tri_nlnd

July 1, 2022

1 1A.algo - Tri plus rapide que prévu

Dans le cas général, le coût d'un algorithme de tri est en $O(n \ln n)$. Mais il existe des cas particuliers pour lesquels on peut faire plus court. Par exemple, on suppose que l'ensemble à trier contient plein de fois le même élément.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

[2]:
```
%matplotlib inline
```

1.1 trier un plus petit ensemble

[3]:
```
import random
ens = [random.randint(0,99) for i in range(10000)]
```

On peut calculer la distribution de ces éléments.

[4]:
```
def histogram(ens):
    hist = {}
    for e in ens:
        hist[e] = hist.get(e, 0) + 1
    return hist

hist = histogram(ens)
list(hist.items())[:5]
```

[4]:
```
[(0, 91), (1, 97), (2, 101), (3, 99), (4, 87)]
```

Plutôt que de trier le tableau initial, on peut trier l'histogramme qui contient moins d'élément.

[5]:
```
sorted_hist = list(hist.items())
sorted_hist.sort()
```

Puis on recontruit le tableau initial mais trié :

[6]:
```
def tableau(sorted_hist):
    res = []
    for k, v in sorted_hist:
        for i in range(v):
            res.append(k)
    return res
```

```
sorted_ens = tableau(sorted_hist)
sorted_ens[:5]
```

[6]: `[0, 0, 0, 0, 0]`

On crée une fonction qui assemble toutes les opérations. Le coût du nivrau tri est en $O(d \ln d + n)$ où d est le nombre d'éléments distincts de l'ensemble initial.

[7]:
```
def sort_with_hist(ens):
    hist = histogram(ens)
    sorted_hist = list(hist.items())
    sorted_hist.sort()
    return tableau(sorted_hist)

from random import shuffle
shuffle(ens)
%timeit sort_with_hist(ens)
```

```
100 loops, best of 3: 3.23 ms per loop
```

[8]:
```
def sort_with_nohist(ens):
    return list(sorted(ens))
```

[9]:
```
shuffle(ens)
%timeit sort_with_nohist(ens)
```

```
100 loops, best of 3: 2.9 ms per loop
```

Les temps d'exécution ne sont pas très probants car la fonction `sort` est immplémentée en C et qu'elle utilise l'algorithme timsort. Cet algorithme est un algorithme adaptatif tel que smoothsort. Le coût varie en fonction des données à trier. Il identifie d'abord les séquences déjà triées, trie les autres parties et fusionne l'ensemble. Trier un tableau déjà trié revient à détecter qu'il est déjà trié. Le coût est alors linéaire $O(n)$. Cela explique le commentaire *The slowest run took 19.47 times longer than the fastest.* ci-dessous où le premier tri est beaucoup plus long que les suivant qui s'appliquent à un tableau déjà trié. Quoiqu'il en soit, il n'est pas facile de comparer les deux implémentations en terme de temps.

[10]:
```
def sort_with_nohist_nocopy(ens):
    ens.sort()
    return ens
shuffle(ens)
%timeit sort_with_nohist_nocopy(ens)
```

```
The slowest run took 21.88 times longer than the fastest. This could mean that
an intermediate result is being cached.
10000 loops, best of 3: 164 µs per loop
```

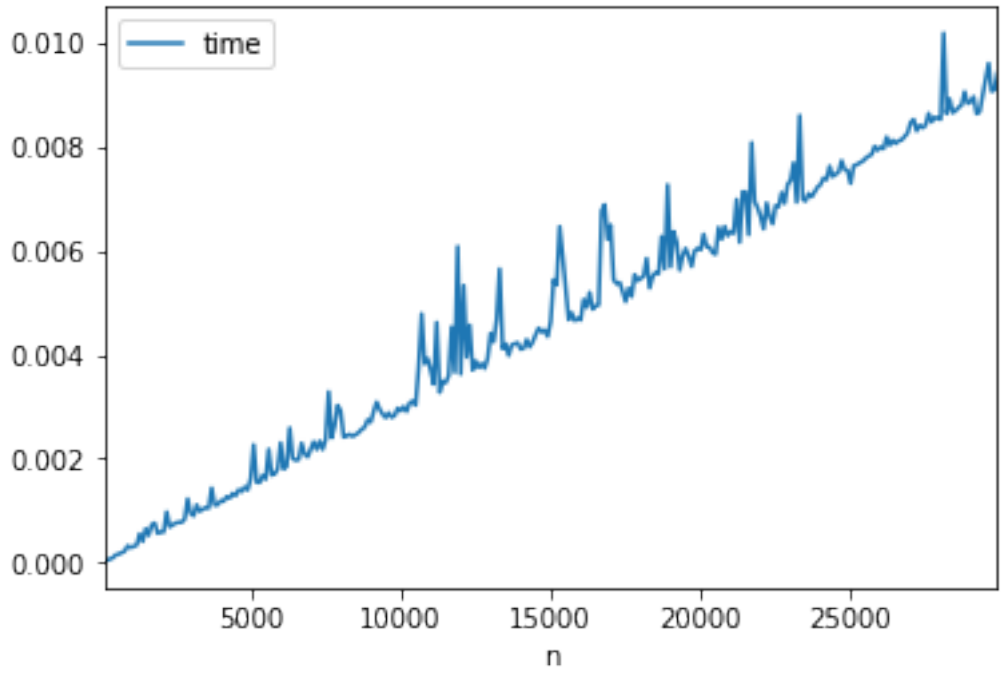
1.2 évolution en fonction de n

Pour réussir à valider l'idée de départ. On regarde l'évolution des deux algorithmes en fonction du nombre d'observations.

[11]:
```
def tableaux_aleatoires(ns, d):
    for n in ns:
```

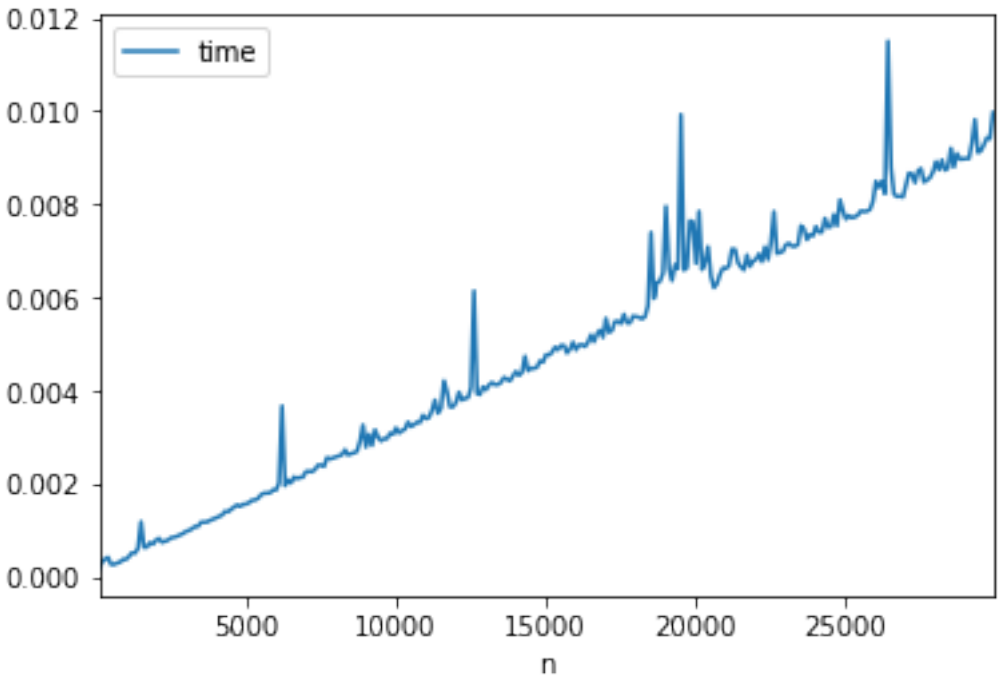
```
        yield [random.randint(0,d-1) for i in range(n)]

import pandas
import time
def mesure(enss, fonc):
    res = []
    for ens in enss:
        cl = time.perf_counter()
        fonc(ens)
        diff = time.perf_counter() - cl
        res.append(dict(n=len(ens), time=diff))
    return pandas.DataFrame(res)

df = mesure(tableaux_aleatoires(range(100, 30000, 100), 100), sort_with_nohist)
df.plot(x="n", y="time")
```

[11]: <matplotlib.axes._subplots.AxesSubplot at 0x219ef99ebe0>



[12]:
```
df = mesure(tableaux_aleatoires(range(100, 30000, 100), 100), sort_with_hist)
df.plot(x="n", y="time")
```

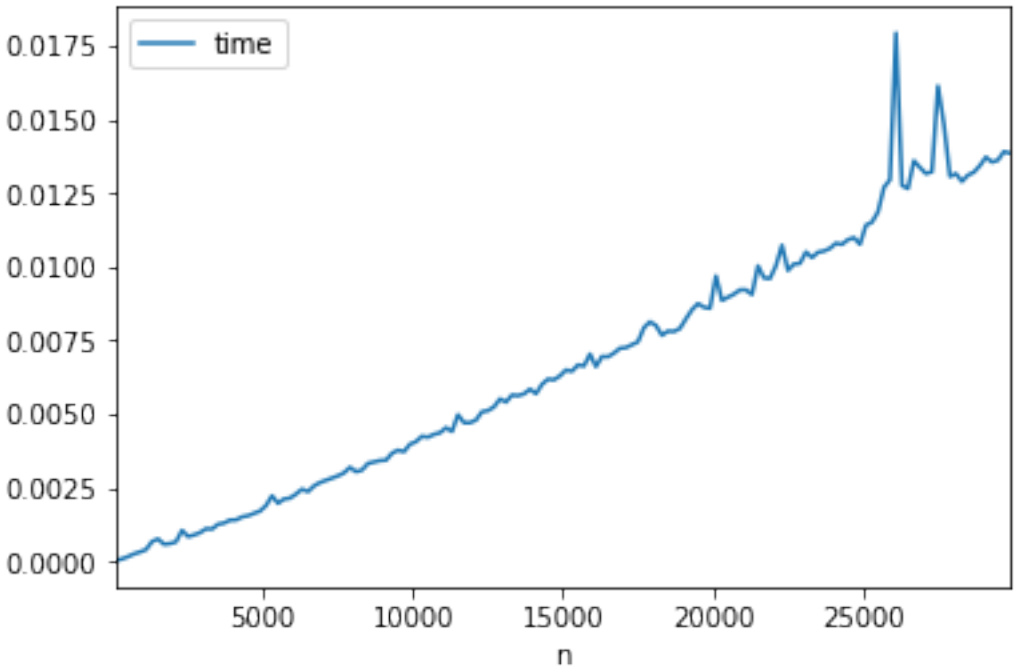
[12]: <matplotlib.axes._subplots.AxesSubplot at 0x219eb2ea2b0>



L'algorithme de tri de Python est plutôt efficace puisque son coût paraît linéaire en apparence.

[13]:
```
df = mesure(tableaux_aleatoires(range(100, 30000, 200), int(1e10)), sort_with_nohist)
df.plot(x="n", y="time")
```

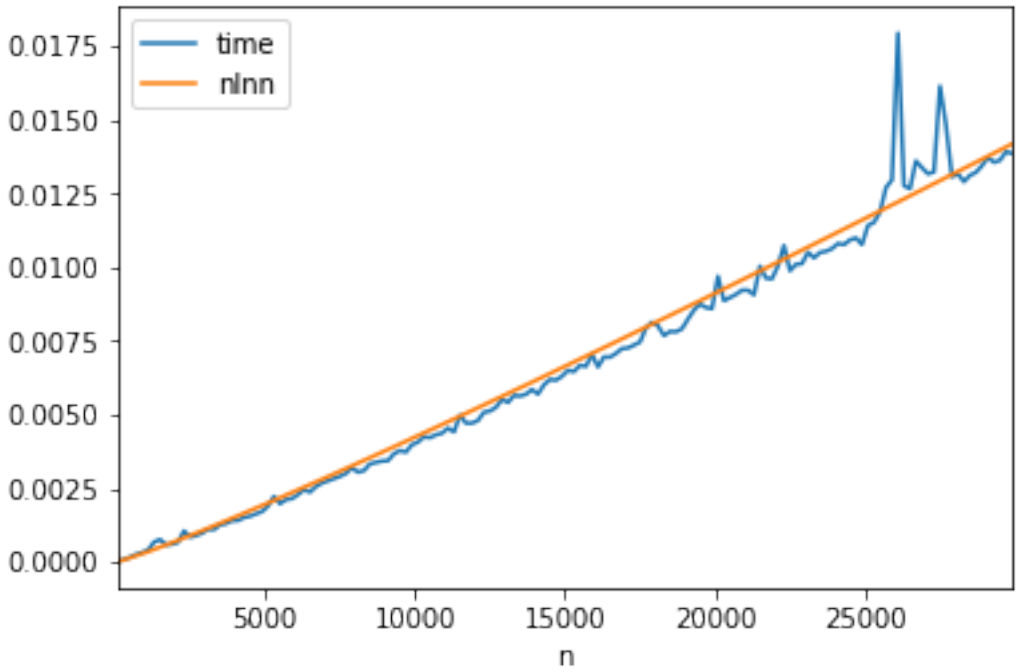
[13]: `<matplotlib.axes._subplots.AxesSubplot at 0x219efdadf98>`



On ajoute un logarithme.

[14]:
```
from math import log
df["nlnn"] = df["n"] * df["n"].apply(log) * 4.6e-8
df.plot(x="n", y=["time", "nlnn"])
```

[14]: <matplotlib.axes._subplots.AxesSubplot at 0x219f00cefd0>



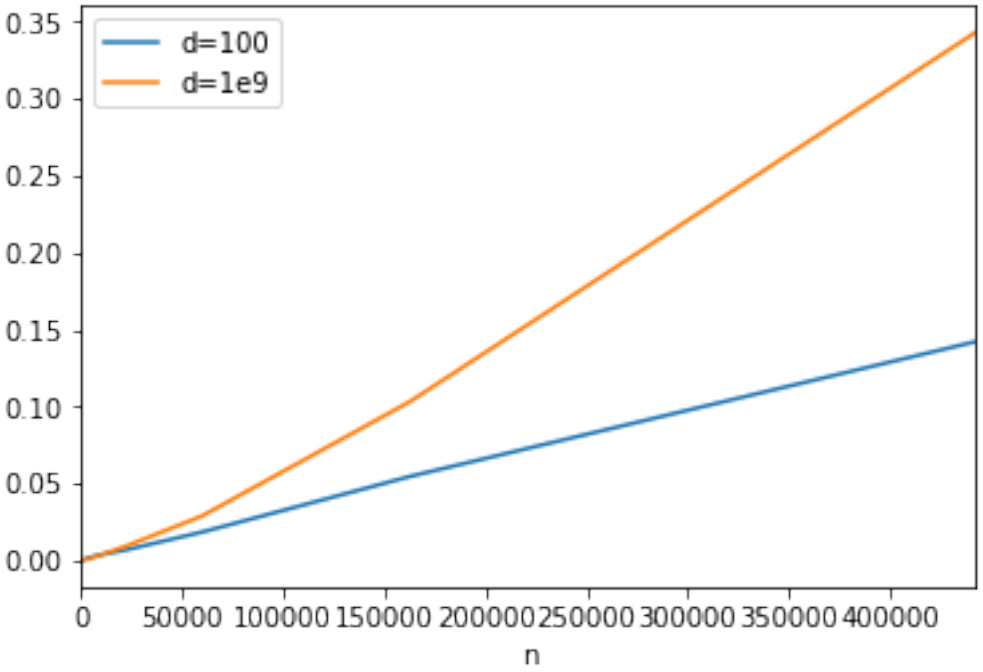
Il faut grossier le trait.

[15]:
```
from math import exp
list(map(int, map(exp, range(5, 14))))
```

[15]: [148, 403, 1096, 2980, 8103, 22026, 59874, 162754, 442413]

[16]:
```
df100 = mesure(tableaux_aleatoires(map(int, map(exp, range(5, 14))), 100),␣
 ↪sort_with_nohist)
dfM = mesure(tableaux_aleatoires(map(int, map(exp, range(5, 14))), 1e9),␣
 ↪sort_with_nohist)
df = df100.copy()
df.columns = ["n", "d=100"]
df["d=1e9"] = dfM["time"]
df.plot(x="n", y=["d=100", "d=1e9"])
```

[16]: <matplotlib.axes._subplots.AxesSubplot at 0x219f147dfd0>



L'algorithme de tri timsort est optimisé pour le cas où le nombre de valeurs distinctes est faible par rapport à la taille du tableau à trier.

[17]: