In [64]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu(first_level=1, last_level=4, header="<font color='blus'>Algorithmic co
```

Out[64]:

run previous cell, wait for 2 seconds

Imports

In [65]:
```
%load_ext autoreload
%autoreload 2

%matplotlib inline
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [66]:
```
# modules de bases
import os
import sys
import copy

# numpy, scipy, pandas
import numpy as np
import scipy.special as sp
import pandas as pd
import random

from timeit import Timer

# pour la visualisation
from IPython.display import display
from matplotlib import pyplot as plt # import matplotlib.pyplot as plt
# plotly
# seaborn
# altair

import random as rd
```

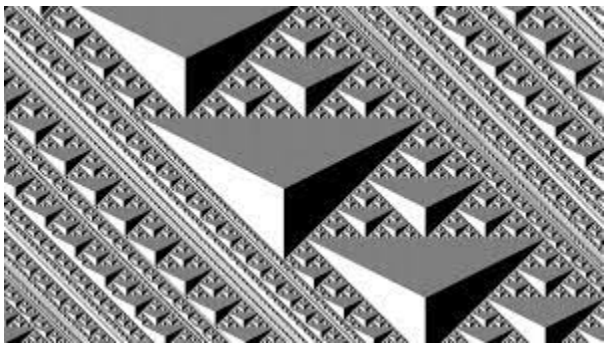
Complexity theory

> 

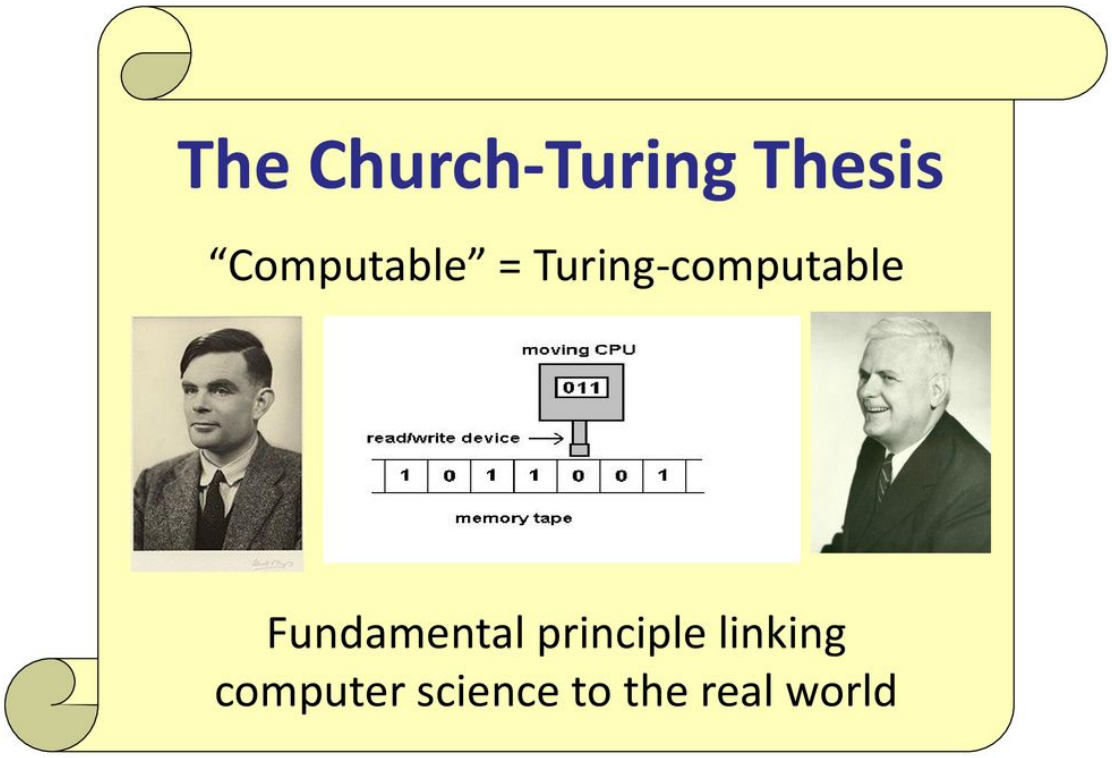
> A system composed of a large number of interacting components, without central control,

whose emergent "global" behavior—described in terms of dynamics, information processing, and/or adaptation —is more complex than can be explained or predicted from understanding the sum of the behavior of the individual components.

Complex systems are generally capable of adapting to changing inputs/environment and in such cases sometimes referred to as complex adaptive systems.



Theory of cellular automaton



The Church-Turing thesis :

Any function that is "computable" -- that is, that can be computed by an algorithm -- can be computed by a universal Turing machine

It is a branch of theoretical computer science.

> This branch deals with the necessary resources, memory and time, to compute an algorithmic problem.
>
> Problems are classified into complexity classes: P, NP, ...
>
> Measuring complexity allows to adapt resources to computers tasks and optimize algorithms.
>
> In this course, we will focus on **time** resource. We use a kind of algorithm, particularly adapted to measure this quantity, both the **searching** and **sorting** algorithms.

Algorithms over `Python` lists

Note: to teach algorithms and illustrate fundamental notions like algorithmic complexities, it is probably better to break python's best practices and avoid its idioms, in short, to use python as a low level programming language.

Search for an item in a list

See the doc for the `%time` magic command.

> > `%%time` is a so-called "magic" command of `IPython`.
> >
> > For the whole cell, `%%time` displays the CPU time and the wait time for the result.
> >
> > `%time` displays the CPU time and the wait time for the result of a single statement line.

> The list comprehension below generates a list of 1 million pseudo-random values, ranging from 1 to 10001

In [67]:
```
%%time
import random as rd
n = int(1e6)
p = int(1e4)
liste_test = [rd.randint(0,p)+1 for _ in range(n)]
print(len(liste_test))
```

```
1000000
CPU times: user 771 ms, sys: 15.3 ms, total: 787 ms
Wall time: 788 ms
```

> **User CPU time** is the amount of time the processor spends in running application / Python code.
>
> **System CPU Time** is the amount of time the processor spends in running the operating system(i.e., kernel), system libraries.

In [68]:
```
print(liste_test[:10])
```

```
min(liste_test), max(liste_test), len(liste_test)
```

```
[2939, 9115, 4771, 2969, 7569, 9967, 6155, 614, 5661, 3130]
```

Out[68]: (1, 10001, 1000000)

In [69]:
```
liste_test.index(7010), liste_test.index(5926), liste_test[-1]
```

Out[69]: (11803, 4813, 4117)

In [ ]:

from collections import Counter sorted(Counter(liste_test).items(), reverse=True, key=lambda val_occurence: val_occurence[1])

Searching for an item in a list

> For each value of the list, we retrieve the corresponding index in the list and we thus build the list of indexes.
>
> In the example below, the index associated with the value `'e'` of the list `list_` is `4`.

In [70]:
```
list_ = ['a', 'b', 'c', 'd', 'e']
print(list_)
list_.index('e')
list_index = [list_.index(l) for l in list_]#
max_index = max(list_index)

element = list_index.index(max_index)#
print(element, max_index)
```

```
['a', 'b', 'c', 'd', 'e']
4 4
```

In [71]:
```
list_.index(list_[-1])
```

Out[71]: 4

In [72]:
```
p
```

Out[72]: 10000

In [73]:
```
list_index = [liste_test.index(l) for l in range(1,p+1)]
max_index = max(list_index)# Get greater index in the list
element = list_index.index(max_index)# Then we retrieve the value in the list for this i
element
```

Out[73]: 7434

In [74]:
```
print(list_index[:10])
max_index
```

```
[5699, 7873, 11999, 2262, 2759, 4560, 24791, 8392, 17036, 13797]
```

Out[74]: 97328

Naive search

> We go through all the elements of the list and we compare them one by one with the element we are looking for.
>
> A test (`if` control structure) is made on each of the elements

In [75]:
```
def naive_search(liste,elt):
    for i in range(len(liste)):
        if liste[i] == elt:
            return i
    return -1
```

> The `%time` statement applies to the cell line.

In [76]:
```
%time naive_search(liste_test,element)
```

```
CPU times: user 10 µs, sys: 0 ns, total: 10 µs
Wall time: 12.6 µs
```

Out[76]:
```
185
```

> In programming, a test (`if` statement)consumes several processor cycles. This is a costly statement that we avoid inserting in a loop (`for`).

Search with the `enumerate` function

In [77]:
```
def enumerate_search(liste,elt):
    for i,item in enumerate(liste):
        if item == elt:
            return i
    return -1
```

In [78]:
```
%time enumerate_search(liste_test,element)
```

```
CPU times: user 9 µs, sys: 0 ns, total: 9 µs
Wall time: 11.7 µs
```

Out[78]:
```
185
```

> Algorithm implemented into `enumerate_search` function is more optimized than our naive search. This is due to the fact that in naive search, a memory access is performed in the list (`list[i]`) and this access is more expensive than going through the native `enumerate` function to retrieve the elements of the list.
>
> Indeed the memory address `list[i]` must be recalculated at each iteration, which consumes several processor cycles.
>
> **NB : When you need an algorithm, make sure that it has not still alerady been implemented in some Python library.** Native algorithms are often higly optimized.

Searching with the `index` method of the list

In [79]:
```
def recherche_index(liste,elt):
    return liste.index(elt)
```

In [80]:
```
%time recherche_index(liste_test,element)
```

```
CPU times: user 5 µs, sys: 0 ns, total: 5 µs
Wall time: 7.87 µs
```

Out[80]:
```
185
```

> - No loop is used in the search program.
> - The element's memory address is calculated and accessed only once.
> - No test is done, which saves processor cycle time and highlights, by the way, the cost of an `if` test.
>
> Note that the methods of the classes are optimized in the `C` language. Most algorithms in `Python` are

underlyingly written in the `C` language. The latter is a so-called **low level** language in the sense that it is necessary to understand the organization of the memory of a processor in order to code efficiently.

> The level of `C` language is just above assembly language.

Searching in a list using recursion algorithm

> A recursive function is related to recurrent sequences, such as:
>
> > $$U_{n+1} = 2U_n + 3$$
>
> The recursive function calls itself.

In [81]:
```
def recursive_search(liste,elt,index=0):
    if liste[0] == elt:
        return index;
    else:
        # Proceed to recursion over the list where 1st element has been removed.
        return recursive_search(liste[1:],elt,index+1)
```

In [82]:
```
if False :
    %time recursive_search(liste_test,element)
```

> The time and resources required to solve an algorithmic problem vary according to the algorithm used.
>
> Enabling recursive search will crash the `Python` kernel, in the sense that the call stack that uses RAM memory will be full before it can exit the recursive process.

> The function below gives the limit depth of recursions.
>
> Using `set` assessor allows to upgrade this limit.

> Then, this limit can be changed, but it is not recommended:
> `sys.setrecursionlimit(124000)`
>
> Using `sys` API (Application Programming Interface), we reach the `system programming` level.

In [83]:
```
import sys
print(sys.getrecursionlimit())
```

```
3000
```

> To avoid the problem of memory saturation, we will truncate the original list and search for the element at the last index, means, the last position in the list

In [84]:
```
len(liste_test)
liste_test_troncated = liste_test[:10000]
print(len(liste_test_troncated))
element = liste_test_troncated[-1]
print(element)
```

```
10000
1208
```

In [85]:
```
%%time
id = recursive_search(liste_test_troncated,element)
print(liste_test_troncated[id])
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<timed exec> in <module>

/tmp/ipykernel_2712659/2781815625.py in recursive_search(liste, elt, index)
      4     else:
      5         # Proceed to recursion over the list where 1st element has been removed.
----> 6         return recursive_search(liste[1:],elt,index+1)

... last 1 frames repeated, from the frame below ...

/tmp/ipykernel_2712659/2781815625.py in recursive_search(liste, elt, index)
      4     else:
      5         # Proceed to recursion over the list where 1st element has been removed.
----> 6         return recursive_search(liste[1:],elt,index+1)

RecursionError: maximum recursion depth exceeded in comparison
```

> **For a list 100 times smaller, the algorithm time is multiplied by a factor of 100 000.**

In [86]:
```
# # on peut changer cette limite, mais cela n'est pas recommandé
# sys.setrecursionlimit(124000)
```

Using data structures under `numpy`

Avec numpy:

`numpy` is an abbreviation of numerical python. It is a library dedicated to scientific programming.

`numpy` allows you to manipulate, among other things, arrays (vectors in the mathematical sense of the term), matrices (bilinear forms), tensors (multilinear forms), statistics, probabilities.

`numpy` covers areas the `scipy` lib covers.

With `pandas`, `numpy` is one of the most widely used libraries in datascience.

In the following examples, lists are converted to tables.

In [87]:
```
import numpy as np # notez que, par convention, l'alias de numpy est np.
```

In [88]:
```
%time test_nparray = np.asarray(liste_test)
```

```
CPU times: user 57.3 ms, sys: 29 µs, total: 57.3 ms
Wall time: 57.2 ms
```

Generate an array under `numpy`

An array of type int, with one dimension, n, is generated with random numbers between 1 and 1000.

In [89]:
```
%%time
n = int(1e6)
test_nparray2 = np.fromfunction(np.vectorize(lambda i: rd.randint(0, 1000)+1), (n,), dty
```

```
CPU times: user 709 ms, sys: 20 ms, total: 729 ms
Wall time: 729 ms
```

Find an element with the `where` method of a `numpy` array

In [90]:
```
%time np.where(test_nparray == element)
```

```
CPU times: user 1.57 ms, sys: 61 µs, total: 1.63 ms
Wall time: 1.35 ms
```

Out[90]:
```
(array([  8542,   9999,  11107,  28223,  46589,  72474,  76533,  82967,
         94687, 104834, 109189, 125888, 181959, 207396, 218815, 227523,
        240377, 240670, 250936, 254831, 259714, 263238, 268107, 279627,
        286179, 291762, 306939, 325966, 331247, 337571, 340144, 347600,
        353602, 380144, 383064, 404580, 405073, 407287, 416202, 420240,
        429655, 430132, 440467, 450420, 453308, 476644, 485352, 487483,
        497288, 504512, 513980, 519779, 523873, 524196, 536819, 543390,
        548474, 549579, 563427, 565692, 577779, 578332, 633546, 638940,
        652490, 664010, 674155, 675193, 676154, 676342, 691848, 707783,
        708388, 751480, 775586, 776156, 781804, 786876, 798609, 800091,
        818692, 820651, 826652, 828109, 842514, 843385, 846691, 850620,
        862519, 869757, 881974, 883470, 898129, 910125, 918253, 924413,
        939871, 941640, 962808, 968068, 973809, 977820, 988846, 992581]),)
```

We have retrieved the list of all the indices from the array `test_nparray`.

These indices corresponds with the value of `element`.

Note the similarity of the `where` keyword with the languages of the `SQL` family.

In [91]:
```
print(test_nparray[966692], element)
print(test_nparray[877119], element)
```

```
6405 1208
53 1208
```

The `shape` property below gives the dimensions of the array, here to an input.

In [92]:
```
test_nparray.shape[0]
```

Out[92]:
```
1000000
```

In [93]:
```
np.where(test_nparray == element)[0].shape
```

Out[93]:
```
(104,)
```

In [94]:
```
len(np.where(test_nparray == element)[0])
```

Out[94]:
```
104
```

`np.where` returns a tuple whose first index gives the list of array indices corresponding to the sought element.

Find an element with an iterator from a `numpy` array

In the next cell, we define an iterator on the array.

An iterator is an object acting as a cursor, which will point to the different indices of a sequence of values or objects. The pointer traverses the addresses of the elements step by step.

An iterator is **inexpensive in memory** and is more general purpose than a list.

The `nd` stem is a contraction of "N-Dimensionnal" and emphasizes numpy's ability to handle multi-dimensional objects, such as matrices.

In [95]:
```
print(element)
element in test_nparray
```

```
1208
```

Out[95]:
```
True
```

In [96]:
```
iterator = np.nditer(test_nparray,flags=['f_index'])
iterator.iternext()
```

Out[96]: True

In [97]:
```
%%time
list_index = list()

iterator = np.nditer(test_nparray,flags=['f_index'])
while iterator.iternext():
    if iterator[0] == element:
        list_index.append(iterator.index)
```

```
CPU times: user 762 ms, sys: 0 ns, total: 762 ms
Wall time: 762 ms
```

In [98]:
```
len(list_index), len(np.where(test_nparray ==element)[0])
```

Out[98]: (104, 104)

In [99]:
```
iterator.iternext()
```

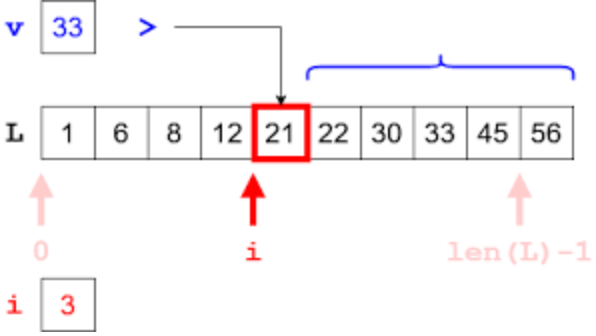
Out[99]: False

Dichotomous search in a `numpy` array

Here-under is a **dynamic programming** representation of the bubble sort algorithm

(wikipedia)

> Dynamic programming is both a mathematical optimization method and a computer programming method. The method was developed by **Richard Bellman** (applied mathematician) in the 1950s and has found applications in numerous fields, from aerospace engineering to economics.



> Also named binary search.
>
> Principle of research:
>
> - 1. the list is ordered
> - As long as the element was not found:

```
 > - 2. the list is divided into two equal parts.
 > - 3. we test if the extreme value of the 1st list corresponds to the
searched value
 > - 4. If the test is positive --> STOP
 > - 5. If the test is negative, we seek in which of the two parts of
the list the element is found
 > - 6. start again from 2.
```

In [100…

```python
def dichotomous_search(liste, valeur):
    first = 0
    last  = len(liste)-1
    index = -1
    count = 0
    while (first <= last) and (index == -1):
        count += 1
        mid = (first+last)//2 # La liste est découpée en deux
        if liste[mid] == valeur: # On regarde si la valeur extrème de la liste correspon
            index = mid
        else:
            # Search for array in which element stands in.
            if valeur < liste[mid]:
                last = mid -1
            else:
                first = mid +1
    #print("nombre d'itérations:",count)
    return index
```

> This search requires ordering the elements of an array. The ordering algorithm needs to be taken into account in order to assess the time consumed by algorithm.

In [101…

```python
len(liste_test)
```

Out[101]: 1000000

In [102…

```python
%time liste_ordonnee = sorted(liste_test)
```

```
CPU times: user 206 ms, sys: 0 ns, total: 206 ms
Wall time: 206 ms
```

In [103…
```
liste_ordonnee[:10]
```

Out[103]:
```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

In [104…
```
%time dichotomous_search(liste_ordonnee, element)
```

```
CPU times: user 10 µs, sys: 0 ns, total: 10 µs
Wall time: 13.4 µs
```

Out[104]:
```
121030
```

> While taking into account all involved algorithms

In [105…
```
%time dichotomous_search(sorted(liste_test),element)
```

```
CPU times: user 224 ms, sys: 72 µs, total: 224 ms
Wall time: 223 ms
```

Out[105]:
```
121030
```

Representation of the complexity of an algorithm

We can carry out an experimental study of the complexity of a search algorithm in a list, by varying the size of the lists analyzed and calculating the average CPU calculation time and by graphically representing this calculation time as a function of the size of the lists analyzed.

Implementation of the measure

In [106…
```
M   = 1000
ELT = 500
N   = 100
# The statement that will be executed for the measurement
statement = "dichotomous_search(liste_ordonnee[:{}],{})".format(M, ELT)

# import of the functions and data necessary for the execution of the instruction.
# Note that __main__ instructs Python interpreter that the notebook is the entry point.
setup = "from __main__ import dichotomous_search, liste_ordonnee"

# Instantiating the Timer Object (see OOP session)
oTimer = Timer(statement, setup)

# Measurement of the elapsed time for the instruction executed N times having the an ave
oTimer.timeit(number=N)
```

Out[106]:
```
0.0005420967936515808
```

Measuring the complexity of the `dichotomous search` algorithm

> We are going to repeat the measure implemented previously on a list of increasing size.
>
> We will obtain a representation of time according to the size of the data, $t = f(size)$.

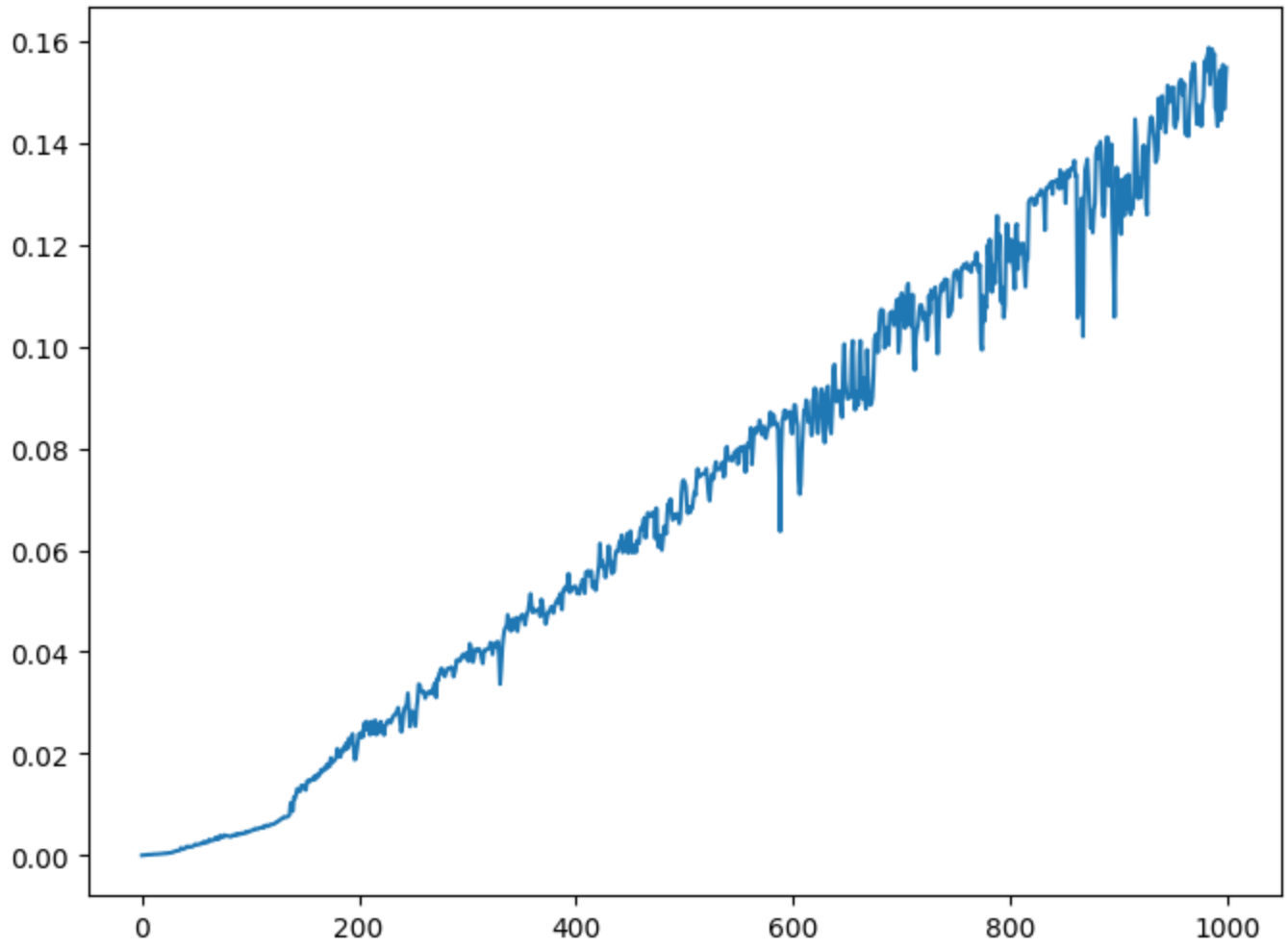
In [107…
```
%%time
```

```
list_computation_time = []
elt = 500
N = 5
for m in range(1000):
    M = 1000*(m+1)# The size of the array ranges from 1000 to 1000001
    t = Timer('''dichotomous_search(liste_ordonnee[:{}],{})'''.format(M, elt),
              "from __main__ import dichotomous_search, liste_ordonnee")
    list_computation_time.append(t.timeit(number=N))# N repetitions of the algorithm for
```

```
CPU times: user 1min 10s, sys: 0 ns, total: 1min 10s
Wall time: 1min 10s
```

In [108…

```
plt.figure(figsize=(8,6))
list_x = range(1000)
list_y = list_computation_time
plt.plot(list_x, list_y);
```



> The graph can be approximated as a straight line; the algorithm is said to have linear complexity. The computation time linearly depends on the size of the data.
>
> This would not be the case for an algorithm with a double loop, for example. The dependency would be like the power of two of the data size.

Sorting algorithms

Sorting algorithms from `numpy`

Sort of a list with numpy:

`quicksort` : quick sort (default `kind = 'quicksort'`)

`heapsort` : sort (by comparison) by heap

`mergesort` : merge sort

`timsort` : hybrid sort derived from merge sort and insertion sort

In [109…
```
len(test_nparray)
```

Out[109]: 1000000

In [110…
```
for kind in {'quicksort', 'mergesort', 'heapsort', 'stable'}:# NB : ths is a set
    print(kind)
    %time np.sort(test_nparray, kind=kind)
```

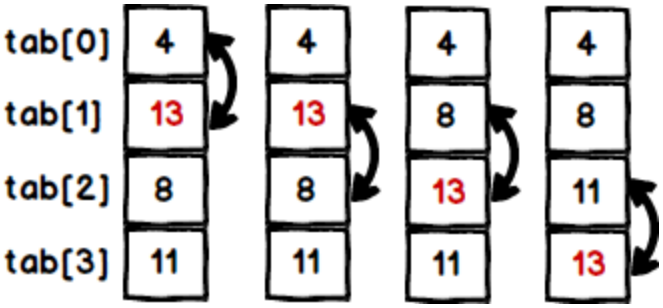
```
heapsort
CPU times: user 124 ms, sys: 4.01 ms, total: 128 ms
Wall time: 128 ms
quicksort
CPU times: user 53.3 ms, sys: 10 µs, total: 53.3 ms
Wall time: 53.3 ms
stable
CPU times: user 68.7 ms, sys: 40 µs, total: 68.7 ms
Wall time: 68.4 ms
mergesort
CPU times: user 65.2 ms, sys: 0 ns, total: 65.2 ms
Wall time: 65.3 ms
```

In [111…
```
import random as rd
```

In [112…
```
LIST_SIZE = 10000
list_ = [i for i in range(0,LIST_SIZE)]
# The list is shuffled for the sort to be valid
liste_a_trier = rd.shuffle(list_)
liste_a_trier = list_
print(len(liste_a_trier), type(liste_a_trier))
```

```
10000 <class 'list'>
```

Bubble sort



> Bubble sorting consists of traversing the table, for example from left to right, comparing the elements side by side and swapping them if they are not in the correct order.

> During a pass of the painting, the largest elements rise gradually to the right like bubbles towards the surface.

In [113…
```
list_ = [i for i in range(0,LIST_SIZE)]
liste_a_trier = rd.shuffle(list_)
liste_a_trier = list_

len(liste_a_trier)
```

Out[113]: 10000

In [114…
```
liste_a_trier[:None] # Liste entière
liste_a_trier[-2-1]  # Element en 3eme position a partir de la fin de la liste
```

Out[114]: 534

In [115…
```
M = None
len(liste_a_trier[:-1])
```

Out[115]: 9999

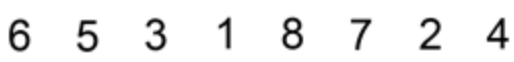
> L'implementation est une forme de tri par selection en partant de la fin de la liste.
>
> Partir de la fin impose de rechercher la valeur max de la liste à positioner en fin de liste.

In [116…
```
def bubble_sort(liste):
    for i,_ in enumerate(liste):
        # We consider all the elements of the list except the last ones
        # If i is 0, we take the whole list.
        # We get the max value of this list.
        val_max = max(liste[:(-i if i else None)]) # NB: liste[:0] == [] vs liste[:None]

        # We get the index of the max value of the truncated list
        ind_val_max = liste.index(val_max)

        # The previous -i value and the max value are swapped; the max remote value ther
    return liste
```

In [117…
```
%time liste_triee = bubble_sort(liste_a_trier)
```

```
CPU times: user 927 ms, sys: 0 ns, total: 927 ms
Wall time: 927 ms
```

Insertion sort



In [118…
```
def insertion_sort1(data):
    for r in range(1, len(data)):
        for l in range(r):
            if data[r] < data[l]:
                temp = data[r]
                data[l+1:r+1] = data[l:r]
                data[l] = temp
    return data
```

In [119…
```
%time liste_triee = insertion_sort1(liste_a_trier)
```

```
CPU times: user 2.6 s, sys: 4.05 ms, total: 2.61 s
Wall time: 2.61 s
```

Some tricks to know

In [120…
```
np_ar1 = np.array([1,2,4,5,6])
print(np_ar1)
np_ar_bool = np_ar1 <= 3
print(np_ar_bool)
```

```
[1 2 4 5 6]
[ True  True False False False]
```

> `argmin` will return the index (the list argument) corresponding to the smallest value.
>
> We recover the index of the smallest value <=3, it is the 1st index to False.

In [121…
```
(np_ar1 <= 3).argmin()
```

Out[121]: 2

In [122…
```
ind_insert= (np_ar1 <= 3).argmin()
ind_insert
```

Out[122]: 2

> Array to be sorted is splited into three parts and is rebuilt using concatenation from numpy `np.concatenate`
>
> Array partitions are :
>
> > - The `True` values part that satisfies condition array <= value_to_be_inserted
> > - The `False` values part that matches with condition array > value_to_be_inserted
> > - The part composed of a single element, value_to_be_inserted (3 here below) matching with element where split takes place.

In [123…
```
# Numpy arrays concatenation
np.concatenate( (np_ar1[:ind_insert],np_ar1[ind_insert:ind_insert],np_ar1[ind_insert:]) )
```

Out[123]: array([1, 2, 4, 5, 6])

> NB : concatenation takes place for arrays.
>
> - `np_ar1[ind_insert]` is a value
> - `np_ar1[ind_insert:ind_insert]` is a numpy array

In [124…
```
# In the example below, the returned value does not matches with a deterministic value i
# does not exists in the array.
(np_ar1 <= 8).argmin() # il faudra traiter ce cas séparément
```

Out[124]: 0

> Building unsorted list that aims to be sorted.

In [125…
```
list_shuffled = [i for i in range(0,LIST_SIZE)]
rd.shuffle(list_shuffled)
len(list_shuffled), list_shuffled[23:33]
```

Out[125]: (10000, [3164, 1641, 906, 8452, 7895, 9948, 4981, 4122, 5804, 3945])

In [126…
```
from typing import List
def insertion_sort(list_for_sorting):
    # Initialization of the array to be sorted that includes the 1st element of the arra
    np_sorted = np.array(list_for_sorting[:1])

    for i,elt in enumerate(list_for_sorting[1:]):
        j = i+1 # Shift indice because liste[1:] is traversed
        if elt >= np_sorted.max():
            # Add the greatest element at the end of the list
            np_sorted = np.concatenate( (np_sorted[:j],[elt]) )
        else:
            # Element has to be inserted in the sorted list at the right place
            # Search for first smaller element in the sorted list : search for
            # the smallest element in the sorted array.
            index_of_smallest_elt = (np_sorted[:j] <= elt).argmin()

            # Add the element just after values limited with this smallest element
            np_sorted = np.concatenate( (np_sorted[:index_of_smallest_elt], [elt], np_so
    return np_sorted
```

In [127…
```
%time liste_sorted = insertion_sort(list_shuffled)
```

```
CPU times: user 150 ms, sys: 4.03 ms, total: 154 ms
Wall time: 147 ms
```

In [128…
```
liste_sorted[-10:]
```

Out[128]: array([9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999])

> Numpy's `max` method makes the performance difference with Python's standard `max` method.

Comparative study of complexity in time

> The device implemented is identical to that seen previously for `Time measurement of`

`the complexity of an algorithm`

The device will be applied to three algorithms, the measurements will be recorded for each of them and compared on a graph.

In [129…
```
LIST_SIZE = 1000
```

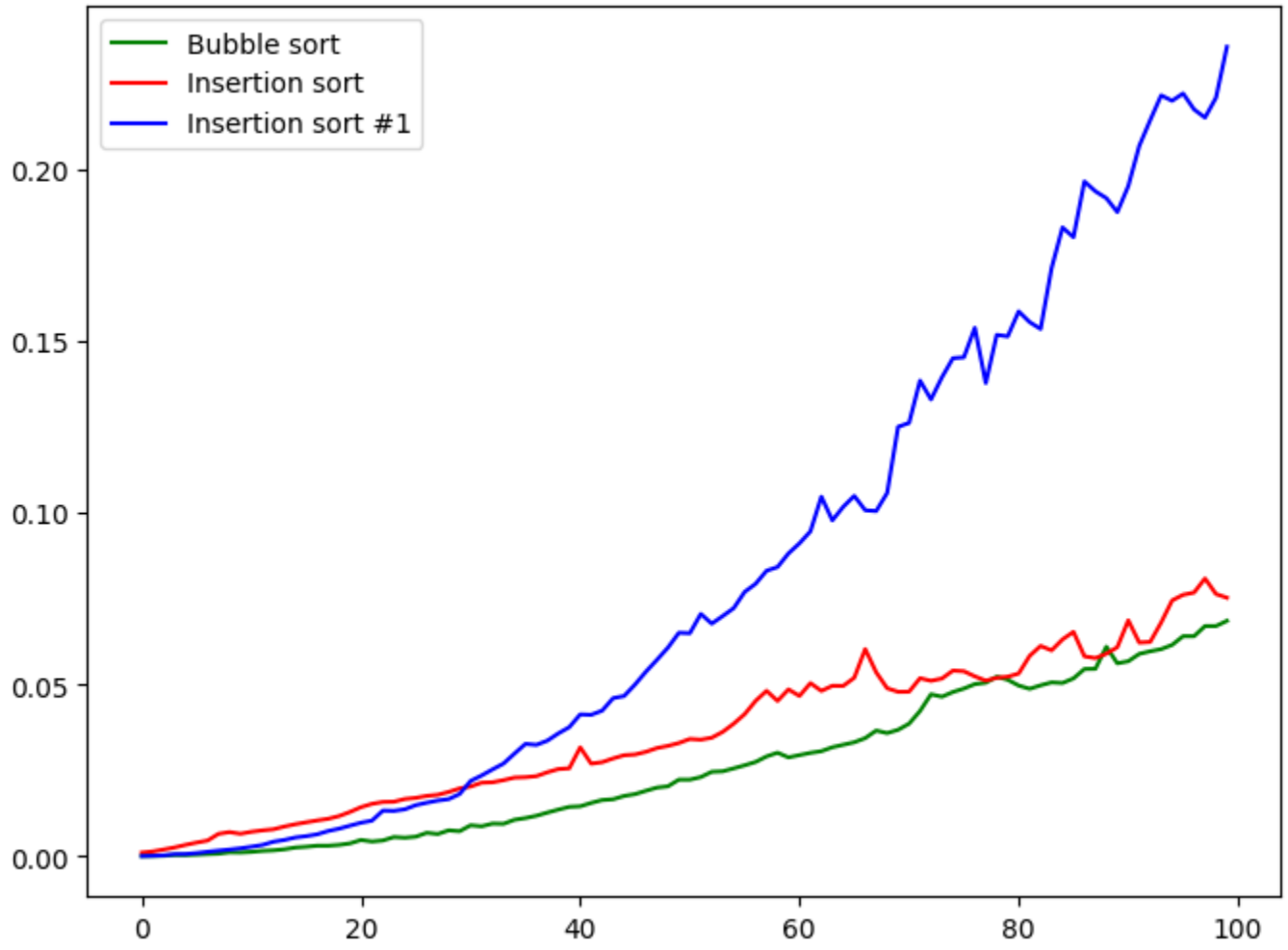
In [141…
```
n = int(1e3)
liste_1 = [i for i in range(0,LIST_SIZE)]
rd.shuffle(liste_1)
liste_2 = liste_1.copy()
liste_3 = liste_1.copy()
```

In [142…
```
time_insertion = []
for m in range(100):
    M = 10*(m+1)
    t = Timer('''insertion_sort(liste_2[:{}])'''.format(M),
              "from __main__ import insertion_sort, liste_2")
    time_insertion.append(t.timeit(number=10))
```

In [143…
```
time_insertion1 = []
for m in range(100):
    M = 10*(m+1)
    t = Timer('''insertion_sort1(liste_2[:{}])'''.format(M),
              "from __main__ import insertion_sort1, liste_2")
    time_insertion1.append(t.timeit(number=10))
```

In [144…
```
time_bubble = []
for m in range(100):
    M = 10*(m+1)
    t = Timer('''bubble_sort(liste_3[:{}])'''.format(M),
              "from __main__ import bubble_sort, liste_3")
    time_bubble.append(t.timeit(number=10))
```

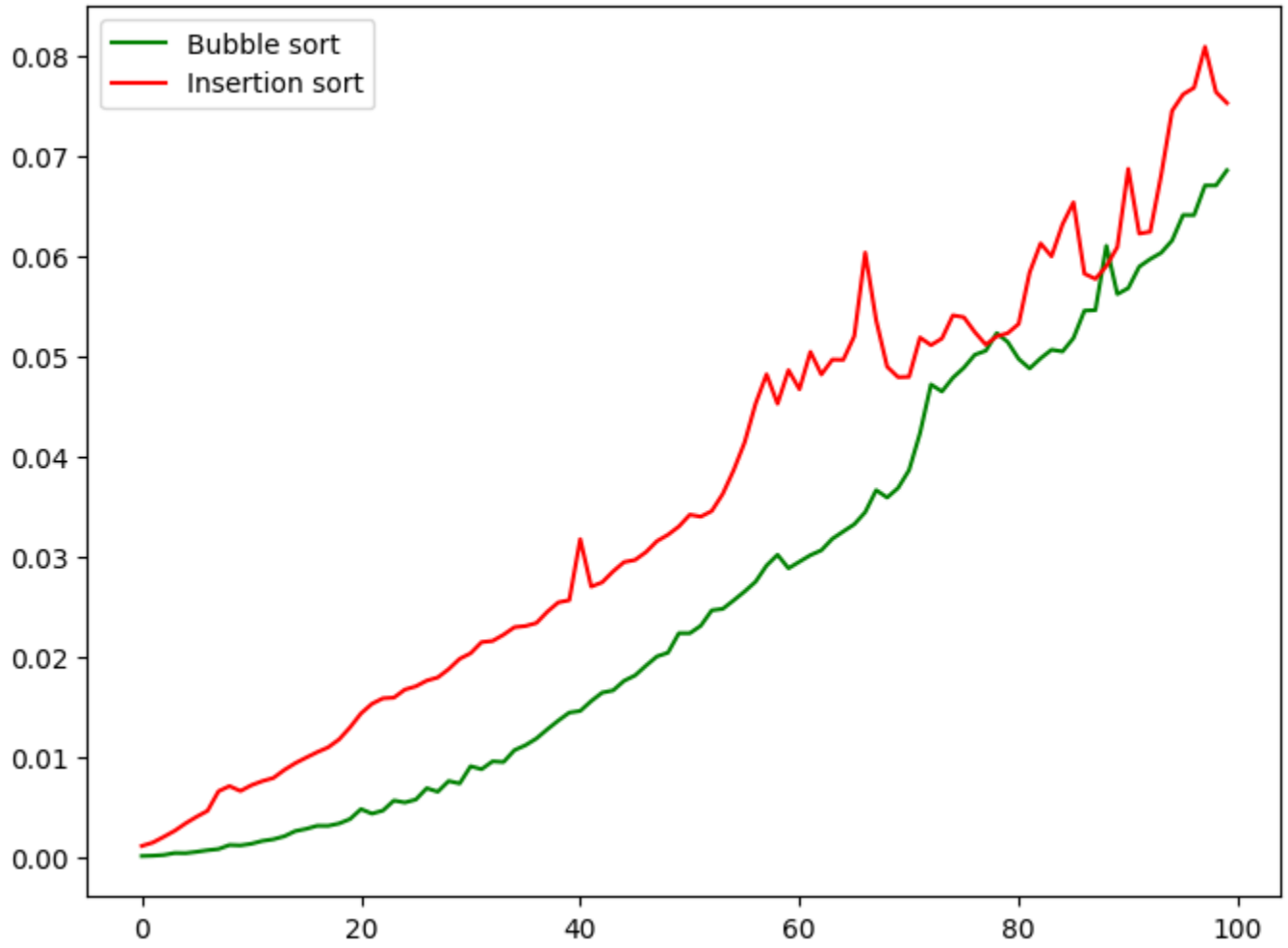
In [145…
```
%matplotlib inline
x = range(100)
plt.figure(figsize=(8,6))
plt.plot(x, time_bubble, 'g', label='Bubble sort')
plt.plot(x, time_insertion, 'r', label='Insertion sort')
plt.plot(x, time_insertion1, 'b', label='Insertion sort #1')
plt.legend();
```



> insertion_sort1 algorithm has $O(t^2)$ time-complexity. This is due to double loop implementation.

In [146…

```
%matplotlib inline
x = range(100)
plt.figure(figsize=(8,6))
plt.plot(x, time_bubble, 'g', label='Bubble sort')
plt.plot(x, time_insertion, 'r', label='Insertion sort')
plt.legend();
```



In [148…
```
n = int(1e3)
liste_1 = [i for i in range(0,LIST_SIZE)]
rd.shuffle(liste_1)
liste_2 = liste_1.copy()
liste_3 = liste_1.copy()
```

In [ ]:
```
time_insertion = []
for m in range(1000):
    M = 1*(m+1)
    t = Timer('''insertion_sort(liste_2[:{}])'''.format(M),
              "from __main__ import insertion_sort, liste_2")
    time_insertion.append(t.timeit(number=10))
```

In [ ]:
```
time_bubble = []
for m in range(1000):
    M = 1*(m+1)
    t = Timer('''bubble_sort(liste_3[:{}])'''.format(M),
              "from __main__ import bubble_sort, liste_3")
    time_bubble.append(t.timeit(number=10))
```

In [ ]:
```
x = range(1000)
plt.figure(figsize=(8,6))
%matplotlib inline
plt.plot(x, time_bubble, 'g', label='Bubble sort')
plt.plot(x, time_insertion, 'r', label='Insertion sort')
plt.legend();
```

Exercises

Exercise 1

> On the graph above, how would you qualify (linearity,...) each one of the time algoritms complexity ?

Exercise 2

Sort by selection on a list

> The principle consists of selecting the smallest value from the list and exchanging this value with the 1st value from the list, redoing this operation with the private list of the 1st value.
>
> - The list does not need to be ordered.
> - We also note the absence of a test which bodes well for good performance.
> - This algorithm uses the `min` method from list that implements an optimized algorithm.
>
> Algorithm
>
> - 1 Traverse all `i` indices in the list
>
> > - 1.1 Finding the minimum value of `list[i:]`, list starting at index `i`
> > - 1.2 The `list[i]` value are swapped with the minimum value
> > - 1.3 Go to 1.1

> 2.1 With the **dynamic programming** method, draw the sort algorithm.

> 2.2 Write a function that implements such algorithm.

> 2.3 Test the function with a random sample range of indices

> 2.4 Evaluate its complexity in time

> 2.5 represent the time-complexity sort algorithm on a graph with a sample of 1000 points

> 2.6 Compare on a same graph, this time-complexity with previous algorithms viewed during this session

> 2.7 Which one of the algorithms should you select for the sort task?

In [ ]:

In [ ]: