Algorithm Analysis tools

Constant function

- For a given argument/variable n, the function always returns a constant value
- It is independent of variable n
- It is commonly used to approximate the total number of primitive operations in an algorithm
- Most common constant function is $g(n) = 1$
- Any constant value c can be expressed as constant function $f(n) = c.g(1)$

Linear function

- For a given argument/variable n, the function always returns n
- This function arises in algorithm analysis any time we have to do a single basic operation over each of n elements
  - For example, finding min/max value in a list of values
  - Time complexity of linear/sequential search algorithm is linear

Quadratic function

- For a given argument/variable n, the function always returns square of n
- This function arises in algorithm analysis any time we use nested loops
  - The outer loop performs primitive operations in linear time; for each iteration, the inner loop also perform primitive operations in linear time
  - For example, sorting an array in ascending/descending order using Bubble Sort (more later on)
  - Time complexity of most algorithms is quadratic

Cubic function

- For a given argument/variable n, the function always returns n x n x n
- This function is very rarely used in algorithm analysis
  - Rather, a more general class "polynomial" is often used
    - $f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ldots + a_d n^d$

Logarithmic function

- ❑ For a given argument/variable *n*, the function always returns logarithmic value of n
- ❑ Generally, it is written as **$f(n) = \log_b n$,** where b is base which is often 2
- ❑ This function is also very common in algorithm analysis
- ❑ We normally approximate the $\log_b n$ to a value *x*. *x* is number of times *n* is divided by *b* until the division results in a number less than or equal to 1
  - $\log_3 27$ is 3, since 27/3/3/3 = 1.
  - $\log_4 64$ is 3, since 64/4/4/4 = 1
  - $\log_2 12$ is 4, since 12/2/2/2/2 = 0.75 ≤ 1

Log linear function

- For a given argument/variable *n*, the function always returns *n log n*
- Generally, it is written as $f(n) = n \log_b n$, where b is base which is often 2
- This function is also common in algorithm analysis
- Growth rate of log linear function is faster as compared to linear and log functions

Exponential function

- For a given argument/variable n, the function always returns b^n, where b is base and n is power (exponent)
- This function is also common in algorithm analysis
- Growth rate of exponential function is faster than all other functions

Algorithmic runtime

- **Worst-case running time**
  - measures the maximum number of primitive operations executed
  - The worst case can occur fairly often
    - e.g. in searching a database for a particular piece of information
- **Best-case running time**
  - measures the minimum number of primitive operations executed
    - Finding a value in a list, where the value is at the first position
    - Sorting a list of values, where values are already in desired order
- **Average-case running time**
  - the efficiency averaged on all possible inputs
  - maybe difficult to define what “average” means

Complexity classes

- Suppose the execution time of algorithm A is a quadratic function of n (i.e. $an^2 + bn + c$)
- Suppose the execution time of algorithm B is a linear function of n (i.e. $an + b$)
- Suppose the execution time of algorithm C is a an exponential function of n (i.e. $a2^n$)
- For large problems higher order terms dominate the rest
- These three algorithms belong to three different "complexity classes"

Big-O and function growth rate

- We use a convention **O-notation** (also called Big-Oh) to represent different complexity classes
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- $g(n)$ is an upper bound on $f(n)$, i.e. maximum number of primitive operations
- We can use the big-O notation to rank functions according to their growth rate

Big-O: functions ranking

BETTER

- $O(1)$ constant time
- $O(\log n)$ log time
- $O(n)$ linear time
- $O(n \log n)$ log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time

WORSE

Simplifications

- ❑ Keep just one term
  - the fastest growing term (dominates the runtime)
- ❑ No constant coefficients are kept
  - Constant coefficients affected by machines, languages, etc
- ❑ Asymptotic behavior (as *n* gets large) is determined entirely by the dominating term
  - Example: $T(n) = 10\,n^3 + n^2 + 40n + 800$
    - *If* $n = 1{,}000$, *then* $T(n) = 10{,}001{,}040{,}800$
    - error is 0.01% if we drop all but the n^3 (the dominating) term

Big Oh: some examples

- $n^3 - 3n = O(n^3)$
- $1 + 4n = O(n)$
- $7n^2 + 10n + 3 = O(n^2)$
- $2^n + 10n + 3 = O(2^n)$

- Moreover
- $7n^2 + 10n + 3 = O(n^3)$
- $7n^2 + 10n + 3 = O(2^n)$
- $7n^2 + 10n + 3$ is NOT $O(n)$

Big Oh: some examples

The difference is a tight bound and non-tight bound:

- $7n^2 + 10n + 3 = O(n^2)$ is called tight bound
- $7n^2 + 10n + 3 = O(n^3)$ is called non-tight bound

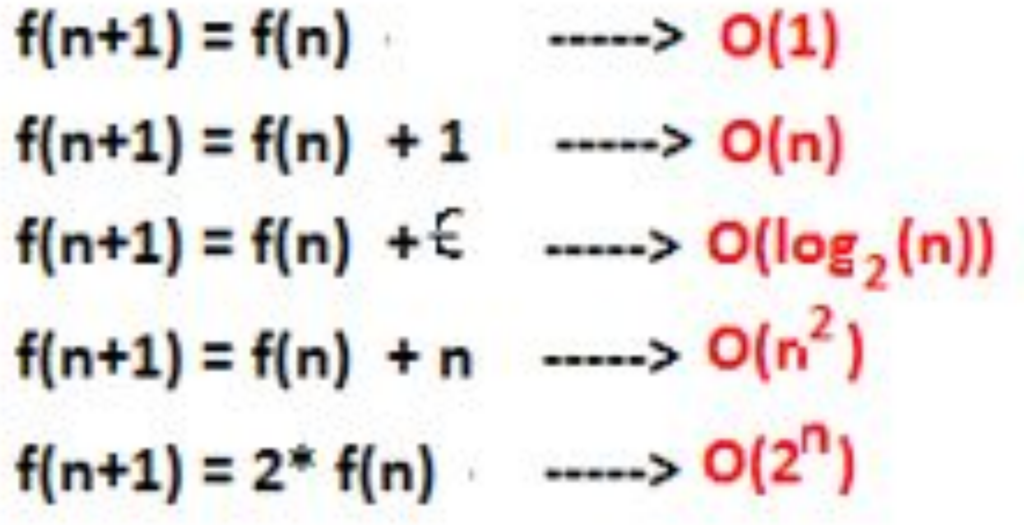
Practice

- ☐ Express the following functions in terms of Big-O notation with a tight bound (a, b and c are constants)
  1. $f(n) = an^2 + bn + c$
  2. $f(n) = 2^n + n \log n + c$
  3. $f(n) = n \log n + b \log n + c$
  4. $f(n) = 2^n + n \log n + 3^n$
  5. $f(n) = 2^n + n \log n + 100 \log n$

Summary & Examples (1)

- four interesting points:
  1. Resources: number of primitive instructions: **time**
  2. Complexity is function of **inputs** (n)
  3. We will focus on the great value of n, Big-O capture the notion of the **asymptotic** value of the number of instructions
  4. **Worst case** (the maximum number of primitive instructions)

Summary & Examples (2)



$f(n+1) = f(n)$ -----> $O(1)$

$f(n+1) = f(n) + 1$ -----> $O(n)$

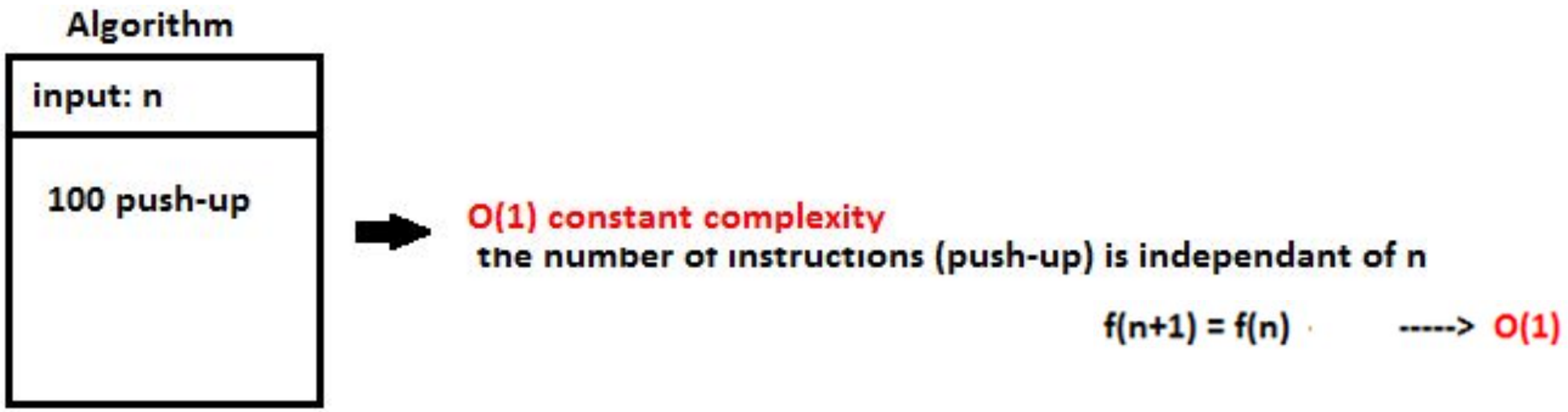
$f(n+1) = f(n) + \epsilon$ -----> $O(\log_2(n))$

$f(n+1) = f(n) + n$ -----> $O(n^2)$

$f(n+1) = 2 * f(n)$ -----> $O(2^n)$

Summary & Examples (3)

- **Problem 1**: prepare a sport competition:
- n: number of remaining days to competition



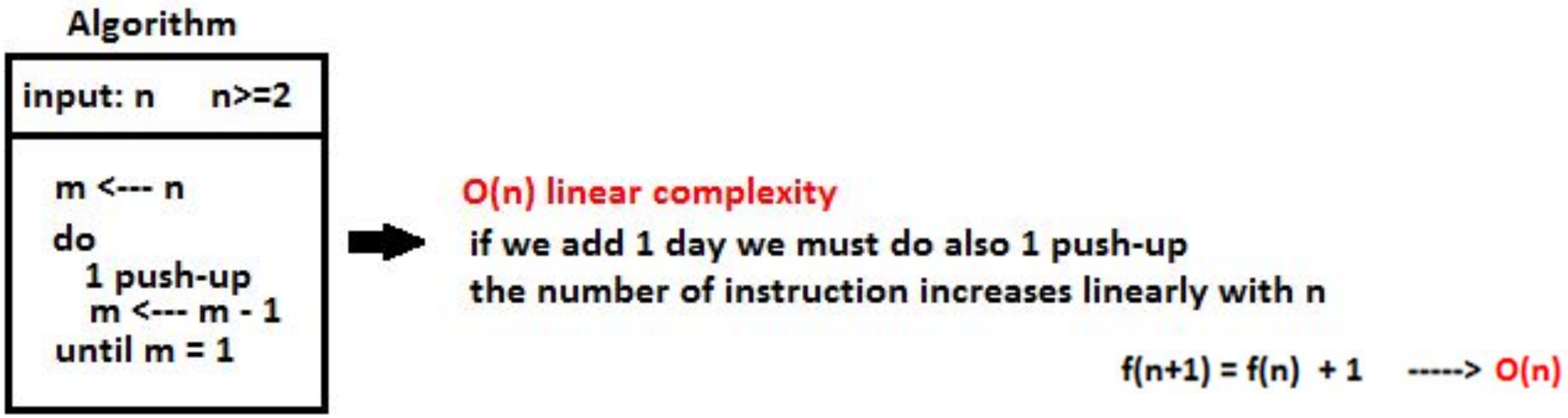
Algorithm

```
input: n
-----------------
100 push-up
```

O(1) constant complexity

the number of instructions (push-up) is independant of n

f(n+1) = f(n) -----> O(1)



Algorithm

```
input: n     n>=2
-----------------
m <--- n
do
   1 push-up
   m <--- m - 1
until m = 1
```

O(n) linear complexity

if we add 1 day we must do also 1 push-up

the number of instruction increases linearly with n

f(n+1) = f(n) + 1 -----> O(n)

Summary & Examples (4)

- **Problem 1**: prepare a sport competition:
- n: number of remaining days to competition

Algorithm

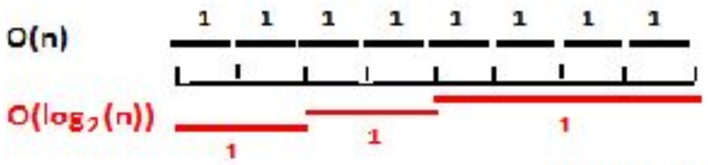
```
input: n      n>=2

m <--- n
do
  1 push-up
  m <--- m/2
until m <= 1
```

O($\log_2(n)$) algorithmic complexity

As a result there will have to be as many "push-up" as we can divide m by 2

For a large n the number of instruction increases too little



$f(n+1) = f(n) + \varepsilon$ -----> O($\log_2(n)$)

Algorithm

```
input: n      n>=2

m <--- n
do
  n push-up
  m <--- m-1
until m = 1
```

O(n^2)) quadratic (polynomial) complexity

+ 1 day -------> "n" push-up

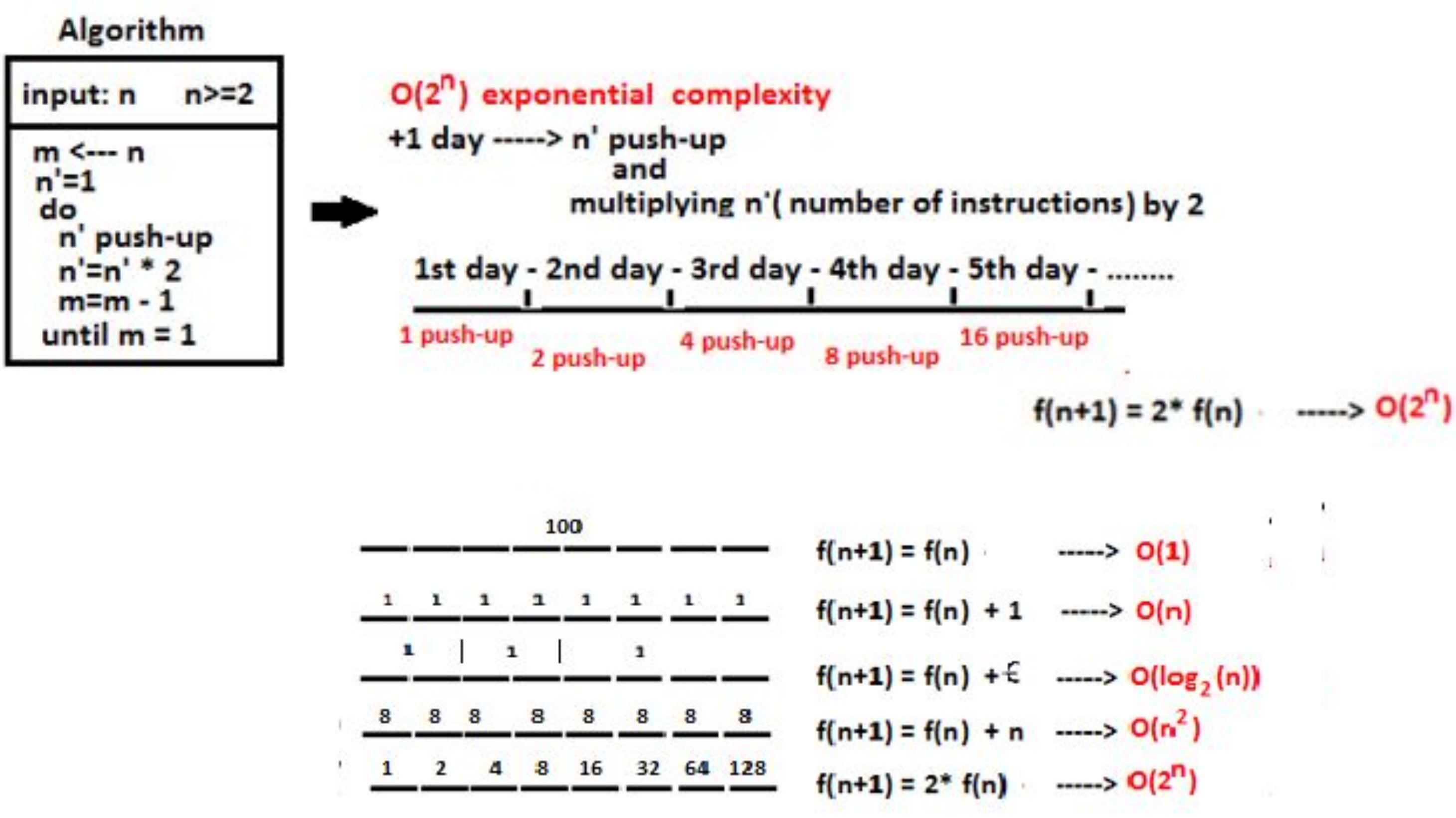
it's like two nested loops :

```
for i= n to 1   for the number of days
  for j=1 to n  for the number of push-up
```

$f(n+1) = f(n) + n$ -----> O(n^2)

Summary & Examples (5)

- **Problem 1**: prepare a sport competition:
- n: number of remaining days to competition



Algorithm

```
input: n      n>=2
m <--- n
n'=1
do
  n' push-up
  n'=n' * 2
  m=m - 1
until m = 1
```

$O(2^n)$ exponential complexity

+1 day -----> n' push-up and multiplying n'(number of instructions) by 2

f(n+1) = 2* f(n) -----> $O(2^n)$

f(n+1) = f(n) -----> O(1)

f(n+1) = f(n) + 1 -----> O(n)

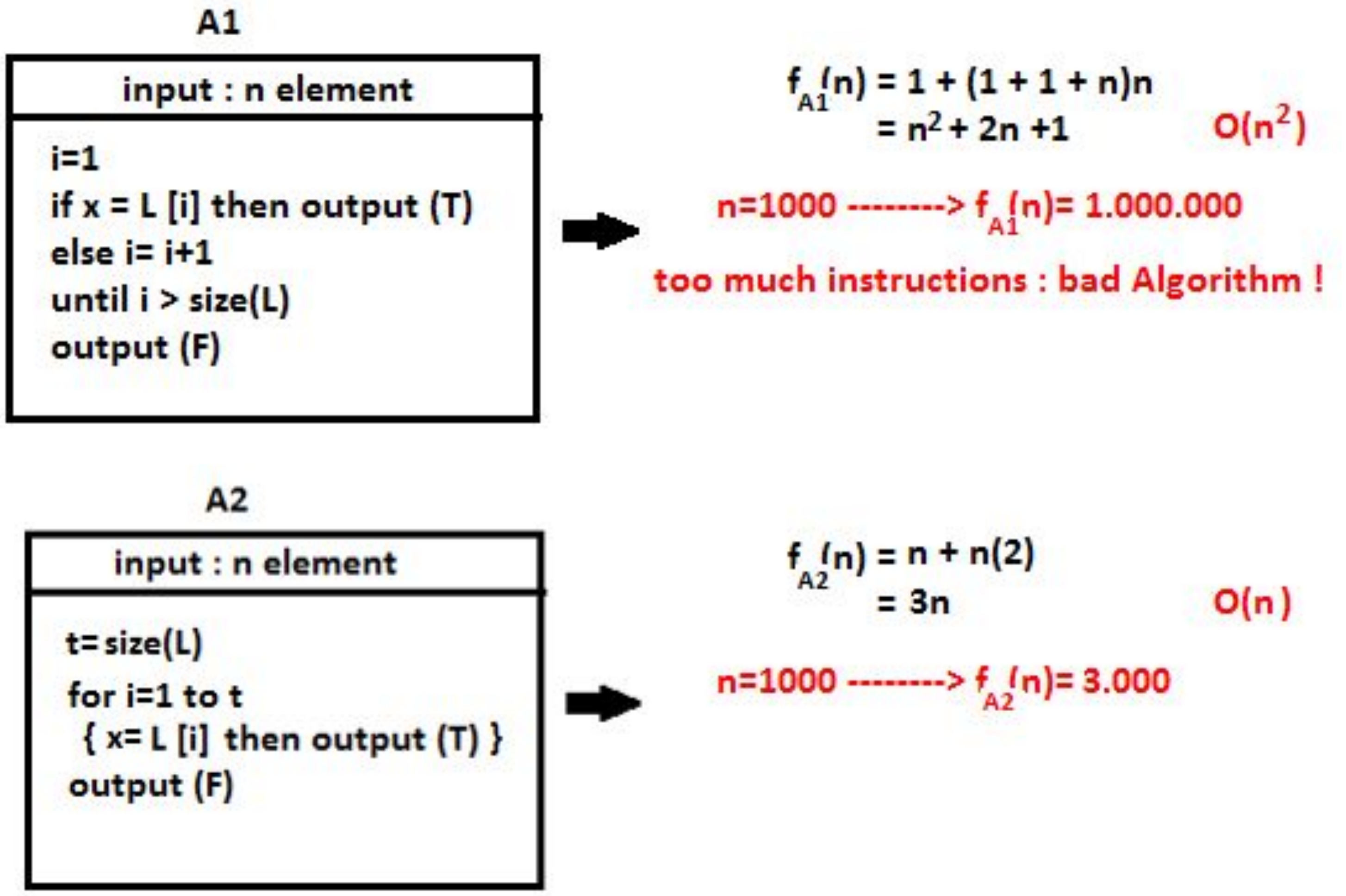
f(n+1) = f(n) + ε -----> $O(\log_2(n))$

f(n+1) = f(n) + n -----> $O(n^2)$

f(n+1) = 2* f(n) -----> $O(2^n)$

Summary & Examples (6)

- **Problem 2**: research (x , L): L[1], L[2],…… L[n]
- n: number of elements



A1

input : n element

```
i=1
if x = L [i] then output (T)
else i= i+1
until i > size(L)
output (F)
```

$f_{A1}(n) = 1 + (1 + 1 + n)n$

$= n^2 + 2n + 1$ $O(n^2)$

n=1000 --------> $f_{A1}(n)$= 1.000.000

too much instructions : bad Algorithm !

A2

input : n element

```
t=size(L)
for i=1 to t
  { x= L [i] then output (T) }
output (F)
```

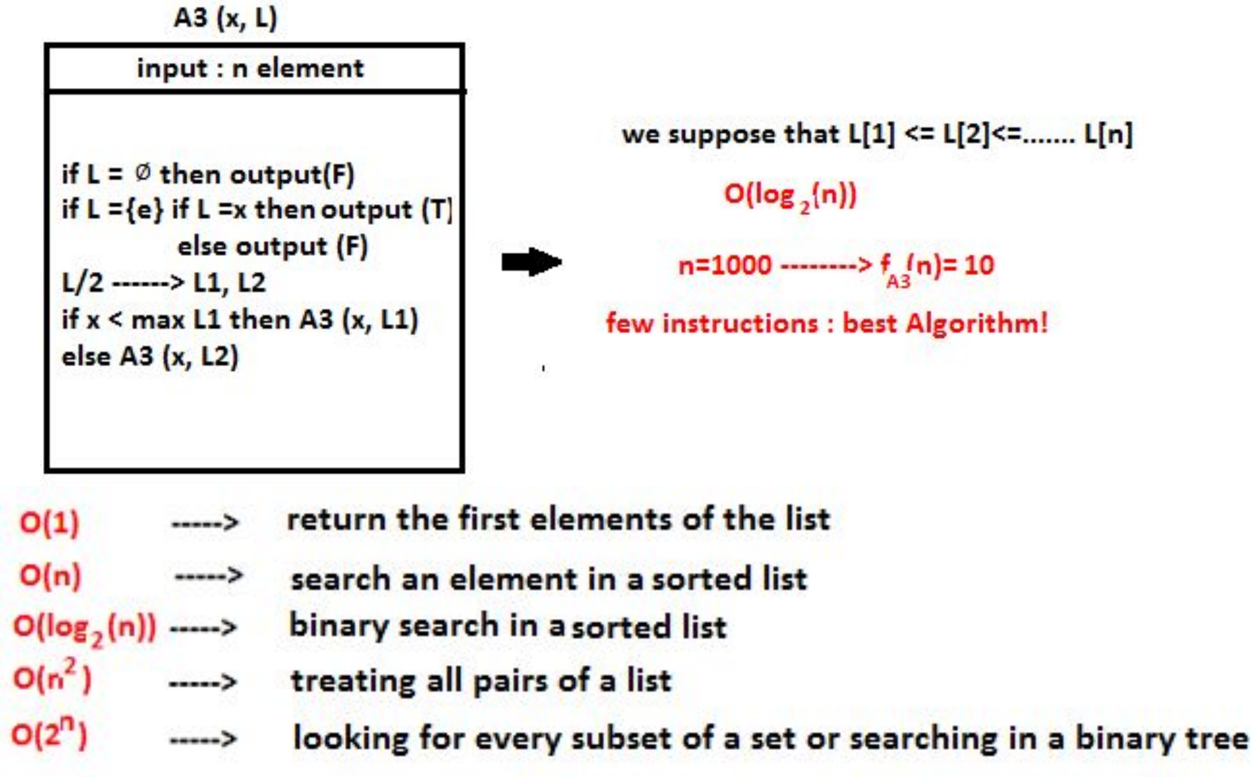
$f_{A2}(n) = n + n(2)$

$= 3n$ $O(n)$

n=1000 --------> $f_{A2}(n)$= 3.000

Summary & Examples (7)

- **Problem 2**: research (x , L): L[1], L[2],…… L[n]
- n: number of elements



A3 (x, L)

input : n element

```
if L = ∅ then output(F)
if L ={e} if L =x then output (T)
          else output (F)
L/2 ------> L1, L2
if x < max L1 then A3 (x, L1)
else A3 (x, L2)
```

we suppose that L[1] <= L[2]<=....... L[n]

$O(\log_2(n))$

n=1000 --------> $f_{A3}(n)= 10$

few instructions : best Algorithm!

$O(1)$	----->	return the first elements of the list
$O(n)$	----->	search an element in a sorted list
$O(\log_2(n))$	----->	binary search in a sorted list
$O(n^2)$	----->	treating all pairs of a list
$O(2^n)$	----->	looking for every subset of a set or searching in a binary tree

Important:
- Count and increment is a fairly simple technique, it allows to get an idea of an algorithm.
- For a complex algorithm it is not always easy to count, but it can provide an interesting reflection track