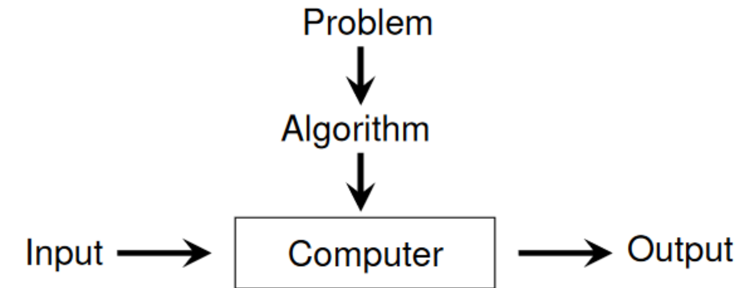
Complexity of algorithms

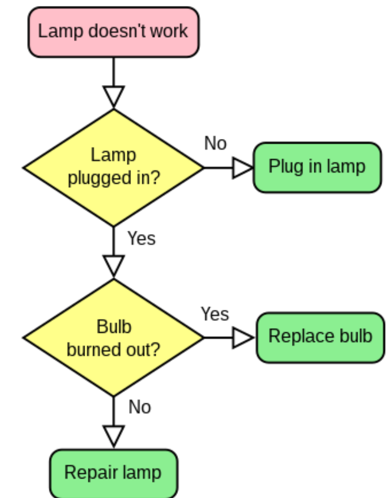
2022/2023

Hemza.loucif@univ-msila,dz

Algorithm

- In mathematics and computer science, an algorithm is a finite sequence of rigorous instructions, typically used to solve a class of specific problems or to perform a computation.
- Algorithms can be expressed in many kinds of notation, including natural languages, pseudocode, flowcharts, or programming languages.
- Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are also often used as a way to define or document algorithms.



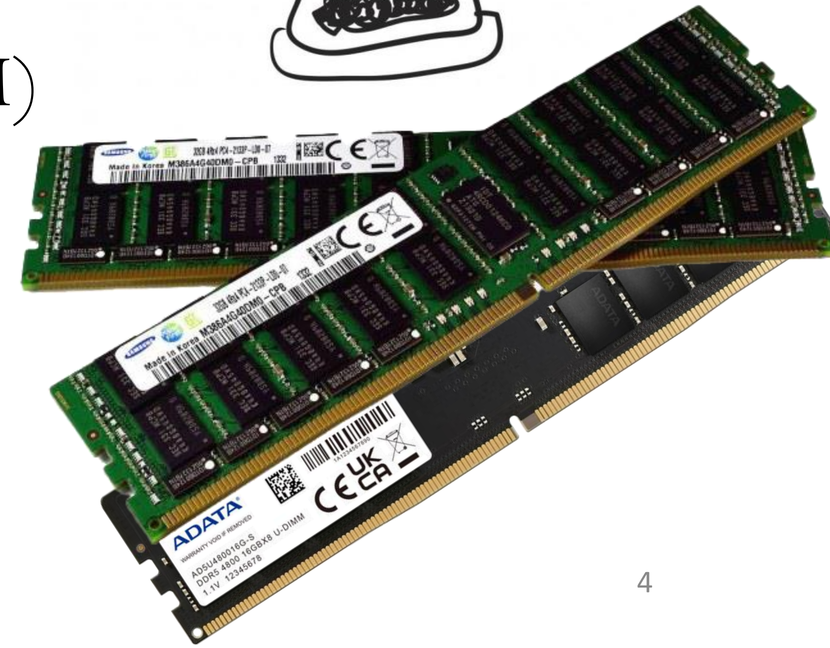


Algorithm
Important Points about Algorithms

- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways,
- There may exist several algorithms for solving the same problem.
  - ✓ Can be based on very different ideas and can solve the problem with dramatically different speeds

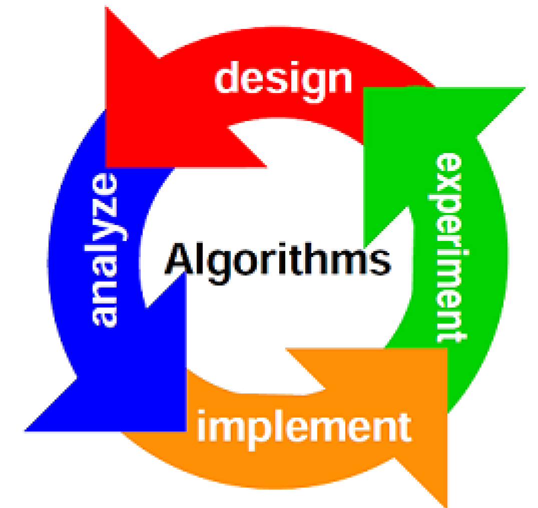
Algorithm
Measures of resource usage

- The two most common measures are:
- Time: how long does the algorithm take to complete?
- Space: how much working memory (typically RAM) is needed by the algorithm?



Algorithmic analysis

- It is frequently important to know how much of a particular resource (such as time or storage) is theoretically required for a given algorithm.
- Methods have been developed for the analysis of algorithms to obtain such quantitative answers (estimates)

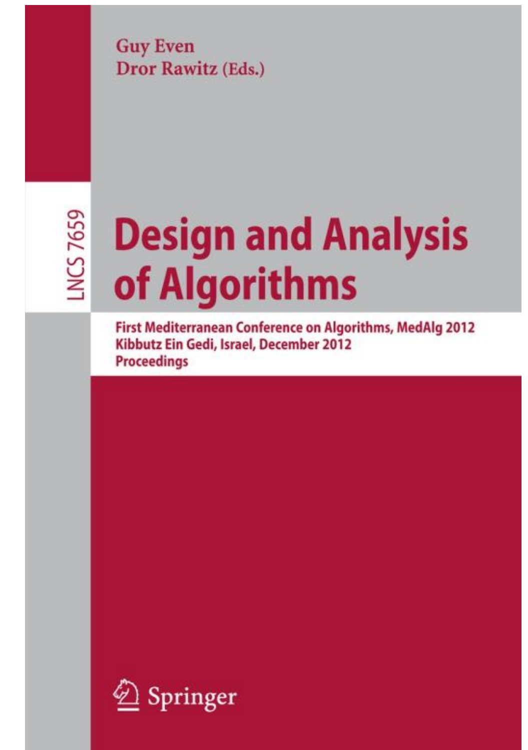


Algorithmic analysis

- Different algorithms may complete the same task with a different set of instructions in less or more time, space, or 'effort' than others.
- For example, a binary search algorithm (with cost O(log n)) outperforms a sequential search (cost O(n)) when used for table lookups on sorted lists or arrays.

Algorithmic analysis

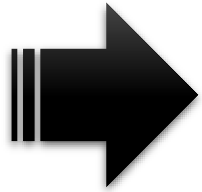
- In computer science, the analysis of algorithms is the process of finding the computational complexity of algorithms—the amount of time, storage, or other resources needed to execute them without the use of a specific programming language or implementation.



Run-time analysis

- Run-time analysis is a theoretical classification that estimates and anticipates the increase in running time (or run-time or execution time) of an algorithm as its input size (usually denoted as n) increases.
- Run-time efficiency is a topic of great interest in computer science: A program can take seconds, hours, or even years to finish executing, depending on which algorithm it implements.

Run-time analysis



- Since algorithms are platform-independent (i.e. a given algorithm can be implemented in an arbitrary programming language on an arbitrary computer running an arbitrary operating system), there are additional significant drawbacks to using an empirical approach to gauge the comparative performance of a given set of algorithms.

Run-time analysis

- Take as an example a program that looks up a specific entry in a sorted list of size n. Suppose this program were implemented on Computer A, a state-of-the-art machine, using a linear search algorithm, and on Computer B, a much slower machine, using a binary search algorithm. Benchmark testing on the two computers running their respective programs might look something like the following:

n (list size)	Computer A run-time (in nanoseconds)	Computer B run-time (in nanoseconds)
16	8	100,000
63	32	150,000
250	125	200,000
1,000	500	250,000

Run-time analysis

- Based on these metrics, it would be easy to jump to the conclusion that Computer A is running an algorithm that is far superior in efficiency to that of Computer B. However, if the size of the input-list is increased to a sufficient number, that conclusion is dramatically demonstrated to be in error:

n (list size)	Computer A run-time (in nanoseconds)	Computer B run-time (in nanoseconds)
16	8	100,000
63	32	150,000
250	125	200,000
1,000	500	250,000
...	...	...
1,000,000	500,000	500,000
4,000,000	2,000,000	550,000
16,000,000	8,000,000	600,000
...	...	...
$63,072 \times 10^{12}$	$31,536 \times 10^{12}$ ns, or 1 year	1,375,000 ns, or 1.375 milliseconds

Run-time analysis

- Computer A, running the linear search program, exhibits a linear growth rate. The program's run-time is directly proportional to its input size.
- Doubling the input size doubles the run-time, quadrupling the input size quadruples the run-time, and so forth.

n (list size)	Computer A run-time (in nanoseconds)	Computer B run-time (in nanoseconds)
16	8	100,000
63	32	150,000
250	125	200,000
1,000	500	250,000
...	...	...
1,000,000	500,000	500,000
4,000,000	2,000,000	550,000
16,000,000	8,000,000	600,000
...	...	...
$63{,}072 \times 10^{12}$	$31{,}536 \times 10^{12}$ ns, or 1 year	1,375,000 ns, or 1.375 milliseconds

Run-time analysis

- On the other hand, Computer B, running the binary search program, exhibits a logarithmic growth rate. Quadrupling the input size only increases the run-time by a constant amount (in this example, 50,000 ns). Even though Computer A is ostensibly a faster machine, Computer B will inevitably surpass Computer A in run-time because it's running an algorithm with a much slower growth rate.

n (list size)	Computer A run-time (in nanoseconds)	Computer B run-time (in nanoseconds)
16	8	100,000
63	32	150,000
250	125	200,000
1,000	500	250,000
...	...	...
1,000,000	500,000	500,000
4,000,000	2,000,000	550,000
16,000,000	8,000,000	600,000
...	...	...
$63{,}072 \times 10^{12}$	$31{,}536 \times 10^{12}$ ns, or 1 year	1,375,000 ns, or 1.375 milliseconds

Time complexity

- In computer science, the time complexity is the computational complexity that describes the amount of computer time it takes to run an algorithm.
- Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform.
- Thus, the amount of time taken and the number of elementary operations performed by the algorithm are taken to be related by a **constant factor**.

Time complexity

- Time complexity is generally expressed as a function of the size of the input.
- Since this function is generally difficult to compute exactly, and the running time for small inputs is usually not consequential, one commonly focuses on the behavior of the complexity when the input size increases—that is, the **asymptotic** behavior of the complexity.
- The most commonly used notation to describe resource consumption or "complexity" is Donald Knuth's Big O notation, representing the complexity of an algorithm as a function of the size of the input n.

Best, Worst and Average Case

- Complexity of algorithms is usually evaluated in the worst case (most unfavorable scenario). This means in the average case they can work faster, but in the worst case they work with the evaluated complexity and not slower.
- Let's take an example: searching in array. To find the searched key in the worst case, we have to check all the elements in the array. In the best case we will have luck and we will find the element at first position. In the average case we can expect to check half the elements in the array until we find the one we are looking for.

Orders of growth/Big O notation

- Big-O, also known as Landau's symbol, is a "symbolism used in complexity theory, computer science, and mathematics to describe the asymptotic behavior of functions. Basically, it tells you how fast a function grows or declines", according to MIT. "The letter O is used because the rate of growth of a function is also called its order."

Orders of growth/Big O notation

- Big O notation is one of the most fundamental tools for computer scientists to analyze the time and space complexity of an algorithm.
- With Big O Notation, you express the runtime in terms of how quickly it grows relative to the input, as the input gets arbitrarily large. Essentially, it's a way to draw insights into how scalable an algorithm is.

Orders of growth/Big O notation

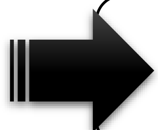
- Note that the big-O expressions do not have constants or low-order terms. This is because, when N gets large enough, constants and low-order terms don't matter (a constant-time method will be faster than a linear-time method, which will be faster than a quadratic-time method).

Orders of growth/Big O notation

- Formal definition:
- A function T(N) is O(F(N)) if for some constant c and for all values of N greater than some value n_0:

$$T(N) \leq c * F(N)$$

- The idea is that T(N) is the exact complexity of a method or algorithm as a function of the problem size N, and that F(N) is an upper-bound on that complexity (i.e., the actual time for a problem of size N will be no worse than F(N)).



In practice, we want the smallest F(N) -- the least upper bound on the actual complexity.

Orders of growth/Big O notation

- For example, consider $T(N) = 3 * N^2 + 5$.
- We can show that T(N) is O(N^2) by choosing $c = 4$ and $n_0 = 2$.

Orders of growth/Big O notation

- For example, consider $T(N) = 3 * N^2 + 5$.
- We can show that T(N) is O(N^2) by choosing $c = 4$ and $n_0 = 2$.

This is because for all values of N greater than 2
Let's try with N =3:

$$3 * 92 + 5 \leq 4 * 92$$

Orders of growth/Big O notation

- For example, consider $T(N) = 3 * N^2 + 5.$
- We can show that T(N) is O(N^2) by choosing $c = 4$ and $n_0 = 2.$

T(N) is not O(N), Why?

because whatever constant c and value n_0 you choose, we can always find a value of N greater than n_0 so that $3 * N^2 + 5$ is greater than c ∗ N.

Orders of growth/Big O notation

Exemples :

$T_1(n)=7=O(1)$
$T_2(n)=12n+5=O(n)$
$T_3(n)=4\ n^2\ +2n+6=O(n^2)$
$T_4(n)=2+(n-1)\times 5=O(n)$

Units for Measuring Running Time

- The running time of an algorithm is to be measured with a unit that is independent of the extraneous factors like the processor speed, quality of implementation, compiler and etc.
- Basic Operation: The operation contributing the most to the total running time of an algorithm.
  - ❖ Examples: Key comparison operation; arithmetic/logic operation (division being the most time-consuming, followed by multiplication), affectation, verification, input/output operations,
- We will count the number of times the algorithm's basic operation is executed on inputs of size n.

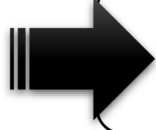
Measuring Running Time

1. Sequence of statements

```
statement 1;
statement 2;
   ...
statement k;
```

- The total time is found by adding the times for all statements:

total time = time(statement 1) + time(statement 2) + ... + time(statement k)



If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: $O(1)$.

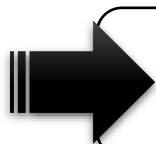
Measuring Running Time

2. if-then-else statements

```
if (condition) {
    sequence of statements 1
}
else {
    sequence of statements 2
}
```

- Here, either sequence 1 will execute, or sequence 2 will execute. Therefore, the worst-case time is the slowest of the two possibilities:

$$\max(\text{time}(\text{sequence } 1), \text{time}(\text{sequence } 2)).$$



For example, if sequence 1 is $O(N)$ and sequence 2 is $O(1)$ the worst-case time for the whole if-then-else statement would be $O(N)$.

Measuring Running Time

3. for loops

```
for (i = 0; i < N; i++) {
    sequence of statements
}
```

- The loop executes N times, so the sequence of statements also executes N times.
- Since we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

Measuring Running Time

4. **Nested loops**

   First we'll consider loops where the number of iterations of the inner loop is independent of the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        sequence of statements
    }
}
```

- The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times.
- As a result, the statements in the inner loop execute a total of $N * M$ times.
- Thus, the complexity is $O(N * M)$.

Measuring Running Time

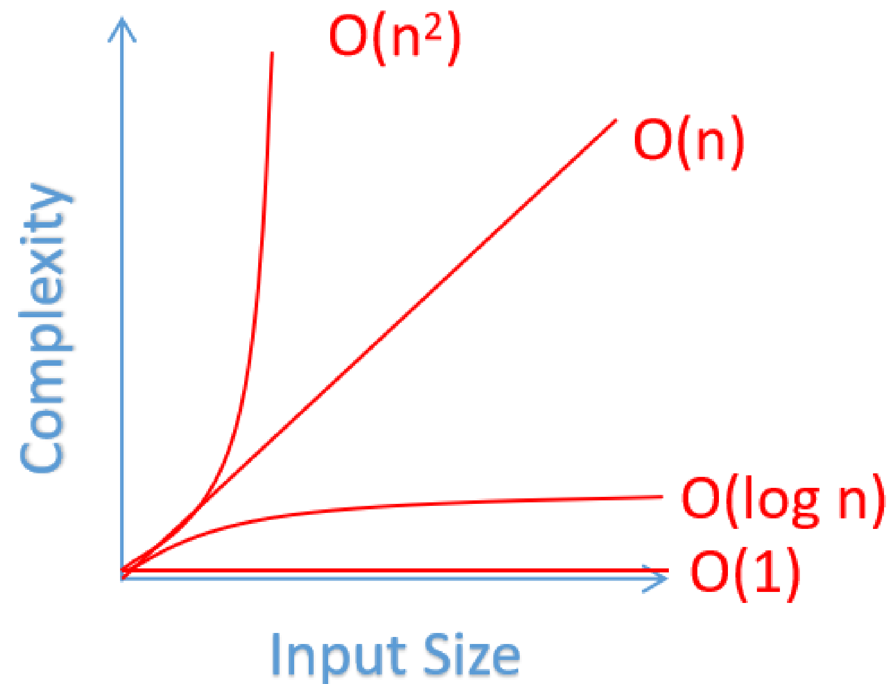
5. Nested loops

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        sequence of statements
    }
}
```

- In a common special case where the stopping condition of the inner loop is $j < N$ instead of $j < M$ (i.e., the inner loop also executes N times), the total complexity for the two loops is $O(N^2)$.

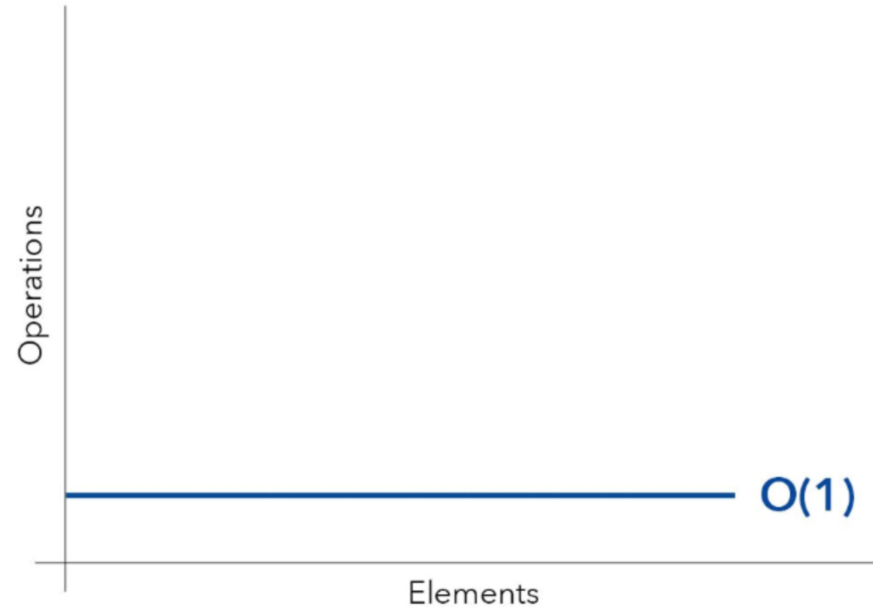
Time complexity Notations

- These following varieties of Big-O Notation aren't the only ones, but they're the ones you're most likely to encounter.



Constant time – O (I)

- This translates to a constant runtime, meaning, regardless of the size of the input, the algorithm will have the same runtime.



Constant time – O (I)

- An algorithm is said to have constant time with order O (I) when it is not dependent on the input size n.
- Irrespective of the input size n, the runtime will always be the same.
- Constant time algorithms will always take same amount of time to be executed. The execution time of these algorithm is independent of the size of the input. A good example of O(I) time is accessing a value with an array index.

```
var arr = [ 1,2,3,4,5];
```

```
arr[2]; // => 3
```

- Other examples include: push() and pop() operations on an array.

Constant time – O (I)

- Examples of constant algorithms:

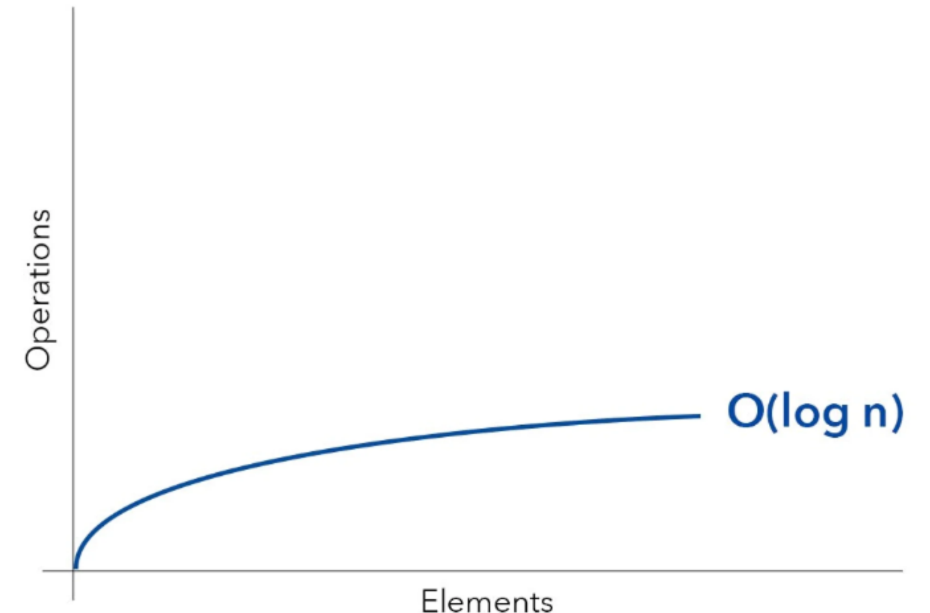
- Check if a number is even or odd
- Print the first element from list
- Remove an item from an object

```
function addUpto(n) {
  return n * (n + 1) / 2
}
```

- Other examples include: push() and pop() operations on an array.

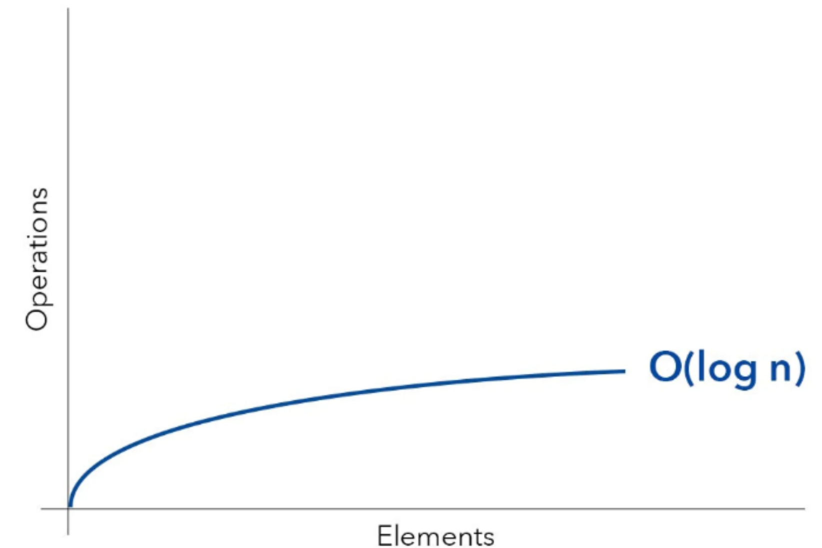
Logarithmic time – O (log n)

- An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step.
- This indicates that the number of operations is not the same as the input size.
- The number of operations gets reduced as the input size increases.



Logarithmic time – O (log n)

- O(log n) means that time goes up linearly, while the n goes up exponentially. So if it takes 1 second to compute 10 elements, it will take 2 seconds to compute 100 elements and so on.
- Algorithms are found in binary trees or binary search functions.
- This involves the search of a given value in an array by splitting the array into two and starting searching in one split.
- This ensures the operation is not done on every element of the data.



Logarithmic time – O (log n)

```
Example 1;
```

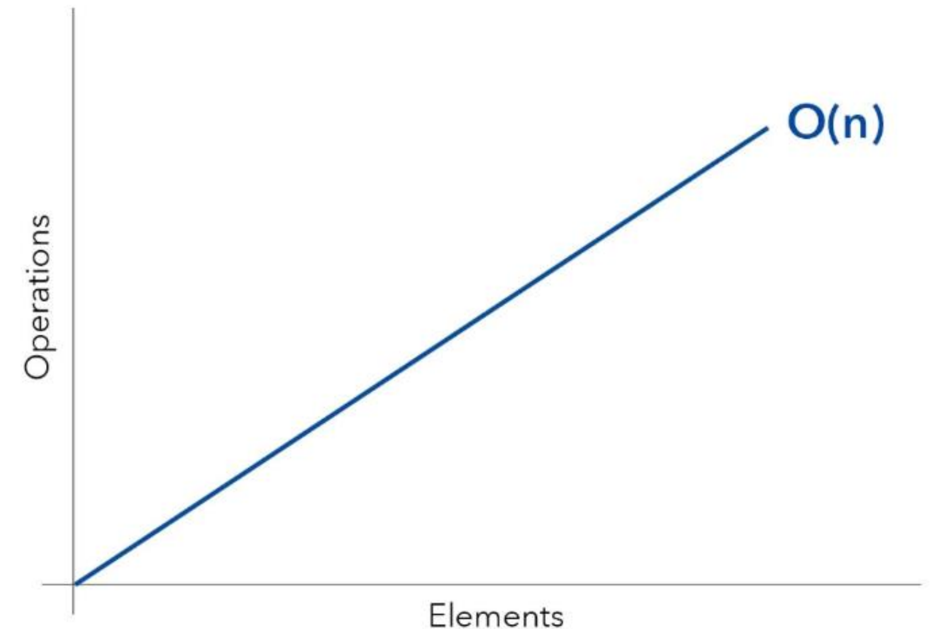
```
for (var i = 1; i < n; i = i * 2)  console.log(i);}
```

```
Example 2;
```

```
for (i = n; i >= 1; i = i/2) console.log(i);}
```

Linear time complexity– O(n)

- An algorithm is said to have a linear time complexity when the running time increases linearly with the length of the input.
- An algorithm has a linear time complexity if the time to execute the algorithm is directly proportional to the input size n. Therefore the time it will take to run the algorithm will increase proportionately as the size of input n increases.
- As the number of inputs/data increases, the number of operations increases linearly.



Linear time complexity– O(n)

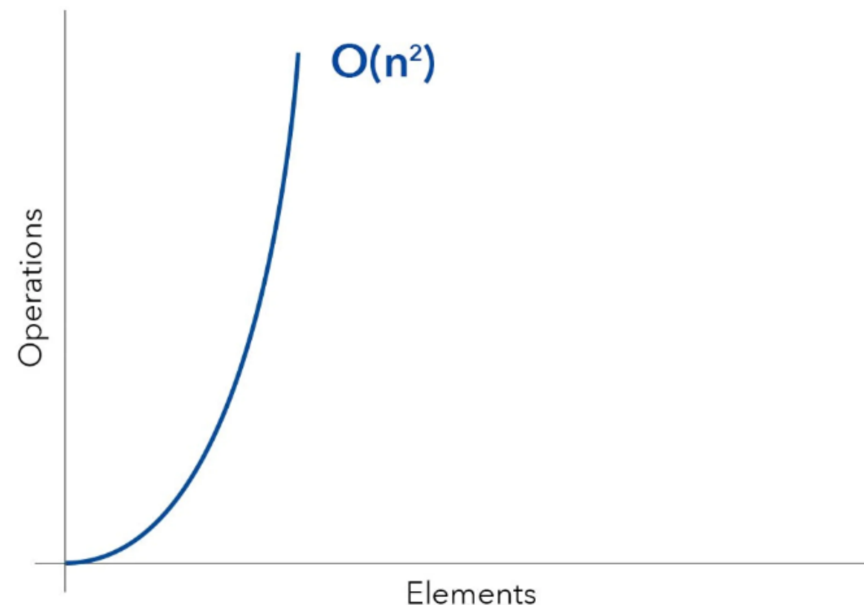
- Examples of linear algorithms:

- Find a given element in a collection
- Get the max/min value
- Return all the values in a list
- Sum up to the number given

```
function addUpto(n) {
  let total = 0;
  for (let i = 1; i <= n; i++) {
    total += 1;
  }
  return total
}
```

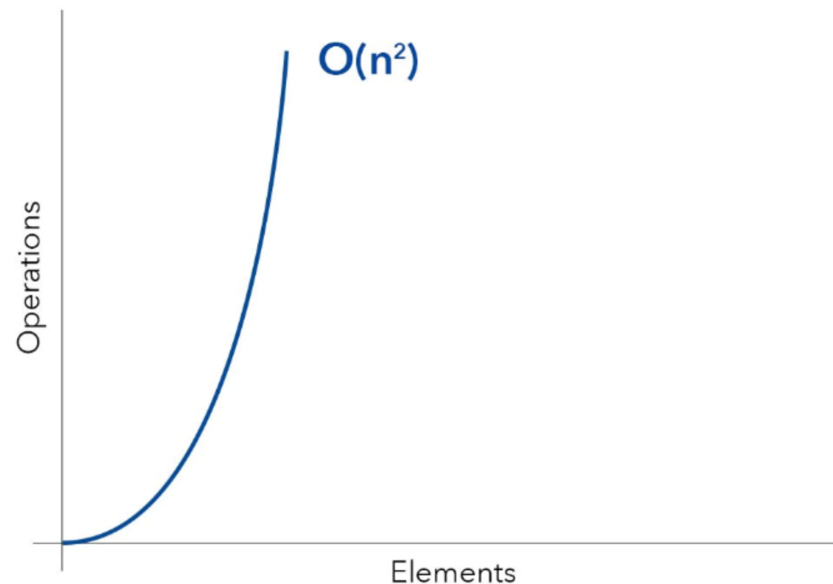
Quadratic time – $O(n^2)$

- An algorithm is said to have a non-linear time complexity where the running time increases non-linearly (n^2) with the length of the input.
- This means that the number of operations it performs scales in proportion to the square of the input.



Quadratic time – $O(n^2)$

- Generally, nested loops come under this order where one loop takes O(n) and if the function involves a loop within a loop, then it goes for $O(n)\times O(n) = O(n^2)$ order.
- Similarly, if there are 'm' loops defined in the function, then the order is given by $O(n^m)$, which are called polynomial time complexity functions.



Quadratic time – O(n²)

- An excellent example of this is checking to see whether there are any duplicates in a list of items.
- This is common with algorithms that involve nested loops or iterations.
- Examples of quadratic algorithms:

- Sorting algorithms (Bubble, Selection, Insertion)
- Check for duplicated values
- Print all ordered pairs in an array

```
function printAllPairs(n) {
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < n; j++){
      console.log(i, j)
    }
  }
}
```

How to calculate time complexity?

- I. Break the code down to different parts
  - Group A:
    - Assignments, statements, accessing a certain element in an array, a comparison
    - Examples

```
array[i]
const b = 5
b > 4
```

How to calculate time complexity?

- I. Break the code down to different parts
  - Group B:
    - Loop or recursion that runs n number of times
    - Examples

```
for (const i = 0; i < n; i++) {
  // do something
}
```

How to calculate time complexity?

- I. Break the code down to different parts
  - Group C:
  - Combining loops that run n times
  - Examples

```
for (const i = 0; i < n; i++) {
  for (const j = 0; j < m; j++) {
    // do something
  }
}
```

How to calculate time complexity?

- I. Break the code down to different parts
  - Group D:
  - Other logic that dictates how many times we iterate over elements
  - Examples

```
while ( low <= high ) {
  mid = ( low + high ) / 2;
  if ( target < list[mid] )
    high = mid - 1;
  else if ( target > list[mid] )
    low = mid + 1;
  else break;
}
```

How to calculate time complexity?

- **2. Decide how 'long' it will take to execute each piece**
  - In Group A are the most discrete pieces of code. The amount of these specific actions are what we are counting. Each comparison, assignment, or array at index access takes about the same amount of time which we consider 1 unit of time.
  - Group B is a simple if statement. If statements will iterate over the whole array or list of length n. For each element we do whatever the actions are inside the loop.

How to calculate time complexity?

- 2. Decide how 'long' it will take to execute each piece
  - **Group C** is combining loops or recursions. Because the loops are nested, we do n actions for the outside loop and for every time it runs we do n actions in the inside loop. $n * n$, or n^2.
  - **Group D** is logic that will affect the time in relation to the logarithm of the input size. In this example we have a while loop that cuts the input size in half each time it runs.

How to calculate time complexity?

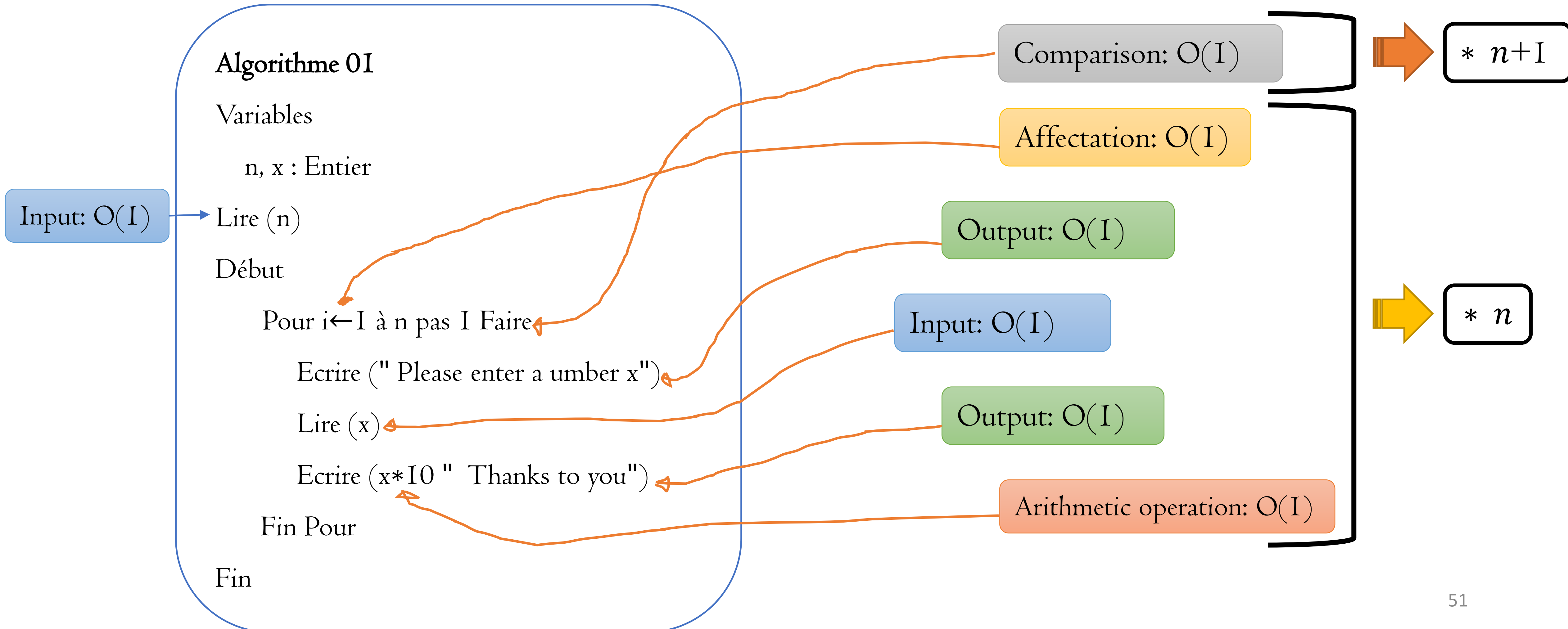
- **3. Add them all up**
- After you've figured out the time for each element we simply need to add them all up and attribute an O() to them!
  - Group A are all worth 1 unit of time, so we just use O(1).
  - Group B take n units of times, so, you guessed it, they are O(n).
  - Group C take n times the amount of time as Group B, which can be expressed as O(n^2). The more iterations you add, the larger the exponent.
  - Finally, Group D. In our example we see that the while loop actually halves the size of the list we're iterating over each time it runs. This run time will be much less than running over the whole list. Dividing the potential time is expressed with O($logn$).

How to calculate time complexity?

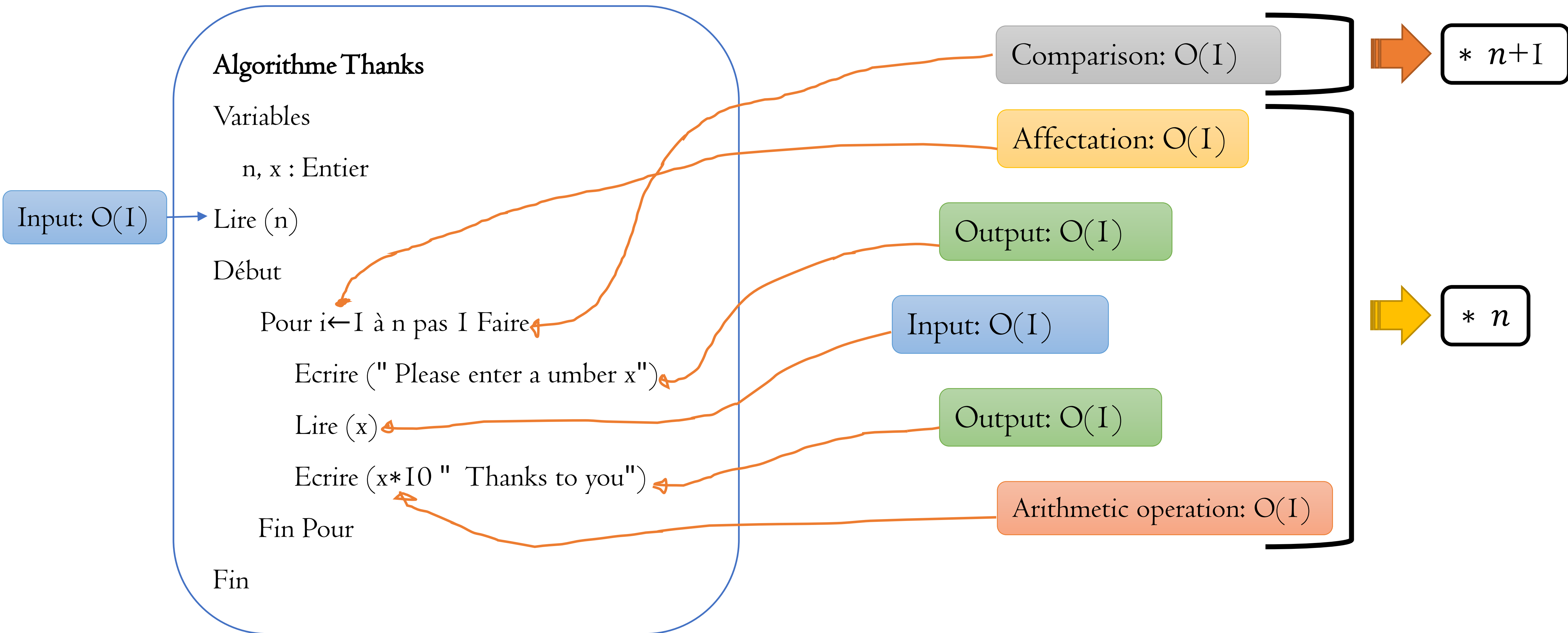
- **4. Don't sweat the small stuff**
  - One of the nice things about Big O is that it isn't concerned with the details.
  - Big O is a comparison of time as n moves towards infinity, where certain elements become trivial.
  - Don't worry about O(1) parts, and don't worry about constants.
  - If there are higher order n values then you can get rid of the lesser n's as well. $5\,n^2 + n + 10$ becomes O(n^2).
  - As n gets larger, the exponential part has the greatest impact on the order of magnitude of the whole expression.

How to calculate time complexity?

- Hence, to determine the complexity of an algorithm, we need to count the number of its basic operations,
- The complexity of each basic operation is constant or O(1),



How to calculate time complexity?



- T(n) = O(1)+ O(1)*(n+1)+ (O(1)+ O(1) +O(1) + O(1)+O(1)) *n
  = 2 O(1) + n O(1) + 5n O(1)
  = (6n +2) O(1)
  = (6n +2) O(1)
  = 6n O(1)
  = O(6n)
  = O(n)

Test yourself

- Determine the complexity in the following codes:

```
a= b + 1;
```

```
while(n>1){
  n=n/2;
}
```

```
for(c=0; c<n; c++){
  a+=1;
}
```

```
for(c=0; c<n; c++){
  for(i=0; i<n; i++){
    a+=1;
  }
}
```

```
for(c=0; c<n; c++){
  for(i=0; i<n; i++){
    for(x=0; x<n; x++){
      a+=1;
    }
  }
}
```

References:

- https://en.wikipedia.org
- https://www.educative.io/blog/a-big-o-primer-for-beginning-devs
- https://medium.com/nerd-for-tech/the-big-o-18fea712ae6b
- https://introprogramming.info/english-intro-csharp-book/read-online/chapter-19-data-structures-and-algorithm-complexity/
- https://www.jsums.edu/nmeghanathan/files/2015/04/CSC323-Sp2015-Module-1-Algorithm-Efficiency.pdf?x61976