Complexité amortie

La méthode du potentiel

Marc Lorenzi

2 juillet 2019

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

[2]
```
plt.rcParams['figure.figsize'] = (12, 6)
```

On se place dans la situation où un algorithme $\mathcal{A}$ prend une structure de données s ayant la "valeur" x en paramètre et modifie la valeur cette structure. Les objets qui nous intéressent sont des listes, des arbres, des graphes, des entiers, etc. La **complexité** de l'algorithme lorsqu'il est exécuté sur la structure est le nombre d'opérations élémentaires effectuées par l'algorithme. Bien évidemment, il s'agit avant toute chose de s'entendre sur ce que l'on appelle "opération élémentaire".

Notons $C_{\mathcal{A}}(x)$ cette complexité. La quantité $C_{\mathcal{A}}(x)$ est souvent difficile à calculer exactement. On définit à partir de celle-ci un certain nombre d'autres quantités, indépendantes de x, et parfois plus faciles à évaluer. Si nous appelons $\mathcal{E}$ l'ensemble des valeurs x qui nous intéressent, on peut considérer

- La complexité en pire cas de l'algorithme $\mathcal{A}$:

$$C'_{\mathcal{A}} = \max_{x \in \mathcal{E}} C_{\mathcal{A}}(x)$$

- La complexité en meilleur cas de l'algorithme $\mathcal{A}$:

$$C''_{\mathcal{A}} = \min_{x \in \mathcal{E}} C_{\mathcal{A}}(x)$$

- La complexité en moyenne de l'algorithme $\mathcal{A}$. Pour cela on commence par mettre une probabilité sur l'ensemble $\mathcal{E}$. La fonction $C_{\mathcal{A}} : \mathcal{E} \to \mathbb{R}$ est alors une variable aléatoire et la complexité en moyenne de l'algorithme est son espérance :

$$C'''_{\mathcal{A}} = E(C_{\mathcal{A}})$$

Par exemple, si $\mathcal{E}$ est fini de cardinal n et muni de la probabilité uniforme, alors

$$C'''_{\mathcal{A}} = \frac{1}{n} \sum_{x \in \mathcal{E}} C_{\mathcal{A}}(x)$$

Dans ce notebook, nous allons parler d'un quatrième type de complexité : la **complexité amortie**.

1. La complexité amortie

1.1 Introduction

Histoire de ne pas parler dans le vide, voici quelques exemples d'algorithmes pour lesquels une analyse amortie aurait un sens.

- On ajoute un objet à la fin d'une liste.
- On retire un objet à la fin d'une liste.
- On ajoute un sommet à un graphe, à un arbre binaire, etc.
- On incrémente un compteur binaire.
- Ayant ordonné totalement l'ensemble $\mathcal{E}$, on passe d'un élément de $\mathcal{E}$ à l'élément suivant.

Ce qui nous intéresse, c'est le "temps" mis par une **succession** de n applications de l'algorithme sur la structure de données. Par exemple, on ajoute n objets à la fin d'une liste, on incrémente n fois un compteur binaire, on énumère les parties d'un ensemble. Bien évidemment, la complexité de cette succession de calculs est inférieure à

$$nC'_{\mathcal{A}}$$

où $C'_{\mathcal{A}}$ est le complexité en pire cas de l'algorithme $\mathcal{A}$. Dans les exemples qui vont suivre, il s'avère que ce majorant est pessimiste : une application particulière de l'algorithme peut être coûteuse, mais si beaucoup d'autres applications ne le sont pas alors la complexité globale de l'algorithme sera faible.

1.2 Considérations énergétiques

Calculer un objet x' à partir d'un objet x demande une certaine quantité de travail. Vous savez tous cela si vous avez déjà résolu une équation de degré 5 ou calculé une intégrale triple :-). Notons $\Delta c = c(x)$ le travail fourni, mesuré en "opérations élémentaires" : chaque

opération élémentaire vaut un quantum de travail. Pour certains calculs nous pouvons interpréter une partie du travail Δc comme une différence de potentiel :

$$\Delta c = \Delta c_a - \Delta \Phi$$

où $\Delta \Phi = \Phi(x') - \Phi(x)$ est la différence du potentiel Φ entre les objets x et x', et $\Delta c_a = c_a(x)$ est le "reliquat" de travail. Que vaut ce potentiel Φ ? Cela dépend évidemment de l'algorithme $\mathcal{A}$ et, comme en physique (en fait beaucoup plus) le choix de Φ contient une dose d'arbitraire. Comme en physique, le potentiel peut travailler contre nous (on monte au sommet de la montagne) ou pour nous (on redescend à ski). Plus nous augmentons notre potentiel, plus nous accumulons de l'énergie . Nous pourrons ensuite utiliser cette énergie en notre faveur.

L'idée de **l'analyse amortie** est de trouver un potentiel tel que le reliquat de travail c_a soit faible : une fois trouvée la fonction magique Φ et calculée la quantité c_a on en déduit c !

Définition : $c_a(x)$ est la **complexité amortie** de l'algorithme $\mathcal{A}$ pour le calcul de x' à partir de x.

Les exemples qui suivent éclaireront (un peu) tout cela.

2. Compteurs binaires

2.1 C'est quoi ?

Commençons par un exemple simple, celui des compteurs binaires.

Nous appelons **compteur binaire** de taille n une liste s de longueur n dont les éléments sont des 0 et des 1, représentant un entier en base 2. Pour reprendre les cotations de l'introduction,

- La structure de données est la liste s.
- Sa valeur x est l'entier représenté par s.
- L'algorithme $\mathcal{A}$ incrémente s, ou plutôt sa valeur, de 1.

Dans la discussion qui va suivre nous identifions la liste s à cet entier. Les éléments de s s'appelleront donc sans surprise ses **bits**. Lorsqu'on ajoute 1 à l'entier (représenté par) s, certains 0 deviennent des 1, certains 1 deviennent des 0. Le nombre de changements de bits (que nous appellerons des **swaps**) est la complexité de l'opération d'incrémentation.

Voici la fonction qui modifie le k-ième bit de s :

[3]
```
def swap(s, k):
    global cmpt
    s[k] = 1 - s[k]
    cmpt.incr()
```

Remarquez la variable `cmpt`: ce sera dans quelques secondes un "compteur de swaps".

[4]
```
class Compteur:

    def __init__(self): self.val = 0
    def incr(self): self.val += 1
    def reset(self): self.val = 0
```

[5]
```
cmpt = Compteur()
```

[6]
```
s = [1, 0, 1, 1, 0]
swap(s, 2)
print(s)
print(cmpt.val)
```

```
[1, 0, 0, 1, 0]
1
```

2.3 Incrémenter un compteur binaire

Quel est le successeur de l'entier 10110111 ? C'est 10111000. Pour l'obtenir on part de la fin de s. Tant qu'on rencontre un 1, on le met à 0. Puis on met à 1 le premier 0 rencontré. La fonction d'incrémentation est donc évidente.

[7]
```
def incr(s):
    n = len(s)
    k = n - 1
    while k >= 0 and s[k] == 1:
        swap(s, k)
        k = k - 1
    if k == -1: return False
    else:
        swap(s, k)
        return s
```

[8]
```
incr([1, 0, 1, 1, 0, 1, 1, 1])
```

```
[1, 0, 1, 1, 1, 0, 0, 0]
```

La liste qui ne contient que des 1 est un peu particulière, puisqu'elle n'a pas de successeur.

[9]
```
incr([1, 1, 1, 1])
```

```
False
```

2.4 Compter

La fonction `count` incrément un compteur binaire initialement égal à $[0, \ldots, 0]$. Elle termine lorsque le compteur est égal à $[1, \ldots, 1]$, ou, plutôt, lorsque la tentative d'incrémenter $[1, \ldots, 1]$ échoue.

[10]
```
def count(n, prt=False):
    s = n * [0]
    while True:
        b = incr(s)
        if b == False: break
        if prt: print(s)
```

[11]
```
count(4, prt=True)
```

```
[0, 0, 0, 1]
[0, 0, 1, 0]
[0, 0, 1, 1]
[0, 1, 0, 0]
[0, 1, 0, 1]
[0, 1, 1, 0]
[0, 1, 1, 1]
[1, 0, 0, 0]
[1, 0, 0, 1]
[1, 0, 1, 0]
[1, 0, 1, 1]
[1, 1, 0, 0]
```

```
[1, 1, 0, 1]
[1, 1, 1, 0]
[1, 1, 1, 1]
```

[12]
```
cmpt.reset()
count(10)
print(cmpt.val)
```

```
2046
```

Une liste de 10 bits peut représenter tous les entiers entre 0 et $2^{10} - 1 = 1023$.

Pour compter de 0 à $2^{10} - 1$, nous faisons 2^{10} incrémentations (dont la dernière échoue). Pourtant, nous n'avons effectué qu'environ 2^{11} swaps, alors que certaines incrémentations d'entiers peuvent demander jusqu'à 10 swaps !

Montrons tout d'abord "à la main" que ceci reste valable en général. Nous le reprouverons ensuite de façon beaucoup plus rusée.

2.5 Nombre total de swaps

Combien de swaps effectue-t-on pour incrémenter un compteur binaire ? Cela dépend de la position du premier 0 en partant de la fin de la liste.

- Si la liste termine par un 0, il y a un seul swap. Et il y a 2^{n-1} telles listes.
- Si la liste termine par 01, il y a 2 swaps. Et il y a 2^{n-2} telles listes.
- Si la liste termine par 011, il y a 3 swaps. Et il y a 2^{n-3} telles listes. ...
- Si la liste est 011...1, il y a n swaps. Et il y a une seule telle liste.
- La liste 11...1 est un peu à part. On y effectue n swaps. Et la fonction d'incrémentation renvoie `False`.

Au total, l'itération de la fonction d'incrémentation effectue un nombre de swaps égal à

$$C_n = \sum_{k=1}^{n} k2^{n-k} + n$$

Proposition : $C_n = 2^{n+1} - 2$.

Démonstration : Posons pour $x \in]0, 1[$,

$$f(x) = \sum_{k=0}^{n} x^k$$

On a

$$C_n = 2^{n-1} f'\left(\frac{1}{2}\right) + n$$

Or,

$$f(x) = \frac{1 - x^{n+1}}{1 - x}$$

On en déduit que

$$f'(x) = \frac{-(1-x)(n+1)x^n + (1 - x^{n+1})}{(1-x)^2} = \frac{nx^{n+1} - (n+1)x^n + 1}{(1-x)^2}$$

d'où

$$f'\left(\frac{1}{2}\right) = 4\left(\frac{n}{2^{n+1}} - \frac{n+1}{2^n} + 1\right) = 4 - \frac{n+2}{2^{n-1}}$$

et

$$C_n = 2^{n+1} - 2$$

2.6 Complexité amortie

Venons en au vif du sujet, la **méthode du potentiel**. Nous allons retrouver le résultat précédent par une toute autre méthode. Pour tout compteur s de n bits autre que la liste qui ne contient que des 1, soit s' la liste obtenue par incrémentation de s. Notons $c(s)$ la complexité de cette opération (nombre de swaps).

Notons $\Phi(s)$ le nombre de bits de s égaux à 1. Enfin, appelons **complexité amortie** de l'algorithme d'incrémentation sur la liste s la quantité

$$c_a(s) = c(s) + \Phi(s') - \Phi(s)$$

Partons de la liste $s_0 = [0, \ldots, 0]$ et effectuons $q \geq 1$ incrémentations pour obtenir successivement les listes $s_1, \ldots, s_q$. On a

$$\sum_{k=0}^{q-1} c_a(s_k) = \sum_{k=0}^{q-1} (c(s_k) + \Phi(s_{k+1}) - \Phi(s_k))$$

Un téléscopage se produit sur les différences de potentiels, et on obtient

$$\sum_{k=0}^{q-1} c_a(s_k) = \Phi(s_q) - \Phi(s_0) + \sum_{k=0}^{q-1} c(s_k)$$

d'où

$$\sum_{k=0}^{q-1} c(s_k) = -\Phi(s_q) + \Phi(s_0) + \sum_{k=0}^{q-1} c_a(s_k)$$

La complexité réelle est donc la somme des complexités amorties, à laquelle on soustrait la différence globale de potentiel.

Que vaut ici la complexité amortie $c_a(s)$?

Supposons que $c(s) = k$. Cela veut dire que pour passer de s à s', on a effectué k swaps. Plus précisément, $k-1$ uns sont devenus des zéros, et un zéro est devenu un 1. La liste s' a donc $k-2$ uns de moins que la liste s. Ainsi

$$\Phi(s') = \Phi(s) - (k-2)$$

De là,

$$c_a(s) = c(s) + \Phi(s') - \Phi(s) = k - (k-2) = 2$$

La complexité amortie est constante, égale à 2. Ainsi, en reprenant les calculs faits plus haut,

$$\sum_{k=0}^{q-1} c(s_k) = -\Phi(s_q) + \Phi(s_0) + 2q$$

Comme $s_0 = [0, \ldots, 0]$, on a $\Phi(s_0) = 0$. Et comme $\Phi(s_q) \geq 0$, on obtient

$$\sum_{k=0}^{q-1} c(s_k) \leq 2q$$

Pour une itération sur toutes les valeurs du compteur binaire, $q = 2^n$, et donc

$$\sum_{k=0}^{2^n-1} c(s_k) \leq 2^{n+1}$$

Remarque : Si nous prenons $q = 2^n - 1$, nous avons $s_q = [1, \ldots, 1]$ donc $\Phi(s_q) = n$. Ainsi,

$$\sum_{k=0}^{q-1} c(s_k) = -\Phi(s_q) + \Phi(s_0) + 2q = -n + 0 + 2(2^n - 1) = 2^n - n - 2$$

Nous retrouvons le résultat obtenu au paragraphe précédent (rappelons-nous qu'il faut ajouter n à cela pour tenir compte du calcul (qui échoue) du successeur de $[1,\ldots,1]$).

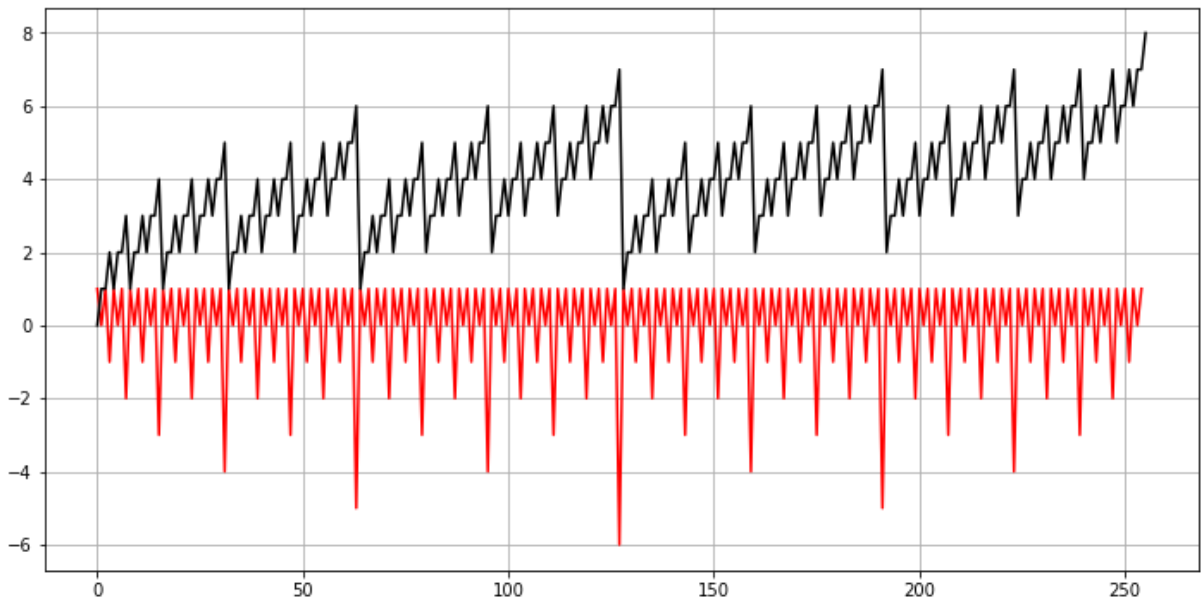
2.7 Commentaires

Traçons le graphe des valeurs du potentiel et aussi celui des différences de potentiel !

[13]
```
def tracer_potentiel(n):
    s = n * [0]
    ps = []
    while True:
        ps.append(sum(s))
        b = incr(s)
        if b == False: break
    qs = [ps[k+1]-ps[k] for k in range(len(ps)-1)]
    plt.plot(qs, 'r')
    plt.plot(ps, 'k')
    plt.grid()
```

Voici le tracé du potentiel au cours des incrémentations. En noir les potentiels, et en rouge les différences de potentiel.

[14]
```
tracer_potentiel(8)
```



Lorsque la liste s est incrémentée de $[0, \ldots, 0]$ à $[1, \ldots, 1]$ le potentiel fluctue assez irrégulièrement de 0 à n. Quelles sont les incrémentations qui augmentent le potentiel ? Rappelons-nous que si $c(s) = k$ alors $\Delta\Phi = -(k-2)$. Ce sont donc les listes telles que $k = 0$ (la liste $[0, \ldots, 0]$) ou $k = 1$ (les listes non nulles qui finissent par 0), c'est à dire la moitié des listes. Lorsqu'on incrémente ces listes, l'augmentation du potentiel fournit en quelque sorte une réserve d'énergie dans laquelle l'algorithme puisera lorsqu'une incrémentation coûteuse surviendra.

Alors pourquoi la complexité amortie est-elle faible ? Parce que la réserve d'énergie stockée dans le potentiel ne génère jamais de dettes importantes ! Regardez par exemple, dans le graphique ci-dessus, le grand pic au milieu : il correspond à $s = [0, 1, \ldots, 1]$. L'incrémentation de s coûte 8 unités d'énergie. À ce moment là le potentiel est à 7. La dette d'énergie après incrémentation n'est donc que de 1. En rajoutant à cela la complexité amortie, qui vaut 2, on se retrouve avec un bénéfice de 1 :-).

3. Files d'attente

3.1 C'est quoi ?

Une **file d'attente** est une structure de données dynamique permettant deux opérations :

- `push`, qui permet d'ajouter un objet x en **queue** de la file.
- `pop`, qui permet de retirer l'objet en **tête** de la file.

Une simple liste Python semble faire l'affaire. Il suffit par exemple d'ajouter des objets à la fin de la liste et de retirer des objets au début de la liste. Quelles sont les complexités de ces opérations ?

- La bonne nouvelle est que l'ajout d'un objet à la fin d'une liste Python a une complexité amortie en $O(1)$. Pour ajouter x à la fin de la liste s, on appelle la méthode `append` : `s.append(x)`. Dans la discussion qui suit nous ferons comme si la complexité en question était toujours $O(1)$.
- La mauvaise nouvelle est que retirer le premier élément d'une liste Python a une complexité en $O(n)$ où n est la longueur de la liste. Pour retirer le premier élément de s il faut faire quelque chose du genre `s = s[1:]`, qui entraîne une recopie des éléments de s.

Remarque : Pour comprendre pourquoi le début et la fin d'une liste Python ne sont pas équivalents, ils faut comprendre comment les listes sont implémentées de facon interne. Nous reviendrons sur ce sujet et sur la complexité de `append` dans la dernière section du notebook. Dans cette section et la suivante, nous ferons comme si la complexité de `append` était effectivement $O(1)$.

3.2 Implémentation

Il y a différentes façons d'écrire une structure de file d'attente efficace en Python. Nous choisirons la suivante (ce n'est pas celle adoptée par les concepteurs du langage, qui utilisent une "liste doublement chaînée"). Un objet de notre classe `File` possède deux champs

- Un champ f (comme "front") qui est une liste. Cette liste contiendra les éléments "en tête de la file", le dernier élément de f étant le premier de la file d'attente.
- Un champ r (comme "rear") qui est aussi une liste. Cette liste contiendra les éléments "en queue de la file", mais rangés à l'envers par rapport à ceux du champ f : le dernier élément de r sera le dernier de la file d'attente.

Par exemple, si s est la file $< a, b, c, d, e >$, où a est le denier élément et e est le premier, on aura par exemple $s.f = [d, e]$ et $s.r = [c, b, a]$, ou peut-être $s.f = [b, c, d, e]$ et $s.r = [a]$, etc.

On rajoute un troisième champ `counter` qui comptera les opérations élémentaires. Nous décidons que ces opérations élémentaires sont :

- Les appels à la méthode `append` (ajouter un élément) sur des listes.
- Les appels à la méthode `pop` (enlever un élément) sur des listes.

Chaque appel à ces méthodes incrémentera un compteur de 1.

Voici un premier jet de la classe `File`. La méthode `reset_counter` remet le compteur à 0.

[15]
```
class File:

    def __init__(self):
        self.f = []
        self.r = []
        self.counter = 0

    def reset_counter(self):
        self.counter = 0
```

Pour ajouter un objet à une file d'attente, on l'ajoute en fin du champ r. Cette opération s'effectue en complexité $O(1)$ (voir remarque un peu plus haut !).

[16]
```
def ajouter(s, x):
    s.r.append(x)
    s.counter += 1
```

[17]
```
s = File()
ajouter(s, 123)
ajouter(s, 456)
ajouter(s, 789)
print(s.f, s.r, s.counter)
```

```
[] [123, 456, 789] 3
```

Pour supprimer l'objet en tête de la file, on supprime le dernier élément du champ f. Complexité en $O(1)$. Euh, oui, mais non ! Que faire si $s.f$ est vide ? Dans ce cas, on renverse $s.r$ puis on le recopie dans $s.f$. Et on efface $s.r$. C'est évidemment là que le bât blesse, car renverser une liste est une opération coûteuse, de complexité $O(n)$ où n est la longueur de la liste à renverser.

[18]
```
def supprimer(s):
    if s.f == []:
        s.r.reverse()
        s.f = s.r[:]
        s.r = []
        s.counter += len(s.f)
    x = s.f.pop()
    s.counter += 1
    return x
```

[19]
```
supprimer(s)
print(s.f, s.r, s.counter)
```

```
[789, 456] [] 7
```

Si vous voulez une justification du fait que l'on ajoute `len(s.f)` au compteur, rappelez-vous que nos opérations élémentaires sont `append` et `pop`. Voici une fonction `renverser` qui fait exactement n appels à `append` pour renverser une liste de longueur n.

[20]
```
def renverser(s):
    s1 = []
    n = len(s)
    for k in range(n - 1, -1, -1):
        s1.append(s[k])
    return s1
```

[21]
```
renverser([1, 2, 3, 4, 5])
```

```
[5, 4, 3, 2, 1]
```

Pour faire plus propre, réécrivons la classe `File`, avec ses deux méthodes `push` et `pop`. Profitons en pour rajouter un champ `maxlen` qui contient la longueur maximale (somme des longueurs de $s.f$ et $s.r$) atteinte par la file s. Chaque appel à `push` met éventuellement à jour le champ `maxlen`. Un appel à `pop` ne peut que diminuer la longueur, bien entendu.

[22]
```python
class File:

    def __init__(self):
        self.f = []
        self.r = []
        self.counter = 0
        self.maxlen = 0

    def reset_counter(self):
        self.counter = 0

    def is_empty(self):
        return self.r == [] and self.f == []

    def push(self, x):
        self.r.append(x)
        self.counter += 1
        l = len(self.f) + len(self.r)
        if l > self.maxlen: self.maxlen = l

    def pop(self):
        if self.f == []:
            self.r.reverse()
            self.f = self.r
            self.counter += len(self.f)
            self.r = []
        x = self.f.pop()
        self.counter += 1
        return x
```

3.3 Tests

Testons tout cela. Faisons n opérations d'insertion et de suppression en partant d'une file vide. À chaque itération on choisit aléatoirement une insertion ou une suppression. On ne supprime évidemment que si la file n'est pas vide !

[23]
```
def test_file(n):
    f = File()
    for k in range(n):
        if random.randint(0, 1) == 0 and not f.is_empty():
            x = f.pop()
        else:
            f.push(random.randint(0, 10))
    return f.counter, f.maxlen
```

[24]
```
print(test_file(100000))
```

```
(149835, 498)
```

Sur l'exemple ci-dessus, on a fait $n = 100000$ opérations. La file a eu une longueur maximale de quelques centaines. Si vous réévaluez la cellule ci-dessus, la longueur maximale de la file varie fortement. En revanche, le nombre d'opérations élémentaires reste à peu près constant, égal à environ $\frac{3}{2}n$.

Comme de mauvais esprits pourraient penser que cela est dû à des appels à `random`, faisons un autre test. Cette fois-ci, faisons $\frac{n}{2}$ insertions suivies de $\frac{n}{2}$ suppressions (le contraire serait délicat à faire :-)).

[25]
```
def test_file2(n):
    f = File()
    for k in range(n // 2):
        f.push(123)
    for k in range(n // 2):
        x = f.pop()
    return f.counter, f.maxlen
```

[26]
```
test_file2(100000)
```

```
(150000, 50000)
```

Bref, se pourrait-il que notre algorithme ait une complexité amortie constante ? Oui, comme nous allons le montrer.

3.4 Complexité amortie

Pour toute file d'attente s, posons

$$\Phi(s) = |s.r|$$

où la valeur absolue désigne la longueur de la liste. À la suite d'une insertion ou d'une suppression, s se transforme en une file s' (plus exactement, les champs de s ont été modifiés, mais ne chipotons pas). Posons

$$c_a(s) = c(s) + \Phi(s') - \Phi(s)$$

où $c(s)$ est le coût réel du passage de s à s'. Examinons les différents cas possibles.

1. On passe de s à s' par une insertion. On a alors $c(s) = 1$ et $\Phi(s') = 1 + \Phi(s)$ puisqu'on ajoute un élément à $s.r$. Ainsi,

$$c_a(s) = 2$$

2. On passe de s à s' par une suppression, et $s.f$ n'est pas vide. On a alors $c(s) = 1$ et $\Phi(s') = \Phi(s)$. Ainsi,

$$c_a(s) = 1$$

3. On passe de s à s' par une suppression, et $s.f$ est la liste vide. On a alors $c(s) = |s.r| + 1$, puisque le renversement coûte $|s.r|$ opérations élémentaires, puis la suppression d'un élément à la fin de $s.f$ coûte une opération. De plus, $\Phi(s') = 0$ et $\Phi(s) = |s.r|$. Ainsi,

$$c_a(s) = |s.r| + 1 - |s.r| = 1$$

Conclusion : Pour toute file s on a $c_a(s) = 1$ pour une opération `pop` et $c_a(s) = 2$ pour une opération `push`.

Partons d'une file s_0 initialement vide et effectuons n opérations, insertion ou suppression. On fabrique successivements n files d'attente $s_1, \ldots, s_n$. On a

$$\sum_{k=0}^{n-1} c_a(s_k) = \sum_{k=0}^{n-1} \left(c(s_k) + \Phi(s_{k+1}) - \Phi(s_k)\right)$$

Par téléscopage, et du fait que $\Phi(s_0) = 0$, il vient

$$\sum_{k=0}^{n-1} c_a(s_k) = \Phi(s_n) + \sum_{k=0}^{n-1} c(s_k)$$

ou encore

$$\sum_{k=0}^{n-1} c(s_k) = -\Phi(s_n) + \sum_{k=0}^{n-1} c_a(s_k)$$

Supposons que l'on ait effectué p push (et donc $n-p$ pop). On obtient

$$\sum_{k=0}^{n-1} c_a(s_k) = 2p + (n-p) = n+p$$

Ainsi,

$$\sum_{k=0}^{n-1} c(s_k) = n + p - \Phi(s_n) \leq n+p$$

Exemple : Prenons n pair et $p = \frac{n}{2}$. On effectue donc autant de push que de pop, ce qui entraîne que $s_n = []$, et donc $\Phi(s_n) = 0$. De là,

$$\sum_{k=0}^{n-1} c(s_k) = n + \frac{n}{2} - 0 = \frac{3}{2}n$$

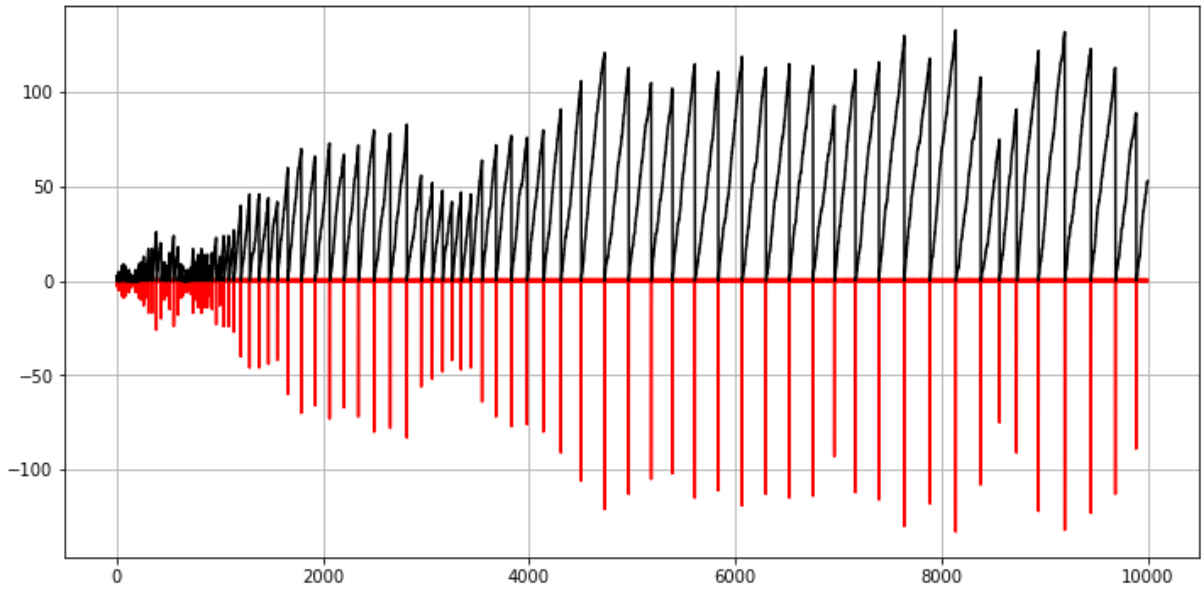
On retrouve le facteur $\frac{3}{2}$ obtenu dans nos tests.

3.5 Tracés

Sans commentaires, voici le tracé du potentiel et des différences de potentiel pour 10000 push ou pop dans une file initialement vide. En noir, le potentiel. En rouge, les différences de potentiel.

[27]
```
def tracer_potentiel2(n):
    f = File()
    ps = []
    for k in range(n):
        r = random.randint(0, 1)
        if r == 0 and not f.is_empty():
            x = f.pop()
        else:
            f.push(random.randint(0, 10))
        ps.append(len(f.r))
    qs = [ps[k+1]-ps[k] for k in range(len(ps)-1)]
    plt.plot(qs, 'r')
    plt.plot(ps, 'k')
    plt.grid()
```

[28]
```
tracer_potentiel2(10000)
```



4. Files à deux extrémités

4.1 C'est quoi ?

Voici un exemple qui généralise le précédent. Je me cantonne au domaine des structures de connées "linéaires" pour ne pas avoir à décrire du code trop compliqué.

Une file à deux extrémités est une structure de données du genre précédent mais dans laquelle il est possible d'insérer ou supprimer en tête ou en queue de la file. Ce nom bizarre `Deque` est celui utilisé par tous les langages de programmation classiques (C++, Java, Python, ...). Voici le code de la classe `Deque` (Double Ended Queue). Il ressemble beaucoup à celui de la classe `File`.

Il va cependant nous falloir être subtils pour les méthodes `pop`. Prenons l'exemple de `pop_rear`, qui supprime l'élément en queue de la file s.

- Si $s.r$ n'est pas vide, aucun problème. Il suffit d'appeler la méthode `pop` de la **liste** $s.r$, qui s'exécute en $O(1)$.
- Si $s.r$ est vide, on procède comme suit : on coupe $s.f$ en deux moitiés, on renverse la première moitié que l'on affecte à $s.r$. Puis on ne garde que la deuxième moitié dans $s.f$.

Une bonne question est : on coupe donc la file en deux moitiés, mais comment décider comment couper en deux ? Où s'arrête la tête et où commence la queue ? Tout cela n'a ni queue ni tête ! Patience ...

Ça a l'air farfelu mais ça marche, comme nous allons le **prouver** par une analyse amortie.

4.2 La classe `Deque`

Voici le code de la classe `Deque`.

- Les méthodes `push_front` et `push_rear` ne présentent aucune difficulté.
- Les méthodes `pop_front` et `pop_rear` sont à méditer. Enfin, il vous suffit de méditer sur l'une des deux, parce qu'on passe de l'une à l'autre en échangeant les "f" et les "r".

[29]
```
class Deque:

    def __init__(self):
        self.f = []
        self.r = []
        self.counter = 0
        self.maxlen = 0

    def reset_counter(self):
        self.counter = 0

    def is_empty(self):
        return self.r == [] and self.f == []

    def push_rear(self, x):
        self.r.append(x)
        self.counter += 1
        l = len(self.f) + len(self.r)
        if l > self.maxlen: self.maxlen = l

    def push_front(self, x):
        self.f.append(x)
        self.counter += 1
        l = len(self.f) + len(self.r)
        if l > self.maxlen: self.maxlen = l

    def pop_front(self):
        if self.f == []:
            n = len(self.r)
            self.f = self.r[:(n+1)//2]
            self.f.reverse()
            self.r = self.r[(n+1)//2:]
            self.counter += n
        x = self.f.pop()
        self.counter += 1
        return x
```

```
    def pop_rear(self):
        if self.r == []:
            n = len(self.f)
            self.r = self.f[:(n+1)//2]
            self.r.reverse()
            self.f = self.f[(n+1)//2:]
            self.counter += n
        x = self.r.pop()
        self.counter += 1
        return x
```

Remarque : Pourquoi une complexité de $n+1$ opérations dans les méthodes `pop` ? Il s'agit de couper une liste en deux et de renverser l'une des deux moitiés. Prenons l'exemple de `pop_front`. Notre code peut être amélioré. On peut en même temps recopier la première "moitié" de $s.r$ dans $s.f$, tout en la renversant. Cela coûte $\lfloor \frac{n+1}{2} \rfloor$ opérations élémentaires. Il faut ensuite ajuster $s.f$ à sa seconde moitié, ce qui n'est pas gratuit ! Cela coûte $n - \lfloor \frac{n+1}{2} \rfloor$ opérations. Au total, n opérations sont suffisantes. Il faut ensuite rajouter l'opération qui enlève un élément à $s.f$.

Pour les amateurs, le code d'une fonction `couper_renverser` (avec compteurs de "append") ressemblerait à cela :

[30]
```
def couper_renverser(s):
    count = 0
    n = len(s)
    p = (n + 1) // 2
    s1 = []
    for k in range(p - 1, -1, -1):
        s1.append(s[k])
        count += 1
    s2 = []
    for k in range(p, n):
        s2.append(s[k])
        count += 1
    return (s1, s2, count)
```

[31]
```
couper_renverser([1, 2, 3, 4, 5, 6, 7])
```

```
([4, 3, 2, 1], [5, 6, 7], 7)
```

[32]
```
couper_renverser([1, 2, 3, 4, 5, 6, 7, 8])
```

```
([4, 3, 2, 1], [5, 6, 7, 8], 8)
```

4.3 Complexité amortie

Qu'allons-nous prendre cette fois ci comme potentiel ? Pour une file s, posons

$$\Phi(s) = ||s.r| - |s.f||$$

où la valeur absolue extérieure est une "vraie" valeur absolue et les valeurs absolues intérieures dénotent des longueurs de listes. Il ne devrait pas y avoir de confusion dans ce qui va suivre.

Pour des raisons de symétrie, les méthodes `push_front` et `push_rear` ont la même complexité amortie, et de même pour les méthodes `pop`.

Commençons par l'analyse de `push_rear`. La file s se transforme en une file s'. La complexité de cette opération est 1. De plus, $|s'.r| = |s.r| + 1$ et $|s'.f| = |s.f|$. Donc

$$\Phi(s') = ||s.r| - |s.f| + 1| \leq ||s.r| - |s.f|| + 1 = \Phi(s) + 1$$

Donc

$$c_a(s) = c(s) + \Phi(s') - \Phi(s) \leq 2$$

Passons à l'analyse de `pop_rear`. Histoire de ne pas nous noyer dans des parties entières supposons n pair. Le cas où n est impair est laissé au lecteur.

- Si $s.r$ n'est pas vide, alors $|s'.r| = |s.r| - 1$ et $|s'.f| = |s.f|$. Donc

$$\Phi(s') = ||s.r| - |s.f| - 1| \leq ||s.r| - |s.f|| + 1 = \Phi(s) + 1$$

et comme $c(s) = 1$,

$$c_a(s) \leq 2$$

- Si $s.r$ est vide, alors $c(s) = |s.f| + 1$. De plus, $\Phi(s) = |s.f|$ et

$$\Phi(s') = ||s'.r| - |s'.f|| = |\frac{1}{2}|s.f| - 1 - \frac{1}{2}|s.f|| = 1$$

Ainsi,

$$c_a(s) = c(s) + \Phi(s') - \Phi(s) \leq 2$$

Comme pour les files d'attente nous obtenons une complexité amortie en $O(1)$.

Faisons un petit test : n insertions et suppressions en tête et/ou en queue, sur une file initialement vide.

[33]
```
def test3(n):
    s = Deque()
    for k in range(n):
        r = random.randint(0, 3)
        if r == 0: s.push_front(1)
        elif r == 1: s.push_rear(1)
        elif r == 2 and not s.is_empty(): s.pop_front()
        elif r == 3 and not s.is_empty() : s.pop_rear()
        #print(s.f, s.r)
    return s.counter, s.maxlen
```

[34]
```
test3(100000)
```

```
(101510, 437)
```

Quelle est la complexité totale ? Avec les notations déjà utilisées pour les files d'attente, nous obtenons

$$\begin{aligned}\sum_{k=0}^{n-1} c(s_k) &= \sum_{k=0}^{n-1} c_a(s_k) + \Phi(s_0) - \Phi(s_n)\\ &= \sum_{k=0}^{n-1} c_a(s_k) - \Phi(s_n)\\ &\leq \sum_{k=0}^{n-1} c_a(s_k)\\ &\leq 2n\end{aligned}$$

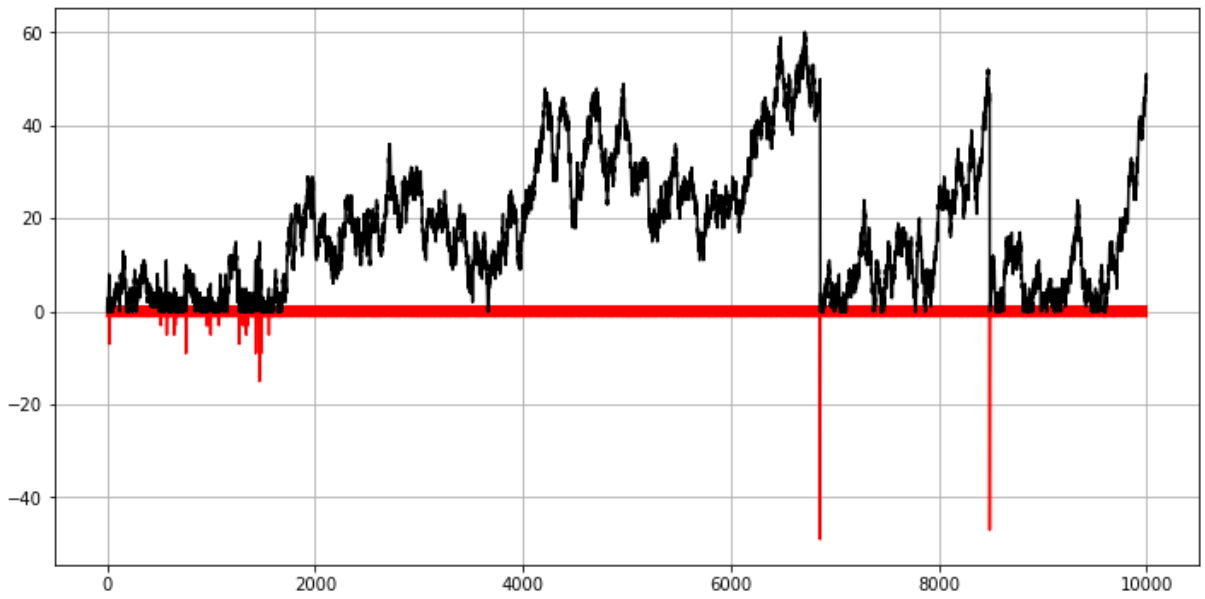
Remarquons pour terminer que notre majorant $2n$ est pessimiste. Les majorations de valeurs absolues du genre $|x-1|$ par $|x|+1$ sont effectivement **très** pessimistes lorsque $x \geq 1$:-). Disons pour faire vite que le 2 du $2n$ est plutôt proche de 1, comme le suggère d'ailleurs notre fonction `test3`. Une analyse plus poussée permettrait sans doute d'effacer le "pour faire vite" et de le remplacer par une preuve. Avis aux amateurs ...

4.4 Tracés

Sans commentaires, voici le tracé du potentiel et des différences de potentiel pour 10000 `push` ou `pop` dans une `Deque` initialement vide.

[35]
```
def tracer_potentiel3(n):
    s = Deque()
    ps = []
    for k in range(n):
        r = random.randint(0, 3)
        if r == 0: s.push_front(1)
        elif r == 1: s.push_rear(1)
        elif r == 2 and not s.is_empty(): s.pop_front()
        elif r == 3 and not s.is_empty() : s.pop_rear()
        ps.append(abs(len(s.r) - len(s.f)))
    qs = [ps[k+1]-ps[k] for k in range(len(ps)-1)]
    plt.plot(qs, 'r')
    plt.plot(ps, 'k')
    plt.grid()
```

[36]
```
tracer_potentiel3(10000)
```



5. Ajout et suppression dans une liste

Nous discutons dans cette section de la complexité de la "fonction" `append` du langage Python. Pour cela, il va nous falloir entrer dans le détail de la façon dont les listes sont implémentées dans notre langage préféré.

5.1 Les méthodes append et pop

Les listes du langage Python possèdent un certain nombre de **méthodes**. Les voici :

[37]
```
s = [1, 2, 3]
print(s.__dir__())
```

```
['__repr__', '__hash__', '__getattribute__', '__lt__', '__le__',
'__eq__', '__ne__', '__gt__', '__ge__', '__iter__', '__init__',
'__len__', '__getitem__', '__setitem__', '__delitem__', '__add__',
'__mul__', '__rmul__', '__contains__', '__iadd__', '__imul__',
'__new__', '__reversed__', '__sizeof__', 'clear', 'copy', 'append',
'insert', 'extend', 'pop', 'remove', 'index', 'count', 'reverse',
'sort', '__doc__', '__str__', '__setattr__', '__delattr__',
'__reduce_ex__', '__reduce__', '__subclasshook__', '__init_subclass__',
'__format__', '__dir__', '__class__']
```

Deux de ces méthodes vont nous intéresser : append et pop.

[38]
```
s.append(4)
print(s)
```

```
[1, 2, 3, 4]
```

[39]
```
x = s.pop()
print(x, s)
```

```
4 [1, 2, 3]
```

Quelle est la complexité de ces méthodes ? Pour faire vite, $O(1)$. En réalité il s'agit là d'une **complexité amortie**. Concentrons nous sur append, une discussion analogue peut être faite pour pop.

5.3 La structure interne des listes

Il existe différentes versions du langage Python. Ce dont je vais parler ici concerne Cython, la version de Python écrite en C. C'est presque certainement celle que vous utilisez vous-mêmes. De façon "interne", une liste est représentée par un tableau de pointeurs sur des objets. Prenons un exemple.

[40]
```
s = [3, False, "pomme"]
```

Lors de la création de s, on peut imaginer (à tort) que Python réserve (le mot exact est **alloue**) trois cases contiguës dans la mémoire.

- La case 0 contient l'adresse d'un objet de valeur l'entier 3 (c'est à dire une autre zone de la mémoire dont Python interprète le contenu comme étant l'entier 3).
- La case 1 contient l'adresse d'un objet de valeur le booléen False.
- La case 2 contient l'adresse d'un objet de valeur la chaîne de caractères "pomme".

Et s dans tout ça ? Eh bien s "connaît" l'adresse de la case 0. Ainsi, lorsque nous demandons $s[2]$, Python peut renvoyer "pomme" en temps $O(1)$:

- En prenant l'adresse du début du tableau et en ajoutant 2 (case contiguës !), ou plutôt deux fois la longueur d'une case.
- Puis en renvoyant l'objet qui se trouve à l'adresse contenue dans la case en question.

Nous voyons donc que l'accès à un élément de s peut se faire en $O(1)$ **parce que les cases du tableau sont contiguës**. Une discussion analogue fonctionne pour l'affectation d'une valeur à une case de s.

5.4 Le fonctionnement de `append`

[41]
```
print(s)
```

```
[3, False, 'pomme']
```

Que se passe-t-il lorsque nous demandons `s.append(3.14)` ? Peut-être que Python rajoute une case contiguë à la case numéro 2 et y met l'adresse d'un objet de valeur le flottant 3.14 ? Oui, mais non ! Parce que en général cette case est déjà occupée par d'autres données ! Alors que faire ? Il faut réserver quelque part dans la mémoire 4 cases contiguës, et déplacer les éléments de s dans ces cases ... Le seul problème est que si l'on fait cela à chaque ajout d'élément, la complexité de `append` est en $O(n)$ où n est la longueur de la liste. Alors que faire ?

La solution adoptée par Python est la suivante :

- Une instruction du genre `s = []` ne réserve aucune place en mémoire.

- Une instruction du genre `s.append(123)`, où s est vide, affecte non pas une mais 4 places en mémoire. Cela nous donne donc un crédit de 3 cases contiguës qui pourront éventuellement être remplies ultérieurement.
- Si s est de longueur 4, `s.append(123)` affecte 8 places en mémoire et déplace les 4 cases de s au nouvel emplacement. On est donc tranquilles pour 4 appels à `append`.

Et après 4 et 8, qu'y a-t-il me direz-vous ? 16, 32, etc. ? Quels sont les nombres magiques déclenchant un déplacement de toutes les valeurs de s ? Eh bien oui pour 16, mais non pour 32. Voici une fonction vous donnant les n premiers nombres. Pour n grand, chaque nombre est environ égal à $\frac{9}{8} \simeq 1.125$ fois le précédent.

[42]
```
def guido(n):
    s = [0]
    for k in range(n):
        y = s[-1] + 1
        z = y // 8
        if y < 9: z = z + 3
        else: z = z + 6
        s.append(s[-1] + z + 1)
    return s
```

[43]
```
s = guido(20)
print(s)
```

```
[0, 4, 8, 16, 25, 35, 46, 58, 72, 88, 106, 126, 148, 173, 201, 233, 269,
309, 354, 405, 462]
```

[44]
```
s = guido(50)
print([s[k] / s[k-1] for k in range(2, len(s))])
```

```
[2.0, 2.0, 1.5625, 1.4, 1.3142857142857143, 1.2608695652173914,
1.2413793103448276, 1.2222222222222223, 1.2045454545454546,
1.1886792452830188, 1.1746031746031746, 1.1689189189189189,
1.1618497109826589, 1.1592039800995024, 1.1545064377682404,
1.1486988847583643, 1.145631067961165, 1.1440677966101696,
1.1407407407407408, 1.1385281385281385, 1.1368821292775666,
1.1354515050167224, 1.1354933726067746, 1.1335927367055771,
1.1327231121281465, 1.1313131313131313, 1.13125, 1.1302288871349644,
1.1298882681564246, 1.1291718170580964, 1.1286261631089218,
1.1280310378273521, 1.1276870163370594, 1.1277163553183378,
1.1271129141311698, 1.126874625074985, 1.1266968325791855,
1.126624143633357, 1.1264416020130006, 1.126209977661951,
1.126115702479339, 1.1259357111404666, 1.1259288228392648,
1.1257381035081626, 1.1256813740615037, 1.1256281407035176,
1.1255681818181817, 1.1254777529386313, 1.1254565259178573]
```

Remarque : Le nom de la fonction, `guido`, est un petit clin d'oeil au concepteur du langage Python. S'il lit ce notebook je serais ravi de savoir pourquoi précisément il a choisi ces nombres :-).

5.5 La complexité réelle de `append`

Quelle est la complexité réelle d'une suite de `append` sur une liste initialement vide ?

Voici tout d'abord une fonction qui calcule, à partir d'un terme de la suite de Guido, le terme suivant.

[45]
```
def g(x):
    y = x + 1
    z = y // 8
    if y < 9: z = z + 3
    else: z = z + 6
    return x + z + 1
```

[46]
```
x = 0
for k in range(10):
    x = g(x)
    print(x)
```

```
4
8
16
25
35
46
58
72
88
106
```

La fonction `complexites` renvoie la liste $[c_0, \ldots, c_n]$ où c_k est la complexité réelle du k ième appel à `append`. Si k n'est pas un nombre de Guido, une opération élémentaire suffit. Sinon, il y a réallocation et $k+1$ opérations sont nécessaires : recopie de la liste initiale, de longueur k, $+$ ajout d'un élément.

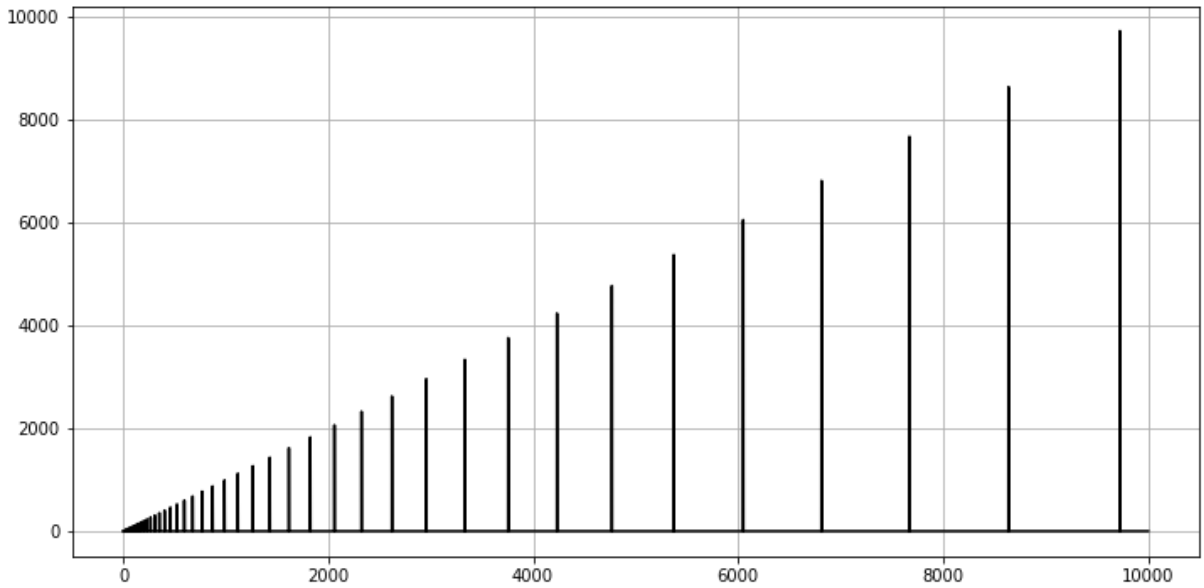
[47]
```python
def complexites(n):
    cs = []
    x = g(0)
    for m in range(n + 1):
        if m < x:
            cs.append(1)
        else:
            x = g(x)
            cs.append(m + 1)
    return cs
```

[48]
```python
print(complexites(100))
```

```
[1, 1, 1, 1, 5, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, 17, 1, 1, 1, 1, 1, 1,
1, 1, 26, 1, 1, 1, 1, 1, 1, 1, 1, 1, 36, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
47, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 59, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 73, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 89, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1]
```

La plupart des complexités valent 1, mais, une fois de temps en temps (devinez quand ?) une réallocation de la liste provoque une complexité importante. Un petit dessin ?

[49]
```python
plt.plot(complexites(10000), 'k')
plt.grid()
```



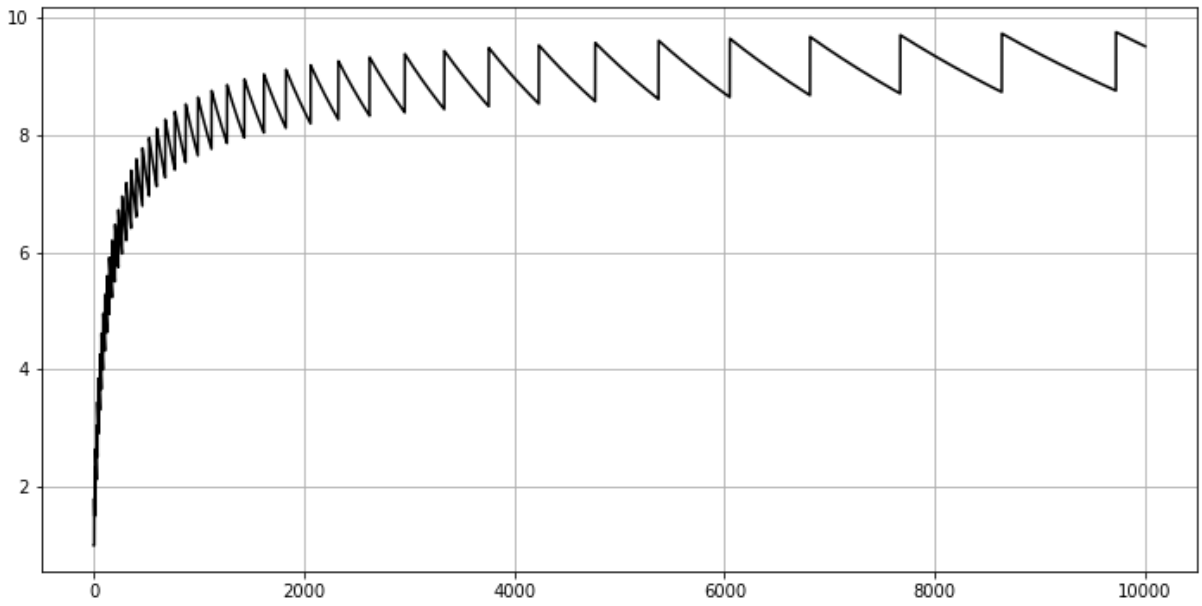
Voici un autre graphique très intéressant. À partir de la liste $[c_0, \ldots, c_n]$ des complexités, calculons $[c'_0, \ldots, c'_n]$ où

$$c'_k = \frac{1}{k+1} \sum_{j=0}^{k} c_j$$

La quantité c'_k est donc la complexité moyenne de append pour $k+1$ insertions à partir de la liste vide.

[50]
```
def moyennes(s):
    s1 = s[:]
    n = len(s)
    for k in range(1, n):
        s1[k] = s1[k] + s1[k - 1]
    for k in range(n):
        s1[k] = s1[k] / (k + 1)
    return s1
```

[51]
```
plt.plot(moyennes(complexites(10000)), 'k')
plt.grid()
```



Il semblerait que cette complexité moyenne soit majorée ... par 10, peut-être ? Nous ne prouverons pas ce résultat, mais dans le paragraphe suivant nous retrouverons le nombre 10.

5.6 La complexité amortie de append

Proposition : La complexité amortie de append est $O(1)$.

Démonstration : Hors de question de faire la démonstration pour les valeurs magiques de n de la suite de Guido. Nous allons prouver ce résultat en supposant qu'à chaque fois que la taille de s atteint une puissance de μ (où $\mu > 1$), il y a une recopie dans une liste μ fois plus longue. Notre suite magique est donc

$$0, 1, \mu, \mu^2, \dots$$

Le fait de prendre des valeurs de μ non entières, comme par exemple $\frac{9}{8}$ est évidemment perturbant. Une démonstration exacte ne peut pas ne pas tenir compte de cela ! Continuons tout de même ainsi. Ce qui suit est donc une démonstration approximative :-).

Question : On prend $\mu = \frac{9}{8}$. Quel est le plus petit entier p tel que $\mu^{p+1} - \mu^p > 1$?

[52]
```
def taille_min():
    mu = 9 / 8
    p = 0
    while mu ** (p + 1) - mu ** p <= 1:
        p += 1
    print(p, mu ** p)
```

[53]
```
taille_min()
```

```
18 8.331925998972208
```

Multiplier la taille allouée par μ ne fait réellement grandir le tableau que pour des tailles de tableau supérieures à 9 (et une puissance de μ supérieure à 18). Voici un début de piste pour la compréhension de la suite de Guido ...

Revenons à notre démonstration : pour toute liste s de longueur n il y a donc en réalité un tableau de longueur μ^p alloué en mémoire, où p est le plus petit entier tel que $n \leq \mu^p$.

Posons

$$\Phi(s) = \beta n - \alpha \mu^p$$

où $\alpha \geq 1$ et $\beta > 0$ seront choisis plus loin de façon génialement astucieuse.

Soient $c(s)$ la complexité de l'ajout d'un élément à s, et $c_a(s) = c(s) + \Phi(s') - \Phi(s)$ où s' est la liste obtenue par l'ajout d'un élément. Deux cas se présentent.

- Si $n < \mu^p$, alors $c(s) = 1$, $\Phi(s) = \beta n - \alpha \mu^p$ et $\Phi(s') = \beta n + \beta - \alpha \mu^p$. De là,

$$c_a(s) = \beta + 1$$

- Si $n = \mu^p$, alors $c(s) = n + 1$, $\Phi(s) = \beta n - \alpha n$ et $\Phi(s') = \beta n + \beta - \alpha \mu n$. De là,

$$c_a(s) = (n + 1) + (\beta n + \beta - \alpha \mu n) - (\beta n - \alpha n) = 1 + \beta + (1 - \alpha(\mu - 1))n$$

Si nous choisissons $\alpha = \frac{1}{\mu - 1}$, nous obtenons $c_a(s) = 1 + \beta$, et la complexité amortie est donc bien constante.

Commment choisir β ? En imposant la condition $\Phi(s) \geq 0$. On a $\mu^{p-1} < n$ donc

$$\Phi(s) = \beta n - \alpha \mu^p > \beta \mu^{p-1} - \alpha \mu^p = \mu^{p-1}(\beta - \alpha \mu)$$

Clairement, $\beta = \alpha \mu$ est la valeur de choix. Finalement,

$$\Phi(s) = \frac{1}{\mu - 1}(\mu n - \mu^p)$$

Conséquence : Si on effectue $n \geq 1$ ajouts d'élément à la fin d'une liste s_0 initialement vide, on fabrique successivement les listes $s_1, \ldots, s_n$. La complexité totale de ces n ajouts est

$$\sum_{k=0}^{n-1} c(s_k) = \sum_{k=0}^{n-1} (c_a(s_k) - \Phi(s_{k+1}) + \Phi(s_k))$$

Par téléscopage :

$$\sum_{k=0}^{n-1} c(s_k) = -\Phi(s_n) + \Phi(s_0) + \sum_{k=0}^{n-1} c_a(s_k) = (1 + \beta)n + \Phi(s_0) - \Phi(s_n)$$

On a $\Phi(s_0) = -1$ car s_0 est vide, et $\Phi(s_n) \geq 0$. Ainsi, en posant $K = 1 + \beta = \frac{2\mu - 1}{\mu - 1}$,

$$\sum_{k=0}^{n-1} c(s_k) \leq Kn - 1$$

Remarqu : K n'est rien d'autre que la complexité amortie de `append`.

Exemples :

- Pour $\mu = 101$, on a $K = 2.01$.
- Pour $\mu = 2$, on a $K = 3$.
- Pour $\mu = \frac{9}{8}$, la valeur asymptotique de la suite de Guido, on a $K = 10$.

Remarque : Clairement, $\mu = 2$ donne une meilleure complexité que $\mu = \frac{9}{8}$. Alors pourquoi le langage Python choisit-il la deuxième solution ? Et pourquoi pas $\mu = 101$???

La complexité "en temps" n'est pas la seule contrainte à étudier lorsqu'on élabore un algorithme. Une autre quantité importante est la **complexité en espace** de l'algorithme. De combien de mémoire l'algorithme a-t-il besoin pour renvoyer le résultat ? Et là, clairement, $\mu = \frac{9}{8}$ est préférable à $\mu = 2$. Quant à $\mu = 101$, il est hors de question d'y penser. Bilan :

- μ grand : insertion rapide, grand gaspillage de mémoire.
- μ petit : insertion lente, petit gaspillage de mémoire.

Le choix de μ est donc un choix délicat ... faisons confiance aux concepteurs de Python pour avoir très longuement réfléchi à cette question.

Remarque :

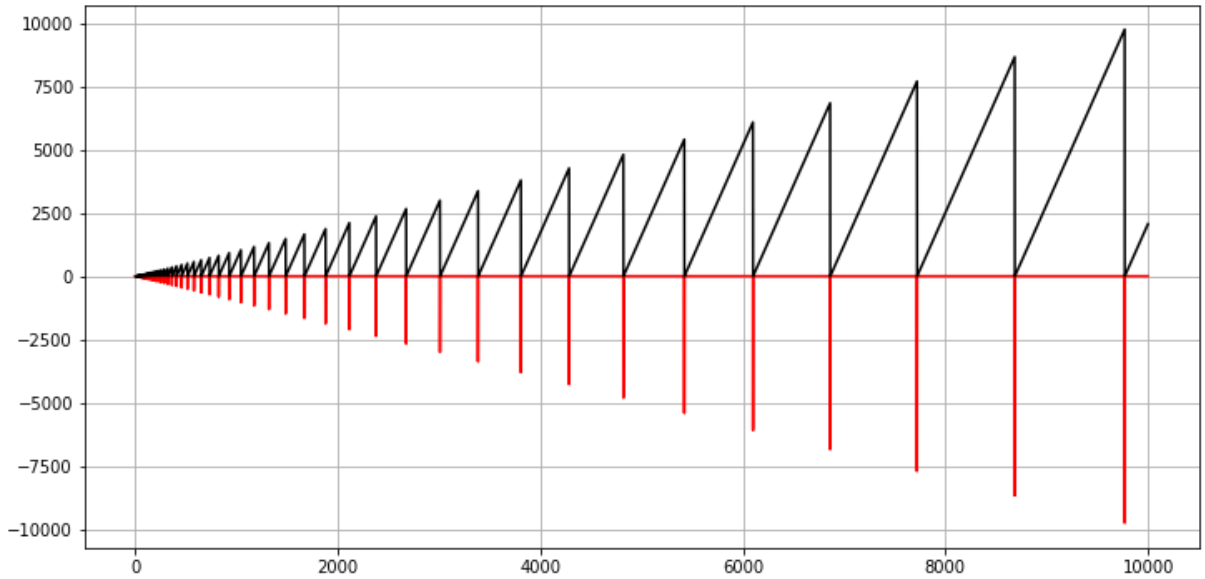
- L'allocation de mémoire n'est pas une opération gratuite. Pour de très grosses listes, cette allocation peut avoir un impact sur la complexité de la fonction `append`.
- Lors du déplacement d'une liste vers un autre emplacement de la mémoire, l'ancien emplacement devient inutilisé et doit être libéré. C'est le rôle du **ramasse-miettes** (Garbage Collector), qui tourne en tâche de fond lorsqu'un programme Python s'exécute. Cette opération n'est pas non plus gratuite.

5.7 Tracé des potentiels

Voici le tracé du potentiel et des différences de potentiel pour 10000 `append` dans une liste initialement vide.

[54]
```
def tracer_potentiel4(n, mu):
    ps = [-1]
    alpha = 1 / (mu - 1)
    for k in range(1, n + 1):
        p = math.ceil(math.log(k) / math.log(mu))
        ps.append((mu * k - mu ** p) / (mu - 1))
    qs = [ps[k + 1] - ps[k] for k in range(n)]
    plt.plot(qs, 'r')
    plt.plot(ps, 'k')
    plt.grid()
```

[55]
```
tracer_potentiel4(10000, 9 / 8)
```



5.8 Et maintenant ?

Il conviendrait sans doute (ce que je ne ferai pas ici) d'analyser la complexité amortie de `pop`, ainsi que la complexité d'appels multiples à `append` et `pop`, comme nous l'avons fait pour les files.

Exercice : Le fonctionnement de `pop` est le suivant : chaque fois que la longueur de la liste devient inférieure à la moitié de la taille allouée pour la liste, il y a allocation d'un nouvel espace et déplacement des données. La taille allouée est le plus petit nombre de Guido supérieur à la longueur de la liste. Analysez la méthode `pop`par la méthode du potentiel. Montrez que la complexité amortie de `pop` est $O(1)$.

Pour les amateurs : Le code source du langage Python (écrit en C) est disponible en ligne. Regardez le code de `append` et `pop`. Il se trouve dans le fichier `Objects/listobject.c`. Regardez en particulier le code de la fonction C `list_resize`. Ce code est commenté, il dit précisément quand ont lieu les réallocations.

Bibliographie :

- T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms (2nd Edition), MIT Press 2001. Contient un chapitre sur l'analyse amortie.
- C. Okazaki, Purely Functional Data Structures, Cambridge Uiversity Press 1998. Mine d'informations sur les structures de données non impératives. Ce livre parle de programmation fonctionnelle et s'adresse à des programmeurs connaissant les langages Standard ML et/ou Haskell mais si vous maîtrisez Ocaml vous n'aurez aucun mal à comprendre le code ML. Si vous ne connaissez que Python ce sera plus difficile ...