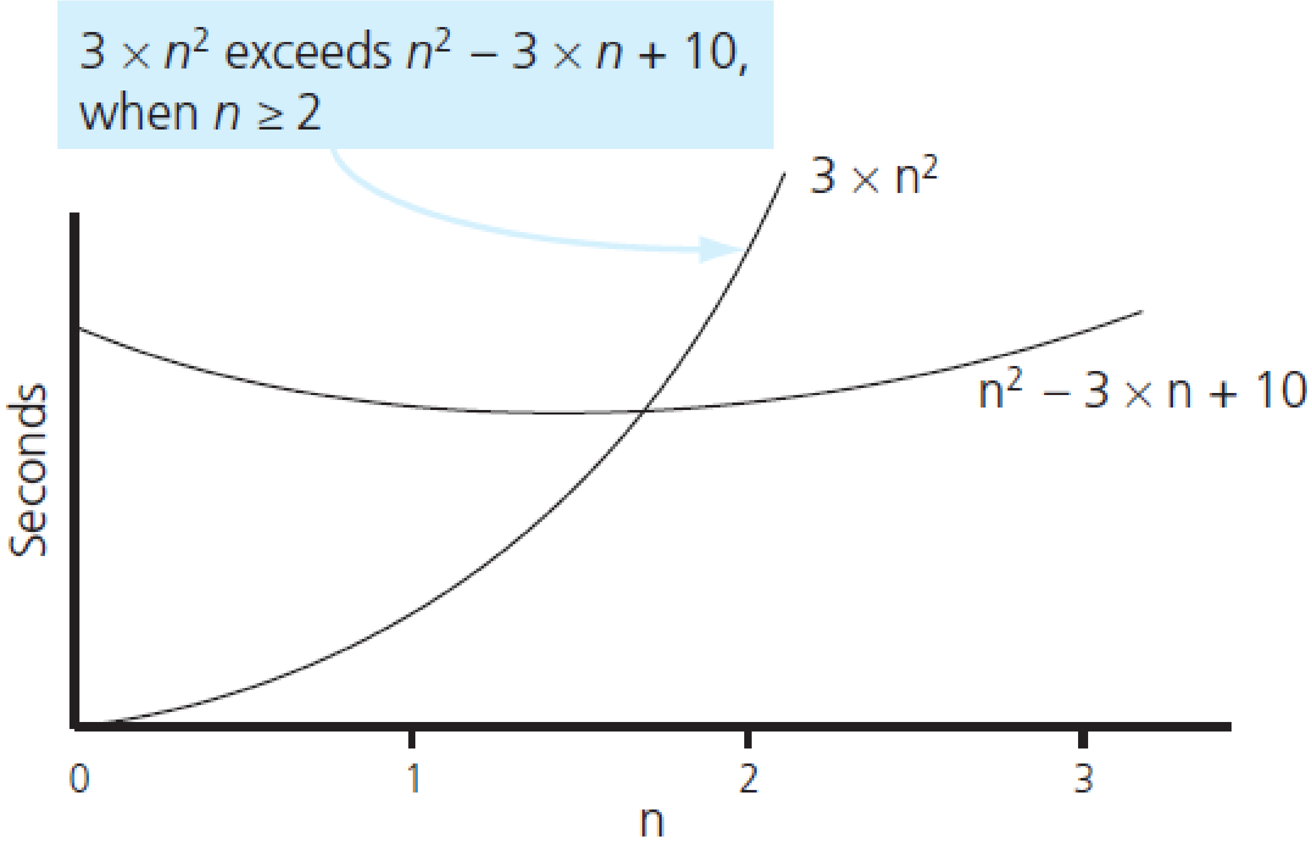


- Definition:
  - Algorithm A with time unit function $g(n)$ is order $f(n)$
    - Denoted $g(n)$ is $O(f(n))$
  - If constants $k$ and $n_0$ exist such that A requires no more than $k \times f(n)$ time units to solve a problem of size $n \geq n_0$.
  - It means that for all $n \geq n_0$, $g(n) \leq c \bullet f(n)$



Graphs of y = x, y = x^2, y = 3x + 3
y = 4x
y = x^2
y = 3x + 3
100
80
60
40
20
0
y
0
2
4
6
8
10
x

- An algorithm requires $\mathbf{g(n)= n^2 - 3n + 10}$ (time units).
- What is the order of algorithm?
  - Hint: Find the values $\mathbf{k}$ and $\mathbf{n_0}$.




- **With c = 3 and $n_0$ = 2, g(n) = O($n^2$)**

- Another algorithm requires $n^2 + 3n + 10$ time units. What is the order of this algorithm?

- How about the order of an algorithm requiring

$$(n + 1) \times (a + c) + n \times w$$

time units?

Where n is the problem size

# Common Growth-Rate Functions



- **f(n) =**
  - 1: Constant
  - $\log_2 n$: Logarithmic
  - n: Linear
  - $n\log_2 n$: Linearithmic
  - $n^2$: Quadratic
  - $n^3$: Cubic
  - $2^n$: Exponential

- **Order of growth of some common functions**

$$O(1) < O(\log_2 n) < O(n) < O(n \times \log_2 n) < O(n^2) < O(n^3) < O(2^n)$$

# Common Growth-Rate Functions

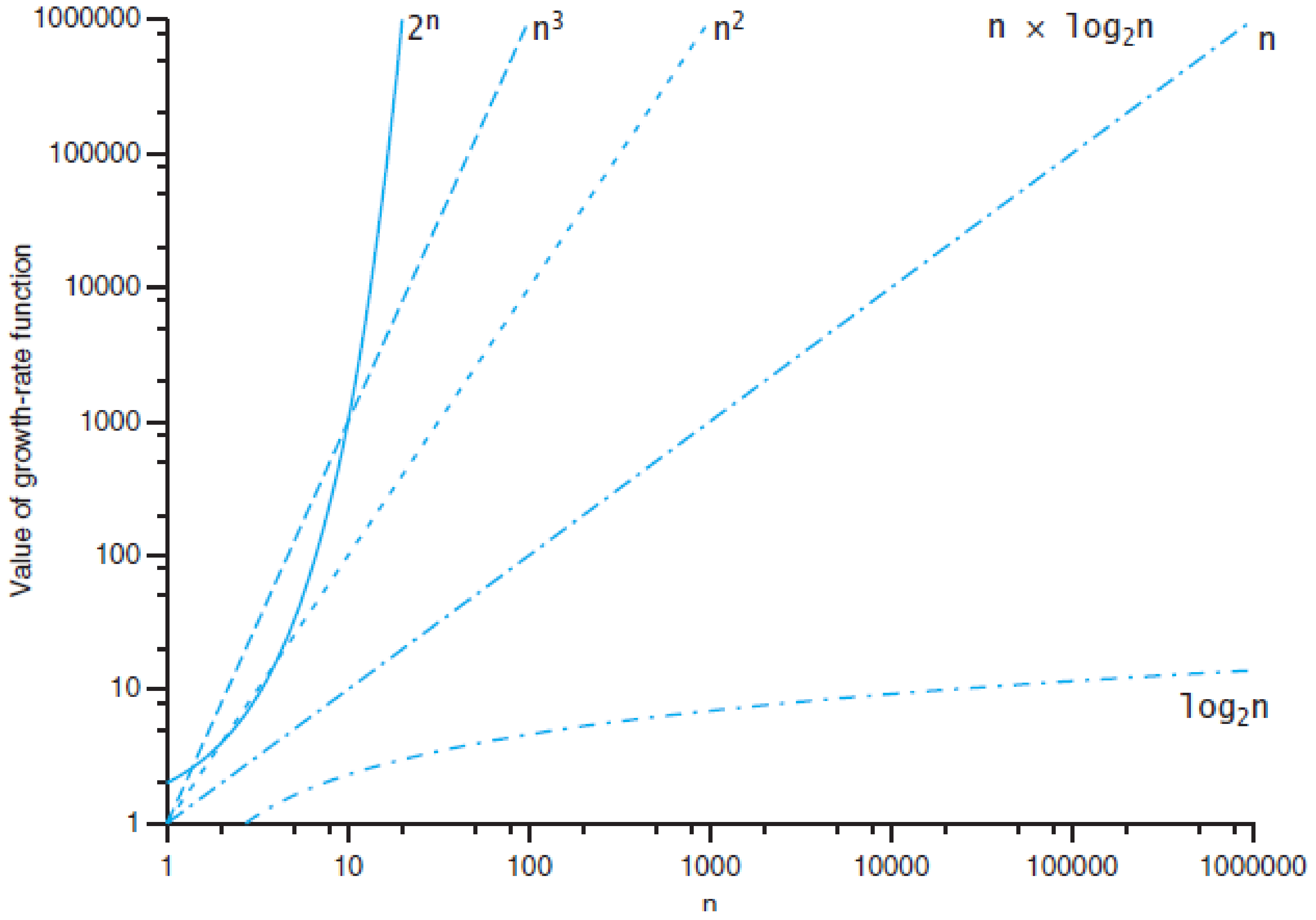
- **A comparison of growth-rate functions in tabular form**

| Function | n = 10 | 100 | 1,000 | 10,000 | 100,000 | 1,000,000 |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $\log_2 n$ | 3 | 6 | 9 | 13 | 16 | 19 |
| n | 10 | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
| $n \times \log_2 n$ | 30 | 664 | 9,965 | $10^5$ | $10^6$ | $10^7$ |
| $n^2$ | $10^2$ | $10^4$ | $10^6$ | $10^8$ | $10^{10}$ | $10^{12}$ |
| $n^3$ | $10^3$ | $10^6$ | $10^9$ | $10^{12}$ | $10^{15}$ | $10^{18}$ |
| $2^n$ | $10^3$ | $10^{30}$ | $10^{301}$ | $10^{3,010}$ | $10^{30,103}$ | $10^{301,030}$ |

# Common Growth-Rate Functions



- **A comparison of growth-rate functions in graphical form**



# Properties of Growth-Rate Functions

- Ignore low-order terms

$$O(n^3 + 4n^2 + 3n) == O(n^3)$$

- Ignore a multiplicative constant in the high-order term

$$O(5n^3) == O(n^3)$$

- Can combine growth rate functions

$$O(f(n)) + O(g(n)) = O(f(n) + g(n))$$

# Some Useful Results

- Constant Multiplication:

If $f(n)$ is $O(g(n))$

then $c*f(n)$ is $O(g(n))$, where $c$ is a constant.

- Polynomial Function:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 \text{ is } O(x^n)$$

# Some Useful Results

- **Summation Function:**
  - If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$
  - Then $f_1(n) + f_2(n)$ is $\mathbf{O(max(g_1(n),g_2(n)))}$

- Multiplication Function:
  - If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$
  - Then $f_1(n) \times f_2(n)$ is $\mathbf{O(g_1(n) \times g_2(n))}$

# Note



- Use like this:
  - `f(x)` **is** `O(g(x))`, or
  - `f(x)` **is of order** `g(x)`, or
  - `f(x)` **has order** `g(x)`

# Quiz

- **Are these functions of order O(x)?**
  - `f(x) = 10`
  - `f(x) = 3x + 7`
  - $f(x) = 2x^2 + 2$

# Quiz

- **Give the order of growth (as a function of n) of the running time of the following function?**
  - `f(n) = (2 + n) * (3 + log2n)`
  - $f(n) = 11 * \log_2 n + n/2 - 3542$
  - `f(n) = n * (3 + n) - 7 * n`
  - $f(n) = \log_2(n^2) + n$

# Quiz

- **Give the order of growth (as a function of n) of the running time of the following function?**
  - `f(n) = (2 + n) * (3 + log2n)`
  - $f(n) = 11 * \log_2 n + n/2 - 3542$
  - `f(n) = n * (3 + n) - 7 * n`
  - $f(n) = \log_2(n^2) + n$

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

$$\log_a x^y = y \log_a x$$

# Algorithm Efficiency

# Algorithm Efficiency

- Best case scenario
- Worst case scenario
- Average case scenario

# Example



- Input: **???**
- Output: **???**
- **Step 1.** Set the first integer the temporary maximum value (`temp_max`).
- **Step 2.** Compare the current value with the `temp_max`.
  - If it is greater than, assign the current value to `temp_max`.
- **Step 3.** If there is other integer in the list, move to next value. Back to step 2.
- **Step 4.** If there is no more integer in the list, stop.
- **Step 5.** return `temp_max` (the maximum value of the list).

# Example



- Input:
- Output:
- Step 1. Assign **i** = 0
- Step 2. **while** (i < n and x ≠ $a_i$)

  i = i + 1
- Step 3.
  - **if** i < n, **return** i
  - Otherwise (i ≥ n), **return** -1 to tell that **x** does not exist in list **a**.

# Example



- Use comparisons for counting.
- Worst case:
  - When it occurs?
  - How many operations?
- Best case:
  - When it occurs?
  - How many operations?

- Use comparisons for counting.

- Average case:
  - If x is found at position $i^{th}$, the number of comparisons is

`2i + 1`

  - The average number of comparisons is:

$$\frac{3+5+7+..+(2n+1)}{n}=\frac{2(1+2+3+...+n)+n}{n}=\frac{2\frac{n(n+1)}{2}+n}{n}=n+2$$

- Decide **n** – the input size
- Identify the algorithm's **basic operation** (as a rule, it is in the innermost loop)
- Check whether the number of times the basic operation is executed depends only on n
  - If it depends on some additional property, specify the **worst-case** for Big-Oh
- Set up a **sum** expressing the number of times the algorithm's basic operation is executed.
- Find a closed-form formula for the count and **establish its order of growth.**

- **Example:** Check whether all the elements in a given array of *n* elements are distinct.

```
UniqueElements(A[0..n − 1])
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
 for i ← 0 to n − 2 do
     for j ← i + 1 to n − 1 do
         if A[i] = A[j]
             return false
 return true
```

Basic operation

- Worst-case:

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2$$

# Exercise

# Exercise



Propose an algorithm to calculate the value of *S* defined below. What order does the algorithm have?

$$S = 1 + \frac{1}{2} + \frac{1}{6} + \ldots + \frac{1}{n!}$$

How many comparisons, assignments are there in the following code fragment with the size *n*?

```
sum = 0;
for (i = 0; i < n; i++)
{
    std::cin >> x;
    sum = sum + x;
}
```

# Exercise



How many assignments are there in the following code fragment with the size *n*?

```
for (i = 0; i < n ; i++)
    for (j = 0; j < n; j++){
        C[i][j] = 0;
        for (k = 0; k < n; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

# Exercise



Give the order of growth (as a function of n) of the running time of the following code fragment:

```
int sum = 0;
for (int i = n; i > 0; i /= 2)
    for (int j = 0; j < n; j++)
        sum++;
```

Give the order of growth (as a function of N) of the running time of the following code fragment:

```
int sum = 0;
for (int i = 1; i < N; i *= 2)
    for (int j = 0; j < N; j++)
        sum++;
```

Give the order of growth (as a function of N) of the running time of the following code fragment:

```
int sum = 0;
for (int i = 1; i < n; i *= 2)
    for (int j = 0; j < i; j++)
        sum++;
```

# Exercise



1. $\sum_{i=l}^{u} 1 = \underbrace{1+1+\cdots+1}_{u-l+1 \text{ times}} = u - l + 1$ ($l$, $u$ are integer limits, $l \le u$); $\sum_{i=1}^{n} 1 = n$

2. $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$

3. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$

4. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1}n^{k+1}$

5. $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1}-1}{a-1}$ $(a \neq 1)$; $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

6. $\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$

# Exercise



Give the order of growth (as a function of n) of the running time of the following code fragment:

```
int sum = 0;
for (int k = 1; k < n; k = k*2)
    for (int i = 0; i < k; i++)
        sum++;
```

# Geometric Sequence



- A geometric sequence, or geometric progression, is a sequence of numbers where each successive number is the product of the previous number and some constant r

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

- common ratio: $r = \dfrac{a_n}{a_{n-1}}$

# Geometric Series



- A geometric series is the sum of the terms of a geometric sequence
- Consider the sequence as follows:

$$S_n = a_1 + a_1 r + a_1 r^2 + \ldots + a_1 r^{n-1}$$

- the formula for the n-th partial sum of a geometric sequence

$$S_n = \frac{a_1(1-r^n)}{1-r} \quad (r \neq 1)$$

# Exercise



What is the goal of the following code and give the order of growth function of its running time

```
void mystery(int*& arr, int n, int a[], int k) {
    arr = new int[n]{0};
    for (int i = 0; i < k; i++) arr[i] = a[i];
    for (int i = k; i < n; ++i) {
        int j = 0;
        while (j < k){
            *(arr + i) += *(arr + i - j - 1);
            j++;
        }
    }
}
```

# Exercise



- Hint: this is the main function for the above fragment of code

```
int main() {
    int *arr = NULL;
    int start[3] = {-2, 0, 3};
    mystery(arr, 7, start, 3);
    for (int i = 0; i < 7; i++)
        std::cout << arr[i] << std::endl;
    if (arr != NULL) delete[] arr;
    return 0;
}
```

- What is the output of the function

THANK YOU

for YOUR ATTENTION