Computational complexity

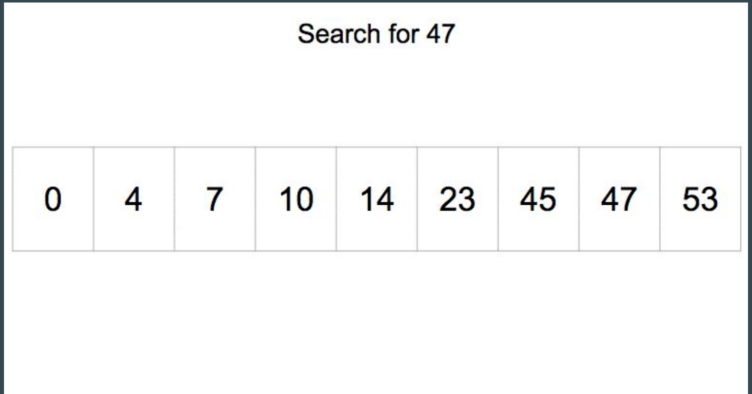
Evaluating algorithms

An intuitive approach to complexity

Search algorithm in a sorted array :

Target value T

Sorted array $A = \{A_1, A_2, A_3 \ldots A_n\}$



Goal: find an algorithm to search if T is present in A

An intuitive approach to complexity

Two algorithms:

Linear search algorithm

```
trouvé = False
i = 0
while not trouvé and i < len(A):
   if(A[i] == T):
       trouvé = True
   i += 1

if trouvé:
   print("trouvé")
else:
   print("pas trouvé")
```

Binary search algorithm

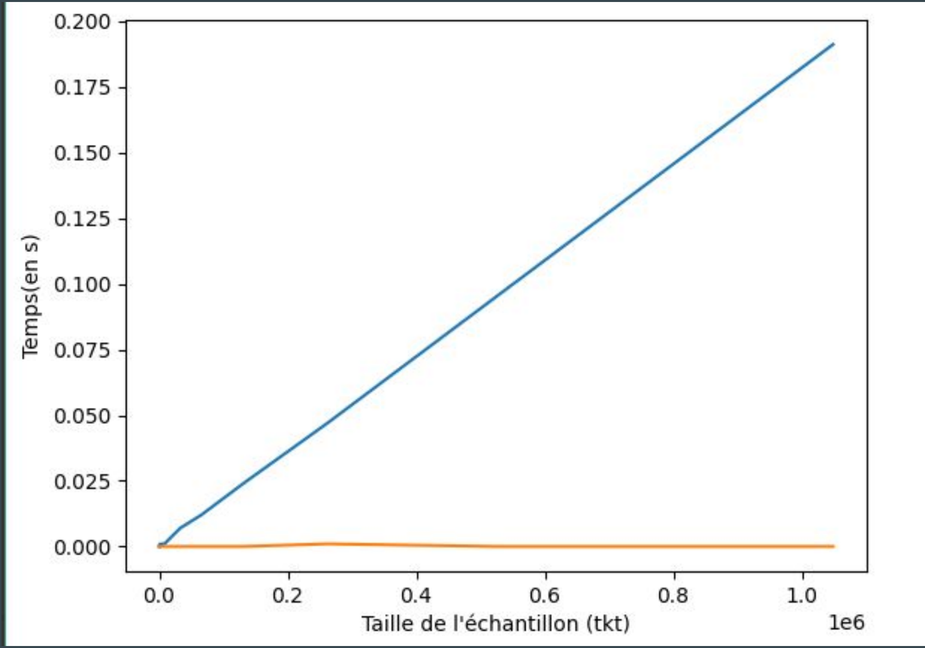
```
start = 0
end = len(A) - 1
resultat = "pas trouvé"
while(start <= end):
    mid = (start + end) // 2
    if(A[mid] > T):
        end = mid - 1
    elif(A[mid] < T):
        start = mid + 1
    else:
        resultat = "trouvé"

print(resultat)
```

How efficient are they?

An intuitive approach to complexity

Let's compare them with the module *time*:



In blue, the Linear search algorithm

In orange, the Binary search algorithm

What happened?

An intuitive approach to complexity

Linear search algorithm

```
trouvé = False
i = 0
while not trouvé and i < len(A):
   if(A[i] == T):
       trouvé = True
   i += 1

if trouvé:
   print("trouvé")
else:
   print("pas trouvé")
```

Loop of size n at worse, with fixed operations inside → complexity is called *linear* in n

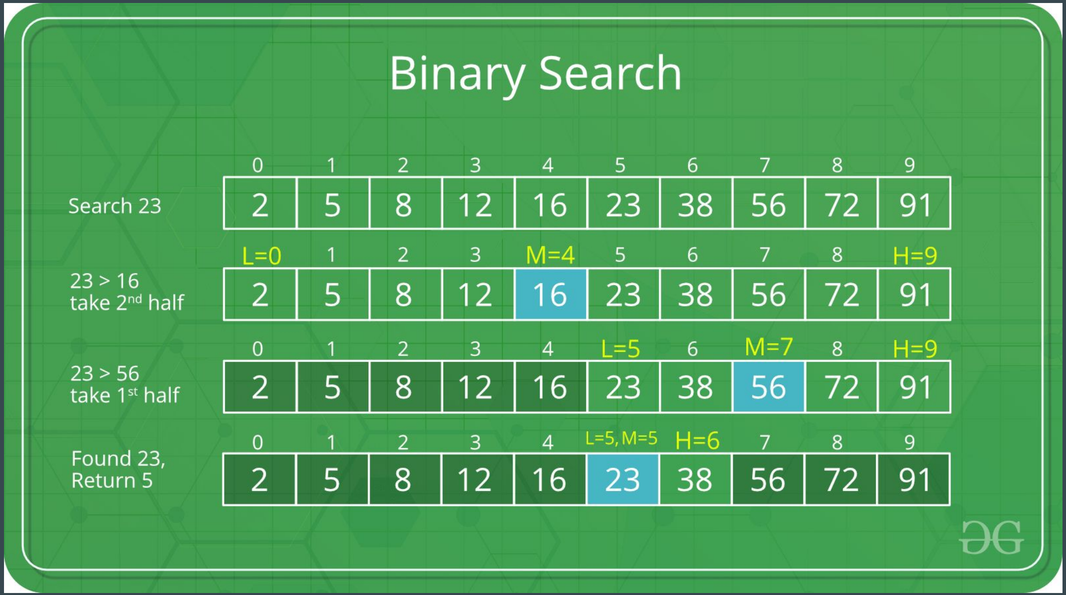
Binary search algorithm

```
start = 0
end = len(A) - 1
resultat = "pas trouvé"
while(start <= end):
    mid = (start + end) // 2
    if(A[mid] > T):
        end = mid - 1
    elif(A[mid] < T):
        start = mid + 1
    else:
        resultat = "trouvé"

print(resultat)
```

Let’s take a closer look at what happens with this algorithm…

An intuitive approach to complexity



At each step the search interval is divided in half.
→ complexity is called *logarithmic* in $\log_2(n)$.

A more formal definition

- What we have evaluated previously is the *time complexity*, i.e the amount of time that it takes to run an algorithm, in function of its parameters
- When comparing memory consumption, we talk about *space complexity*

A more formal definition

When evaluating complexity, we use the *big O notation*:

$f(x) = O(g(x))$
if and only if there is x_0 and M such that:
$|f(x)| \leq M\, g(x)$ for all $x > x_0$

For example:

- $n^2 - 3n + 5 = O(n^2)$
- $42 = O(1)$
- $-7\, n \log_2 n = O(n \log n)$

This notation allows to evaluate how the program behaves depending on the size of the parameters.

Examples of time complexity

Let n be the size of an array.

Here are the time complexities of a few operations:

- $O(\log n)$ find an element in a sorted array with binary search
- $O(n)$ explore all elements of the array
- $O(n \log n)$ sort the array with a merge sort
- $O(n^2)$ find all the couples of elements of the array
- $O(n!)$ find all the permutations of the array

etc

Average VS worst case complexity

Sometimes the time needed to run an algorithms varies for inputs of the same size.

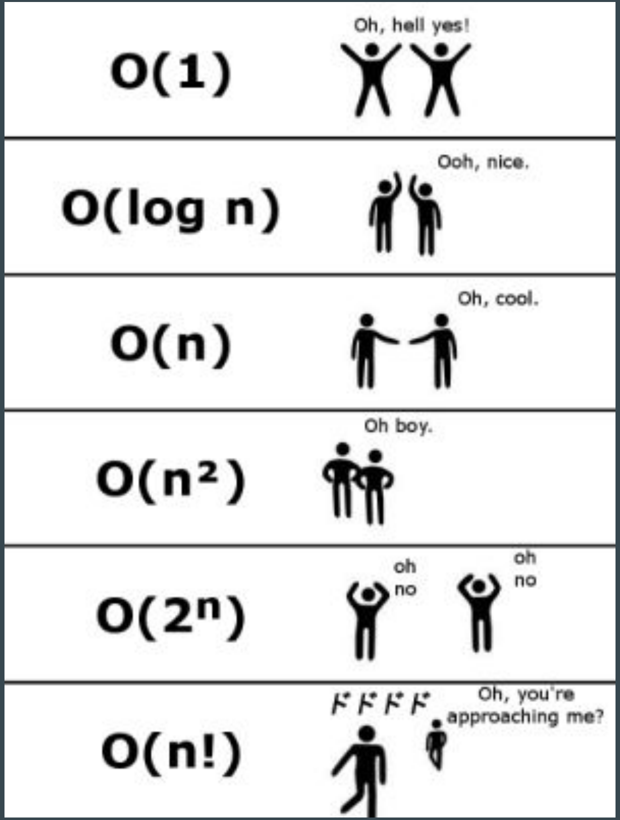
For example, the complexity of the quick sort is:

- $O(n \log n)$ on average
- $O(n^2)$ in the worst case

When talking about complexity, we usually refer to the *worst case* complexity.

Common complexities

- 1 constant
- log n logarithmic
- n linear
- n log n linearithmic
- n^2 quadratic
- n^3 cubic
- n^k polynomial
- k^n exponential
- n! factorial



Equivalence time - complexity

total time = number of operations / operations per second

In python3 ...

total time = number of operations / 10^7 - 10^8

Why is it useful?

Problem's constraints : $0 \leq N \leq 10^2$, $0 \leq C \leq 10^3$, Time limit = 1s

$N*C$	=>	10^5 operations	=>	0.001s
N^2*C	=>	10^7 operations	=>	0.1s
$N*C^2$	=>	10^8 operations	=>	1s
$N!*C$	=>	∞? operations	=>	NO

Some traps easily avoidable

```
array = ACollectionImplementation()
targets = [1,2,5,6]
for element in targets :
   if(array.found(element)):      ← we don’t know how this function is implemented
     print(element,"was found")
   else :
     print(element,"was not found")
```

The Complexity of this algorithm is unknown

Common complexities

1	constant
$\log n$	logarithmic
n	linear
$n \log n$	linearithmic
n^2	quadratic
n^3	cubic
n^k	polynomial
k^n	exponential
$n!$	factorial

Now, some exercises :)