Quicksort

December 22, 2018

1 Le tri rapide

Marc Lorenzi
22 décembre 2018

```
In [1]: import matplotlib.pyplot as plt
        import random
        import math
```

```
In [2]: plt.rcParams['figure.figsize'] = (10, 6)
```

1.1 1 Introduction

1.1.1 1.1 Quicksort

Le tri rapide (quicksort) est un algorithme permettant de trier des listes en pire cas en complexité $\Theta(n^2)$ où n est la longueur de la liste à trier. Malgré cette mauvaise complexité en pire cas, quicksort possède une complexité en moyenne en $\Theta(n \log n)$. De plus il possède le grand avantage de pouvoir trier les listes sur place, sans création de nouvelles listes. Sa "constante cachée" est relativement petite, ce qui en fait un algorithme de choix pour de nombreux problèmes où un tri s'avère nécessaire.

Avant de nous attaquer à quicksort, nous allons mettre en place quelques outils qui nous seront bien utiles.

1.1.2 1.2 Un compteur d'opérations

Nous allons parler de complexité, et donc compter des opérations. L'intérêt d'exposer un algorithme dans un notebook est que nous allons pouvoir confronter la belle théorie à la vile pratique …

Qu'allons nous compter ? Quicksort est un algorithme de tri **comparatif**, c'est à dire que pour trier une liste il compare entre-eux des éléments de la liste. Mais, me direz-vous, pour trier il faut bien comparer ! Point du tout. Il existe des algorithmes de tri qui ne comparent pas, mais ceci est un autre sujet. Qu'allons nous compter, dis-je ? Eh bien essentiellement des comparaisons d'éléments de listes. Et aussi, histoire de comparer (si j'ose dire), des échanges d'éléments de listes. Pour cela, il nous sera bien utile de posséder un compteur qui comptera à notre place. Plutot que de polluer le code de nos belles fonctions de tri avec des incrémentations de variables hors-sujet, voici une classe `Compteur` … qui compte.

Un objet de la classe compteur possède deux champs, `comp` (compare) et `xch` (exchange). Ces deux champs contiennent des entiers qui sont incrémentés de 1 via les méthodes `incr_comp` et `incr_xch`. Un compteur peut aussi (heureusement) être réinitialisé via la méthode `reset`.

```
In [3]: class Compteur:

            def __init__(self):
                self.comp = 0
                self.xch = 0

            def incr_comp(self):
                self.comp += 1

            def incr_xch(self):
                self.xch += 1

            def reset(self):
                self.comp = 0
                self.xch = 0

            def __repr__(self):
                return 'comparaisons : %d\nechanges : %d' % (self.comp, self.xch)
```

Nous allons définir un Compteur Global `CG` qui sera LE compteur utilisé dans tout le notebook.

```
In [4]: CG = Compteur()
```

Testons notre compteur.

- On le réinitialise.
- On incrémente 10000 fois l'un ou l'autre des champs du compteur, au hasard.
- On affiche le compteur.
- On le réinitialise.
- On affiche le compteur (qui devrait être à zéro).

```
In [5]: CG.reset()
        for k in range(10000):
            r = random.uniform(0, 1)
            if r < 0.5: CG.incr_comp()
            else: CG.incr_xch()
        print(CG)
        CG.reset()
        print(CG)
```

```
comparaisons : 4955
echanges : 5045
comparaisons : 0
echanges : 0
```

Tout va bien, notre compteur a l'air de fonctionner.

1.1.3 1.3 Les opérations à compter

Bien entendu, utiliser le compteur va nous obliger à un peu de discipline. Chaque fois que nous comparerons deux éléments $s[i]$ et $s[j]$ d'une liste, nous nous obligerons à appeler `inférieur(s, i, j)`, qui renvoie `True` si et seulement si $s[i] \leq s[j]$.

```
In [6]: def inferieur(s, i, j):
            CG.incr_comp()
            return s[i] <= s[j]
```

```
In [7]: CG.reset()
        s = [1, 4, 2, 3]
        print(inferieur(s, 1, 3))
        print(CG)
```

```
False
comparaisons : 1
echanges : 0
```

Et chaque fois que nous échangerons les éléments d'indices i et j de la liste s, notre devoir sera d'appeler `echanger(s, i, j)`.

```
In [8]: def echanger(s, i, j):
            CG.incr_xch()
            s[i], s[j] = s[j], s[i]
```

```
In [9]: print(s)
        echanger(s, 1, 2)
        print(s)
        print(CG)
```

```
[1, 4, 2, 3]
[1, 2, 4, 3]
comparaisons : 1
echanges : 1
```

1.1.4 1.4 Listes aléatoires

Pour tester un algorithme de listes il est bon de pooséder une fonction renvoyant des listes aléatoires. La fonction ci-dessous prend un entier n en paramètre et renvoie une permutation aléatoire de l'ensemble des entiers entre 0 et $n-1$. L'algorithme utilisé est celui de **Fisher-Yates** : il renvoie toute liste avec une probabilité égale à $\frac{1}{n!}$. Ce qui est heureux puisqu'il y a $n!$ permutations possibles.

Remarquez que notre algorithme effectue 0 comparaison et n échanges.

```
In [10]: def liste_aleatoire(n):
             s = list(range(n))
             for i in range(n):
```

```
            j = random.randint(i, n - 1)
            echanger(s, i, j)
        return s
```

```
In [11]: CG.reset()
         print(liste_aleatoire(10))
         print(CG)
         CG.reset()
```

```
[1, 5, 2, 9, 6, 8, 3, 7, 0, 4]
comparaisons : 0
echanges : 10
```

1.1.5 1.5 Une liste est-elle triée ?

Pour savoir si une liste est triée il n'y a qu'à regarder ? Sans doute, mais on s'usera les yeux quand nos listes auront un million d'éléments. Alors écrivons une fonction qui regarde pour nous.

Une liste $[x_0, x_1, \ldots, x_{n-1}]$ est triée si et seulement si pour tout $k < n - 1$ on a $x_k \leq x_{k+1}$. D'où le code ci-dessous. La dernière ligne `k >= n - 1` permet au code de renvoyer la bonne valeur si la liste est vide (la liste vide est triée !).

```
In [12]: def est_triee(s):
             n = len(s)
             k = 0
             while k < n - 1 and s[k] <= s[k + 1]:
                 k = k + 1
             return k >= n - 1
```

```
In [15]: est_triee([1, 2, 3, 4, 5])
```

```
Out[15]: True
```

```
In [16]: est_triee([1, 2, 4, 3, 5])
```

```
Out[16]: False
```

```
In [17]: est_triee([])
```

```
Out[17]: True
```

1.1.6 1.6 Statistiques

Automatisons aussi nos futures statistiques sur quicksort (ou n'importe quel algorithme sur les listes qui compare ou qui échange). L'idée est simple. Étant donnée une fonction `algo` prenant une liste en paramètre, on exécute `algo` sur un certain nombre de listes (`sample`). On fait cela pour des tailles différentes. On renvoie pour chacune des tailles le nombre moyen de comparaisons et le nombre moyen d'échanges.

- `min` est la taille minimale des listes testées.

- max est la taille maximale des listes testées.
- step est le pas permettant de passer d'une taille de liste à la suivante.

Lisez le code, il ne présente aucune difficulté.

```
In [18]: def stats(algo_tri, step=1, min=1, max =100, sample=1):
             tailles = list(range(min, max+step, step))
             temps_comp = []
             temps_xch = []
             for k in tailles:
                 vcomp = 0
                 vxch = 0
                 for j in range(sample):
                     s = liste_aleatoire(k)
                     CG.reset()
                     algo_tri(s)
                     vcomp += CG.comp
                     vxch += CG.xch
                 temps_comp.append(vcomp / sample)
                 temps_xch.append(vxch / sample)
             return (tailles, temps_comp, temps_xch)
```

Testons la fonction stats sur un algorithme idiot (?) qui ne trie certainement pas la liste. Mais bon, il compare et il échange ...

```
In [19]: def algo_idiot(s):
             for i in range(len(s) - 1):
                 if inferieur(s, i, i + 1):
                     echanger(s, i, i + 1)
```

```
In [20]: s = liste_aleatoire(20)
         print(s)
         CG.reset()
         algo_idiot(s)
         print(CG)
```

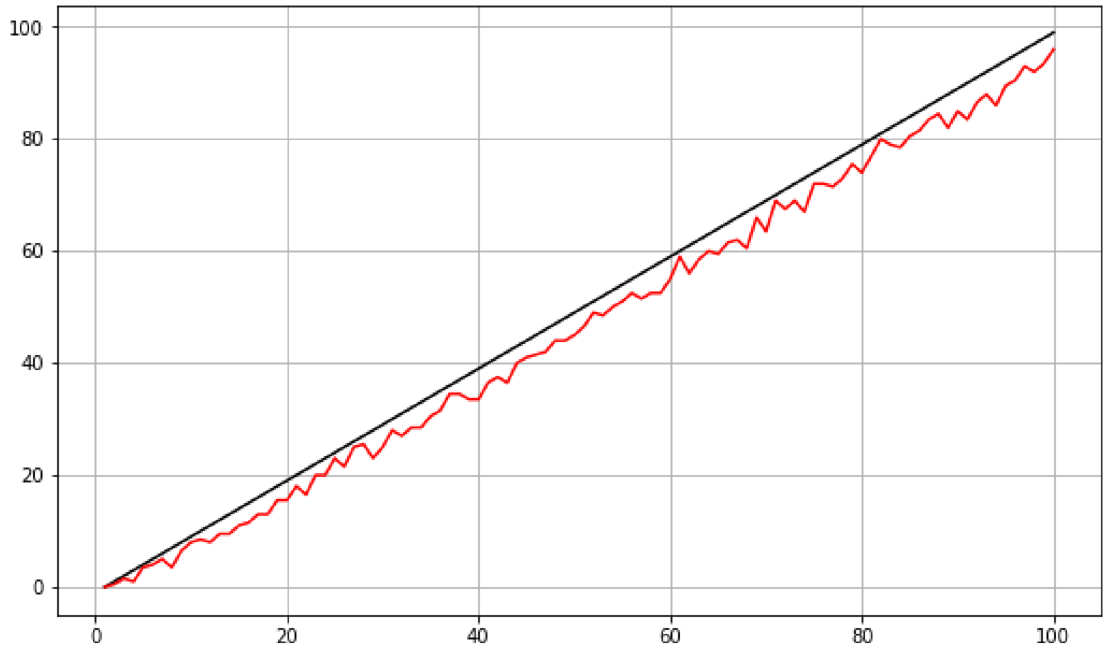
```
[3, 14, 17, 15, 10, 8, 5, 6, 12, 0, 4, 18, 2, 1, 7, 11, 9, 19, 16, 13]
comparaisons : 19
echanges : 18
```

```
In [22]: stts = stats(algo_idiot, 1, 1, 100, 2)
         print(stts)
```

```
([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
```

Ceci n'est pas très parlant, alors affichons un graphique.

```
In [23]: xs, ys, zs = stts
         plt.plot(xs, ys, 'k')
         plt.plot(xs, zs, 'r')
         plt.grid()
         plt.show()
```



En noir, le nombre de comparaisons, qui ne devrait surprendre personne. Pour une liste de taille n, c'est $n-1$. Que le nombre d'échanges soit inférieur au nombre de comparaisons ne devrait pas non plus être une surprise.

1.2 2 Le tri rapide

1.2.1 2.1 Le principe

Quicksort est un algorithme du type **diviser pour régner**.

Soit s une liste à trier entre les indices minimal l (left) et maximal r (right). Si $l > r$ il n'y a rien à faire. Sinon, soit $x = s[r]$.

1- On place dans la "moitié" gauche de la liste les éléments de s inférieurs ou égaux à x.

2- On place dans la "moitié" droite de la liste les éléments de s strictement supérieurs à x.

3- On trie la "moitié" gauche privée de x.

4- On trie la "moitié" droite.

Toutes ces opérations se passent bien entendu entre les indices l et r. Remarquez également les guillemets autour du mot "moitié" : il n'y a aucune raison que les moitiés soient égales, nous devrions plutôt dire "parties". Mais j'aime bien l'idée des deux moitiés non égales.

Exercice : que valent les deux "moitiés' lorsque la liste est déjà triée ?

Exercice : Quand les deux moitiés sont-elles exactement de même longueur ?

1.2.2 2.2 Partitionner

La fonction ci-dessous effectue les opérations 1 et 2 dont je viens de parler. Un indice i est mis à jour à l'intérieur d'une boucle indexée par un entier j.

J'ai rajouté un paramètre `dbg` qui, mis à `True`, permet de visualiser les étapes de l'algorithme. Lorsque `dbg=True`, on affiche les valeurs de i, j et s à la fin de chaque itération.

```
In [24]: def partition(s, l, r, dbg=False):
             i = l - 1
             for j in range(l, r):
                 if inferieur(s, j, r):
                     i = i + 1
                     echanger(s, i, j)
                 if dbg: print('i=%d, j=%d, s=%s'%(i, j, s))
             echanger(s, i + 1, r)
             return i + 1
```

Proposition : Soit $x = s[r]$. À l'issue de la j-ième itération ($l \leq j < r$) :

- $i \leq j$
- Tous les éléments de s dont l'indice est compris entre l et i sont inférieurs ou égaux à x.
- Tous les éléments de s dont l'indice est compris entre $i+1$ et j sont strictement supérieurs à x.

Démonstration : La preuve se fait par récurrence sur j.

Ainsi, à l'issue de la dernière itération (celle où $j = r - 1$, puisque `range(l, r)` $= [l, \ldots, r-1]$:

- $i \leq r - 1$.
- Tous les éléments de s dont l'indice est compris entre l et i sont inférieurs ou égaux à x.
- Tous les éléments de s dont l'indice est compris entre $i+1$ et $r-1$ sont strictement supérieurs à x.

Reste à placer $s[r]$ au bon endroit. On l'échange avec $s[i+1]$ et on renvoie $i+1$. Est-ce une bonne idée ? Deux cas se présentent :

- $i < r - 1$. Dans ce cas, $s[i+1] > x$ et c'est donc une bonne idée.
- $i = r - 1$. Dans ce cas, on fait un échange inutile mais cela ne nuit pas.

Testons sans crainte, puisque la théorie nous dit que tout va bien. N'hésitez pas à évaluer la cellule ci-dessous un certain nombre de fois. Regardez les étapes de l'algorithme et comprenez ce qui se passe. Si on comprend `partition`, on comprend le tri rapide.

```
In [25]: N = 10
         s = liste_aleatoire(N)
         print(s)
         CG.reset()
         print('pivot : ', partition(s, 0, N - 1, dbg=True))
         print(s)
         print(CG)
```

```
[4, 6, 2, 0, 9, 3, 7, 8, 1, 5]
i=0, j=0, s=[4, 6, 2, 0, 9, 3, 7, 8, 1, 5]
i=0, j=1, s=[4, 6, 2, 0, 9, 3, 7, 8, 1, 5]
i=1, j=2, s=[4, 2, 6, 0, 9, 3, 7, 8, 1, 5]
i=2, j=3, s=[4, 2, 0, 6, 9, 3, 7, 8, 1, 5]
i=2, j=4, s=[4, 2, 0, 6, 9, 3, 7, 8, 1, 5]
i=3, j=5, s=[4, 2, 0, 3, 9, 6, 7, 8, 1, 5]
i=3, j=6, s=[4, 2, 0, 3, 9, 6, 7, 8, 1, 5]
i=3, j=7, s=[4, 2, 0, 3, 9, 6, 7, 8, 1, 5]
i=4, j=8, s=[4, 2, 0, 3, 1, 6, 7, 8, 9, 5]
pivot :  5
[4, 2, 0, 3, 1, 5, 7, 8, 9, 6]
comparaisons : 9
echanges : 6
```

1.2.3 2.3 Nombre de comparaisons effectuées par `partition`

Proposition : Un appel à `partition(s, l, r)` effectue $n-1$ comparaisons, où $n = r - l + 1$ est la taille de la sous-liste à partitionner.

Démonstration : Il suffit de compter le nombre d'itérations d'une boucle `for`. Le lecteur motivé devrait y arriver :-).

1.2.4 2.4 Nombre d'échanges effectués par `partition`

Si vous avez, comme je l'ai conseillé, exécuté la cellule précédente un certain nombre de fois vous avez dû remarquer que certains échanges (à chaque fois que i est incrémenté) ne modifient pas la liste. C'est certes inefficace mais du coup il est évident de compter le nombre d'échages. Ne prenez surtout pas exemple sur moi, on ne rend pas une fonction inefficace pour pouvoir l'analyser plus facilement !

On fait un échange à chaque fois que i est incrémenté, c'est à dire à chaque fois qu'on rencontre un élément de la liste inférieur ou égal au pivot x. D'où la

Proposition : Un appel à `partition(s, l, r)` effectue k échanges, où k est le nombre d'éléments inférieurs ou égaux au pivot.

Quelles sont les valeurs possibles pour k ? Eh bien, $1, 2, \ldots, n$, où $n = l - r + 1$ est la taille de la sous-liste à partitionner.

Quelle est la valeur "moyenne" de k ? En admettant que les valeurs possibles pour k sont uniformément réparties, c'est

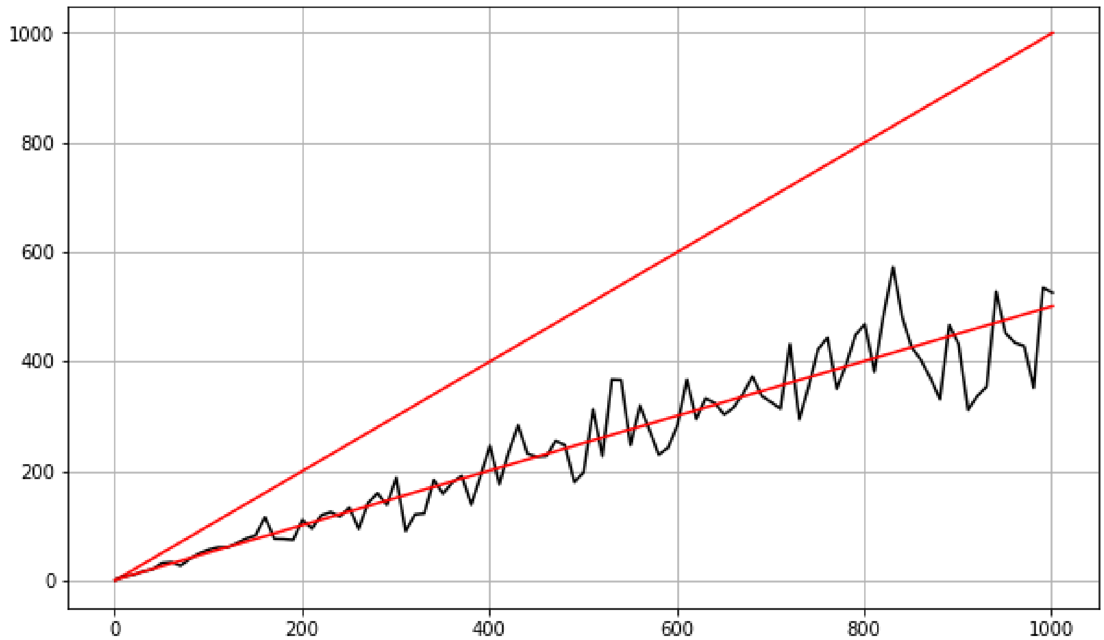
$$\frac{1}{n}\sum_{k=1}^{n} k = \frac{n+1}{2}$$

soit environ la moitié du nombre de comparaisons.

Pour illustrer tout cela, voici en rouge le nombre de comparaisons et ce nombre plus un divisé par deux, comme disaient les mathématiciens avant l'invention des symboles. Et en noir le nombre moyen d'échanges pour des échantillons de listes de diverses tailles.

```
In [26]: tailles, temps_comp, temps_xch = stats(lambda s:partition(s, 0, len(s) - 1), step=10, m
         plt.plot(tailles, temps_comp, 'r')
```

```
plt.plot(tailles, temps_xch, 'k')
ys = [(x + 1) / 2 for x in tailles]
plt.plot(tailles, ys, 'r')
plt.grid()
plt.show()
```



Notre valeur moyenne de k dépasse nos espérances mathématiques.

1.2.5 2.4 Trier

La fonction de tri est maintenant évidente.

- `tri_aux` prend en paramètres une liste s et deux indices l et r. Elle trie la liste s entre les indices l et r inclus.
- `tri_rapide` trie toute la liste s en appelant `tri_aux` entre les indices 0 et `len(s)-1`.

```
In [28]: def tri_aux(s, l, r):
             if l <= r:
                 q = partition(s, l, r)
                 tri_aux(s, l, q - 1)
                 tri_aux(s, q + 1, r)

         def tri_rapide(s):
             tri_aux(s, 0, len(s) - 1)
```

La correction de la fonction `tri_rapide` se fait par récurrence forte sur la longueur des listes. Si `partition` fonctionne, `tri_rapide` aussi.

Exercice : Prouvez la correction de `tri_rapide`. Idée : que voulez vous prouver ?
Testons sur un exemple (pléonasme).

```
In [29]: s = liste_aleatoire(10)
         print(s)
         CG.reset()
         tri_rapide(s)
         print(s)
         print(CG)
```

```
[7, 3, 2, 8, 9, 1, 6, 0, 4, 5]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
comparaisons : 23
echanges : 22
```

Testons sur une **GROS** exemple.

```
In [30]: s = liste_aleatoire(10000)
         #print(s)
         CG.reset()
         tri_rapide(s)
         print('Triée ?', est_triee(s))
         print(CG)
```

```
Triée ? True
comparaisons : 155779
echanges : 80911
```

Avant de faire de jolis dessins, un peu de théorie ...

1.2.6 2.4 La complexité en moyenne du tri rapide

Notons C_n le nombre moyen de comparaisons effectuées par `tri_rapide` sur une liste de taille n. Ne désirant pas entrer dans des détails probabilistes, je me contente de quelques explications pour "établir" une relation de récurrence sur les C_n.

Soit à trier une liste s de taille $n \geq 1$. Un appel à `partition` effectue $n-1$ comparaisons. Puis la fonction `tri_aux` est appelée récursivement sur deux sous-listes s_1 et s_2 de s, l'une de taille k et l'autre de taille $n-1-k$, où $0 \leq k \leq n-1$. Toutes les valeurs de k ont la même probabilité, à savoir $\frac{1}{n}$. Admettons que la fonction `partition` ne perturbe pas les probabilités, c'est à dire que pour une valeur donnée de k toutes les listes s_1 de taille k ont la même probabilité d'apparaître (et de même pour les listes s_2 de taille $n-1-k$). On peut alors (je n'entre pas dans les détails) en déduire :

$$\forall n \geq 1,\ C_n = n - 1 + \frac{1}{n}\sum_{k=0}^{n-1}(C_k + C_{n-1-k})$$

Comme, bien entendu, $C_0 = 0$, nous disposons d'une récurrence clés en main qu'il ne nous reste qu'à résoudre.

Exercice : Que vaut C_1 ? Que vaut C_2 ? Que vaut C_3 ? Bon j'arrête :-).

Soit $n \geq 1$. Commençons par remarquer (changement d'indice $k' = n-1-k$) que $\sum_{k=0}^{n-1} C_{n-1-k} = \sum_{k=0}^{n-1} C_k$.

On a donc $C_n = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} C_k$, d'où

$$nC_n = n(n-1) + 2\sum_{k=0}^{n-1} C_k$$

De là, pour $n \geq 2$, $nC_n - (n-1)C_{n-1} = n(n-1) - (n-1)(n-2) + 2C_{n-1} = 2(n-1) + 2C_{n-1}$, et donc

$$nC_n = 2(n-1) + (n+1)C_{n-1}$$

Remarquons que cette égalité reste valable pour $n = 1$, puisque $C_0 = C_1 = 0$. Divisons par $n(n+1)$. Nous obtenons, pour tout $n \geq 1$,

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + 2\frac{n-1}{n(n+1)}$$

Une simple récurrence montre alors que

$$\frac{C_n}{n+1} = 2\sum_{k=1}^{n} \frac{k-1}{k(k+1)}$$

Poursuivons en décomposant les fractions en éléments simples : $\frac{C_n}{n+1} = 2\sum_{k=1}^{n}(\frac{-1}{k} + \frac{2}{k+1}) = 2H_{n+1} - 4 + \frac{2}{n+1}$ où $H_n = \sum_{k=1}^{n}\frac{1}{k}$ est le $n$ième nombre harmonique. Rappelons le résultat bien connu (ou pas)

$$H_n = \ln n + \gamma + O(\frac{1}{n})$$

où $\gamma \simeq 0.577216$ est la constante d'Euler. On en déduit la

Proposition :

- Pour tout $n \geq 0$, $C_n = 2(n+1)H_{n+1} - 4(n+1) + 2$.
- $C_n = 2(n+1)\ln(n+1) + (2\gamma - 4)(n+1) + O(1)$.

Exercice : Euh, ça marche aussi pour $n = 0$? Vérifiez.

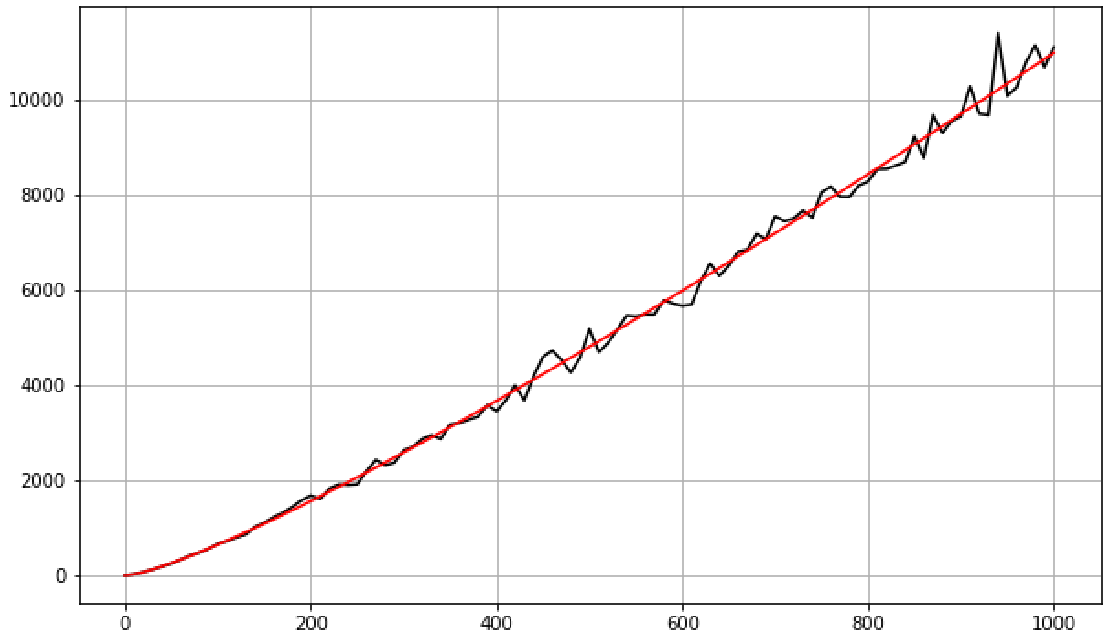
Comme annoncé dans l'introduction, la complexité moyenne du tri rapide est donc $\Theta(n \log n)$, où n est la longueur de la liste à trier.

```
In [31]: def complexite_theo(n):
             ga = 0.577215664
             return 2 * (n + 1) * math.log(n + 1) + (2 * ga - 4) * (n + 1)
```

1.2.7 2.5 Statistiques

Réconcilions théorie et pratique sur un même graphique. En rouge, la complexité moyenne théorique. En noir, le nombre de comparaisons constatées sur des tests.

```
In [32]: tailles, temps_comp, temps_xch = stats(tri_rapide, step=10, min=0, max=1000, sample=3)
         plt.plot(tailles, temps_comp, 'k')
         #plt.plot(tailles, temps_xch, 'b')
         ys = [complexite_theo(k) for k in tailles]
         #zs = [(k+1)*(math.log(k+1)+0.577215+1/(2*(k+1))-1) for k in tailles]
         plt.plot(tailles, ys, 'r')
         #plt.plot(tailles, zs, 'r')
         plt.grid()
         plt.show()
```



Ça fait plaisir à voir.

1.2.8 2.6 L'écart-type du tri rapide

Savoir que le tri rapide est rapide en moyenne c'est bien. Mais cela ne prouve pas qu'il se comporte bien tout le temps. Une mesure intéressante d'une variable aléatoire, outre son espérance, est son écart-type. Un petit écart-type indique des valeurs de la variable aléatoire "resserrées" autour de sa moyenne. Alors, quel est l'écart-type du quicksort ?

Proposition : L'écart-type du tri rapide pour des listes de taille n est

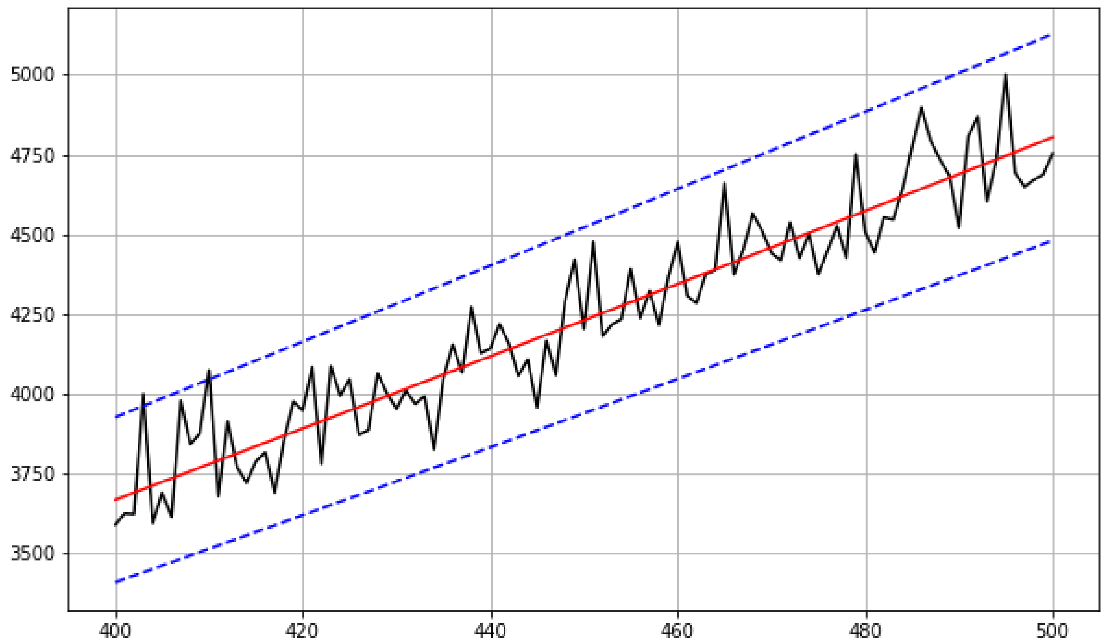
$$\sigma_n \sim n\sqrt{7 - \frac{2}{3}\pi^2} \simeq 0.648n$$

Démonstration : Bien au delà des ambitions de ce notebook. Contentons-nous de remarquer que $\sigma_n = o(C_n)$: l'écart-type est **négligeable** devant la moyenne ! Retraçons le graphique ci-dessus en y rajoutant deux courbes en pointillés, correspondant à moyenne $\pm$ écart-type.

```
In [33]: math.sqrt(7 - 2 * math.pi ** 2 / 3)
```

Out[33]: 0.6482775120325356

```
In [34]: tailles, temps_comp, temps_xch = stats(tri_rapide, step=1, min=400, max=500, sample=5)
         plt.plot(tailles, temps_comp, 'k')
         #plt.plot(tailles, temps_xch, 'b')
         ys = [complexite_theo(k) for k in tailles]
         zs1 = [complexite_theo(k) - 0.648 * k for k in tailles]
         zs2 = [complexite_theo(k) + 0.648 * k for k in tailles]
         plt.plot(tailles, ys, 'r')
         plt.plot(tailles, zs1, 'b--')
         plt.plot(tailles, zs2, 'b--')
         plt.grid()
         plt.show()
```



C'est presque trop beau pour être vrai. L'introduction du notebook annonçait un pire cas du tri rapide en $\Theta(n^2)$. Comment est-il possible que nous n'ayons pas sur notre graphique un ou deux points complètement gagas, correspondant à des listes sur lesquelles le comportement du tri est très mauvais ? Eh bien de telles listes sont TRÈS TRÈS rares. Si vous vous ennuyez, évaluez encore et encore la cellule ci-dessus et appelez-moi dès que vous verrez un point bizarre :-).

1.3 3 Compléments

1.3.1 3.1 La complexité en pire cas du tri rapide

Oui, le tri rapide peut se comporter très mal. Et le problème c'est que cela arrive dans la vraie vie. Nous allons étudier la complexité en pire cas du tri rapide, et montrer que cette complexité est atteinte pour les listes déjà triées. C'est très ennuyeux parce qu'il s'avère que dans les applications réelles on a souvent à trier des listes **presques** triées. D'où un comportement catastrophique du quicksort. Nous verrons dans la section suivante commment remédier à cela au moyen d'une **randomisation** (anglicisme navrant).

Notons $C(s)$ le nombre de comparaisons d'éléments de listes effectuées par un appel à `tri_rapide(s, 0, len(s) - 1)`. Notons également $C_n = \max\{C(s), |s| = n\}$ le nombre maximal de comparaisons effectuées par `tri_rapide` sur les listes de longueur n.

On a $C_0 = 0$. Soit $n \geq 1$. Soit s une liste de longueur n. On a

$$C(s) = n - 1 + C(s_1) + C(s_2)$$

où s_1 et s_2 sont les listes sur lesquelles la fonction de tri se rappelle récursivement. Soit $k = |s_1|$. On a $0 \leq k \leq n-1$ et donc $C(s_1) \leq C_k$. Bien entendu, $|s_2| = n - 1 - k$, et donc $C(s_2) \leq C_{n-1-k}$. Ainsi,

$$C(s) \leq n - 1 + C_k + C_{n-1-k}$$

où le $n-1$ est le nombre de comparaisons effectuées par la fonction de partition. De là, $C(s) \leq n - 1 + \max\{C_k + C_{n-1-k}, 0 \leq k \leq n-1\}$. Ceci étant vrai pour toute liste de longueur n, on en déduit

$$\forall n \geq 1, C_n \leq n - 1 + \max\{C_k + C_{n-1-k}, 0 \leq k \leq n-1\}$$

Proposition : Pour tout $n \in \mathbb{N}$, $C_n \leq \frac{1}{2}n(n-1)$.

Démonstration : On fait une récurrence forte sur n. Pour $n = 0$ c'est clair puisque $C_0 = 0$. Soit $n \geq 1$. Supposons l'inégalité vraie pour tous les entiers $0 \leq k < n$. On a alors

$$C_n \leq n - 1 + \frac{1}{2}\max\{k(k-1) + (n-1-k)(n-2-k), 0 \leq k \leq n-1\}$$

Soit $f : [0, n-1] \to \mathbb{R}$ définie par $f(x) = x(x+1) + (n-1-x)(n-2-x)$. On a $f'(x) = 2(2x - n + 1)$. La fonction f décroit de 0 jusqu'à $\frac{n-1}{2}$ puis croît jusqu'à $n-1$. De plus, $f(0) = f(n-1) = (n-2)(n-1)$. Ainsi, $C_n \leq n + \frac{1}{2}(n-2)(n-1)) = \frac{1}{2}n(n-1)$.

Proposition : Soit s une liste de taille n triée dans l'ordre croissant. On a $C(s) = \frac{1}{2}n(n-1)$.

Démonstration : On fait une récurrence sur n. Si $n = 0$ c'est évident. Soit donc $n \geq 0$. Supposons le résultat vrai pour les listes de taille n triées dans l'ordre croissant et donnons nous une liste s de taille $n+1$ triée dans l'ordre croissant. `tri_rapide(s)` apelle `tri_aux(s, 0, n)` qui appelle `partition(s, 0, n)`. La fonction `partition` effectue alors n échanges contre-productifs d'éléments de s avec eux-mêmes (on en a déjà parlé plus haut), c'est à dire que s n'est pas modifiée. Puis `partition` renvoie n. Si nous notons s' la liste s privée de son dernier élément, nous avons donc $C(s) = n + C(s')$. Par l'hypothèse de récurrence, $C(s) = n + \frac{1}{2}n(n-1) = \frac{1}{2}n(n+1)$.

Corollaire : $C_n \geq \frac{1}{2}n(n-1)$.

Démonstration : Il existe une liste s de taille n telle que $C(s) = \frac{1}{2}n(n-1)$.

Proposition : $C_n = \frac{1}{2}n(n-1) = \Theta(n^2)$.

Démonstration : Immédiat par ce qui précède.

1.3.2 3.2 Tri randomisé

Voici une nouvelle version du tri rapide. Dans la fonction de partition, on échange $s[r]$ et $s[k]$, où k est un entier aléatoire entre l et r. Je ne ferai aucune théorie sur le sujet mais on conçoit que de cette façon on règle le problème des listes triées. En fait, avec cet algorithme, sauf malchance inouie, nous trions toute liste avec une complexité en $\Theta(n \log n)$.

Pour être tout à fait exacts, un individu infiniment malveillant connaissant l'état de mon générateur de nombres aléatoires à tout instant **pourrait** fabriquer une liste qui serait triée avec un nombre de comparaisons de l'ordre de n^2. Si ça l'amuse :-) ...

```
In [35]: def partition_rand(s, l, r):
             i = l - 1
             echanger(s, random.randint(l, r), r)
             for j in range(l, r):
                 if inferieur(s, j, r):
                     i = i + 1
                     echanger(s, i, j)
             echanger(s, i + 1, r)
             return i + 1
```

```
In [36]: def tri_aux_rand(s, l, r):
             if l <= r:
                 q = partition_rand(s, l, r)
                 tri_aux_rand(s, l, q - 1)
                 tri_aux_rand(s, q + 1, r)

         def tri_rapide_rand(s):
             tri_aux_rand(s, 0, len(s) - 1)
```

Histoire de comparer les deux versions du tri rapide, trions la liste $[0, \ldots, 999]$.

```
In [37]: s = list(range(2000))
         CG.reset()
         tri_rapide(s)
         print(CG)
```

```
comparaisons : 1999000
echanges : 2001000
```

Avec le quicksort classique, nous avons environ 2 millions de comparaisons.

```
In [38]: s = list(range(2000))
         CG.reset()
         tri_rapide_rand(s)
         print(CG)
```

```
comparaisons : 26119
echanges : 17668
```

Avec le quicksort randomisé, nous avons environ 25000 comparaisons (eh oui, cela change à chaque exécution, c'est le propre des algorithmes randomisés !).

Clairement, l'ajout d'une unique ligne de code a rendu notre algorithme plus efficace (sauf individu infiniment malveillant dans les parages). Peut-on faire encore mieux ? Oui, il y a plein de possibilités. Je vais en explorer une, sans entrer dans les détails théoriques.

1.3.3 3.3 Médiane de trois

Une raison claire de l'inefficacité du quicksort sur certaines listes est que `partition` les coupe en des moitiés très inégales. Alors comment faire pour remédier à cela ? Une première idée serait de rechercher la médiane de la liste à partitionner et de choisir cette médiane comme pivot de la partition. Le problème est qu'il faudrait calculer la médiane. On pourrait par exemple trier la liste et prendre l'élément du milieu ? Hi, hi :-). Bon, il existe des algorithmes de calcul de médiane plus sophistiqués, mais leur coût serait inaccpetable parce que que nous voulons faire **baisser** le coût du quicksort.

Un autre idée est de prendre trois éléments au hasard dans la liste et de choisir comme pivot la médiane de ces trois éléments (c'est à dire celui qui n'est le plus petit, ni le plus grand). Risqué, mais jouable. L'espoir que nous avons est évidemment que la liste sera coupée en deux moitiés "un peu plus égales" que dans la fonction de partition classique. La fonction `partition3` fait le travail. Elle choisit trois indices au hasard entre l et r, puis calcule, par un appel à `mediane3` l'indice k de la médiane. Elle place ensuite $s[k]$ à l'indice r puis fait ensuite comme la fonction `partition` standard.

```
In [39]: def partition3(s, l, r):
             a = random.randint(l, r)
             b = random.randint(l, r)
             c = random.randint(l, r)
             k = mediane3(s, a, b, c)
             i = l - 1
             echanger(s, k, r)
             for j in range(l, r):
                 if inferieur(s, j, r):
                     i = i + 1
                     echanger(s, i, j)
             echanger(s, i + 1, r)
             return i + 1
```

Reste à écrire `mediane3`. En fait `mediane3` trie physiquement dans l'ordre croissant les éléments $s[a], s[b], s[c]$. Puis elle renvoie b puisque $s[b]$ contient la médiane cherchée.

Remarque : Ce calcul nous coûte 3 comparaisons et au plus 3 échanges.

```
In [40]: def mediane3(s, a, b, c):
             if inferieur(s, c, a): echanger(s, a, c)
             if inferieur(s, b, a): echanger(s, a, b)
             if inferieur(s, c, b): echanger(s, b, c)
             return b
```

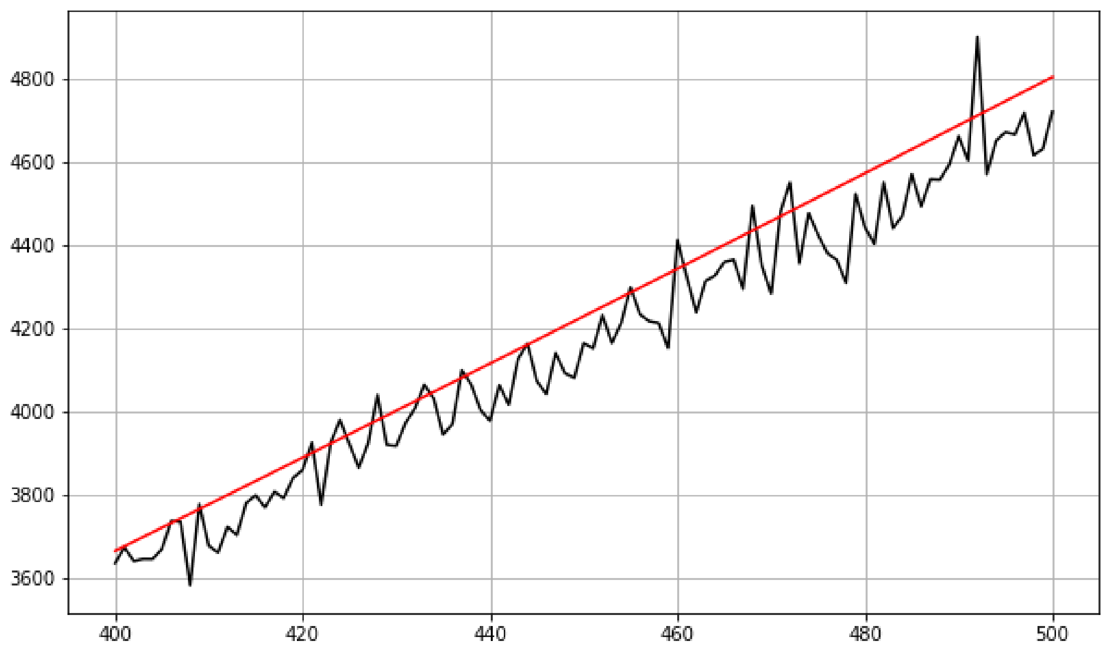
```
In [41]: def tri_aux3(s, l, r):
             if l + 3 <= r:
                 q = partition3(s, l, r)
                 tri_aux3(s, l, q - 1)
                 tri_aux3(s, q + 1, r)
             elif l <= r:
                 q = partition(s, l, r)
                 tri_aux3(s, l, q - 1)
```

```
            tri_aux3(s, q + 1, r)

    def tri_rapide3(s):
        tri_aux3(s, 0, len(s) - 1)
```

Voici les résultat pratiques obtenus. En rouge, la courbe théorique pour le tri rapide **standard**. On constate effectivement que la courbe noire est plutôt **sous** la courbe rouge. Il y a donc a priori un gain de performance dans notre nouvel algorithme.

```
In [42]: tailles, temps_comp, temps_xch = stats(tri_rapide3, step=1, min=400, max=500, sample=5)
         plt.plot(tailles, temps_comp, 'k')
         #plt.plot(tailles, temps_xch, 'b')
         ys = [complexite_theo(k) for k in tailles]
         plt.plot(tailles, ys, 'r')
         plt.grid()
         plt.show()
```



Le gain n'est pas énorme mais il existe. On peut démontrer que la complexité moyenne du tri rapide "médiane de 3" est

$$C'_n \sim \frac{12}{7} n \ln n$$

alors que pour le tri rapide classique cette complexité était

$$C_n \sim 2n \ln n$$

Comme $\frac{12}{7} < 2$ il y a bien un gain de performance en nombres de comparaisons !

Exercice : Affichez sur un même graphique les courbes pour le tri rapide standard et pour le tri rapide médiane de 3. Vous constaterez l'existence du léger mieux.

1.4 4 Et ensuite ?

Il y aurait encore beaucoup de choses à dire sur le tri rapide. J'ai parlé presque exclusivement de comparaisons mais il serait aussi très intéressant de s'intéresser aux **échanges** effectués par l'algorithme. En l'état, la première fonction de tri rapide que nous avons écrite (celle qui s'appelle `tri_rapide`) effectue un nombre moyen d'échanges égal à

$$E_n = (n+1)(H_{n+1} - 1)$$

où $H_n = \sum_{k=1}^{n} \frac{1}{k}$. Vous avez juste trois lignes à décommenter dans une cellule au paragraphe 2.5 pour voir le phénomène. Et pour comprendre qu'il y a des choses à étudier de ce côté.

En particulier, améliorer la fonction de partition pour abaisser le nombre d'échanges serait une bonne idée. La nôtre effectue en moyenne environ $\frac{n}{2}$ échanges. Avec beaucoup de soin on peut descendre à $\frac{n}{6}$ ...

Une étude de la **complexité en espace** du tri rapide serait éussi à envisager. Tel quel, notre code est **récursif non terminal**. Chaque appel récursif utilise de l'espace et on peut montrer qu'en meilleur cas l'espace utilisé (en dehors de la liste s à trier, cela va de soi) est $O(\log n)$ où n est la taille de la liste. En pire cas, l'espace utilisé monte à $O(n)$. En dérécursifiant le code, ou en le rendant récursif terminal, on peut descendre à $O(\log n)$ en pire cas. Cela permet d'envisager l'utilisation du quicksort sur des machines ayant très peu de mémoire.