Anlayse des algorithmes 4

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).

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

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.

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.

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

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 $n \leq 2^k$.

```python
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 $n \leq 2^k$.

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

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

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

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.

```
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 Mise en évidence du problème

Prenons l'exemple de la recherche d'un élément dans une liste :

```
def is_number_in_list(nb,tab):
    """Renvoie True si le nombre nb est dans la liste de nombres tab
    Entrées :
      * nb, int -- nombre entier
      * tab, list -- liste de nombres entiers
    """
    for i in range(len(tab)):
        if tab[i]==nb:
            return True
    return False
```

Exemple –

À partir de l'algorithme précédent, évaluer :

- le nombre de tour de boucles dans le pire des cas ;
- le nombre de tour de boucles dans le meilleur des cas.

```
def is_number_in_list_dicho(nb,tab):
    """
    Recherche d'un nombre par dichotomie dans un tableau trié.
    Renvoie l'index si le nombre nb est dans la liste de nombres tab.
    Renvoie None sinon.
    Entrées :
     * nb,int -- nombre entier
     * tab,list -- liste de nombres entiers triés
    """
    g, d = 0, len(tab)-1
    while g <= d:
        m = (g + d) // 2
        if tab[m] == nb:
            return m
        if tab[m] < nb:
```

```
            g = m+1
        else:
            d = m-1
    return None
```

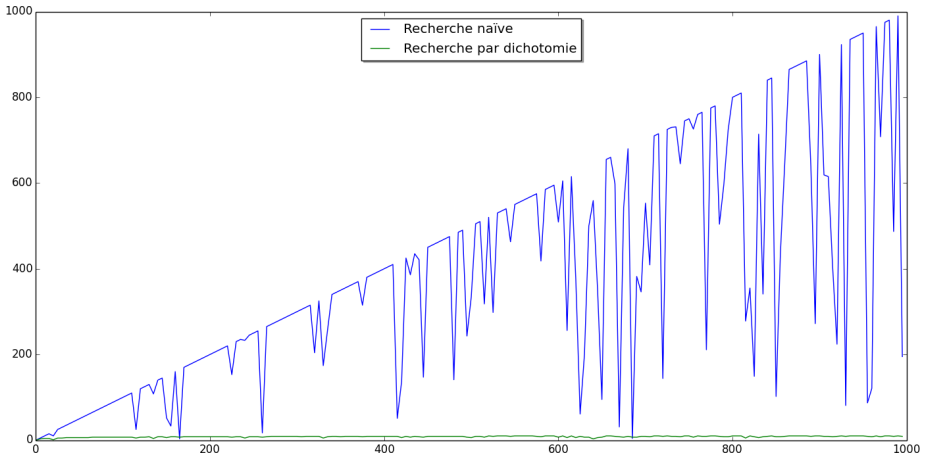

FIGURE 4.1 – Évolution du nombre d'opérations pour rechercher un nombre dans une liste de 1 à 1000 nombres

4.5 Complexité des algorithmes

4.5.1 Présentation

Il existe souvent plusieurs façons de programmer un algorithme. Si le nombre d'opérations à effectuer est peu important et les données d'entrée de l'algorithme sont de faibles tailles, le choix de la solution importe peu. En revanche, lorsque le nombre d'opérations et la taille des données d'entrée deviennent importants, deux paramètres deviennent déterminants : le temps d'exécution et l'occupation mémoire.

Définition – Complexité en temps

La complexité en temps donne le nombre d'opérations effectuées lors de l'exécution d'un programme. On appelle C_o le coût en temps d'une opération o.

Définition – Complexité en mémoire (ou en espace)

La complexité en mémoire donne le nombre d'emplacements mémoires occupés lors de l'exécution d'un programme.

Remarque

On distingue la complexité dans le pire des cas, la complexité dans le meilleure des cas, ou la complexité en moyenne. En effet, pour un même algorithme, suivant les données à manipuler, le résultat sera déterminé plus ou moins rapidement. Généralement, on s'intéresse au cas le plus défavorable à savoir, la complexité dans le pire des cas.

Définition – Complexité algorithmique – Notation $\mathcal{O}$ – [Bournez]

Soient f et g deux fonctions $f, g : \mathbb{N} \longrightarrow \mathbb{R}_*^+$. On note $f(n) = \mathcal{O}(g(n))$ lorsqu'il existe $c \in \mathbb{R}^+$ et $n_0 \in \mathbb{N}$ tels que pour tout $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

Intuitivement, cela signifie que f est inférieur à g à une constante multiplicative près pour les données suffisamment grandes.

Exemple –

Par ordre de complexité croissante on a :

- $\mathcal{O}(1)$: algorithme s'exécutant en temps constant, quelle que soit la taille des données ;
- $\mathcal{O}(\log(n))$: algorithme rapide (complexité logarithmique) (Exemple : recherche par dichotomie dans un tableau trié) ;
- $\mathcal{O}(n)$: algorithme linéaire ;
- $\mathcal{O}(n \cdot \log(n))$: algorithme quasi-linaire ;
- $\mathcal{O}(n^2)$: complexité quadratique ;
- $\mathcal{O}(n^3)$: complexité cubique ;
- $\mathcal{O}(2^n)$: complexité exponentielle.

4.5.2 Coût temporel d'un algorithme et d'une opération

Résultat –

On considère que le coût élémentaire C_e correspond au coût d'une affectation, d'une comparaison ou de l'évaluation d'une opération arithmétique.

Exemple –

Chacune de ces 3 instructions ont le même coût temporel C_e. On a donc $C_{\text{foo}}(n) = 3C_e$.
`foo` prendra (a priori) toujours le même temps d'exécution. On a donc $C_{\text{foo}}(n) = \mathcal{O}(1)$.
On parle de complexité constante.

```python
def foo():
    a=20
    a<=100
    a+a
```

Résultat –

Pour une séquence de deux instructions de coûts respectifs C_1 et C_2, le coût total est de la séquence est de $C_1 + C_2$.

Exemple –

Soit la fonction ci-contre.
Le coût temporel correspond à l'addition du coût élémentaire de l'affectation ajouté au coût de l'affichage. On a donc $C_{\text{foo}}(n) = C_e + C_{\text{print}}$. Le côut du `print` est variable en fonction, par exemple, de la longeur de la chaîne à afficher.

```python
def foo():
    a=20
    print(a)
```

`foo` prendra (a priori) toujours le même temps d'exécution. On a donc $C_{\text{foo}}(n) = \mathcal{O}(1)$. On parle de complexité constante.

Résultat –

Le coût d'un test `if test : inst_1 else : inst_2` est inférieur ou égal au maximum du coût de l'instruction 1 et du coût de l'instruction 2 additionné au coût du test (coût élémentaire).

```
def foo(x) :
    if x<0 :
        x=x+1
        x=x+2
    else :
        x=x+1
```

Exemple –

Soit le programme suivant (sans application réelle) :
La comparaison a un coût élémentaire C_e. Dans le « pire » des cas, on réalise deux additions et deux affectations. Le coût temporel total est est donc $C_{\text{foo}}(n) = C_e + \max(4C_e, 2C_e) = 5C_e$. `foo` prendra (a priori) toujours le même temps d'exécution. On a donc $C_{\text{foo}}(n) = \mathcal{O}(1)$. On parle de complexité constante.

Résultat –

Le coût d'une boucle `for i in range(n) : inst` est égal à : n fois le coût de l'instruction `inst` si elle est indépendante de la valeur de i.

```
def foo(n) :
    res = 0
    for i in range(n) :
        res = res+i
    return res
```

Exemple –

On peut considérer que l'incrémentation de `i` a un coût C_e. $C_{\text{foo}}(n) = C_e + n \times 3C_e = \mathcal{O}(n)$. La durée de l'algorithme croît linéairement avec la valeur de n.

Résultat –

Soit la boucle `while cond : inst`, la condition `cond` faisant intervenir un variant de boucle. Il est donc possible de connaître le nombre n d'itérations de la boucle.

Résultat –

Dans la pratique, on cherche toujours à **majorer** le coût temporel d'un algorithme. En conséquences :

- il est inutile de compter exactement le nombre d'opérations ;
- il « suffit » de se se placer dans le pire des cas et de compter le nombre de fois qu'est réalisée l'opération se réalisant le plus de fois.

4.5.3 Exemple

Exemple – Calcul de factorielle

Complexité en mémoire C_M : lors de l'exécution du programme, il sera nécessaire de stocker les variables suivantes : `n`, `res`, `i`.

```
def factorielle(n) :
    if n == 0 :
        return 1
    else :
        i = 1
        res = 1
        while i <= n :
            res = res * i
            i = i + 1
    return res
```

La complexité en mémoire est donc constante : $C_M = \mathcal{O}(1)$.
Complexité en temps C_T (1)
Compte-tenu d'un des réultats précédents, dans le pure des cas, on est dans le `else`. La boucle `while` est réalisée n fois ; donc on peut directement dire que que $C_T(n)$ est un $\mathcal{O}(n)$.
Complexité en temps C_T (2) Une justification plus exhaustive serait la suivante.
La première comparaison a un coût élémentaire C_e.
Pour $n = 0$ le coût du retour est C_r.
Pour $n \neq 0$:

- les deux affectations ont un coût respectif C_e ;
- la boucle tant que sera réalisée n fois. Pour chaque itération,
  - la multiplication ainsi que l'affectation ont un chacun un coût C_e ;
  - l'incrémentation et l'affectation ont chacun un coût C_e ;
- le coût du retour est C_r.

En conséquence, la complexité en temps s'élève à :

$$C_T(n) = C_e + \max\left(C_r; C_e + C_e + n\left(4C_e\right) + C_r\right)$$

Ainsi $C_T(n) = C_e(3 + 4n) + C_r$ et $C_T(n) \underset{+\infty}{\sim} 4C_e n$ lorsque n tend vers l'infini. On parle d'une complexité algorithmique linéaire, notée $\mathcal{O}(n)$.

Il est fréquent que la complexité en temps soit améliorée au prix d'une augmentation de la complexité en espace, et vice-versa. La complexité dépend notamment :

- de la puissance de la machine sur laquelle l'algorithme est exécuté ;
- du langage et compilateur / interpréteur utilisé pour coder l'algorithme ;
- du style du programmeur.

4.5.4 D'autres exemples

Recherche d'un maximum

Exemple – Recherche d'un maximum

Soit une liste de nombre entiers désordonnés. Comment déterminer le plus grand nombre de la liste ?
Intuitivement, une solution est de parcourir la liste d'élément et de déterminer le plus grand élément par comparaisons successives.
Dans ce cas, le coût temporel est : $C_T(n) = C_e + n\left(2C_e\right)$. Ici encore, la complexité de cet algorithme est linéaire car $C_T(n) \underset{+\infty}{\sim} 2C_e n$.
Par ailleurs, en comptant le nombre de compairaisons, on observe aussi qu'il y a environ n comparaisons. On a donc $C_T(n)$ est en $\mathcal{O}(n)$.

```python
def cherche_max(L : list) ->
    int :
    maxi = L[0]
    for i in range(1,len(L)):
        if tab[i] > maxi :
            maxi = tab[i]
    return maxi
```

Suite

Soit la suite u_n définie par récurrence pour tout $n \in \mathbb{N}^*$ par $\begin{cases} u_1 = 1 \\ u_{n+1} = \dfrac{u_n + 6}{u_n + 2} \end{cases}$.

```python
def un_it (n) :
    if n == 1 :
        return 1
```

```
    else :
        u = 1
        for i in range(2,n+1):
            u = (u+6)/(u+2)
        return u
```

La boucle `cor` s'exécute `n` fois. Cet algorithme est en $\mathcal{O}(n)$.

```
def un_rec (n) :
    if n == 1 :
        return 1
    else :
        return (un_rec(n-1)+6)/(un_rec(n-1)+2)
```

A l'itération `n`, $C(n) = 2 \times C(n-1)$. Il s'agit donc d'une suite géométrique et $C(n) = C(0) \times 2^n$. On a donc une complexité exponentielle en $\mathcal{O}(2^n)$.

```
def un_rec_v2 (n):
    if n == 1 :
        return 1
    else :
        v = un_rec_v2(n-1)
        return (v+6)/(v+2)
```

A l'itération `n`, $C(n) = 1 + C(n-1)$. Il s'agit donc d'une suite arithmétique et $C(n) = C(0) + n$. On a donc une complexité linéaire en $\mathcal{O}(n)$.

Diviser pour régner – recherche dichotomique

```
def recherche_dichotomique(x,
     a):
    g, d = 0, len(a)-1
    while g <= d:
        m = (g + d) // 2
        if a[m] == x:
            return m
        elif a[m] < x:
            g = m+1
        else:
            d = m-1
    return None
```

Exemple –

On peut montrer que la suite $d - g$ décroit strictement (car d décroit et g croit). Dans ce cas, la difficulté consiste en déterminer le nombre de fois que sera exécutée la boucle `while`. On note $C_w = C_e + \max(C_e + C_r; 2C_e + 2C_e; 3C_e + C_e) = C_e + \max(C_e + C_r; 4C_e)$ le coût d'une itération de la boucle `while`.
Au cours de l'algorithme, on va devoir diviser en 2 la taille le tableau jusqu'à ce qu'on trouve (ou pas) l'élément recherché. On cherche donc combien de fois m on peut diviser par 2 la taille du tableau n :

$$\frac{n}{2^m} \geq 1 \iff n \geq 2^m \iff \ln(n) \geq m \ln(2)$$

On parlera ici de complexité logarithmique.

Résultat –

Pour une opération ayant un temps d'exécution de $10^{-9}s$, on peut calculer le temps d'exécution en fonction du nombre de données et de la complexité de l'algorithme :

Données	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(2^n)$
100	$2 \cdot 10^{-9}\ s$	$0,1 \cdot 10^{-6}\ s$	$0,2 \cdot 10^{-6}\ s$	$10 \cdot 10^{-6}\ s$	$1,26765 \cdot 10^{21}\ s$
1 000	$3 \cdot 10^{-9}\ s$	$1 \cdot 10^{-6}\ s$	$3 \cdot 10^{-6}\ s$	$0,001\ s$	$1,0715 \cdot 10^{292}\ s$
10 000	$4 \cdot 10^{-9}\ s$	$10 \cdot 10^{-6}\ s$	$40 \cdot 10^{-6}\ s$	$0,1\ s$	$+\infty$

4.5.5 Trompe l'œil – coûts cachés –

```
def recherche(e,L):
    return e in L
```

```
L1 = L.copy()
```

```
L.append(e)
```

Il faut être vigilant lorsqu'on manipule des tableaux avec Python. En effet, de simples instructions cachent en effet des coûts pas toujours visibles. Ainsi, l'instruction `e in L` est linéaire $\mathcal{O}(n)$ car Python dont inspecter chacun des éléments d'une liste de taille `n` pour savoir si `e` est dedans ou non.

Il en est de même pour l'instruction `copy`.

Concernant la méthode `append` on pourra faire l'hypothèse que cela s'effectue en temps constant ($\mathcal{O}(1)$). Cependant, lorsqu'on ajoute une liste Python préréserve des espaces d'allocations juxtaposés. Tant que la liste ne dépasse pas la taille de l'espace alloué, l'ajout d'un élément se fait à temps constant. Quand cette taille est dépassée, la liste est déplacée à un autre endroit en mémoire où plus d'espace est nécessaire. Cette copie se fait donc en temps linéaire. Ainsi, pour append on parle de temps amorti constant.

4.6 Profiling des algorithmes

Afin d'évaluer la performance des algorithmes, il existe des fonctionnalités permettant de compter le temps consacré à chacune des fonctions ou à chacune des instructions utilisées dans un programme `http://docs.python.org/2/library/profile.html`.

Exemple –

Voici un exemple du crible d'Eratosthène.

```
def crible(n):
    tab=[]
    for i in range(2,n):
        tab.append(i)
    # Liste en comprehension tab=[x for x in range(2,n)]
    for i in range(0,len(tab)):
        for j in range(len(tab)-1,i,-1):
            if (tab[j]%tab[i]==0):
                tab.remove(tab[j])
    return tab

import cProfile
cProfile.run('crible(10000)')
```

`cProfile` renvoie alors le message suivant :

```
 28770 function calls in 1.957 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    1.957    1.957 <string>:1(<module>)
        1    0.420    0.420    1.957    1.957 eratosthene.py:4(crible)
        1    0.000    0.000    1.957    1.957 {built-in method exec}
     9999    0.015    0.000    0.015    0.000 {built-in method len}
```

```
10      9998    0.016    0.000    0.016    0.000 {method 'append' of 'list'
    objects}
11         1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.
    Profiler' objects}
12      8769    1.505    0.000    1.505    0.000 {method 'remove' of 'list'
    objects}
```

On a alors le bilan du temps passé à effectuer chacune des opérations. Ainsi pour améliorer notablement l'algorithme, le plus intéressant serait d'optimiser la méthode `remove`.

Bibliographie

[1] François Denis http://pageperso.lif.univ-mrs.fr/~francois.denis/algoL2/chap1.pdf

[2] Alain Soyeur http://asoyeur.free.fr/

[3] François Morain, Cours de l'Ecole Polytechnique, http://www.enseignement.polytechnique.fr/profs/informatique/Francois.Morain/TC/X2004/Poly/www-poly009.html.

[4] Renaud Kerivent et Pascal Monasse, La programmation pour ... , Cours de l'École des Ponts ParisTech - 2012/2013 http://imagine.enpc.fr/~monasse/Info.

[5] Olivier Bournez, Cours INFO 561 de l'Ecole Polytechnique, Algorithmes et programmation, http://www.enseignement.polytechnique.fr/informatique/INF561/uploads/Main/poly-good.pdf.

[6] Wack et Al., *L'informatique pour tous en classes préparatoires aux grandes écoles*, Editions Eyrolles.