Anlayse des algorithmes 4

- Terminaison.
- Correction partielle.
- Correction totale.
- Variant. Invariant.
- Complexité.

4.1 Anlayse des algorithmes

4.1.1 Définitions

Définition – Terminaison d'un algorithme

Prouver la terminaison d'un algorithme signifie montrer que cet algorithme se terminera en un temps fini. On utilise pour cela un **variant de boucle**.

Définition – Correction d'un algorithme

Un algorithme est dit (partiellement) correct s'il est correct dès qu'il termine.

Prouver la correction d'un algorithme signifie montrer que cet algorithme fournit bien la solution au problème qu'il est sensé résoudre. On utilise pour cela un **invariant de boucle**.

Définition – Invariant de boucle

Un invariant de boucle est une propriété dépendant des variables de l'algorithme, qui est vérifiée à chaque passage dans la boucle.

Définition – 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 :

- le temps nécessaire à l'exécution de l'algorithme ;
- la mémoire nécessaire à l'exécution de l'algorithme (en plus des données d'entrée).

4.1.2 Un exemple ...

Objectif

L'objectif est ici de montrer la nécessité d'utiliser un invariant de boucle.
Pour cela, on propose la fonction suivante sensée déterminer le plus petit entier `n` strictement positif tel que $1 + 2 + \ldots + n$ dépasse strictement la valeur entière strictement positive `v`. Cette fonction renvoie-t-elle le bon résultat ? Desfois ? Toujours ?

```
1 def foo(v:int) -> int:
2     r = 0
3     n = 0
4     while r < v :
5         n = n+1
6         r = r+n
```

```
7     return n
```

Montrer intuitivement que `foo(v)` se termine pour $v \in \mathbb{N}^*$.

L'algorithme se terminera si on sort de la boucle `while`. Il faut pour cela que la condition `r<v` devienne fausse (cette condition est vraie initialement). Pour cela, il faut que `r` devienne supérieure ou égale à `v` dont la valeur ne change jamais.

`n` étant incrémenté de 1 à chaque itération, la valeur de `r` augmente donc à chaque itération. Il y aura donc un rang `n` au-delà duquel `r` sera supérieur à `v`. L'algorithme se termine donc.

Que renvoie `foo(9)` ? Cela répond-il au besoin ?

Début de la i^e itération	r	n	r < v
Itération 1	0	0	0 < 9 ⇒ True
Itération 2	1	1	1 < 9 ⇒ True
Itération 3	3	2	3 < 9 ⇒ True
Itération 4	6	3	6 < 9 ⇒ True
Itération 5	10	4	10< 9 ⇒ False

La fonction renvoie 4. On a $1 + 2 + 3 + 4 = 10$. On dépasse strictement la valeur 9. La fonction répond au besoin dans ce cas.

Que renvoie `foo(10)` ? Cela répond-il au besoin ?

Début de la i^e itération	r	n	r < v
Itération 1	0	0	0 < 10 ⇒ True
Itération 2	1	1	1 < 10 ⇒ True
Itération 3	3	2	3 < 10 ⇒ True
Itération 4	6	3	6 < 10 ⇒ True
Itération 5	10	4	10 < 10 ⇒ False

La fonction renvoie 4. On a $1 + 2 + 3 + 4 = 10$. On ne dépasse pas strictement la valeur 10. La fonction ne répond pas au besoin dans ce cas.

Résultat –

La fonction proposée ne remplit pas le cahier des charges. Aurait-on pu le prouver formellement ?

4.2 Terminaison d'un algorithme

4.2.1 Variant de boucle

Définition – Variant de boucle

Un variant de boucle permet de prouver la terminaison d'une boucle conditionnelle. Un variant de boucle est une **quantité entière positive** à l'entrée de chaque itération de la boucle et qui **diminue strictement à chaque itération**.

Théorème –

Si une boucle admet un variant de boucle, elle termine.

Propriété –

Un algorithme qui n'utilise ni boucles inconditionnelles (boucle `for`) ni récursivité termine toujours. Ainsi, la question de la terminaison n'est à considérer que dans ces deux cas.

Reprenons l'exemple précédent.

```
def foo(v:int) -> int:
    r = 0
    n = 0
    while r < v :
        n = n+1
        r = r+n
    return n
```

Dans cet exemple montrons que la quantité $u_n = v - r$ est un variant de boucle :

- initialement, $r = 0$ et $v > 0$; donc $u_0 > 0$;
- à la fin de l'itération n, on suppose que $u_n = v - r > 0$ et que $u_n < u_{n-1}$;
- à l'itération $n + 1$:
  - cas 1 : $r \geq v$. Dans ce cas, n et r n'évoluent pas l'hypothèse de récurrence reste vraie. On sort de la boucle `while`. L'algorithme termine,
  - cas 2 : $r < v$. Dans ce cas, à la fin de l'itération $n + 1$, montrons que $u_{n+1} < u_n$: $u_{n+1} = v - (r + n + 1) = u_n - n - 1$ soit $u_{n+1} = u_n - n - 1$ et donc $u_{n+1} < u_n$. L'hypothèse de récurrence est donc vraie au rang $n + 1$.

Au final, $u_n = v - r$ est donc un variant de boucle et la boucle se termine.

4.2.2 Un second exemple ressemblant...[1]

1: https://marcdefalco.github.io/pdf/complet_python.pdf

Considérons l'algorithme suivant qui, étant donné un entier naturel n strictement positif (inférieur à 2^{30}), détermine le plus petit entier k tel que $n \leq 2^k$.

```
def plus_grande_puissance2(n):
    k = 0
    p = 1
    while p < n:
        k = k+1
        p = p*2
    return k
```

[1] Dans l'exemple précédent, la quantité $n - p$ est un variant de boucle :

- au départ, $n > 0$ et $p = 1$ donc $n - p \geq 0$;
- comme il s'agit d'une différence de deux entiers, c'est un entier. Et tant que la condition de boucle est vérifiée $p < n$ donc $n - p > 0$.
- lorsqu'on passe d'une itération à la suivante, la quantité passe de $n - p$ à $n - 2p$ or $2p - p > 0$ car $p \geq 1$. Il y a bien une stricte diminution.

[2] Montrons que, la quantité $u_j = n - p$ est un variant de boucle :

- intialement, $n > 0$ et $p = 1$ donc $n - p \geq 0$;
- à la fin de l'itération j, on suppose que $u_j = n - p > 0$ et$u_j < u_{j-1}$;

- à la fin de l'itération suivante, $u_{j+1} = n - 2p = u_j - p$. p est positif donc u_{j+1} est un entier et $u_{j+1} < u_j$. Par suite, ou bien $u_{j+1} < 0$ c'est à dire que $n - p < 0$ soit $p > n$. On sort donc de la boucle. Ou bien, $u_{j+1} > 0$, et la boucle continue.

$n - p$ est donc un variant de boucle.

4.3 Correction d'un algorithme

4.3.1 Invariant de boucle

Méthode –

Pour montrer qu'une propriété est un invariant de boucle dans une boucle `while` :

- le propriété doit être vérifiée avant d'entrer dans la boucle ;
- la propriété doit être vraie en entrée de boucle ;
- la propriété doit être vraie en fin de boucle.

Reprenons un des exemples précédents. Reconsidérons l'algorithme suivant qui, étant donné un entier naturel n strictement positif (inférieur à 2^{30}), détermine le plus petit entier k tel que $n \leq 2^k$.

```
1 def plus_grande_puissance2(n):
2     k = 0
3     p = 1
4     while p < n:
5         k = k+1
6         p = p*2
7     return k
```

Montrons que la propriété suivante est un invariant de boucle : $p = 2^k$ et $2^{k-1} < n$.

- **Initialisation :** à l'entrée dans la boucle $k = 0$ et $p = 1$, $n \in \mathbb{N}^*$
  - d'une part on a bien $1 = 2^0$;
  - d'autre part $2^{-1} < n$.
- On considère que la propriété est vraie au n^e tour de boucle c'est à dire $p = 2^k$ et $2^{k-1} < n$.
- Au tour de boucle suivant :
  - **ou bien** $p >= n$. Dans ce cas, on sort de la boucle et on a toujours $p = 2^k$ et $2^{k-1} < n$ (propriété d'invariance). La propriété est donc vraie au tour $n + 1$.
  - **ou bien** $p < n$. Dans ce cas, il faut montrer que $p = 2^{k+1}$ et $2^k < n$. Etant entrés dans la boucle, $p < n \Rightarrow 2^k < n$. De plus, en fin de boucle, $p \longrightarrow p * 2$ et $k \longrightarrow k + 1$. On a donc $p \longleftarrow 2^k * 2 = 2^{k+1}$.

La propriété citée est donc un invariant de boucle.

4.3.2 Un « contre exemple »

Reprenons le tout premier exemple où on cherche le plus petit entier `n` strictement positif tel que $1 + 2 + ... + n$ dépasse strictement la valeur entière strictement positive `v` .

```python
def foo(v:int) -> int:
    assert v>0
    r = 0
    n = 0
    while r <= v :
        n = n+1
        r = r+n
    return n
```

La propriété suivante est-elle un invariant de boucle : $r = \sum_{i=0}^{n} i$ et $\sum_{i=0}^{n-1} i \leq v, n \in \mathbb{N}^*$?

La réponse est directement NON, car la phase d'initialisation n'est pas vérifiée car $n = 0$ et $n \notin \mathbb{N}^*$. Cela signifie donc que l'algorithme proposé ne répond pas au cahier des charges.

Modifions alors l'algorithme ainsi.

```python
def foo2(v:int) -> int:
    assert v>0
    r = 1
    n = 1
    while r <= v :
        n = n+1
        r = r+n
    return n
```

Montrons que la propriété suivante est un invariant de boucle : $r = \sum_{i=0}^{n} i$ et $\sum_{i=0}^{n-1} i \leq v$, $n \in \mathbb{N}^*$.

- **Initialisation :** à l'entrée dans la boucle $r = 1$ et $n = 1$, $n \in \mathbb{N}^*$
  - d'une part on a bien $r = \sum_{i=0}^{1} i = 1$;
  - d'autre part $\sum_{i=0}^{0} i = 0 < v$ et $v > 0$ (spécification de la fonction).
- On considère que la propriété est vraie au début du n^e tour de boucle c'est-à-dire $r = \sum_{i=0}^{n} i$ et $\sum_{i=0}^{n-1} i \leq v$.
- Á la fin du n^e tour de boucle, $n_{n+1} = n_n + 1$ et $r_{n+1} = r_n + n_{n+1} = r_n + n_n + 1 = \sum_{i=0}^{n} (i) + n_n + 1 = \sum_{i=0}^{n+1} i$ (car $n_n = n$). On a alors,
  - ou bien $r_{n+1} > v$ et on sort de la boucle ; on peut renvoyer n.
  - ou bien $r_{n+1} \leq v$ et donc $\sum_{i=0}^{n} i \leq v$.

La propriété citée est donc un invariant de boucle.

4.4 Complexité algorithmique

4.4.1 Préliminaire : parlons du temps. . .

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

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

[*]. D'ailleurs, c'est l'unité de temps du système international

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.* π s $\approx$ 1 nano-siècle.

$\log_{10}(t)$	Durée	$\log_{10}(t)$	Durée
$-9,0$	exécution d'une instruction	$10,5$	un millénaire
$0,0$	une seconde	$13,1$	âge de la maîtrise du feu
$1,8$	une minute	$13,5$	un million d'années
$3,6$	une heure	$14,0$	âge de Lucy
$4,9$	un jour	$16,5$	un milliard d'années
$5,8$	une semaine	$17,0$	âge de la vie sur Terre
$6,4$	un mois (30 jours)	$17,6$	âge de l'univers
$7,5$	un an	$34,0$	temps avant extinction des dernières étoiles
$9,5$	un siècle		

4.4.2 Rappels d'arithmétique : le petit théorème de Fermat

Théorème –

Théorème 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 considérera 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 ».

4.4.3 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):
        #Invariant : x=2**i
        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):
        #Invariant : x=2**i mod p
        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*.

```python
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.

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

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

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

4.4.4 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 `premierprobablei(p)`.

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

$\log_{10}(p)$	$\log_{10}(T_1(p))$	$\log_{10}(T_2(p))$	$\log_{10}(T_3(p))$
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 [†] 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. Il ne reste plus qu'à regarder, pour un programme donné, combien d'instructions élémentaires il effectue.

4.5 Étude théorique

4.5.1 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.

4.5.2 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 [‡] et le seront encore probablement dans 50 ans.

4.5.3 Étude théorique de `premierprobable2`

```
1 def expo2(p):
2     """Renvoie 2**(p-1) mod p"""
3     x = 1
4     for i in range(p-1):
5         x = (2*x) % p
6     return x
7
8 def premierprobable2(p):
9     return expo2(p) == 1
```

[†] En inventant un algorithme ou en combinant des algorithmes existants

[‡] 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.

Comment estimer le temps d'exécution de `premierprobable2(p)` ?

Les opérations effectuées dans `premierprobable2(p)` sont les suivantes.

- Une affectation.
- Une boucle effectuant `p` -1 tours. À chaque tour on effectue les opérations suivantes :
  - une multiplication ;
  - 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 $p_0 \geq 1$:

On a vu que	$T_2(p) = c_1 + (p-1)(c_2+c_3+c_1) + c_4$.
De plus	$p_0c_1 > c_1, p_0(c_2+c_3+c_1) > (p-1)(c_2+c_3+c_1)$ et $p_0c_4 > c_4$.
Au final	$T_2(p) \leq c_1p_0 + p_0(c_2+c_3+c_1) + c_4p_0$
et	$T_2(p) \leq (2c_1+c_2+c_3+c_4)p_0$.

On a donc montré qu'il existait un entier p_0 et un réel $C > 0$ tel que $\forall p \geq p_0 \quad T_2(p) \leq Cp_0$.

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

Définition –

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 $\left|\frac{T_2(p)}{p}\right| \leq C$ et $T_2(p) = O(p)$.

Remarque

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.

4.5.4 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 $(p-1)$ 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 $O(p)$.

4.5.5 Étude expérimentale de `premierprobable2`

Pour $p = \left\lfloor \frac{p_0}{q^k} \right\rfloor$, 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.

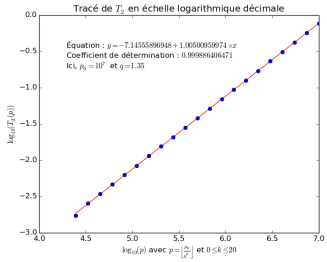

FIGURE 4.1 – Tracé expérimental de T_2 en échelle logarithmique décimale.

Cette étude expérimentale (voir figure ci-contre)nous donne :$T_2(p) \approx b_2 p^{\alpha_2}$ où $\alpha_2 \approx 1,005$ et $b_2 = 10^{\beta_2} \approx 10^{-7,16} \approx 6,9 \times 10^{-8}$.

- 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 $T_2(p) \approx 70p \times t$.

Remarque

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 $\log_2(p)$.

4.5.6 Étude théorique de `premierprobable3`

```python
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
```

```python
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 $T_3(p) = O(\log p)$.

4.5.7 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 ci-contre) 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 $O(\log p)$.

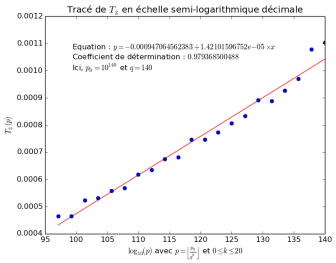

Figure 4.2 – Tracé expérimental de T_3 en échelle semi-logarithmique décimale.

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

4.5.8 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
```

```python
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 (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 $O(p)$.

4.5.9 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 ci-contre) nous donne : $T_1(p) \approx b_1 p^{\alpha_1}$ où $\alpha_1 \approx 1,76$ et $b_1 = 10^{\beta_1} \approx 10^{-9,52} \approx 3,02 \times 10^{-10}$.

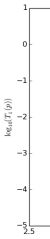

FIGURE
échelle

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 $C \log y$.

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.

4.6 Étude dans le pire des cas

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

```
1 def appartient(e, t):
2     """Retourne un booléen disant si e appartient à t
3        Précondition : t est un tableau"""
4     for x in t:
5         # Invariant : e n'est pas positionné dans t avant x
6         if e == x:
7             return True # On a trouvé e, on s'arrête
8     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 [§] » 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.

4.7 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.

4.7.1 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).

4.7.2 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.

[§] D'un point de vue probabiliste, on calcule une espérance.

4.8 Complexités usuelles

Voir la table 4.1.

Table 4.1 – Temps de calculs pour des instructions de 10^{-9}s.

	$f(n)$					
n	$\log_2 n$ logarithmique	n linéaire	$n \log n$ quasi-linéaire	n^2 quadratique	n^3 cubique	2^n exponentielle
($\log_{10}$)	($\log_{10}$ de la durée en secondes)					
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					

4.9 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.