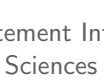
Cours 1 : Méthodes d'analyse des algorithmes - Illustration sur les algorithmes de tri

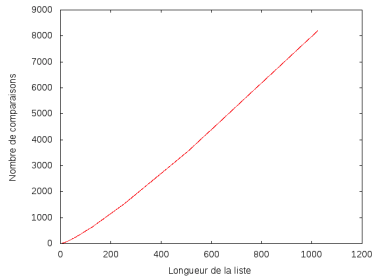
Jean-Stéphane Varré

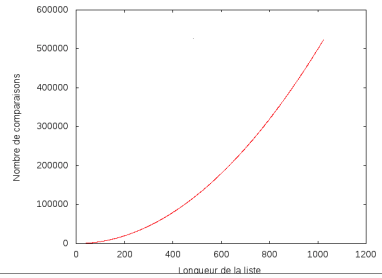
Département Informatique
Faculté des Sciences et Technologies
Université de Lille

jean-stephane.varre@univ-lille.fr



Que pensez-vous de cela ?





Etant donné un algorithme :

- combien de temps va-t-il mettre à s'exécuter ?
- combien d'espace mémoire va-t-il utiliser ?
- a-t-il toujours le même comportement ou bien existe-t-il des données plus ou moins favorables ?

Etant donné un algorithme :

- combien de temps va-t-il mettre à s'exécuter ?
- combien d'espace mémoire va-t-il utiliser ?
- a-t-il toujours le même comportement ou bien existe-t-il des données plus ou moins favorables ?

?

- **que compter ?**
- **comment compter ?**

Mais avant cela, parlons tableaux

Les tableaux ne sont pas des listes

- suite ordonnée d'éléments (i.e. il existe un suivant et un précédent)
- de taille bornée
- chaque élément est associé à un indice
- qui supporte les opérations :
  - d'accès à un élément par son indice
  - d'accès au nombre d'éléments : la longueur

Tableaux en Python

- n'existent pas nativement
- mais sont implantés dans la bibliothèque `numpy`

```
1  >>> import numpy as np
2  >>> t = np.array([1, 2, 3])
3  >>> t
4  array([1, 2, 3])
5  >>> print(t)
6  [1 2 3]
7  >>> type(t)
8  <class 'numpy.ndarray'>
9  >>> t.dtype
10 dtype('int64')
```

exemples . . . voir portail pédagogique

- extrait de la doc NumPy

Internal memory layout of an ndarray

An instance of class `ndarray` consists of a contiguous one-dimensional segment of computer memory (owned by the array, or by some other object), combined with an indexing scheme that maps *N* integers into the location of an item in the block. The ranges in which the indices can vary is specified by the `shape` of the array. How many bytes each item takes and how the bytes are interpreted is defined by the data-type object associated with the array.

Le type `Element`

```
1  from functools import total_ordering
2
3  @total_ordering
4  class Element:
5
6      def __init__(self,value):
7          assert(type(value) == int)
8          self.value = value
9
10     def __eq__(self, other):
11         return self.value == other.value
12
13     def __ne__(self, other):
14         return not (self == other)
15
16     def __lt__(self, other):
17         return self.value < other.value
18
19     def __repr__(self):
20         return "{}".format(self.value)
```

Présence d'un élément dans un tableau (v1)

```
1  def contains(t,v):
2      """
3      :param a:
4      :type t: a NumPy array of Element
5      :param b:
6      :type v: Element
7      :return:
8         - True if v is in t
9         - else False
10     """
11     found = False
12     for i in range (0,len(t)):
13         if t[i] == v:
14             found = True
15     return found
```

- temps d'exécution ?
- espace mémoire occupé ?

Observons

- on utilise le module `cProfile` qui permet de suivre les appels de fonctions et le temps passé
- recherche de l'élément de valeur 500 dans un tableau de 10,000,000 éléments

```
1  # python3 -m cProfile -s time testContains.py
2  True
3         20050832 function calls (20048641 primitive calls) in 54.368 seconds
4
5     Ordered by: internal time
6
7     ncalls  tottime  percall  cumtime  percall filename:lineno(function)
8         77   22.426    0.291   22.426    0.291 {built-in method numpy.core.multiarra
9          1   12.550   12.550   21.528   21.528 testContains.py:20(<listcomp>)
10  10000001    8.977    0.000    8.977    0.000 Element.py:7(__init__)
11         1    6.151    6.151    9.799    9.799 testContains.py:4(contains_v1)
12  10000000    3.647    0.000    3.647    0.000 Element.py:13(__eq__)
```


Variation de la longueur du tableau

longueur 10,000

```
         70802 function calls (68611 primitive calls) in 0.359 seconds

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    17/16    0.116    0.007    0.122    0.008 {built-in method _imp.create_dynamic}
      124    0.023    0.000    0.023    0.000 {built-in method marshal.loads}
       77    0.021    0.000    0.021    0.000 {built-in method numpy.core.multiarray.array}
   319/7    0.017    0.000    0.301    0.043 {built-in method builtins.__import__}
       15    0.011    0.001    0.011    0.001 {built-in method posix.listdir}
```

longueur 100,000

```
         250802 function calls (248611 primitive calls) in 0.915 seconds

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       77    0.207    0.003    0.207    0.003 {built-in method numpy.core.multiarray.array}
      124    0.134    0.001    0.134    0.001 {method 'read' of '_io.FileIO' objects}
    17/16    0.099    0.006    0.103    0.006 {built-in method _imp.create_dynamic}
   100001    0.095    0.000    0.095    0.000 Element.py:7(__init__)
        1    0.080    0.080    0.175    0.175 testContains.py:20(<listcomp>)
```

longueur 1,000,000

```
         2050802 function calls (2048611 primitive calls) in 5.715 seconds

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       77    2.145    0.028    2.145    0.028 {built-in method numpy.core.multiarray.array}
  1000001    1.319    0.000    1.319    0.000 Element.py:7(__init__)
        1    0.903    0.903    2.222    2.222 testContains.py:20(<listcomp>)
        1    0.611    0.611    0.939    0.939 testContains.py:4(contains_v1)
  1000000    0.328    0.000    0.328    0.000 Element.py:11(__eq__)
```


J'arrive à observer le comportement de mon algorithme sur de grandes instances du problème

Observation de certaines built-in functions

```
taille  ncalls tottime  percall filename:lineno(function)

   10k: 17/16  0.116    0.007  {built-in method _imp.create_dynamic}
  100k: 17/16  0.099    0.006  {built-in method _imp.create_dynamic}
 1000k: 17/16  0.126    0.007  {built-in method _imp.create_dynamic}
10000k: 17/16  0.110    0.006  {built-in method _imp.create_dynamic}

   10k: 124    0.003    0.000  {method 'read' of '_io.FileIO' objects}
  100k: 124    0.134    0.001  {method 'read' of '_io.FileIO' objects}
 1000k: 124    0.080    0.001  {method 'read' of '_io.FileIO' objects}
10000k: 124    0.072    0.001  {method 'read' of '_io.FileIO' objects}

   10k: 19/7   0.017    0.000  {built-in method builtins.__import__}
  100k: 19/7   0.001    0.000  {built-in method builtins.__import__}
 1000k: 19/7   0.001    0.000  {built-in method builtins.__import__}
10000k: 19/7   0.001    0.000  {built-in method builtins.__import__}
```

Seules certaines fonctions sont impactées par le changement de taille du problème que je soumets à mon algorithme.

Variation du temps pris pour faire la comparaison

hypothèse : l'opération de comparaison des `Element` influence le temps d'exécution de mon algorithme

cas où le temps d'exécution de `__eq__` augmente

Temps de eq	total	`contains_v1`	`__init__`	`__eq__`
x1000	0.601	0.128	0.001	0.126
x10000	1.652	1.465	0.001	1.457
x100000	13.523	13.295	0.001	13.284
x1000000	132.910	132.422	0.001	132.398

cas où le temps d'exécution de `__init__` augmente

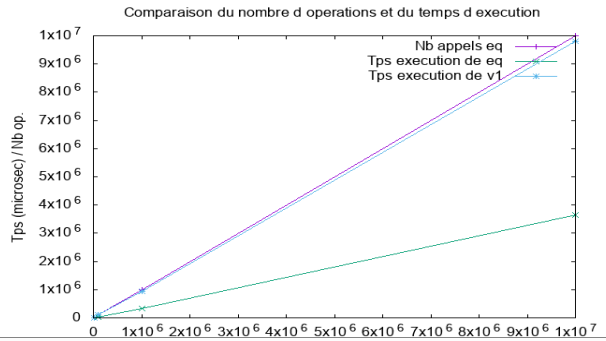
Temps de `init`	total	`contains_v1`	`__init__`	`__eq__`
x1000	0.590	0.001	0.134	0.000
x10000	1.727	0.001	1.227	0.000
x100000	12.650	0.001	12.167	0.001
x1000000	138.088	0.006	137.777	0.001

J'observe correctement le comportement de mon algorithme si je sais choisir l'opération d'intérêt

Au revoir le temps ... bonjour le nb d'opérations

nombre d'appels de fonctions

			contains_v1	
	nb total	nb	nb	nb
Temps de eq	__init__	contains_v1	__init__	__eq__
x1000	1001	1	0	1000
x10000	1001	1	0	1000
x100000	1001	1	0	1000
x1000000	1001	1	0	1000



Et si j'ai plusieurs algorithmes qui traitent la même tache, comment les comparer ?

un algorithme A qui nécessite $10^{-4} \times 2^n$ *instructions d'intérêt est-il moins performant qu'un algorithme B qui requiert* $10^{-2} \times n^5$ *instructions d'intérêt pour traiter une donnée de taille n ?*

Il suffit de compter le nombre d'opérations d'intérêt pour approcher, à un facteur multiplicatif près, le temps pris par mon algorithme

Au menu

- notions de pire des cas et meilleur des cas
- revisite des tris bulle, sélection, insertion
- comportement asymptotique
- le tri par insertion récursive et dichotomique
- résolution des équations de partition
- analyse des tris fusion, rapide