Quickselect

Marc Lorenzi

22 décembre 2018

In [1]:
```
import matplotlib.pyplot as plt
import math
import random
import utils
from utils import CG, inferieur, echanger, liste_aleatoire
```

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

Avertissement : Avant de lire ce notebook, regardez celui sur le tri rapide. Un certain nombre d'idées étudiées dans le notebook sur le tri rapide seront reprises ici, sans être détaillées.

0 Listes aléatoires, compteurs, comparaisons

Nous avons mis en place dans le notebook "tri rapide" un certain nombre de choses qui seront reprises ici. Le module `utils` contient

- Une fonction `liste_aleatoire` prenant en paramètre un entier n et renvoyant une permutation aléatoires des entiers entre 0 et $n-1$.
- Un "compteur global" `CG` qui compte les comparaisons d'objets. `CG.reset()` remet le compteur à zéro. Il est impératif, à chaque fois que nous aurons à compter une comparaison entre deux objets x et y, d'appeler `inferieur(x, y)` , où la fonction `inferieur` est définie dans le module `utils` . Cette fonction incrémente `CG` de 1 et renvoie `True` si $x \leq y$ et `False` sinon.
- Une fonction `stats` dont nous reparlerons plus loin.
- Une fonction `tri_rapide` prenant en paramètre une liste s et triant sur place la liste s avec un nombre moyen de comparaisons en $O(n \ln n)$.

1 Introduction

1.1 Le problème de la sélection

Soient s une liste de longueur $n \geq 1$ et $0 \leq k < n$. Quel est le k-ième plus petit éléments de s (le zéroième étant le minimum de s) ? Si nous notons $\min(s, k)$ l'élément en question, le problème que nous allons chercher à résoudre dans ce notebook est :

Problème : Calculer $\min(s, k)$ pour toute liste s et pour tout k.

Remarque : Un appel à `min(liste_aleatoire(n), k)` renvoie k. Ce n'est pas la valeur retournée qui nous intéressera mais le nombre de comparaisons effectuées !

Pour certaines valeurs de k c'est évidemment très facile. Par exemple, si $k = 0$, $\min(s, 0) = \min(s)$ est tout simplement le plus petit élément de s. La fonction `plus_petit` fait le travail sur une liste non vide.

In [3]:
```
def plus_petit(s):
    m = s[0]
    for x in s:
        if inferieur(x, m): m = x
    return m
```

In [4]:
```
s = liste_aleatoire(1000)
CG.reset()
print(plus_petit(s))
print(CG)
```

```
0
comparaisons : 1000
```

La fonction `plus_petit` effectue n comparaisons d'éléments sur une liste s de taille n. Nous avons donc un algorithme qui trouve le plus petit élément en temps linéaire. Tentons quelque chose d'un peu plus difficile, mais à peine. Comment trouver le **second** plus petit élément de s, c'est à dire $\min(s, 1)$?

La fonction `second_plus_petit` ci-dessous calcule $min(s, 1)$ pour une liste s ayant au moins deux éléments.

In [5]:
```
def second_plus_petit(s):
    m = s[0]
    p = s[1]
    if not inferieur(m, p): m, p = p, m
    for x in s:
        if inferieur(x, m): m = x
        elif inferieur(x, p): p = x
    return p
```

In [6]:
```
s = liste_aleatoire(1000)
#s = list(range(999, -1, -1))
#s = list(range(1000))
CG.reset()
print(second_plus_petit(s))
print(CG)
```

```
1
comparaisons : 1996
```

Cette fois-ci encore, nous voici en possession d'un algorithme qui effectue un nombre de comparaisons linéaire en n, la longueur de s. Précisément, le nombre $C(s)$ de comparaisons **d'éléments de** s effectuées par la fonction `second_plus_petit` sur une liste de taille n vérifie

$$n \leq C(s) \leq 2n$$

Exercice : Exécuter la cellule ci-dessus en remplaçant la liste aléatoire par ;

- Une liste triée dans l'ordre croissant
- Une liste triée dans l'ordre décroissant

Expliquer le nombre de comparaisons obtenu.

Aucun problème pour trouver le troisième, le quatrième, ou le cinquième plus petit élément de s, mis à part le fait que le nombre de comparaisons devient plus important (tout en restant linéaire en la longueur de la liste). Peut-on écrire une fonction générale résolvant le problème bâtie sur le modèle des deux fonctions ci-dessus ? Oui, sans doute, mais le nombre de comparaisons nécessaires à la recherche du k-ième plus petit élément de s serait proportionnel à $k+1$.

Et alors, me direz-vous ? Ce nombre sera toujours proportionnel à n ? Eh non ! Qu'arrive-t-il si k dépend de n ?

Définition : Soit s une liste de longueur $n \geq 1$.

- Si n est impair, la **médiane** de s est $\mu(s) = \min(s, \frac{n-1}{2})$.
- Si n est pair, la **médiane** de s est $\mu(s) = \min(s, \frac{n}{2})$.

Remarque : Si n est pair, la *vraie* définition de la médiane est $\mu(s) = \frac{1}{2}(\min(s, \frac{n}{2} - 1) + \min(s, \frac{n}{2}))$. Mais je m'en tiendrai à ma propre définition pour simplifier les discussions et les calculs.

Notre très hypothétique algorithme naïf aurait sans doute une complexité quadratique pour le calcul de la médiane, ce qui, avouons-le, serait très décevant. Il va nous falloir trouver autre chose.

1.2 Sélection un peu moins naïve

Remarquons que si une liste s est triée, alors notre problème est évident puisque $\min(s, k) = s[k]$. On en déduit un algorithme de sélection :

- Trier s.
- Renvoyer $s[k]$.

Le module `utils` contient une fonction `quicksort` qui trie une liste avec un nombre moyen de comparaisons en $O(n \ln n)$.

In [7]:
```
def select_sort(s, k):
    utils.quicksort(s)
    return s[k]
```

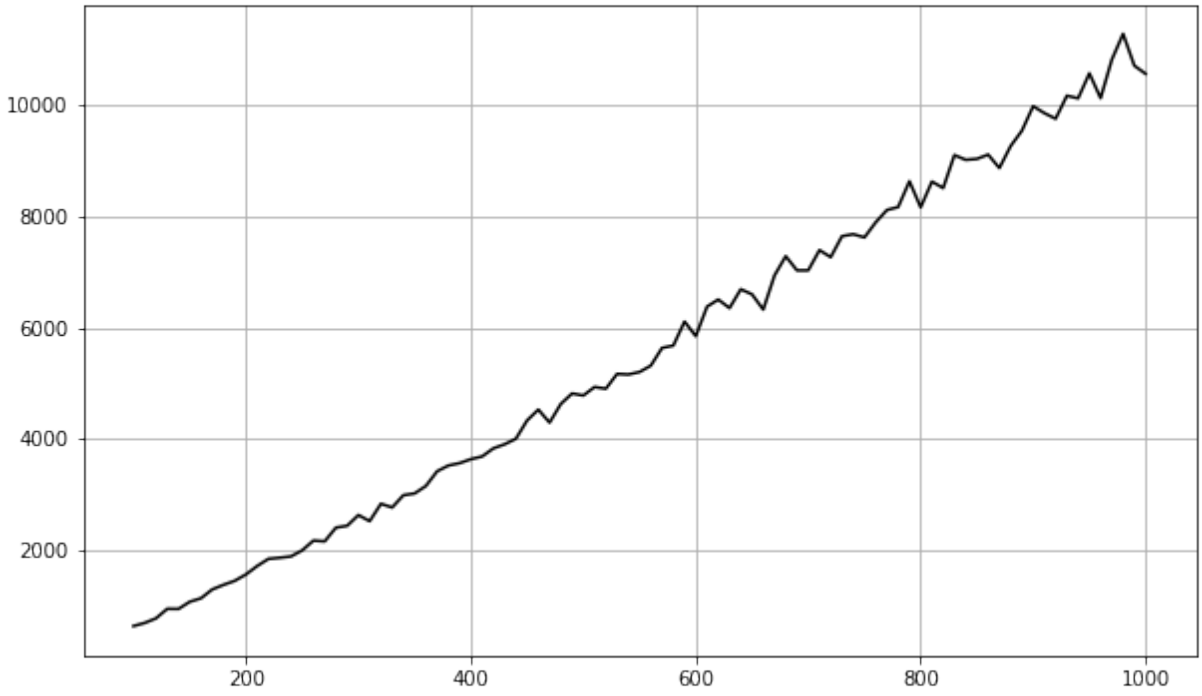
In [8]:
```
s = liste_aleatoire(1000)
CG.reset()
print(select_sort(s, 500))
print(CG)
```

```
500
comparaisons : 10525
```

Affichons quelques statistiques ! La fonction `stats` du module `utils` effectue le travail suivant : pour des tailles de listes comprises entre `min` et `max` par pas de `step` , elle appelle `sample` fois un algorithme (`mediane` dans la cellule ci-dessous) sur des listes aléatoires. Elle calcule ensuite le nombre moyen de comparaisons effectuées par l'algorithme pour les listes d'une taille données. La fonction renvoie un couple `(tailles, tcomp)` où

- `tailles` est la liste des tailles de listes échantillonnées.
- `tcomp` est la liste des nombres moyens de comparaisons correspondants.

In [9]:
```
tailles, tcomp = utils.stats(lambda s: select_sort(s, len(s) // 2), st
plt.plot(tailles, tcomp, 'k')
plt.grid()
plt.show()
```



Bon, nous sommes en mesure de calculer $\min(s, k)$ en $O(n \ln n)$ comparaisons, ceci pour toute liste de longueur n et pour tout entier $0 \leq k < n$. C'est mieux que rien. Mais cette complexité est clairement mauvaise pour $k = 0$ ou 1. Nous avions une complexité en temps linéaire avec notre algorithme très naïf. Ensuite, on a vraiment l'impression que trier **toute** la liste juste pour en extraire **une** valeur est une dépense d'énergie absurde. Nous allons faire mieux en étudiant un algorithme qui s'exécute en nombre moyen de comparaisons $O(n)$ pour toute liste de taille n et pour tout k tel que $0 \leq k < n$. Il s'agit de l'algorithme `quickselect` .

2 L'algorithme de sélection rapide

2.1 L'idée

Quickselect s'inspire du tri rapide. Soit s une liste. Soient l (left) et r (right) deux indices. Soit $0 \leq k < r - l$. Nous désirons trouver le k-ième plus petit élément de $s[l : r]$.

1- On partitionne la liste s entre les indices l et r (voir le notebook "tri rapide"). Soit j la valeur renvoyée par la fonction de partition.

2a- Si $k < j - l$, alors l'élément que nous cherchons est le k-ième de la sous-liste $s[l : j - 1]$.

2b- Si $k = j - l$, alors nous avons trouvé : l'élément que nous cherchons est $s[j]$.

2c- Si $k > j - l$, alors l'élément que nous cherchons est le $k - j + l - 1$-ième de la sous-liste $s[j + 1 : r]$.

Contrairement à l'algorithme du tri rapide qui effectue **deux** appels récursifs, la fonction de sélection en effectue **un seul**.

2.2 Partitionner

Je ne ré-explique pas la fonction de partition, déjà étudiée dans le notebook sur le tri rapide. Remarquez la randomisation de cette fonction, effectuée aux lignes 2 et 3 de celle-ci.

In [10]:
```
def partition(s, l, r, dbg=False):
    k = random.randint(l, r)
    echanger(s, k, r)
    i = l - 1
    for j in range(l, r):
        if inferieur(s[j], s[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
```

In [11]:
```
s = liste_aleatoire(10)
CG.reset()
pivot = partition(s, 0, 9, True)
print('Pivot : ', pivot)
print(CG)
```

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

2.3 Quickselect

La fonction de sélection est maintenant évidente.

In [12]:
```
def select_aux(s, l, r, k):
    j = partition(s, l, r)
    if k < j - l:    return select_aux(s, l, j - 1, k)
    elif k == j - l: return s[j]
    else:            return select_aux(s, j + 1, r, k - j + l - 1)

def quickselect(s, k):
    return select_aux(s, 0, len(s) - 1, k)
```

On en déduit évidemment une fonction qui renvoie la médiane d'une liste.

In [13]:
```
def mediane(s):
    return quickselect(s, len(s) // 2)
```

Histoire de comparer les performances de `quickselect` et `select_sort`, cherchons la médiane d'une liste de 100000 éléments.

In [14]:
```
s = liste_aleatoire(100000)
CG.reset()
print(mediane(s))
print(CG)
```

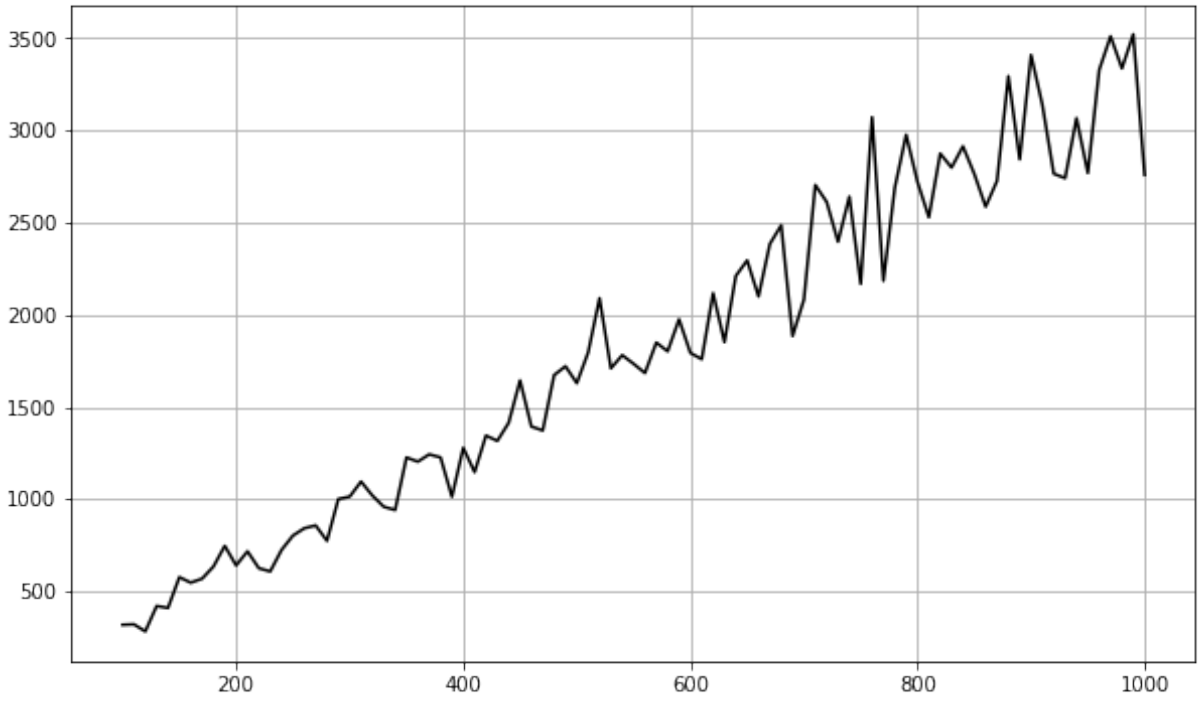
```
50000
comparaisons : 209080
```

In [15]:
```
s = liste_aleatoire(100000)
CG.reset()
print(select_sort(s, 50000))
print(CG)
```

```
50000
comparaisons : 2024349
```

Affichons quelques statistiques sur `quickselect` pour des calculs de médianes.

In [16]:
```
tailles, tcomp = utils.stats(mediane, step=10, min=100, max=1000, samp
plt.plot(tailles, tcomp, 'k')
plt.grid()
plt.show()
```



Exercice : Mettez `sample` à 1 et exécutez la cellule ci-dessus. Si vous avez lu (ce dont je ne doute point) le notebook sur le tri rapide, ceci devrait vous suggérer que l'écart-type de quickselect est "moins" négligeable devant sa moyenne que ce que nous avions observé pour quicksort.

3 La théorie

3.1 Nombre moyen de comparaisons

L'analyse précise du comportement en moyenne de quickselect est difficile. Je me contenterai pour l'instant de donner le résultat. Un peu plus loin dans le notebook nous prouverons que cette complexité moyenne est linéaire.

Pour $n \geq 1$ et $0 \leq k < n$, notons $C_{n,k}$ le nombre moyen de comparaisons effectuées par quickselect pour calculer $\min(s, k)$ sur les listes s de taille n. L'expression de $C_{n,k}$ fait intervenir les nombres harmoniques H_n où

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

On peut montrer que $H_n = \ln n + \gamma + \frac{1}{2n} + -\frac{1}{12n^2} + o(\frac{1}{n^2})$ où $\gamma \simeq 0.577216$ est la *constante d'Euler*. Ci-dessous deux fonctions renvoyant H_n et son approximation.

In [17]:
```
def harmo(n):
    s = 0
    for k in range(1, n + 1):
        s = s + 1 / k
    return s
```

In [18]:
```
def approx_harmo(n):
    if n == 0: return 0
    else:
        ga = 0.577215664
        return math.log(n) + ga + 1 / (2 * n) - 1 / (12 * n ** 2)
```

Remarquez que l'approximation donnée pour H_n est vraiment bonne, même pour de petites valeurs de n.

In [19]:
```
for k in range(1, 21):
    print('%3d%15.10f%15.10f' % (k, harmo(k), approx_harmo(k)))
```

```
  1   1.0000000000   0.9938823307
  2   1.5000000000   1.4995295112
  3   1.8333333333   1.8332353601
  4   2.0833333333   2.0833016918
  5   2.2833333333   2.2833202431
  6   2.4500000000   2.4499936517
  7   2.5928571429   2.5928537042
  8   2.7178571429   2.7178551223
  9   2.8289682540   2.8289669903
 10   2.9289682540   2.9289674237
 11   3.0198773449   3.0198767770
 12   3.1032106782   3.1032102768
 13   3.1801337551   3.1801334633
 14   3.2515623266   3.2515621093
 15   3.3182289932   3.3182288281
 16   3.3807289932   3.3807288654
 17   3.4395525226   3.4395524221
 18   3.4951080782   3.4951079980
 19   3.5477396571   3.5477395924
 20   3.5977396571   3.5977396042
```

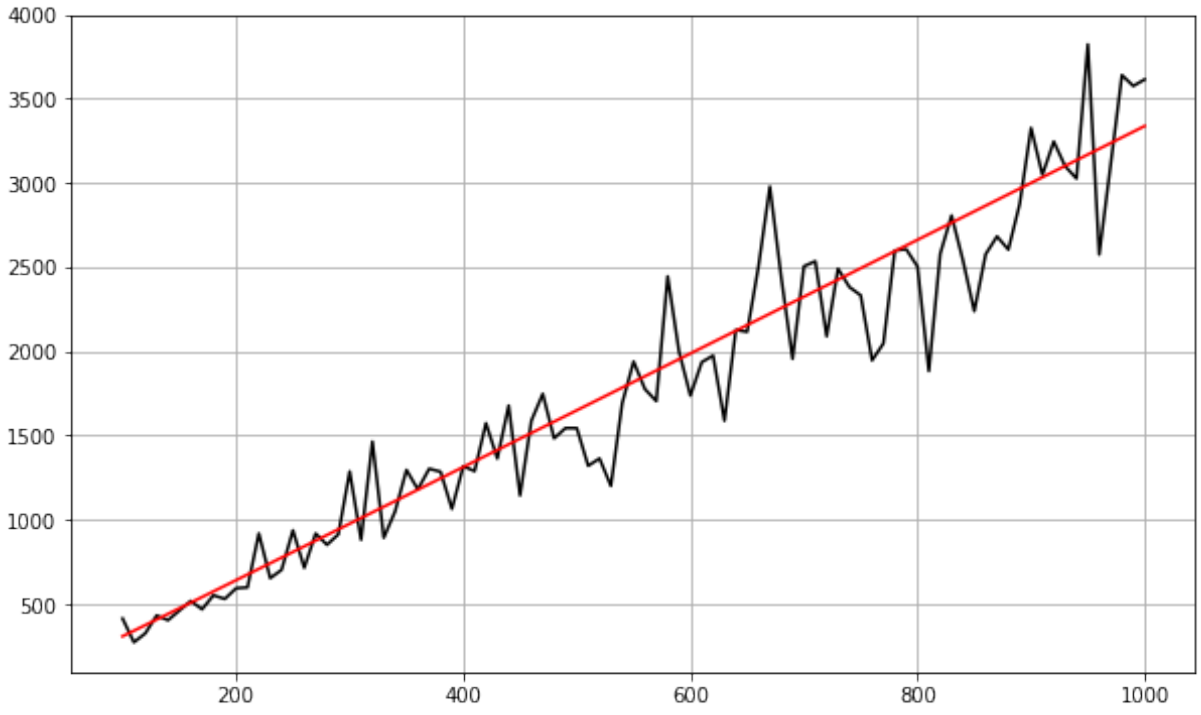
Proposition : Soit $n \geq 1$. Soit $0 \leq k < n$. On a

$$C_{n,k} = 2((n+1)H_n - (n+3-k)H_{n+1-k} - (k+2)H_k + n + 3)$$

Démonstration : Hors de notre portée. Dans "The Art of Computer Programming", Donald Knuth classe cette démonstration de difficulté M40, c'est à dire "projet mathématique à long terme". Mais comme nous ne sommes pas obligés de croire sans voir, faisons une petite simulation :-).

In [20]:
```
def theo(n, k):
    return 2 * ((n + 1) * harmo(n) - (n + 3 - k) * harmo(n + 1 - k) -
```

In [21]:
```
tailles, tcomp = utils.stats(lambda s:quickselect(s, len(s) // 2), ste
plt.plot(tailles, tcomp, 'k')
ys = [theo(k, k // 2) for k in tailles]
plt.plot(tailles, ys, 'r')
plt.grid()
plt.show()
```



La formule donnant $C(n,k)$ est loin d'être limpide. Il n'est même pas évident que $C(n,k) = O(n)$. Alors faisons quelques petits calculs relatifs à la médiane. Que vaut $C_{2n,n}$? Eh bien,

$$C_{2n,n} = 2((2n+1)H_{2n} - (n+3)H_{n+1} - (n+2)H_n + 2n + 3)$$

ou encore

$$C_{2n,n} = 2((2n+1)H_{2n} - (2n+5)H_n + \frac{2n^2+4n}{n+1})$$

Arrangeons tout cela. Tout d'abord,

$$(2n+1)H_{2n} = (2n+1)(\ln n + \ln 2 + \gamma + O(\frac{1}{n}) = (2n+1)\ln n + (2n+1)(\ln 2 + \gamma) + C$$

De même,

$$(2n+5)H_n = (2n+5)\ln n + (2n+5)\gamma + O(1)$$

Enfin,

$$\frac{2n^2+4n}{n+1} = 2n\frac{n+2}{n+1} = 2n + O(1)$$

En combinant le tout, on obtient

$$C_{2n,n} = 4(1+\ln 2)n - 8\ln n + O(1)$$

Un calcul analogue nous donnerait quelque chose de très analogue pour $C_{2n+1,n}$. On en déduit une expression de la complexité moyenne M_n du calcul de la médiane par quickselect :

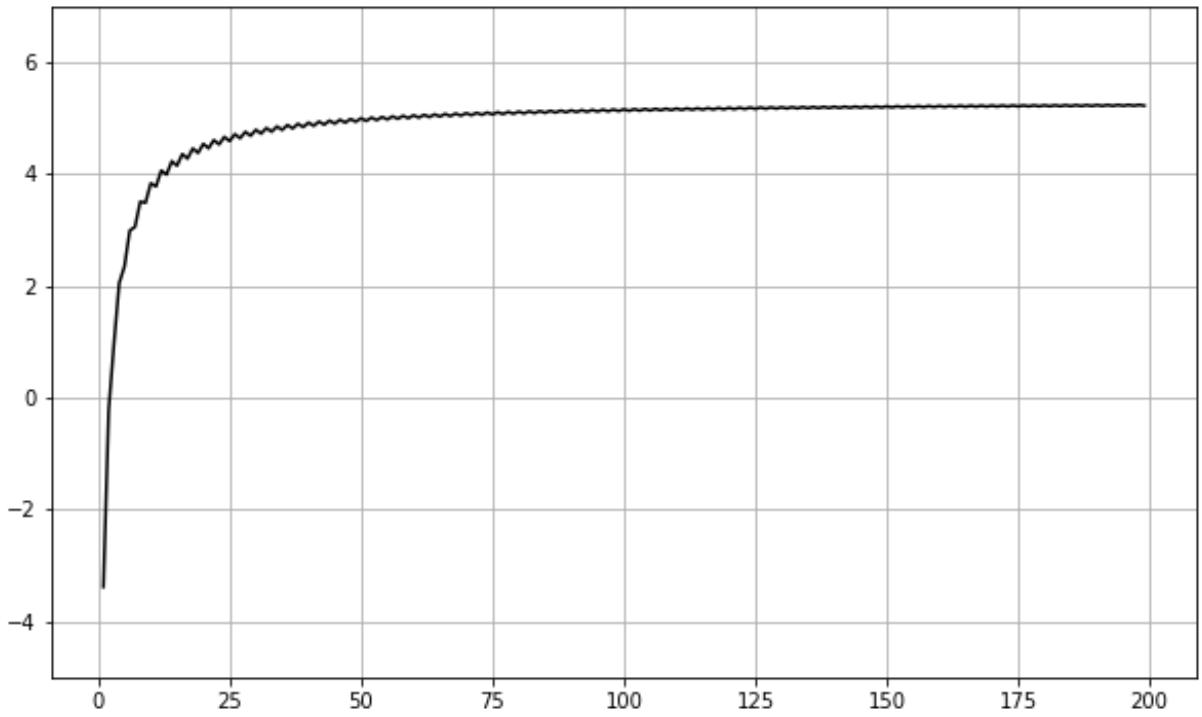
Proposition : $M_n = 2(1+\ln 2)n - 8\ln n + O(1)$.

In [22]:
```
def theo2(x):
    return 2 * (1 + math.log(2)) * x - 8 * math.log(x)
```

In [23]:
```
theo(10000, 5000) - theo2(10000)
```

Out[23]:
```
5.312096678004309
```

In [24]:
```
xs = list(range(1, 200))
ys = [theo(k, k//2) - theo2(k) for k in xs]
plt.plot(xs, ys, 'k')
plt.ylim(-5, 7)
plt.grid()
plt.show()
```



3.2 Le pire cas de quickselect

Si vous avez **vraiment** lu le notebook sur le tri rapide, vous vous doutez que le pire cas de `quickselect` est en $O(n^2)$. C'est effectivement ce qui se passe. Comme je n'ai pas très envie de reprendre des choses déjà dites dans le notebook sur le tri rapide, posons-nous une question différente : est-il fréquent qu'à chaque appel à `partition` le pivot soit le plus petit ou le plus grand élément de la liste ? Pour $i \geq 1$, notons $\mathcal{E}_i$ l'événement : "pour une liste aléatoire de longueur i, la fonction `partition` choisit comme pivot le plus grand ou le plus petit élément de la liste". L'événement qui nous intéresse est donc

$$\mathcal{E} = \bigcap_{i=1}^{n} \mathcal{E}_i$$

La question est : que vaut $P(\mathcal{E})$? Les événements $\mathcal{E}_i$ sont indépendants. Nous avons donc $P(\mathcal{E}) = \prod_{i=1}^{n} P(\mathcal{E}_i)$. Par ailleurs, si $i > 1$ on a $P(\mathcal{E}_i) = \frac{2}{i}$ car le nombre d'éléments égaux au plus petit ou au plus grand élément de la liste est pléonasmiquement 2. Et $P(\mathcal{E}_1) = 1$ parce que ... une liste de longueur un n'a qu'un élément et que partition choisit donc forcément pour pivot le plus petit ou le plus grand élément. Ainsi,

$$P(\mathcal{E}) = \prod_{i=2}^{n} \frac{2}{i} = \frac{2^{n-1}}{n!}$$

L'événement $\mathcal{E}$ est donc extrêmement improbable. Évidemment, me direz-vous, il est possible que moins pire que pire soit déjà assez pire. Nous y reviendrons d'ici la fin de ce notebook.

In [25]:
```
def fact(n):
    p = 1
    for k in range(1, n): p = p * k
    return p
```

In [26]:
```
2 ** 19 / fact(20)
```

Out[26]:
```
4.3099804121821765e-12
```

Pour une liste à 20 éléments, il y a en gros 4 chances sur mille milliards que $\mathcal{E}$ se produise. C'est pas beaucoup :-).

3.3 Une majoration de la complexité en moyenne de quickselect

Montrons que la complexité moyenne de quickselect (en nombre de comparaisons d'éléments de liste) est linéaire. Nous pouvons penser à quickselect comme à une chaîne d'appels récursifs. Nous allons regrouper ces appels récursifs en paquets, chaque paquet constituant ce que nous appellerons une "phase" de l'algorithme. Précisément :

Définition : Une phase de l'algorithme se termine lorsque la taille de la liste est inférieure à 75% de celle de la liste de départ.

Pourquoi 75% ? À cause de la remarque suivante : supposons qu'un pivot soit choisi dans les 50% des valeurs médianes de la liste. Il y a au moins 25% de valeurs inférieures au pivot et au moins 25% de valeurs supérieures au pivot, donc nous sommes assurés qu'une phase se termine après un appel à `partition`. Numérotons les phases 0, 1, 2, ... Dans la phase k, la taille de la liste est inférieure à $n\left(\frac{3}{4}\right)^k$, où n est la taille de la liste de départ. On vérifie alors facilement qu'il y a au plus $N = 1 + \lceil \log_{4/3} n \rceil$ phases (N est le plus petit entier tel que $n\left(\frac{3}{4}\right)^N$ soit inférieur à 1).

Notons X_k la variable aléatoire égale au nombre d'appels récursifs dans la phase k. Le nombre de comparaisons effectuées dans la phase k est alors inférieure à $nX_k\left(\frac{3}{4}\right)^k$. Le nombre total C_n de comparaisons effectuées est donc majoré par

$$C_n \leq \sum_{k=0}^{\lceil \log_{4/3} n \rceil} nX_k \left(\frac{3}{4}\right)^k$$

Pouvons-nous espérer majorer subtilement l'espérance de C_n ? Mais oui, car l'espérance est **linéaire** :

$$E(C_n) \leq n \sum_{k=0}^{\lceil \log_{4/3} n \rceil} E(X_k) \left(\frac{3}{4}\right)^k$$

Reste donc à évaluer $E(X_k) = \sum_{i=0}^{\infty} iP(X_k = i)$.

Exercice : Montrer que $P(X_k = i) \leq \frac{1}{2^i}$. Je sais, poser cet exercice est une sournoiserie. Mais c'est à cette seule condition que le concepteur du notebook sera certain que le lecteur sera réellement impliqué.

On a $P(X_k = i) \leq \frac{1}{2^i}$ donc $E(X_k) \leq \sum_{i=0}^{\infty} \frac{i}{2^i} = 2$. Ainsi,

$$E(C_n) \leq 2n \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \left(\frac{3}{4}\right)^k \leq 2n \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k = 8n$$

Pour ceux d'entre-vous qui auraient des doutes, $\sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k = \frac{1}{1-\frac{3}{4}} = 4$.

Notre fonction `quickselect` a donc bien une complexité linéaire en moyenne.

Proposition : $E(C_n) \leq 8n$.

Remarque : Pourquoi diantre a-t-on $\sum_{i=0}^{\infty} \frac{i}{2^i} = 2$? Pour aller au plus vite, considérons la série entière $\sum_{i\geq 0} \frac{x^i}{2^i}$ Le rayon de convergence de cette série est 2. De plus, pour tout $x \in]-2,2[$, $\sum_{i=0}^{\infty} \frac{x^i}{2^i} = \frac{1}{1-\frac{x}{2}} = \frac{2}{2-x}$. Dérivons : pour tout $x \in]-2,2[$, $\sum_{i=1}^{\infty} \frac{ix^{i-1}}{2^i} = \frac{2}{(2-x)^2}$. Évaluons en 1 : $\sum_{i=1}^{\infty} \frac{i}{2^i} = \sum_{i=0}^{\infty} \frac{i}{2^i} = \frac{2}{(2-1)^2} = 2$.

4 Et après ?

4.1 Il faut toujours tenir ses promesses

J'avais promis de revenir sur le moins pire cas que le pire cas. Alors revenons-y. Soit r un entier non nul. Quelle est la probabilité que chacune des phases de quickselect se termine en au plus r appels récursifs à `partition` ? Quelle est la probabilité que quickselect n'ait besoin de faire "que" $O(n \log n)$ comparaisons ?

4.1.1 Majorons le nombre de comparaisons

Avec les notations ci-dessus, lorsque cela arrive le nombre de comparaisons effectuées par quickselect est majoré par

$$n \sum_{k=0}^{\lceil \log_{4/3} n \rceil} X_k \left(\frac{3}{4}\right)^k \leq nr \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \left(\frac{3}{4}\right)^k \leq 4nr$$

4.1.2 Probabilité d'un tel événement

Soit $\mathcal{E}_{n,r}$ l'événement "Toutes les phases se terminent en au plus r appels récursifs". Le complémentaire de $\mathcal{E}_{n,r}$ est donc "Il existe au moins une phase qui se termine en au moins $r+1$ appels récursifs".

Soit $k \in \mathbb{N}$. On a $P(X_k > r) = P(\bigcup_{i=r+1}^{\infty} [X_k = i]) = \sum_{i=r+1}^{\infty} P(X_k = i)$ car ces événements sont disjoints deux à deux. Avez-vous fait l'exercice un peu plus haut ? Alors tout va bien. On en déduit que $P(X_k > r) \leq \sum_{i=r+1}^{\infty} \frac{1}{2^i} = \frac{1}{2^r}$.

La probabilité pour que **l'une** des phases se termine en plus de $r+1$ étapes est

$$P\left(\bigcup_{i=0}^{\lceil \log_{4/3} n \rceil} [X_i > r]\right) \leq \sum_{i=0}^{\lceil \log_{4/3} n \rceil} P(X_i > r) \leq \sum_{i=0}^{\lceil \log_{4/3} n \rceil} \frac{1}{2^r} = \frac{1 + \lceil \log_{4/3} n \rceil}{2^r} = \frac{1}{2^s}$$

où l'on a posé $r = s + \log_2(1 + \lceil \log_{4/3} n \rceil)$. Nous avons donc pour résumer :

- $P(\mathcal{E}_{n,r}) \geq 1 - \frac{1}{2^s}$
- Le nombre de comparaisons effectuées par quickselect sur une liste appartenant à $\mathcal{E}_{n,r}$ est majoré par $4nr = O(ns + n \log \log n)$.

4.1.3 Quickselect en temps quasi-linéaire

Soit $\mathcal{F}_n$ l'événement : "Le nombre de comparaisons effectuées par quickselect est $O(n \log n)$".

Soit $k > 0$ un réel fixé. Soient $s = \log_2 n^k = k \log_2 n$ et r le nombre correspondant (eh oui, on n'a jamais dit que r ne dépendait pas de n !). On a d'après ce qui précède

- $P(\mathcal{E}_{n,r}) \geq 1 - \frac{1}{2^s} = 1 - \frac{1}{n^k}$
- Le nombre de comparaisons effectuées par quickselect sur une liste appartenant à $\mathcal{E}_{n,r}$ est majoré par $4nr = O(kn \log_2 n + n \log \log n) = O(n \log n)$.

Ainsi, $\mathcal{E}_{n,r} \subset \mathcal{F}_n$ et donc :

$$P(\mathcal{F}_n) \geq P(\mathcal{E}_{n,r}) \geq 1 - \frac{1}{n^k}$$

Ainsi, on a la

Proposition : Pour tout réel $k > 0$, le nombre de comparaisons effectuées par quickselect est un $O(n \log n)$ avec une probabilité supérieure à $1 - \frac{1}{n^k}$.

Bon, et alors ? Nous constatons que

- La probabilité que quickselect effectue $O(n \log n)$ comparaisons tend vers 1 lorsque n tend vers l'infini.
- Cette probabilité tend **vite** vers 1, le choix de k étant arbitraire.

Pour faire vite, nous dirons que quickselect effectue $O(n \log n)$ comparaisons avec une **grande probabilité**. Et, donc, La probabilité que $\Omega(n^2)$ somparaisons soient nécessaires à quickselect tend vers 0 lorsque n tend vers l'infini, et ce plus vite que $\frac{1}{n^k}$, quel que soit k.

4.2 Sélection linéaire en pire cas

Il existe un algorithme qui permet de trouver la médiane d'une liste en temps linéaire **en pire cas**. Son analyse sort du cadre de ce notebook. Je me contente de le décrire brièvement. Voici en quelques mots son fonctionnement. Soit à trouver la médiane d'une liste s.

- Découper la liste en blocs de 5 éléments (plus un bloc avec les éléments restants, si la longueur de la liste n'est pas un multiple de 5).
- Déterminer la médiane de chacun des blocs (cela se fait en temps constant puisque chacun des blocs est de taille constante).
- Déterminer récursivement la médiane de ces médianes.
- Utiliser cette médiane comme pivot de la fonction `partition`.

Je n'en dirai pas plus. Pour ceux d'entre vous qui seraient intéressés, une recherche sur "median of medians" vous dira tout ce que vous voulez savoir.

In [ ]: