Université Paris 8 Année 2022–2023

Master Mathématiques et applications, parcours ACC

Aide : mesure expérimentale de complexité d'un algorithme

18 octobre 2022

Résumé

Ce document a pour but de présenter une méthode expérimentale et graphique pour estimer la complexité en temps d'une implantation. Cette analyse n'est pas parfaite mais permet d'obtenir une première approximation de la complexité.

Considérons la fonction `f` suivante (implantée en `python`), où `x` est une liste d'entiers de taille `k` et `n` est un entier supérieur à 2.

```python
def f(x, k, n):
    res = 1
    for i in range(k):
        for j in range(i):
            res = (res + x[i]*x[j]) % n
    return res
```

On cherche ici à estimer expérimentalement la complexité algorithmique de la fonction `f`. Pour cela, notons que celle-ci dépend essentiellement de deux paramètres : la longueur de la liste k et la valeur maximale n des entiers impliqués.

Remarque. Notre estimation se fera via le temps d'exécution de la fonction. Pour cela, on dispose de la fonction `python time()` incluse dans la bibliothèque `time`. Celle fonction retourne le temps écoulé depuis le 1er janvier 1970, en millisecondes. Pour calculer le temps d'exécution d'un algorithme, on l'encadre donc deux prises de temps, et on en calcule la différence.

```python
from time import time
t1 = time.time()
_ = f(x, k, n)
t2 = time.time()
temps_ecoule = t2 - t1
```

Pour mesurer l'influence de n ou de k sur le temps de calcul, il est préférable de fixer l'un de ces paramètres, puis de faire varier l'autre. Supposons d'abord que l'on fixe n, à une valeur raisonnable et pas trop grande (pour que les exécutions ne soient pas trop longues). Dans l'exemple, on va prendre $n = 1027$. Puis, on fait varier k sur une plage assez grande et on étudie l'évolution du temps d'éxecution « moyen » en fonction de k, que l'on note $T(k)$.

Que cherche t-on exactement ? On cherche à *vérifier une hypothèse* sur la forme de la complexité. Pour cela, il faut savoir que, **graphiquement, on ne sait reconnaître que des droites affines, desquelles on peut déduire un coefficient directeur et une ordonnée à l'origine**.

— Si l'on souhaite vérifier que la complexité est **linéaire** en k, c'est-à-dire, que $T(k)$ est de la forme $ak + b$. Alors, on va tracer le graphe de $T(k)$ en fonction de k, puis retrouver graphiquement une approximation de a et b.

— Si l'on souhaite vérifier que la complexité est **polynomiale** en k, c'est-à-dire, que $T(k)$ est de la forme $ak^d + o(k^d)$. Alors, on va tracer le graphe de $\log(T(k))$ en fonction de $\log(k)$ (peu importe la base du logarithme). On doit alors obtenir

$$\log(T(k)) \simeq d\log(k) + \log(a),$$

autrement dit, le coefficient directeur de la droite obtenue fournit l'exposant de complexité, et l'ordonnée à l'origine donne le logarithme du coefficient dominant.

— Si l'on souhaite vérifier que la complexité est **exponentielle** en k (c'est-à-dire, $T(k)$ est de la forme $ac^k + o(c^k)$. Alors, on va tracer le graphe de $\log(T(k))$ en fonction de k. On doit alors obtenir

$$\log(T(k)) \simeq \log(c)k + \log(a),$$

autrement dit, le coefficient directeur de la droite obtenue fournit le logarithme de la base de l'exponentielle, et l'ordonnée à l'origine donne le logarithme du coefficient dominant.

— Si l'on souhaite vérifier que la complexité est **polylogarithmique** en k (c'est-à-dire, $T(k)$ est de la forme $a\log(k)^d + o(\log(k)^d)$. Alors, on va tracer le graphe de $\log(T(k))$ en fonction de $\log\log(k)$ (peu importe la base du logarithme). On doit alors obtenir

$$\log(T(k)) \simeq d\log\log(k) + \log(a).$$

— **Exercice :** comment vérifier expériementalement une complexité de la forme $T(k) \simeq 2^{a\sqrt{k}}$? et de la forme $T(k) \simeq a\log(k)\log(\log(k))$?

Maintenant que l'on sait ce que l'on chercher à vérifier, il faut choisir une plage de valeurs pour k, et mesurer le temps d'éxecution correspondant. La plage de valeurs aura : une valeur minimale, une valeur maximale, et un type de progression (arithmétique, géométrique).

Quelles valeurs minimales/maximales prendre pour k ? Éviter les trop petites valeurs ou les valeurs singulières. Pour de très grandes valeurs, il peut y avoir des problèmes d'accès mémoire (plus lent) qui peuvent fausser l'analyse.

Comment faire varier k ? Cela dépend de la forme de complexité ciblée. Il faut que les points mesurés soient uniformément répartis sur l'axe des abscisses. Si l'on s'attend à une complexité polynomiale en k, alors on va tracer $\log(T)$ en fonction de $\log(k)$, donc il est intéressant de faire croitre k de manière exponentielle (car la croissance de $\log(k)$ sera alors linéaire). Par exemple, à chaque étape on multiplie k par une constante et on en prend la partie entière.

Combien de tirages effectuer pour chaque (n,k) ? Cela dépend de la variance de l'algorithme utilisé (surtout s'il est probabiliste). Cela peut aller de quelques tirages (par exemple, 5, 7) jusqu'à plusieurs milliers. Dans tous les cas, pour une valeur de (n,k) fixée, il est préférable de garder uniquement la médiane (ou la moyenne des valeurs "centrales") des temps d'exécutions.

Voici un exemple de script pour collecter les mesures temporelles :

```
M = 17
TIMES = []
KMIN, KMAX, KSTEP = 10, 1000, 1.2
k = KMIN
while k < KMAX:
    print("on traite la valeur :", k)
    TMP_TIMES = []
    for m in range(M):
        x = [ randint(0,n-1) for i in range(k) ]
        t1 = time()
        _ = f(x,k,n)
        t2 = time()
        TMP_TIMES.append(t2-t1)
    TMP_TIMES.sort()
    TMP_TIMES = TMP_TIMES[3:-3]
    TIMES += [ [k,t] for t in TMP_TIMES]
    k *= KSTEP
    k = ceil(k)
```

Dans cet exemple, pour chaque valeur de k on choisit d'effectuer $M = 17$ exécutions, et d'éliminer les 3 temps les plus courts et les 3 temps les plus longs. On obtient les graphiques de la Figure 1, tracés grâce aux fonctions d'affichage de `sagemath` :

```
scatter_plot(TIMES, markersize=1)
scatter_plot( [ [log(val[0],2), log(val[1],2)] for val in TIMES ], markersize=1)
```

La sous-figure de gauche montre une évolution convexe, mais on n'arrive pas à déterminer si c'est une exponentielle, une cubique, une parabole, etc. En revanche, la sous-figure de droite est bien plus lisible. On trouve graphiquement deux points de la « droite » représentée par cette alignement approximatif de points : par exemple $(5.8, -12)$ et $(11, -2)$. On en déduit que le coefficient directeur est d'environ $\frac{10}{5.2} \simeq 1.92$. C'est relativement proche la valeur attendue (2).

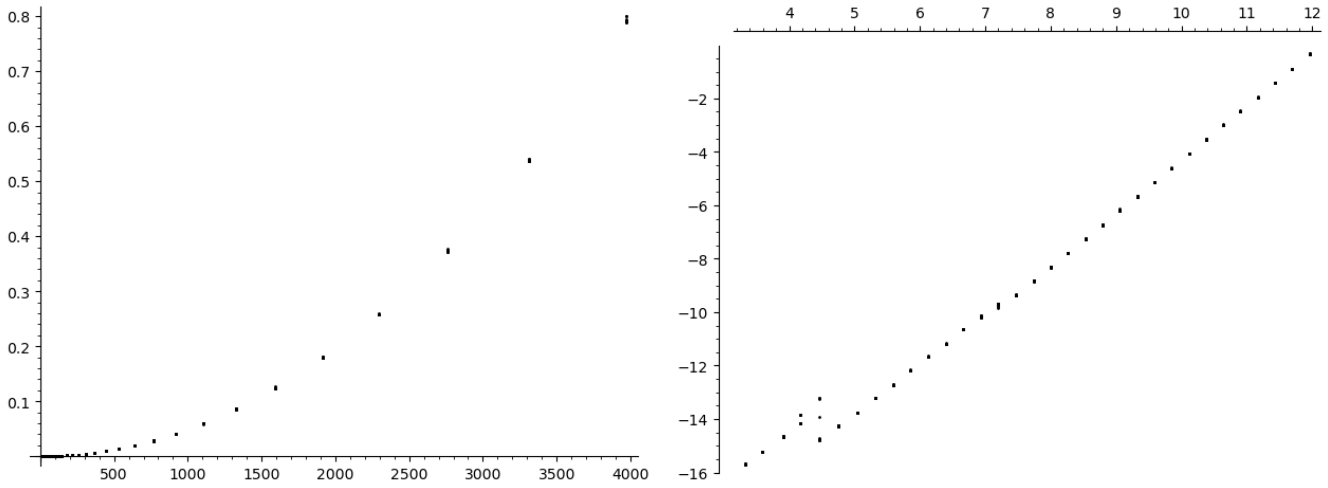

FIGURE 1 – Complexité lorsque k varie de manière géométrique. À gauche : en fonction de k ; à droite : en fonction de $\log(k)$.

Observons que si nous n'avions opéré une progession géométrique k, mais une progression arithmétique, nous aurions obtenu les graphiques de la Figure 2, avec un étalement des points sur la sous-figure de droite qui rend l'analyse moins précise.

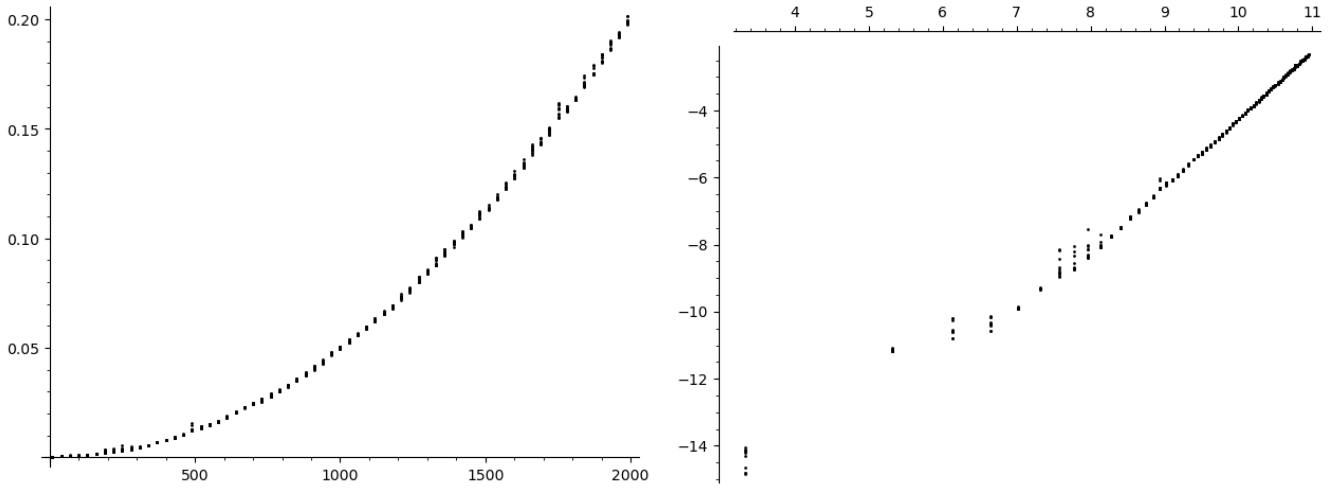

FIGURE 2 – Complexité lorsque k varie de manière arithmétique. À gauche : en fonction de k ; à droite : en fonction de $\log(k)$.

Pour la dépendance en n de la complexité, observons que l'on s'attend plutôt à une complexité polylogarithmique. Traçons donc $\log(T(n))$ en fonction de $\log\log(n)$ après avoir collecté les mesures dans le script suivant :

```
M = 17
TIMES = []
LOGNMIN, LOGNMAX, LOGNSTEP = 100, 20000, 1.1
log_n = LOGNMIN
while log_n < LOGNMAX:
    print("on traite", log_n)
    n = randint(2**log_n, 2**(log_n +1))
    TMP_TIMES = []
    for m in range(M):
        x = [ randint(0,n-1) for i in range(k) ]
        t1 = time()
        _ = f(x,k,n)
        t2 = time()
        TMP_TIMES.append(t2-t1)
    TMP_TIMES.sort()
    TMP_TIMES = TMP_TIMES[2:-2]
    TIMES += [ [log_n,t] for t in TMP_TIMES]
    log_n *= LOGNSTEP
    log_n = ceil(log_n)
```

Les graphes donnés en Figure 3 révèlent une complexité en $\log(n)^d$ avec $d \simeq 1.62$.

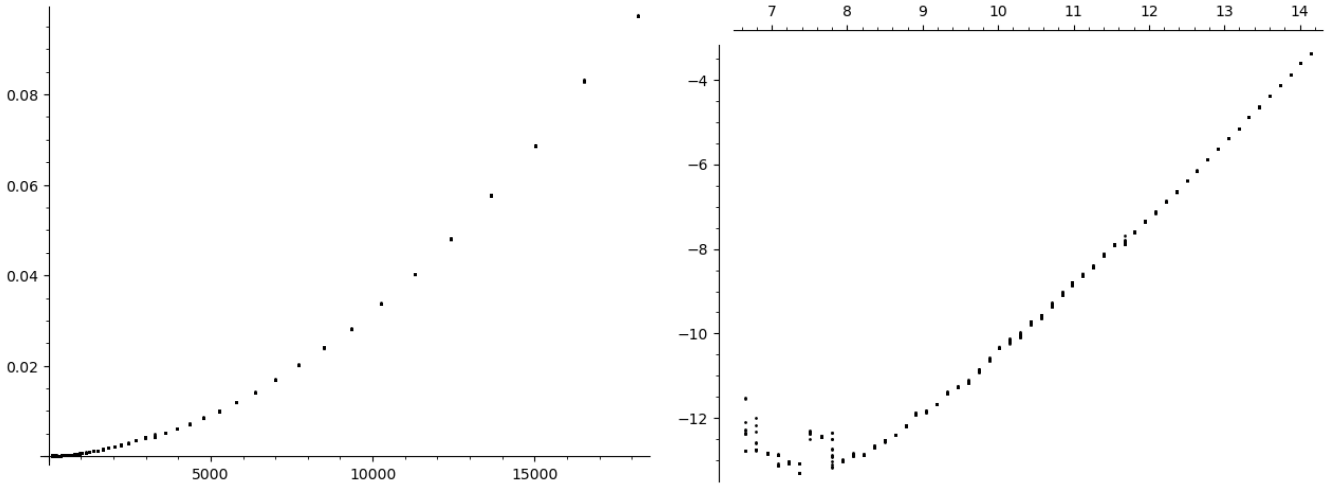

FIGURE 3 – Complexité lorsque n varie de manière géométrique. À gauche : en fonction de n ; à droite : en fonction de $\log\log(n)$.