Data structure and Algorithms

Thanh-Hai Tran

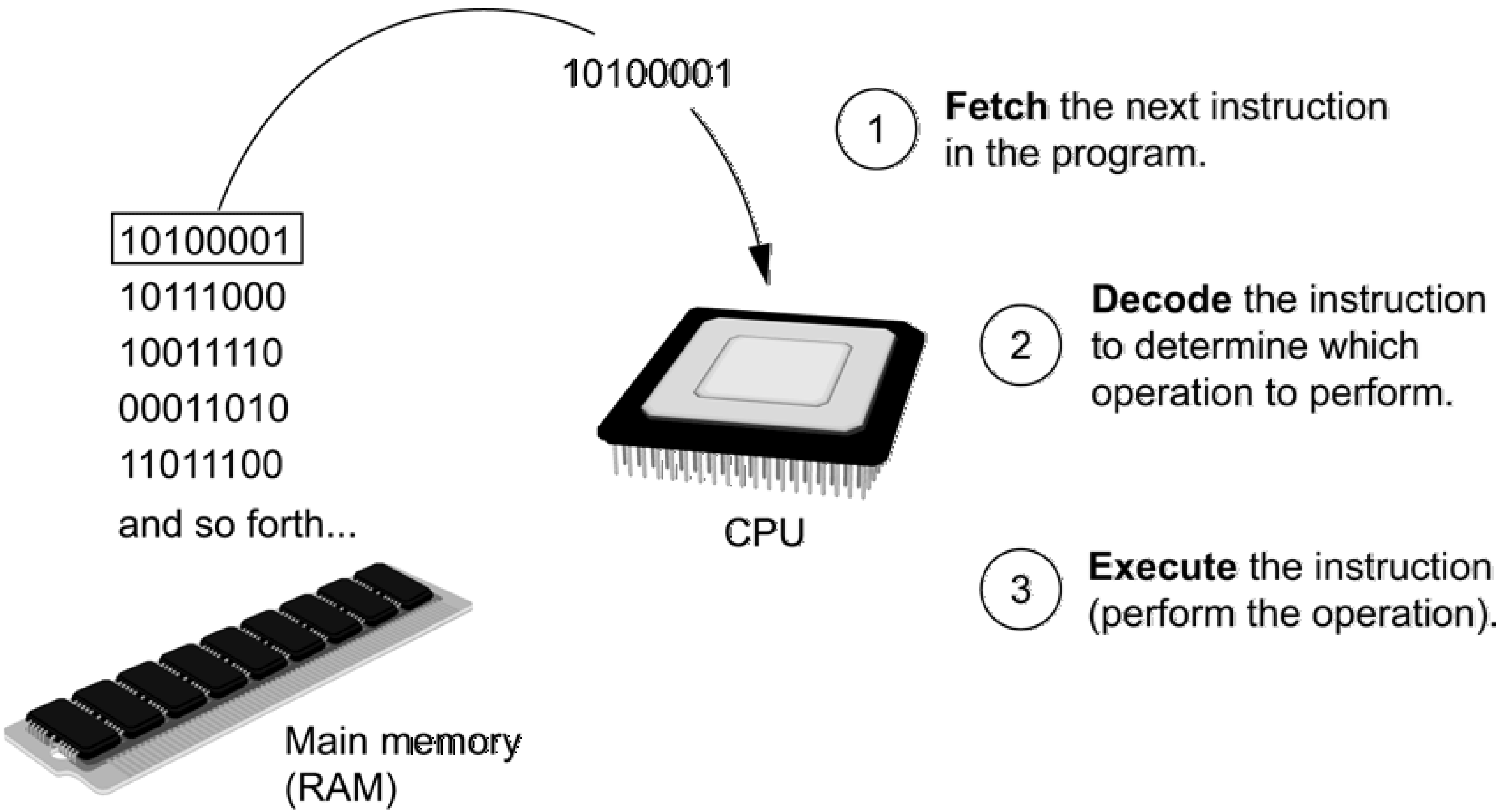
Electronics and Computer Engineering
School of Electronics and Telecommunications

Hanoi University of Science and Technology
1 Dai Co Viet - Hanoi - Vietnam

Outline

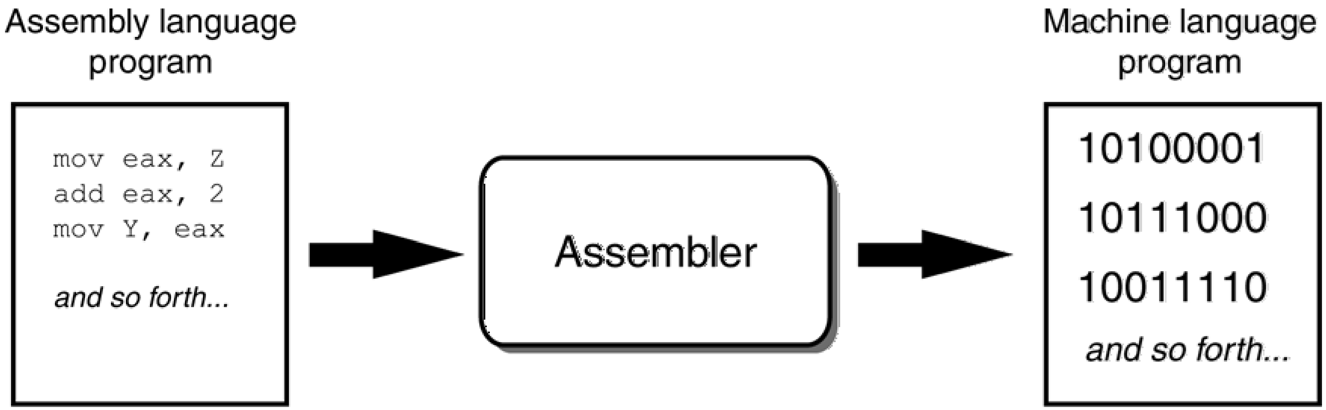
- Objectives and contents of the course
- Basic concepts of DS & algorithms
- Languages for expressing algorithms
- Design and Analysis of algorithms

How a Computer Program Works



Programming Languages

- **Assembly language is referred to as a low-level language**



- **High-level languages allow you to create powerful and complex programs without knowing how the CPU works, using words that are easy to understand**
  - C/C++, Java, JavaScript, Python, Visual Basic, C#,…
  - Compilers or interpreters needed

Language to express algorithms

There two rules when choosing a language to explain algorithms:

- **Independence of algorithms**: used language helps readers to make clear the ideas (logics) of explained algorithms.

  → Suitable languages are natural languages and formal languages (like algorithm diagram, mathematical symbols).
- **Implementable of algorithms**: used language helps developers to understand how the explained algorithm can be implemented by computer program.

  → Suitable languages are programming languages.

Language to express algorithms

- Natural languages
- Algorithm diagrams: Using visual symbols
- Pseudo codes (Ngôn ngữ tựa lập trình): the language is between natural language and programming language
- Programming languages, like C/C++, Java, …

Pseudo codes

- **A pseudo code consists of:**
  - Declaration of new data types
  - Declaration of data objects
  - Access to components (fields) of data objects
  - Definition of operations for data types
  - Call/Use operations

Pseudo codes

- **Declaration of new data type:**

  `new datatype` *typename {fields}*

- **Exp:**

arrays:

new datatype *vector {array[1..100] of integer}*

new datatype *matrix {array[1..10][1..20] of integer}*

record (struct):

new datatype person {ID, name, age}

new datatype person {ID string, name string, age integer}

new datatype node {info, next →node}

pointers:

new datatype →node

Pseudo codes

- **Some simple data types:**
  - *integer*:
  - *float*:
  - *number*: maybe integer or float
  - *char*: character
  - *string*: string of characters
  - *bool*: boolean (true/false)
  - *array[min..max]*: array with elements indexed from *min* to *max* (max-min+1 elements)

Pseudo codes

- **Declaration of data objects:**

```
object-name typename
obj-name1, obj-name2 typename
```

- **Exp:**

```
v vector
v1, v2 vector
m matrix
p node
q →node
```

Pseudo codes

- **Access to elements (fields) of data objects:**

```
new datatype vector {array[1..100] of integer}
v vector
Elements of v:
      v[1], v[2], …, v[100]

new datatype matrix {array[1..10][1..20] of
integer}
m matrix
Elements of m:
      m[1,1], m[1,2], … , m[10,20]
```

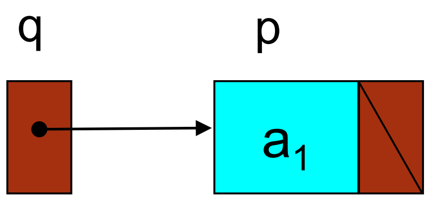
Pseudo codes

- **Access to elements of data objects:**

```
new datatype node {info, next →node}
p node
Elements of p:
      p.info
      p.next

q →node
Elements of q:
      q→info
      q→next
```

Pseudo codes




```
p node
p.info = a1
p.next = null
q →node
q = →p
print q→info  //a1
print q→next  //null
```

Pseudo codes

- **Definition of operation:**

```
opName([in|out|in-out par1 typename,
…])
{
    statements
}
```

- **With:**
  - *opName*: name of operation
  - *par1, …*: names of parameters, each belongs to some datatype. There are 3 types of parameters:
    - *in*: input only parameter
    - *out*: output only parameter
    - *in-out*: parameter with both roles input and output
  - *Statements*: sequence of statements that change inputs into expected outputs.

Pseudo codes

- **Exp:**

```
sum(in a, in b, out c)
{
    c = a + b;
}
min(in a array[1..N], out m)
{   m = 1;
    for i=2 to N
        if (a[i]<a[m]) m = i;
}
```

Pseudo codes

- **Some simple statements and compound statements (also called *control statements*)**
  - ‘=’ : assignment
  - *if, if..else*: conditional (branching)
  - *for, while, repeat..until*: loop
  - *new p*: memory allocation for pointer p;
  - *delete p*: deallocation of memory occupied by p;

Outline

- Objectives and contents of the course
- Basic concepts of DS & algorithms
- Languages for expressing algorithms
- Design and Analysis of algorithms

Design and Analysis of algorithms

- **Design of algorithms:** The process of transforming specification of algorithm into structure of the program that implements the algorithm
- In general, it includes two steps (phases):
  - **General design (Thiết kế sơ bộ):** this step needs to identify clearly components (also called modules) of the algorithm. The method is normally used in this step is top-down design which helps to identify functionalities of each module, and their relationships.
  - **Detail design (Thiết kế chi tiết):** this step begins to implement (coding) modules, one by one. Then all implementations need to be combined into a complete program. This step usually uses the design method called stepwise refinement method (phương pháp *tinh chỉnh từng bước*).

Analysis of algorithms

- **Determine the correctness of algorithms:**
  - Proof by induction (Chứng minh bằng quy nạp)
  - Proof by counter-example (Chứng minh bằng phản ví dụ)
- **Estimation of runtime**
  - Manual measures: using clocks (usually in programs)
  - By theory: estimation of algorithm complexity (xác định độ phức tạp của giải thuật)
- **Estimation of used memory space**

Analysis of algorithms

- **Algorithms need to be analyzed for their efficiency, in terms of**
  - Running time ➔ computational complexity
  - Size of used memory ➔ memory complexity
- **An algorithm may run faster/slower and use more/less on certain data sets than on others**
  ➔ many indicators for assessing the efficiency:
  - Average case
  - Best case (lower bound)
  - Worst case (upper bound)
  - Most common case
- **How to measure complexity?**
  - Experimental studies
  - Theoretical analysis with pseudo-code, flowcharts

Analyses of computational times

- **Experimental method**
  - Setting algorithm in programming language
  - Run the program with different input data
  - Measure program execution time and evaluate the increase
  - Size relative to the size of the input data
- **Limitations:**
  - Limitation in the quantity and quality of test samples
  - Requires testing environment (hardware and software) should be uniform, stable

Analyses of computational times

- **Theoretical methods**
  - Able to review any input data
  - Used to evaluate algorithms independent of testing environment
  - Used with high-level descriptions of the algorithm
- **Implementing this method must take care of**
  - Language describing algorithm
  - Determination of the time measurement
  - An approach to generalizing time complexity

Analyses of computational times

- **Measuring time used in the method theoretical analysis**
  - Basic math operation is performed with time intercepted by an constant that is independent of data size
- **Measuring time of a algorithm is determined by counting the number of basic operations that the algorithm realizes**

$$T(n) \approx c_{op*}C(n)$$

- **Typical operations:** assignment, function call, arithmetic operation, array reference, return, comparasion

An example

- **Line 1: 2 typical operations**
- **Line 2:**
  - Assignment i= 1 (1)
  - Comparison i< n (n times)
- **Inside loop (n-1) times**
  - One comparison: 2(n-1)
  - One assignment 2(n-1)
  - Increase I = I +1 2(n-1)
- **Line 3: a return 1**
- **Totally: 2 + 1 + n + 6(n-1) +1 = 7n -2**

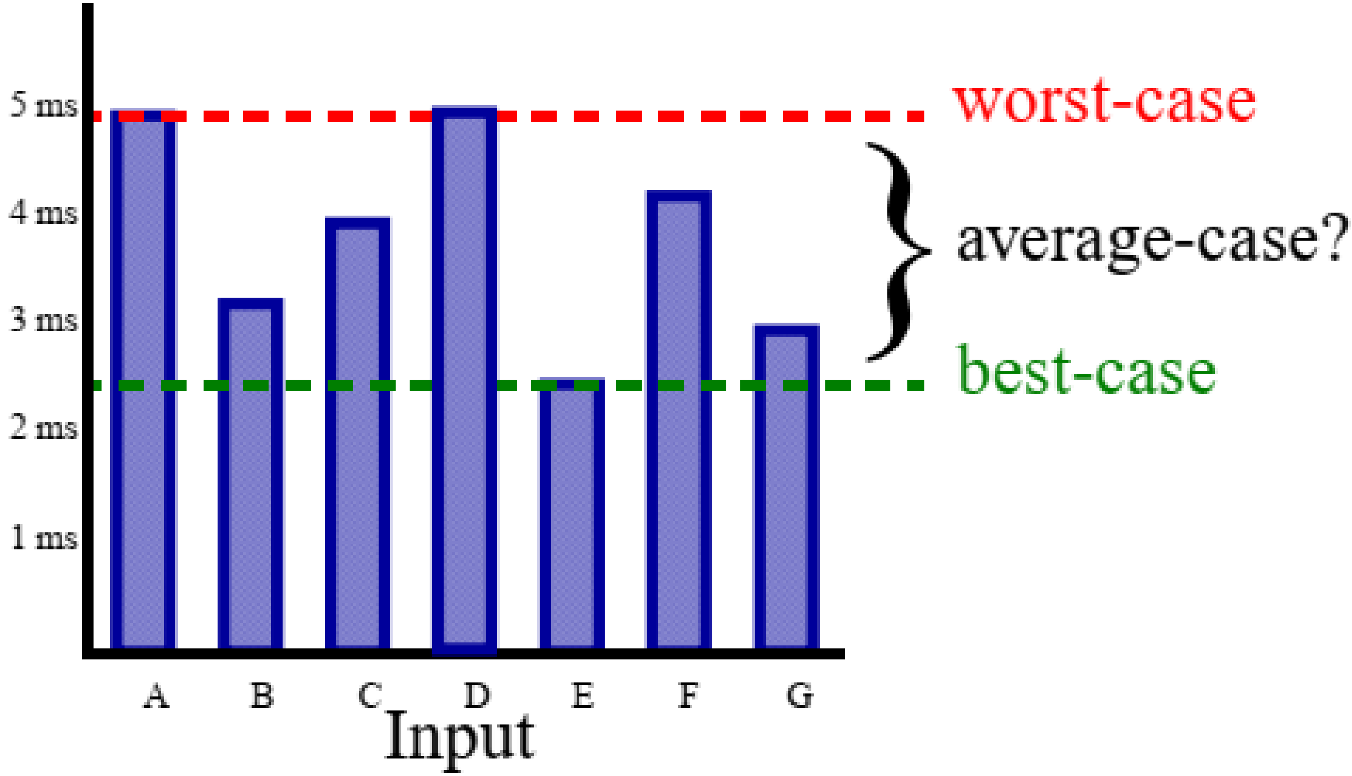
```
Function ARRAY-MAX(A,n)
Đâu vào : mảng A gồm n phần tử.
Đầu ra: phần tử lớn nhất trong
mảng
Begin
1. currentMax = A[0]
2. for i = 1 to n-1 do
      if currentMax < A[i] then
      currentMax = A[i]

3. return currentMax
End.
```

Estimation of algorithm complexity

- **Objective:**
  - Given an algorithm *A* with *n* representing its data size. We need to find a formula (function) $T_A(n)$ that expresses the running time of *A* in computers (we usually use $T(n)$ for short).
- **Normally, there are 3 cases for T(n):**
  - Best case $T_b(n)$: the case that algorithm can be run the most quickly
  - Worse case $T_w(n)$: the case that algorithm can be run the most slowly
  - Average case $T_a(n)$: the average of all cases.

Estimation of algorithm complexity



An example

- **Sequential search algorithm:** search for a value in an array

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]
4	8	7	10	21	14	22	36	62	91	77	81

- Worst case: n
- Best case: 1
- Average case: T(n) = $\sum ip_i$ where is p_i is the probability that value of interest appears at a[i]. When p_i = 1/n the total number will be (n+1)/2

Analysis of algorithms

- Many algorithms are simply too hard to analyze mathematically.
- There may not be sufficient information to calculate the behavior of the algorithm in the average case.
- Big O analysis only tells us how the algorithm grows with the size of the problem, not how efficient it is, as it does not consider the programming effort.
- It ignores important constants.
- For example, if one algorithm takes O(n2) time to execute and the other takes O(100000n2) time to execute, then as per Big O, both algorithm have equal time complexity. In real-time systems, this may be a serious consideration.

Big O concept (ô lớn)

- *Definition*: Given an integer n that is not negative, and two functions t(n) và g(n) defined in non-negative domains. We say that:

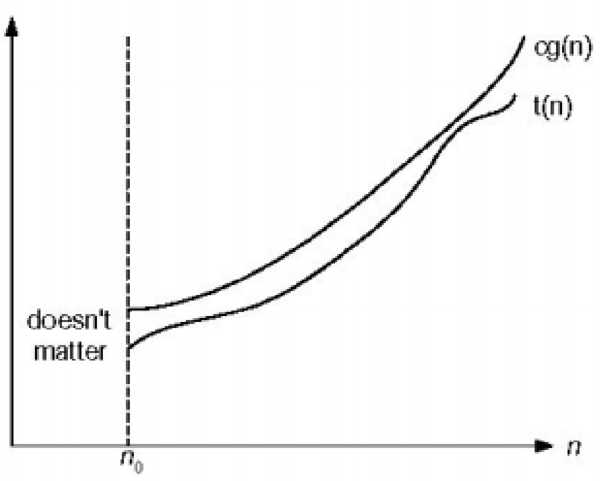
  $$t(n) = O(g(n)) \text{ (t(n) is big O of g(n))}$$

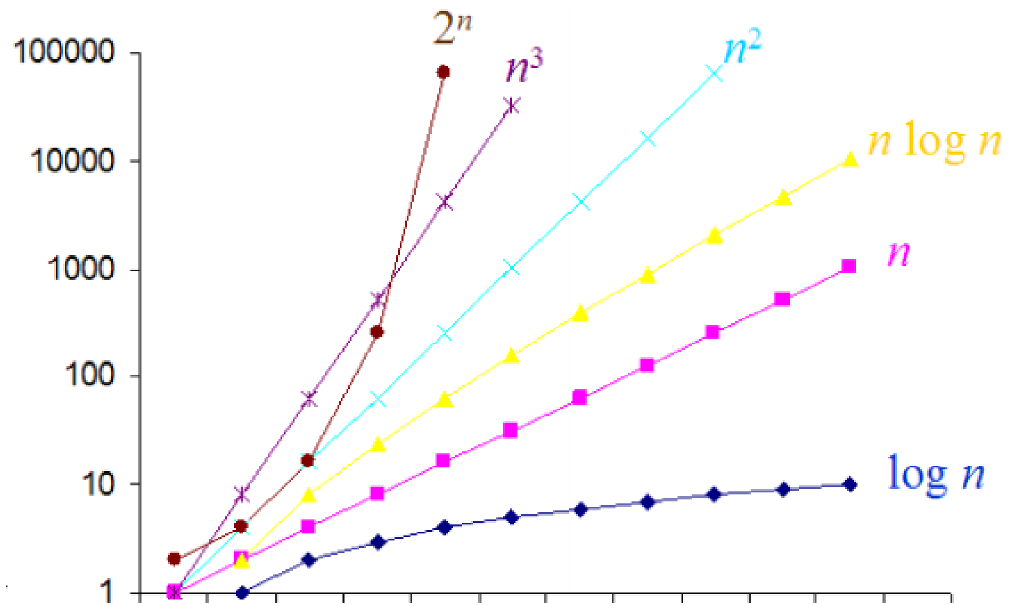
  if and only if there exists a constant C and n_0, so that:

  $$\forall\, n \geq n_0 \text{ then } t(n) \leq C.g(n)$$

- Meaning of big O: t(n) = O (g(n)) means that for $\forall\, n \geq n_0$ g(n) is above asymptote of t(n)

Big O concept (ô lớn)





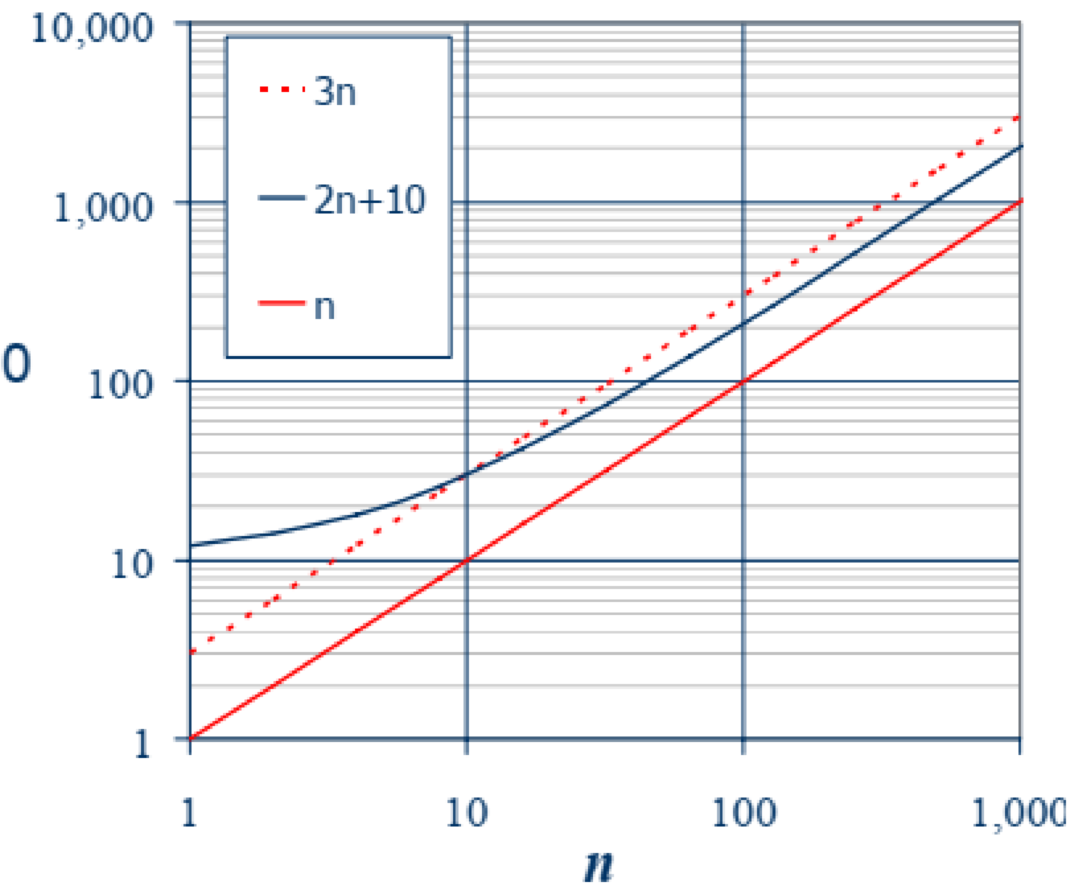
Example

- **Exp: Given t(n) = 3n. We can easily find some functions f(n) that are big O of t(n) as follows:**
  - with f(n) = n: T(n) = O (n), because with C = 3 and

    $n_0 = 0$, we have $\forall$ n $\geq$ 0 then 3n $\leq$ 3.n.
  - with f(n) = n^2: T(n) = O (n^2), because with $n_0 = 3$, C = 1, we have $\forall$ n $\geq$ 3 then 3n $\leq$ 1.n^2.
- **7n -2 => O(n)**
- **$3n^3+20n^2+5$ => O(n^3)**
- **3log(n)+5 => O(log(n))**

Example

– Ví dụ: Giải thuật có T(n) = 2***n*** + 10 thì có độ phức tạp là ***O***(***n***)
  - $2n + 10 \le cn$
  - $(c - 2)\, n \ge 10$
  - $n \ge 10/(c - 2)$
  - Lấy $c = 3$ và $n_0 = 10$



Example

- P1: If T(n) = O (f(n)) and f(n) = O(g(n))

  ➔ T(n) = O(g(n)).
- Among many functions f(n) being big O of T(n), we always choose the smallest and most simple one f(n) so that T(n) = O(f(n)). In that case f(n) is called *complexity function* (hàm độ lớn hay độ phức tạp).
- Complexity functions usually have forms: 1 (constant), $\log_2(n)$, n, $n.\log_2(n)$, n^2, n^3, n^k, 2^n, k^n.

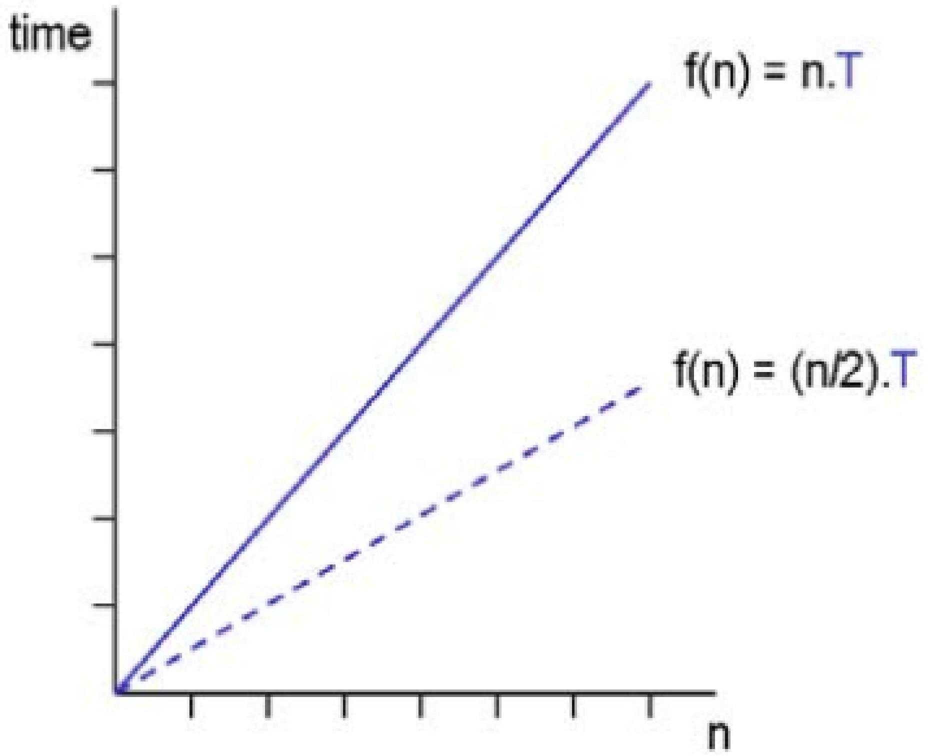
Rules to determine complexity

- Time function T(n) of a algrithm of polynomal rank k then $T(n) = O(n^k)$
- $n^x = O(a^n)$ x>0, a>1
- $\log(n^x) = O(\log(n))$ x > 0
- Additional rule: P1, P2 wwith corresponding T1(n), T2(n) then
  - The computational time of P1 then P2 is T1(n)+T2(n)
  - The complexity could be: O(max(f(n), g(n))) where T1(n) = O(f(n)); T2(n) = O(g(n))
- Multiplicative rule:
  - The computational time of P1 in P2 is T1(n)T2(n)
  - The complexity could be: O(f(n) * g(n)) where T1(n) = O(f(n)); T2(n) = O(g(n))

Example

```
for i = 1 to n
begin
     P; {đoạn giải thuật với thời
     gian thực hiện T}
end
```

```
i: = 1
while (i <= n) do
begin
     P; {đoạn giải thuật với thời
     gian thực hiện T}
     i: = i+2;
end
```



Common classes

- **Linear: O(n)**
- **Quadratic: O(n^2)**
- **Polynomial: O(n^k), $k \geq 1$**
- **Exponential: O(a^n), $n > 1$**
- **Logarithmic: O($\log n$)**
- **Factorial: O($n!$)**

- **Efficiency comparison of classes**
  - O$(\log n)$ < O$(\sqrt{n})$ < O(n) < O(n^2) < O(n^3) < O(2^n) < O(3^n) < O$(n!)$ < O(n^n)