Chapitre $\underline{1001}_2$
Complexité

14 mars 2017

1 Préliminaire : parlons du temps...

...ou plutôt des ordres de grandeur des durées.

On choisit comme unité la seconde [1], on donne des logarithmes décimaux des durées étudiées.

Quelques repères à connaître.

1. $\log_{10} t = x$ signifie $t = 10^x$, en particulier $10^n \leq t < 10^{n+1}$ où $n = \lfloor x \rfloor$.
2. $10^{0,5} = \sqrt{10} \approx 3$.
3. $10^{0,3} \approx 2$.
4. Un an vaut environ $3,15 \times 10^7$s, *i.e.* $\pi\,\mathrm{s} \approx 1$ nano-siècle.

TABLE 1 – Durées : ordre de grandeur (t en secondes).

$\log_{10}(t)$	Durée
$-9,0$	exécution d'une instruction
$0,0$	une seconde
$1,8$	une minute
$3,6$	une heure
$4,9$	un jour
$5,8$	une semaine
$6,4$	un mois (30 jours)
$7,5$	un an
$9,5$	un siècle
$10,5$	un millénaire
$13,1$	âge de la maîtrise du feu
$13,5$	un million d'années

1. D'ailleurs, c'est l'unité de temps du système international

TABLE 1 – Durées : ordre de grandeur (suite).

$\log_{10}(t)$	Durée
$14,0$	âge de Lucy
$16,5$	un milliard d'années
$17,0$	âge de la vie sur Terre
$17,6$	âge de l'univers
$34,0$	temps avant extinction des dernières étoiles

2 Introduction

Définition 2.0.1 (Complexité). La complexité algorithmique est l'étude des ressources requises pour exécuter un algorithme, en fonction d'un paramètre (souvent, la taille des données d'entrée). Les deux ressources en général étudiées sont :

1. Le temps nécessaire à l'exécution de l'algorithme
2. La mémoire nécessaire à l'exécution de l'algorithme (en plus des données d'entrée).

Nous avons déjà vu dans le TP06 que deux algorithmes résolvant un même problème peuvent avoir des temps d'exécution radicalement différents.

Nous allons étudier un exemple plus simple.

2.1 Rappels d'arithmétique : le petit théorème de Fermat

Théorème 2.1.1 (de Fermat). Soit p un nombre premier et $a \in [\![1, p[\![$, alors $a^{p-1} = 1[p]$.

On s'intéresse au problème de l'étude de la primalité d'un entier naturel donné. Plusieurs méthodes existent pour répondre de manière déterministe à cette question, mais elles ne sont pas nécessairement satisfaisantes. Notamment, dire si un grand nombre entier est premier est assez difficile.

On considèrera alors le « test » de primalité de Fermat, en base 2, en fonction d'un entier p.

— On calcule 2^{p-1} modulo p.

— Si cela ne donne pas 1, p n'est pas premier.

— Si cela donne 1, on dit que « p est probablement premier ».

2.2 Algorithmes d'exponentiation

La question qui se pose naturellement est de calculer 2^{p-1} modulo p (sous entendu, sans exploiter l'exponentiation de Python). Voici trois manières différentes de le faire.

1. Une première idée est de calculer naïvement la puissance, de proche en proche, puis de considérer la division euclidienne de ce nombre par p.

```
def expo1(p):
    """Renvoie 2**(p-1) mod p"""
    x = 1
    for i in range(p-1):
        x = 2*x
    return x % p
```

2. Une seconde idée est de réaliser chaque multiplication modulo p, ce qui permet de ne multiplier que de « petits » nombres à chaque fois.

```
def expo2(p):
    """Renvoie 2**(p-1) mod p"""
    x = 1
    for i in range(p-1):
        x = (2*x) % p
    return x
```

3. Une troisième idée est de voir que si $n = 2k$, alors $2^n = \left[\left(2^k\right)\right]^2$ et si $n = 2k+1$, alors $2^n = 2\left[\left(2^k\right)\right]^2$. Pour calculer une puissance (ici, de 2), il suffit donc de calculer une puissance bien plus petite, suivie d'une mise au carré et d'une multiplication éventuelle. C'est l'idée de l'algorithme *d'exponentiation rapide*.

```
def expo3(p):
    """Renvoie 2**(p-1) mod p par exponentiation rapide"""
    y = 1
    n = p-1
    x = 2
    while n > 0:
        # Invariant : y * (x**n) = 2**(p-1) mod p
        # Variant : n
        if n % 2 != 0: # n est impair
            y = (y * x) % p
            n = n - 1
        # n est pair et y * (x**n) = 2**(p-1) mod p
        x = (x * x) % p
        n = n // 2
    # n == 0 et y = 2 ** (p-1) modulo p
    return y
```

On a donc construit les « tests » suivants.

```
def premierprobable1(p):
    return expo1(p) == 1

def premierprobable2(p):
```

```
    return expo2(p) == 1

def premierprobable3(p):
    return expo3(p) == 1
```

2.3 Temps d'exécution

On se doute bien que :

1. les temps d'exécution varient en fonction du nombre p à tester ;
2. il croissent avec p (grosso-modo).

Étudions cela plus précisément, pour $i \in \{1, 2, 3\}$ on note $T_i(p)$ le temps d'exécution de `premierprobable(p)`.

Dressons un tableau des temps de calcul $T_i(n)$ pour les fonctions que nous avions données, pour $i = 1, 2, 3$.

TABLE 2 – Logarithmes des temps d'exécution (en secondes).

$\log_{10}(p)$	$\log_{10}(T_1(p))$	$\log_{10}(T_2(p))$	$\log_{10}(T_3(p))$
6	$1,2$	$-1,0$	$-4,7$
8	*5, 2*	$1,0$	$-4,6$
9	*7, 2*	*2, 0*	$-4,5$
10	*9, 2*	*3, 0*	$-4,4$
14	*17, 2*	*7, 0*	$-4,3$
24		*17, 0*	$-4,0$
100			$-2,9$
2000			$0,0$

(les valeurs en italique sont des extrapolations)

Ces temps varient **énormément** en fonction de l'algorithme utilisé. Suivant l'algorithme, on a une exécution quasi-instantanée ou au contraire interminable.

Ainsi, avoir une idée du temps d'exécution d'un programme est nécessaire avant de l'utiliser réellement dans un contexte scientifique ou industriel.

Ainsi, étudier expérimentalement le temps de calcul d'un algorithme est :

1. nécessaire pour «valider» l'usage d'un algorithme ;
2. insuffisant pour trouver [2] des algorithmes performants.

Il est donc nécessaire d'étudier théoriquement la complexité des algorithmes.

Comment faire cette étude théorique ?

1. On se donne un *modèle* de l'exécution des programmes. Ce modèle précise le temps de calcul de chaque opération élémentaire.

2. En inventant un algorithme ou en combinant des algorithmes existants

2. Il ne reste plus à regarder, pour un programme donné, combien d'instructions élémentaires il effectue.

3 Étude théorique

3.1 Deux mauvaises nouvelles

1. Compter exactement le nombre d'opérations faites par le programme est pénible voire difficile (mathématiquement).
2. Savoir précisément combien de temps met chaque opération élémentaire est difficile et change suivant les machines.

3.2 Deux bonnes nouvelles

1. Il n'est pas nécessaire d'être extrêmement précis : on se contente d'un ordre de grandeur quand le paramètre devient grand. On parle d'**estimation asymptotique de la complexité**.
2. Les principes de cette estimation étaient déjà valables il y a 70 ans [3] et le seront encore probablement dans 50 ans.

3.3 Étude théorique de `premierprobable2`

```python
def expo2(p):
    """Renvoie 2**(p-1) mod p"""
    x = 1
    for i in range(p-1):
        x = (2*x) % p
    return x

def premierprobable2(p):
    return expo2(p) == 1
```

Comment estimer le temps d'exécution de `premierprobable2(p)` ?
Les opérations effectuées dans `premierprobable2(p)` sont les suivantes.

- Une affectation.
- Une boucle effectuant $\mathtt{p} - 1$ tours. À chaque tour on effectue les opérations suivantes :
  — une multiplication ;

3. Le Harvard Mark III, construit en 1949, était l'ordinateur le plus rapide du monde. Il avait coûté de l'ordre de 10^5\$ à l'époque, soit 10^6\$ actuels. Il faisait une addition en 4×10^{-3}s et avait environ 8ko de mémoire vive. Depuis les performances des ordinateurs ont été multipliées par 10^6 environ.

  — un calcul de reste ;
  — une affectation.
- Une comparaison.

On considère le modèle usuel suivant.

- Le temps mis par une affectation est constante.
- Le temps mis par opération arithmétique (somme, produit, division, calcul de reste) est constant.
- Le temps mis par une comparaison de deux nombres est constant.

Appelons alors

- c_1 le temps d'une affectation ;
- c_2 le temps d'une multiplication ;
- c_3 le temps d'un calcul de reste ;
- c_4 le temps d'une comparaison.

Cela donne,

$$T_2(p) = c_1 + (p-1)(c_2 + c_3 + c_1) + c_4$$

C'est une expression déjà un peu compliquée et qui, à première vue, ne nous apprend pas grand-chose, car on ne connaît pas les constantes en jeu. Ces constantes dépendent de la machine sur lequel on exécutera le programme.

On a quand même une information intéressante : $T_2(p)$ n'augmente pas plus vite que p multipliée par une constante.

Montrons ce second point. Pour $p \geq 1$:

$$\begin{aligned} T_2(p) &\leq c_1 p + p(c_2 + c_3 + c_1) + c_4 p \\ &\leq (2c_1 + c_2 + c_3 + c_4)p \end{aligned}$$

On a donc montré qu'il existait un entier p_0 et un réel $C > 0$ tel que

$$\forall p \geq p_0 \qquad T_2(p) \leq Cp$$

On dit alors que $T_2(p)$ est *dominé* par p et on note

$$T_2(p) = O(p)$$

Définition 3.3.1. On dit qu'une suite (u_n) est dominée par une suite (v_n) et on note $u_n = O(v_n)$ si, à partir d'un certain rang, la suite (u_n/v_n) est bien définie et bornée.

Ici, on a, pour $p \geq 1$:

$$0 \leq T_2(p) \leq Cp$$

Donc

$$\left| \frac{T_2(p)}{p} \right| \leq C$$

donc $T_2(p) = O(p)$.

Remarque 3.3.2. Bien entendu, on a aussi $T_2(p) = O(p^2)$, même si cela ne nous intéresse pas ... Nous aimerions donc bien aussi certifier qu'il existe une constante C' telle que, pour tout $p \geq 1$, $C'p \leq T_2(p)$. Cela se traduirait par $p = O(T_2(p))$. Conjugué avec $T_2(p) = O(p)$, cela se note $T_2(p) = \Theta(p)$.

Souvent, obtenir une telle borne inférieure est bien plus difficile qu'obtenir une majoration, nous ne nous en soucierons généralement pas.

3.4 Rédaction

Nous avons détaillé l'étude de `premierprobable2`. On peut aller beaucoup plus vite. Il suffit de dire :

1. On considère que les temps d'exécution d'une affectation, d'une opération arithmétique et d'une comparaison sont constant.
2. Dans `premierprobable2(p)`, on effectue une affectation et une comparaison ainsi que $(\mathtt{p} - 1)$ tours de boucle, avec un nombre constant d'opérations de temps constant à chaque tour.
3. Donc le temps d'exécution de `premierprobable2(p)` est un $O(\mathtt{p})$.

3.5 Étude expérimentale de `premierprobable2`

Pour $p = \left\lfloor \dfrac{p_0}{q^k} \right\rfloor$, avec $q = 1,35$, $p_0 = 10^7$ et $k \in [\![0, 20[\![$, on calcule $T_2(p)$ et l'on effectue ensuite une régression linéaire par moindres carrés. Cette étude expérimentale (voir figure 1) nous donne :

$$T_2(p) \approx b_2 p^{\alpha_2} \text{ où } \alpha_2 \approx 1,005 \text{ et } b_2 = 10^{\beta_2} \approx 10^{-7,16} \approx 7,15 \times 10^{-8}.$$

- La constante b_2 ne pouvait de toute façon pas être prédite par notre étude théorique.
- On est (à peu de choses près) sur du $O(p)$. La différence vient d'erreurs liées à notre modèle et aux erreurs expérimentales.
- Si on prend comme référence qu'une instruction machine prend un temps $t = 10^{-9}s$, l'étude expérimentale semble montrer que $T_2(p) \approx 120p \times t$.

Remarque 3.5.1. Ce n'est pas toujours la quantité p qui est pertinente, mais plutôt la taille qu'occupe l'entier p en mémoire, qui est de l'ordre de $\log_2(p)$.

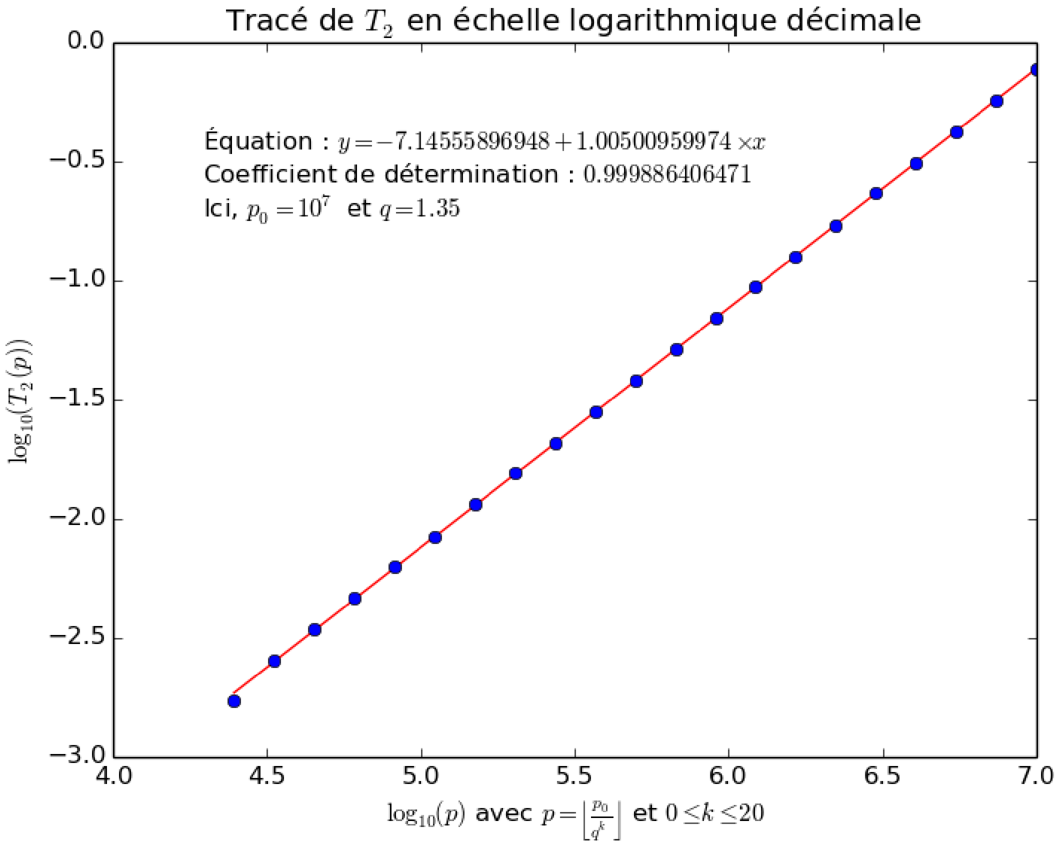

FIGURE 1 – Tracé expérimental de T_2 en échelle logarithmique décimale.

3.6 Étude théorique de `premierprobable3`

```
def expo3(p):
    """Renvoie 2**(p-1) mod p par exponentiation rapide"""
    y = 1
    n = p-1
    x = 2
    while n > 0:
        # Invariant : y * (x**n) = 2**(p-1) mod p
        # Variant : n
        if n % 2 != 0: # n est impair
            y = (y * x) % p
            n = n - 1
        # n est pair et y * (x**n) = 2**(p-1) mod p
        x = (x * x) % p
        n = n // 2
```

```python
    # n == 0 et y = 2 ** (p-1) modulo p
    return y

def premierprobable3(p):
    return expo3(p) == 1
```

On effectue un nombre constant d'opérations de temps constant, puis une boucle `while` effectuant à chaque tour un nombre borné d'opérations de temps constant, puis une comparaison.

Toute la difficulté est d'estimer le nombre de tours effectués par la boucle.

- Il est inférieur ou égal à p (n est un variant de la boucle et est initialisé à $p-1$), donc $T_3(p) = O(p)$.
- En fait n est divisé (au moins) par deux à chaque tour de boucle. Donc $\lfloor \log_2(n) \rfloor$ est un variant de boucle.
- Donc on effectue au plus $1 + \lfloor \log_2(p-1) \rfloor = O(\log_2(p)) = O(\log p)$ tours de boucle.

Donc
$$T_3(p) = O(\log p).$$

3.7 Comparaison avec l'étude expérimentale

Pour $p = \left\lfloor \frac{p_0}{q^k} \right\rfloor$, avec $q = 140$, $p_0 = 10^{140}$ (! !) et $k \in [\![0, 20[\![$, on calcule $T_3(p)$ et l'on effectue ensuite une régression linéaire par moindres carrés. Cette étude expérimentale (voir figure 2) nous donne : $T_3(p)$ et $\log p$ une relation affine :

$$T_3(p) \approx \alpha_3 \log_{10} p + \beta_3.$$

Donc $T_3(p)$ semble bien être un $O(\log p)$.

NB : en général, quand on note $\log$, il faut comprendre

— $\ln$ si c'est dans un texte de mathématiques en anglais ;

— $\log_{10}$ si c'est dans un texte de mathématiques ou de physique en français ;

— $\log_2$ si c'est dans un texte d'informatique.

Mais de toute façon, ici on cela n'a pas d'importance car pour tout $x > 0$,

$$\ln x = \ln 2 \, \log_2 x = \ln 10 \, \log_{10} x.$$

Donc
$$O(\ln n) = O(\log_2 n) = O(\log_{10} n).$$

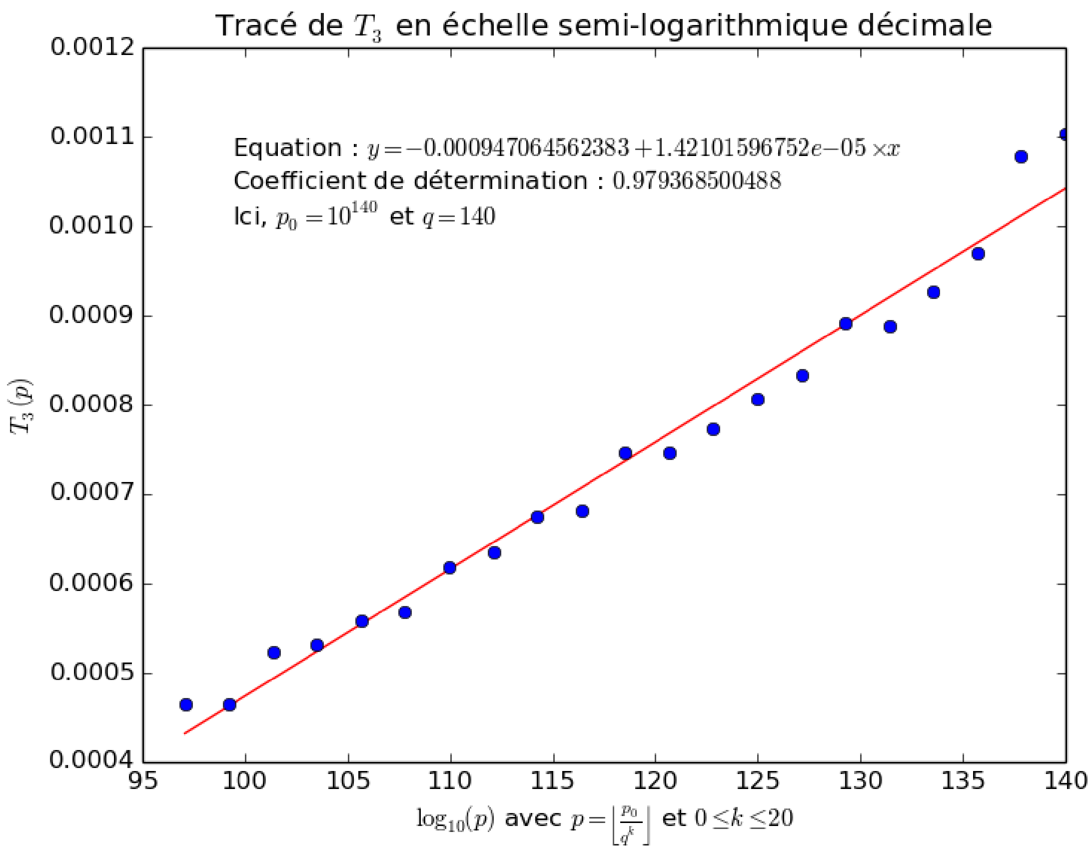

FIGURE 2 – Tracé expérimental de T_3 en échelle semi-logarithmique décimale.

3.8 Étude théorique de `premierprobable1`

```python
def expo1(p):
    """Renvoie 2**(p-1) mod p"""
    y = 1
    for i in range(p-1):
        y = 2*y
    return y % p

def premierprobable1(p):
    return expo1(p) == 1
```

- On considère que les temps d'exécution d'une affectation, d'une opération arithmétique et d'une comparaison sont constants.
- Dans `premierprobable1(p)`, on effectue une affectation, un calcul de reste et une comparaison ainsi que $(\mathtt{p}-1)$ tours de boucle, avec un nombre constant d'opérations de temps constant à chaque tour.

- Donc le temps d'exécution de `premierprobable1(p)` est un $O(\mathtt{p})$.

3.9 Étude expérimentale de `premierprobable1`

Pour $p = \left\lfloor \frac{p_0}{q^k} \right\rfloor$, avec $q = 1,35$, $p_0 = 3 \times 10^5$ et $k \in [\![0, 20[\![$, on calcule $T_1(p)$ et l'on effectue ensuite une régression linéaire par moindres carrés. Cette étude expérimentale (voir figure 1) nous donne :

$$T_1(p) \approx b_1 p^{\alpha_1} \text{ où } \alpha_1 \approx 1,76 \text{ et } b_1 = 10^{\beta_1} \approx 10^{-9,52} \approx 3,02 \times 10^{-10}$$

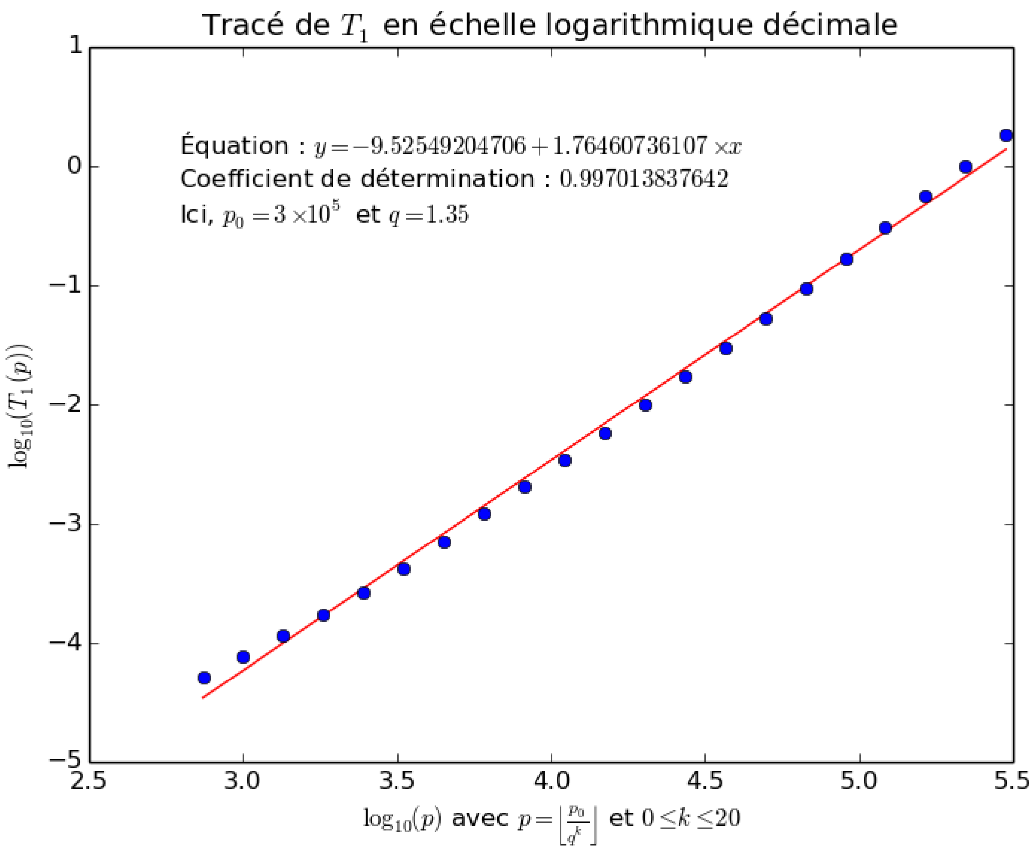

FIGURE 3 – Tracé expérimental de T_1 en échelle logarithmique décimale.

Il semble alors que $T_1(p)$ n'est pas en $O(p)$! ! !
Les causes possibles :

- Cela peut provenir d'erreurs expérimentales. Mais ici, l'erreur paraît vraiment élevée.

- Le modèle peut être erroné : les opérations sont-elles vraiment de temps constant ?

Ici, les nombres manipulés sont très grands puisqu'on calcule 2^{p-1}. Or, 2^{p-1} possède environ $\log_{10}(2^{p-1}) = (p-1)\log_{10} 2$ chiffres.

Une hypothèse : chaque multiplication de y par 2 coûte un temps proportionnel à la longueur de y, est en temps $C \log \mathtt{y}$.

Au i^e tour de boucle, y vaut 2^i, donc le temps de calcul de la multiplication est $Ci \log 2$. Donc le temps de calcul de l'ensemble de la boucle est

$$\sum_{i=0}^{p-2} Ci \log 2 \leq (p-1)C(p-2)\log 2 \leq p^2 C \log 2$$

Donc le temps de calcul de l'ensemble de la boucle est un $O(p^2)$.

Il reste à estimer le temps de calcul du reste de 2^{p-1} dans la division par p. On peut penser que ce temps de calcul est au plus proportionnel au nombre de chiffres de 2^{p-1} multiplié par le nombre de chiffres de p, donc est dominé par $p \log p$, donc par p^2.

Au final, le temps de calcul serait donc un $O(p^2)$.

Retour sur l'expérimentation :

- On trouve un $O(p^{1,76})$, ce qui est déjà plus proche de $O(p^2)$.
- Si on regarde la courbe expérimentale, on a l'impression qu'elle est plutôt convexe. Comme ce qui nous intéresse est la pente pour p élevé, on peut penser que nous l'avons expérimentalement sous-estimée.

4 Étude dans le pire des cas

Revenons sur la recherche d'un élément dans un tableau (fait dans le cours sur les tableaux).

```python
def appartient(e, t):
    """Retourne un booléen disant si e appartient à t
       Précondition : t est un tableau"""
    for x in t:
        # Invariant : e n'est pas positionné dans t avant x
        if e == x:
            return True # On a trouvé e, on s'arrête
    return False
```

Si l'on cherche à étudier la complexité d'une exécution `appartient(e, t)`, on tombe vite sur un écueil : le nombre de tour effectués dans la boucle `for` est variable ! Il dépend en effet de la position de la première occurence de `e` dans `t`.

Dans ce cas là, la notion de complexité n'est pas bien définie. On peut alors s'intéresser à d'autres notions de complexité.

Complexité en moyenne – On suppose que `e` ou `t` sont tirés selon une certaine loi de probabilité, et l'on compte « en moyenne [4] » le nombre d'opérations effectuées. La loi utilisée est une hypothèse de modélisation. Par exemple, on peut supposer que `e` est un élément de `t` et que sa première occurrence est distribuée uniformément dans `t`. Ce type de calcul est souvent compliqué à mener (la modélisation en elle même n'étant souvent pas évidente), nous ne nous y intéresseront pas.

Complexité dans le pire des cas – On suppose que `e` n'est pas dans `t` (ou qu'il apparaît uniquement en dernière position), on peut dans ce cas assez facilement calculer le nombre d'opérations : il y a `len(t)` tours de boucle, dans chaque tour de boucle il y a une comparaison. La complexité *dans le pire des cas* est donc en $O($`len(t)`$)$.

Généralement, on vous signalera que l'on étudie une complexité dans le pire des cas. Si ce n'est pas précisé, vous devez signaler que vous prenez l'initiative d'étudier le pire des cas, tout autre choix étant souvent déraisonnable.

5 Temps d'exécution des instructions élémentaires en Python

Il est parfois difficile de savoir quel nombre d'opérations `Python` effectue pour réaliser une instruction précise. En première approche, vous pouvez considérer ceci.

5.1 Opérations en temps constant

- Affecter une variable à un objet.
- Accéder (en lecture et en écriture) à un élément d'une liste `Python`, d'une chaîne de caractère ou d'un tuple.
- Calculer la longueur d'une liste `Python`, d'une chaîne de caractères ou d'un tuple par la fonction `len`.
- Créer une liste, chaîne de caractères ou un tuple vide.

Les opérations suivantes peuvent être considérées comme étant en temps constant, même si elles ne le sont pas formellement.

- Opérations usuelles sur les entiers (sauf sur de très très grands nombres ; la complexité réelle dépend des nombres de bits dans les écritures binaires des entiers considérés).
- Ajouter un élément à une liste par la méthode `append()` (temps constant amorti).

4. D'un point de vue probabiliste, on calcule une espérance.

5.2 Les autres

- Concaténer deux listes `Python`, chaînes de caractères ou tuple avec `+` : en $O(n)$ où n est la taille de la plus grande des deux listes.
- Extraire une tranche d'une liste `Python`, d'une chaîne de caractères ou d'un tuple : en $O(k)$ où k est la longueur de la tranche.
- Copier une liste `Python`, chaînes de caractères ou tuple avec `copy()` : en $O(n)$ où n est la taille de la plus grande des deux listes.

6 Complexités usuelles

Voir la table 3.

TABLE 3 – Temps de calculs pour des instructions de 10^{-9}s.

	$f(n)$					
n (log$_{10}$)	$\log_2 n$ (log$_{10}$ de la durée en secondes)	n	$n \log n$	n^2	n^3	2^n
1	-8.5	-8.0	-7.5	-7.0	-6.0	$-6,0$
2	-8.2	-7.0	-6.2	-5.0	-3.0	$21,1$
4	-7.9	-5.0	-3.9	-1.0	3.0	
5	-7.8	-4.0	-2.8	1.0	6.0	
6	-7.7	-3.0	-1.7	3.0		
7	-7.6	-2.0	-0.6	5.0		
8	-7.6	-1.0	0.4			
9	-7.5	0.0	1.5			
12	-7.4	3.0	4.6			
18	-7.2	6.0				
1000	$-5,5$					

7 Conclusion

1. On doit étudier la complexité du point de vue expérimental et théorique.
2. On a besoin d'un modèle. Lequel prendre ?
   — Le plus simple qui convient !
   — En général, le «modèle usuel du programmeur».

— En cas de manipulation de grands nombres : prendre un modèle où le coût des opérations arithmétiques dépend de la taille des opérandes.

3. Pour l'estimation théorique, on se contente de donner une estimation asymptotique.

8 Exercices

Exercice 8.0.1. Étudier la complexité théorique de la fonction `maxi` du cours nº 4.

Exercice 8.0.2. Étudier les complexité théoriques (dans le pire des cas) des fonctions `appartient` et `appartient_dicho` du cours nº 4. Les comparer.

Exercice 8.0.3. Étudier la complexité théorique dans le pire des cas de la fonction `recherche` du cours nº 5. On pourra être amené à la reformuler légèrement.

Exercice 8.0.4. Étudier la complexité théorique de la fonction `conv_b2` du cours nº 7.

Exercice 8.0.5. Étudier les complexités théoriques des fonctions `calc_b2_naif` et `calc_b2_horner` du cours nº 7. Les comparer.

Exercice 8.0.6. On considère la suite u à valeurs dans $[\![0; 64\,007[\![$ définie par

$$u_0 = 42, \quad \forall n \in \mathbb{N},\ u_{n+1} = 15\,091 u_n\ [64\,007],$$

ainsi que, pour tout $n \in \mathbb{N}$, $S_n = \sum_{k=0}^{n} u_k$.

On propose l'algorithme suivant pour calculer les valeurs de S.

```
def u(n):
    """u_n, n : entier naturel"""
    v = 42
    # Inv : v = u_0
    for k in range(n):
        # Inv : v = u_k
        v = 15091 * v % 64007
        # Inv : v = 15091*u_k % 64007 = u_k+1
    # Inv : au dernier tour, k = n-1, donc v = u_n
    return v

def S(n):
    """u_n, n : entier naturel"""
    s = u(0)
    # Inv : s = S_0
    for k in range(n):
        # Inv : s = S_k
```

```
        s = s + u(k+1)
        # Inv : v = S_k+u_k+1 = S_k+1
    # Inv : au dernier tour, k = n-1, donc s = S_n
    return s
```

1. Étudier les complexités des fonctions `u` et `S`, en fonction de n.
2. Écrire une fonction donnant la valeur de S_n en temps $O(n)$.

Exercice 8.0.7. On considère la suite u définie par

$$u_0 = 2, \quad \forall n \in \mathbb{N},\ u_{n+1} = u_n^2.$$

On se considère la fonction suivante, permettant de calculer les valeurs de u.

```
def u(n):
    """u_n, n : entier naturel"""
    v = 2
    # Inv : v = u_0
    for k in range(n):
        # Inv : v = u_k
        v = v*v
        # Inv : v = u_k**2 = u_k+1
    # Inv : au dernier tour, k = n-1, donc v = u_n
    return v
```

Pour étudier le temps d'exécution d'une fonction, on pourra utiliser le morceau de code suivant.

```
import timeit

REPEAT=3

def duree(f, x):
  """Calcule le temps mis par Python pour calculer f(x).
  Cette fonction effectue en fait le calcul de f(x) REPEAT fois
  et garde la valeur la plus petite
  (l'idée est d'éliminer les éventuelles perturbations provoquées
  par d'autres processus tournant sur la machine)"""
  t = timeit.Timer(stmt=lambda : f(x))
  time = min(t.repeat(REPEAT, number=1))
  return time
```

1. Étudier en fonction de n la complexité asymptotique de la fonction `u`, dans le modèle standard.
2. Tracer les temps de calculs de u_k pour $k \in [\![0;30[\![$ par la fonction `u`. Discuter le résultat.
   Indication : on pourra utiliser une échelle semi-logarithmique.

3. Proposer un modèle de complexité plus réaliste et étudier dans ce modèle n la complexité asymptotique de la fonction `u`. On pourra déterminer explicitement u_n.