1 Preuves & Complexité

1.1 Preuves

1.1.1 Précondition et postcondition

Voici un algorithme de tri : c'est-à-dire un algorithme qui permet de trier les valeurs d'un tableau d'entiers dans l'ordre croissant.

Algorithm 1: Counting Sort

```
1  def counting_sort(t, borne):
2      n ← len(t)
3      for i ← 0 to borne − 1 do
4          c[i] ← 0
5      for i ← 0 to n-1 do
6          c[t[i]] ← c[t[i]] + 1
7      for i ← 1 to borne-1 do
8          c[i] ← c[i] + c[i − 1]
9      for i ← n − 1 to 0 do
10         c[t[i]] ← c[t[i]] − 1
11         b[c[t[i]]] ← t[i]
12     return b
```

Et, voici son implémentation en python. Si le code se lit bien, on ne comprend pas facilement ce qu'il fait.

```python
def counting_sort(t : list[int], borne: int)->list[int]:
      c = [0 for __ in range(borne)]
      n = len(t)
      b = [0 for __ in range(n)]
      for i in range(len(t)):
            c[t[i]] += 1
      for i in range(1,borne):
            c[i] += c[i-1]
      for i in range(n-1,-1,-1):
            c[t[i]] -= 1
            b[c[t[i]]] = t[i]
      return b
```

Avec une docstring c'est un peu mieux :

```
def counting_sort(t, borne):
      """
      Sorting a list of bounded integers by bound, by creating the array
      c of increasing cumulative frequencies, then copying the
      elements of t into a new list b starting from the end of t at the
      place defined for t[i] by c[t[i]].
      Args :
        t : list of positive int
        borne : an integer greater than all elements of t

      Returns :
                a sorted list of elements of t
      """
        ...
```

Avec des commentaires c'est beaucoup mieux :

```
def counting_sort(t : list[int], borne: int)->list[int]:
      c = [0 for __ in range(borne)]
      n = len(t)
      b = [0 for __ in range(n)]
# n contient la longueur de la liste t
# c et b sont deux listes
# de longueurs borne et n
# initialisées avec des 0
      for i in range(len(t)):
            c[t[i]] += 1
# c[t[i]] contient le nombre d'occurences de l'élément t[i]
# dans le tableau t
      for i in range(1,borne):
            c[i] += c[i-1]
# c contient maintenant les effectifs cumulés croissants
# c[i] est le nombre d'éléments de t plus petits que i
# b ne contient que des 0
      for i in range(n-1,-1,-1):
 # en lisant t de la droite vers la gauche, c[t[i]] est le nombre
 # d'éléments de t plus petits que t[i]
            c[t[i]] -= 1
 # c[t[i]] est diminué de 1 car c[t[i]]-1 est le nombre de places
 # nécessaires pour stocker les éléments plus petits que t[i],
 # qui est placé dans b à la place c[t[i]]
            b[c[t[i]]] = t[i]
 # b contient les éléments de t dans l'ordre croissant
      return b
```

Les commentaires que nous avons ajouté aux blocs d'instructions sont de trois natures différentes :

- Les préconditions : qui sont les conditions que doivent vérifier les différents objets avant d'être traités par une ou des instructions. Elles garantissent que le traitement est possible sans erreur.
- Les postconditions : qui sont les conditions vérifiées par les objets à la sortie du traitement. Elles garantissent que le traitement a donné le bon résultat.
- Les justifications ou explications du code.

Les deux premiers types sont inspirés par le paradigme de programmation par contrat.

Ces commentaires sont très utiles au correcteur de votre copie pour comprendre ce que fait votre code.

Alors si vous avez le temps, ou si c'est demandé explicitement, n'hésitez surtout pas à agrémenter votre code de commentaires pertinents.

Avec un schéma c'est le must :

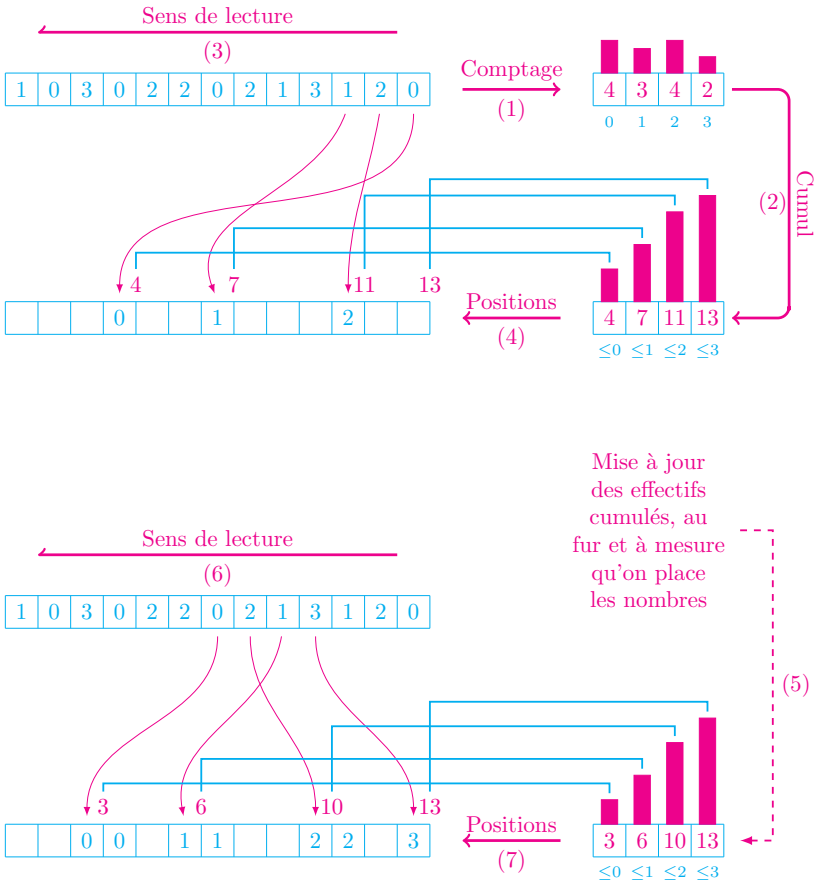


FIGURE 1.1 – counting_sort stable.

1.1.2 Correction et terminaison

Lorsqu'on écrit un algorithme il faut s'assurer de deux choses :

1. qu'il se termine, c'est-à-dire que quel que soit l'état initial l'algorithme s'arrête après un nombre fini d'opérations ;

2. qu'il fournit le bon résultat.

Définition 1.1.

On dit qu'un algorithme est partiellement correct si quelque soit l'état initial qui fait qu'il se termine, il retourne un résultat correct.

On dit qu'un algorithme est totalement correct si quelque soit l'état initial il se termine et le résultat est correct.

Exemples 1.1.

Algorithm 2: Un algorithme qui ne finit pas.

```
1 def f(n):
      Data: n ≥ 2 : un entier
2     while n ≥ 2 do
3         n ← n + 1
4     return n
```

Algorithm 3: Un algorithme partiellement correct.

```
1 def max(x, y):
      Data: x, y : deux réels
      Result: r = max(x, y)
2     if x < y then
3         r ← y
4     return r
```

Algorithm 4: Un algorithme totalement correct.

```
1 def max(x,y):
      Data: x, y : deux réels
      Result: r = max(x, y)
2     if x < y then
3         r ← y
4     else
5         r ← x
6     return r
```

Algorithm 5: Un algorithme partiellement correct qui ne finit pas toujours.

```
1 def max(x,y):
      Data: x, y : deux réels
      Result: r = max(x, y)
2     if x ≤ y then
3         r ← y
4     else
5         while x ≠ y do
6             r ← x
7     return r
```

Une boucle `for` se termine toujours !
Enfin est censée se terminer... Celle-ci en python ne s'arrête pas...

```
l = [1]
for elt in l:
    l.append(elt) # Python permet la modification de l'itérable
                  # alors qu'on le parcourt...
```

Mais une boucle `for` du type : `for i in range(n)`, elle, se terminera toujours.

On peut même remarquer que l'objet `range` est défini au moment de lancement de la boucle et ne change plus après :

```
>>> n = 3
>>> for i in range(n):
        n += 1
        print(n)
4
5
6
```

En revanche en Php ou en C la boucle suivante sera infinie :

```
<?php
$n = 3;
for ($i = 0 ; $i < $n ; $i++){
    n++;
}?>
```

Mais comment démontrer qu'une boucle conditionnelle s'arrête ?

En pratique 1.1.

Pour démontrer la terminaison d'une boucle conditionnelle, on identifie une suite d'entiers strictement décroissante dont les valeurs diminuent à chaque itérations de la boucle.

Ce résultat repose sur le théorème mathématique suivant :

Théorème 1.1.

Toute suite d'entiers strictement décroissante ne peut prendre qu'un nombre fini de valeurs distinctes.

Prouver que le premier algorithme s'arrête. Quant est-il du second ?

Exemple 1.2.

Algorithm 6: Algorithme 1

Data: n : un entier naturel

```
1 x ← 0
2 while n ≥ 0 do
3 |   n ← n − 1
4 |   x ← x^n
```

Algorithm 7: Algorithme 2

Data: $x > 0$: un réel

```
1 n ← 0
2 while x > 0 do
3 |   x ← e^{−n}
4 |   n ← n + 1
```

Remarque 1.1.

L'algorithme précédent n'est pas censé s'arrêter, mais dans la pratique il s'arrête à cause de la représentation des flottants.
Essayez et vous verrez...

Exemple 1.3 (Algorithme de la division euclidienne)**.**

Algorithm 8: Algorithme de la division euclidienne.

Data: $a, b \neq 0$: deux entiers naturels
Result: q, r tels que $a = bq + r$ avec $r < b$

```
1 q ← 0
2 r ← a
3 while b <= r do
4 |   q ← q + 1
5 |   r ← r − b
```

— Quelle est la quantité entière qui diminue à chaque itération de la boucle ?
— Les valeurs prises par la variable r constituent une suite d'entiers strictement décroissante. Ce qui prouve que cet algorithme se termine.

Exemple 1.4 (Suite de Syracuse).

Algorithm 9: Suite de Syracuse.

Data: n : un entier naturel
1 **while** $n \neq 1$ **do**
2 **if** n *est pair* **then**
3 $n \leftarrow n//2$
4 **else**
5 $n \leftarrow 3n+1$

— Est-ce que cet algorithme s'arrête ?
— La conjecture de Syracuse prétend que oui. Mais personne n'a pu le démontrer, d'après Paul Erdös : « les mathématiques ne sont pas encore prêtes pour de tels problèmes ». Certains pensent même que ce problème pourrait être indécidable.

Définition 1.2 (Variant de boucle).

On appelle variant de boucle un entier, positif ou nul, qui décroît à chaque itération.

La question que l'on se pose maintenant est : « est-ce que mon algorithme donne le bon résultat ? »
— Pour s'en convaincre on peut évidemment tester l'algorithme pour un nombre fini de conditions initiales. Mais ce n'est pas très satisfaisant.
— Comme en mathématique on préfèrera faire une preuve de notre algorithme.

Définition 1.3 (Invariant de boucle).

Un invariant de boucle est une propriété P qui vérifie :
— P est vraie avant d'entrer dans la boucle pour la première fois.
— Si P est vraie avant une itération elle l'est aussi après.
— Une fois la boucle terminée P permet de montrer la validité de l'algorithme.

En pratique 1.2.

Prouver une boucle à l'aide d'un invariant de boucle c'est faire une récurrence finie.
Il y a trois étapes :
— Initialisation : Montrer que P est vraie avant de rentrer dans la boucle.
— Conservation : Montrer que si P est vraie avant une itération de la boucle, P est vraie avant la suivante itération.
— Terminaison : Prouver la validité de l'algorithme (ou d'une partie de l'algorithme) grâce à P.

Exemple 1.5.

Prouvons l'algorithme de la division euclidienne :

Data: $a, b \neq 0$: deux entiers naturels
Result: q, r tels que $a = bq + r$ avec $r < b$

```
1  q ← 0
2  r ← a
3  while b <= r do
4      q ← q + 1
5      r ← r − b
```

Considérons la propriété P : « $a = bq + r$ ».

— Initialisation : P est vraie avant de rentrer dans la boucle $a = b \times 0 + a$.
— Conservation : Notons b', q' et r' les valeurs de b, q et r après une itération. Alors :

$$b' = b,\ r' = r - b \text{ et } q' = q + 1.$$

Donc $b'q' + r' = b(q+1) + r - b = bq + r = a$, et donc P est vraie après une itération.

— Terminaison : La boucle s'arrête lorsque $r < b$, (nous avons déjà montrer que cet algorithme termine) à la fin on aura donc :

$$a = bq + r \text{ et } r < b.$$

Exercice 1.1.

1. Que vaut r à la sortie ?
2. Prouver que cet algorithme est totalement correct.

Algorithm 10: Algorithme mystère.

Data: a, n : deux entiers naturels

```
1  A ← a
2  r ← 1
3  p ← n
4  while p > 0 do
5      if p pair then
6          A ← A × A
7          p ← p/2
8      else
9          r ← r × A
10         p ← p − 1
```

Exercice 1.2.

Prouver que l'algorithme suivant est totalement correct.

Algorithm 11: Insertion Sort

```
1 def insertion_sort(t):
      Data: t : tableau de nombres
      Result: Le tableau t trié
2     n ← len(t)
3     for i ← 1 to n − 1 do
4         key ← t[i]
5         j ← i − 1
6         while j ≥ 0 and t[j] > key do
7             t[j + 1] ← t[j]
8             j ← j − 1
9         t[j + 1] ← key
```

1.2 Complexité

1.2.1 Un problème plusieurs solutions

Plusieurs algorithmes peuvent répondre à un même problème. La question de leur efficacité se pose alors. Un des critères pour évaluer l'efficacité est le temps d'exécution, qu'on appelle aussi coût de l'algorithme.

Pour mesurer ce coût on utilise la notion de complexité algorithmique, que nous allons étudier.

Pour comparer des algorithmes qui résolvent un même problème, ou non d'ailleurs, il faudrait que la complexité :

— soit indépendante du langage de programmation
— soit indépendante de l'ordinateur sur lequel le programme tourne
— soit peu dépendante des détails d'implémentation.

Pour analyser les algorithmes nous supposerons travailler sur un modèle d'ordinateur simplifié : une machine RAM.
Entre autres cela supposera :

— un monoprocesseur (pas de parrallélisme)
— les instructions sont exécutées séquentiellement
— la mémoire est infinie et n'est pas hiérarchisée (pas de cache, de swap etc)
— un accès aléatoires à la mémoire en temps constant.

Dans ca cadre, pour mesurer le temps d'execution d'un algorithme, il suffit de compter les instructions élémentaires et de les pondérer par leur coût temporel.

En général, nous noterons T la complexité temporelle d'un algorithme, c'est une fonction croissante de ses paramètres.

1.2.2 Des exemples

Recherche du maximum dans une tableau linéaire

Commençons par observer un algorithme de recherche du maximum dans un tableau linéaire de nombres. L'algorithme doit retourner le maximum mais aussi la première position à laquelle il se trouve dans le tableau.

Nous allons déterminer le coût algorithmique de cette fonction en fonction de la taille n du tableau passé en argument.

Algorithm 12: search_max(t)

```
1 def search_max(t):
2     n ← len(t)
3     i_max ← 0
4     t_max ← t[0]
5     for i ← 1 to n − 1 do
6         if t_max < t[i] then
7             i_max ← i
8             t_max ← t[i]
9     return max, i_max
```

Le module `line_profiler` nous permet de voir combien de fois chaque ligne est exécutée lors de l'appel à une fonction.

Commençons avec le tableau des entiers de 0 à 999, la maximum se trouve donc à la dernière place du tableau.

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     0                                           @profile
     1                                           def max_search(t):
     2         1          1.0      1.0      0.2      n = len(t)
     3         1          0.0      0.0      0.0      imax = 0
     4         1          0.0      0.0      0.0      tmax = t[0]
     5       999        162.0      0.2     24.9      for i in range(1,n):
     6       999        178.0      0.2     27.3          if tmax < t[i]:
     7       999        133.0      0.1     20.4              imax = i
     8       999        177.0      0.2     27.2              tmax = t[i]
     9         1          0.0      0.0      0.0      return tmax, imax
```

Continuons avec le tableau des entiers de 0 à 999 trié dans l'ordre décroissant :

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     0                                           @profile
     1                                           def max_search(t):
     2         1          0.0      0.0      0.0      n = len(t)
     3         1          0.0      0.0      0.0      imax = 0
     4         1          0.0      0.0      0.0      tmax = t[0]
     5       999        158.0      0.2     45.4      for i in range(1,n):
     6       999        190.0      0.2     54.6          if tmax < t[i]:
     7                                                     imax = i
     8                                                     tmax = t[i]
     9         1          0.0      0.0      0.0      return tmax, imax
```

On observe, que dans les deux cas les lignes 5 et 6 sont exécutée 999, et dans le cas croissant le test de la ligne 6 étant toujours vrai les lignes 7 et 8 sont aussi exécutées 999 fois. Dans le cas décroissant au contraire le test n'est jamais vérifié et du coup ces mêmes lignes ne sont jamais évaluées.

Faisons maintenant un calcul précis du coût. Notons c_i le coût temporelle de la ligne i, et comptons combien de fois chaque ligne est exécutée.

Algorithm 13: search_max(t)

		Coût	nb fois
1	**def** $search\_max(t)$**:**		
2	$n \leftarrow$ **len**(t)	c_1	1
3	$i_{max} \leftarrow 0$	c_2	1
4	$t_{max} \leftarrow t[0]$	c_3	1
5	**for** $i \leftarrow 1$ **to** $n-1$ **do**	c_4	$n-1$
6	**if** $t_{max} < t[i]$ **then**	c_5	$n-1$
7	$i_{max} \leftarrow i$	c_6	nb
8	$t_{max} \leftarrow t[i]$	c_7	nb
9	**return** max, i_{max}	c_8	1

Ici nb est le nombre de fois où l'inégalité de la ligne 7 sera vérifiée au cours de l'exécution de l'algorithme. Ce nombre vérifie $0 \leq nb \leq n-1$.

Si on désigne par $T(n)$ le temps d'exécution de cet algorithme en fonction de la taille n du tableau, on alors :

$$c_1 + c_2 + c_3 + c_8 + (c_4 + c_5 + c_6) \times (n-1) \leq T(n) \leq c_1 + c_2 + c_3 + c_8 + (2 + c_4 + c_5 + c_6) \times (n-1).$$

Il semblerait donc qu'il existe $\alpha_1, \beta_1, \alpha_2, \beta_2 \in \mathbb{R}$, avec $\alpha_1 \neq 0$ et $\alpha_2 \neq 0$ tels que :

$$\alpha_1 \cdot n + \beta_1 \leq T(n) \leq \alpha_2 \cdot n + \beta_2$$

.

Autrement dit, la complexité temporelle de cet algorithme a une croissance linéaire.

Ce qui signifie que si la taille du tableau est multipliée par deux le temps d'exécution est lui aussi quasiment multiplié par deux.

Vérifions si notre calcul semble juste avec une simulation. Sur la figure (1.2), sont représentés le temps d'exécution de la fonction `search_max` et le graphe d'une approximation affine d'équation $y = 6 \times 10^{-8}x - 2 \times 10^{04}$, déterminée par la fonction `polyfit` du module `Numpy`.

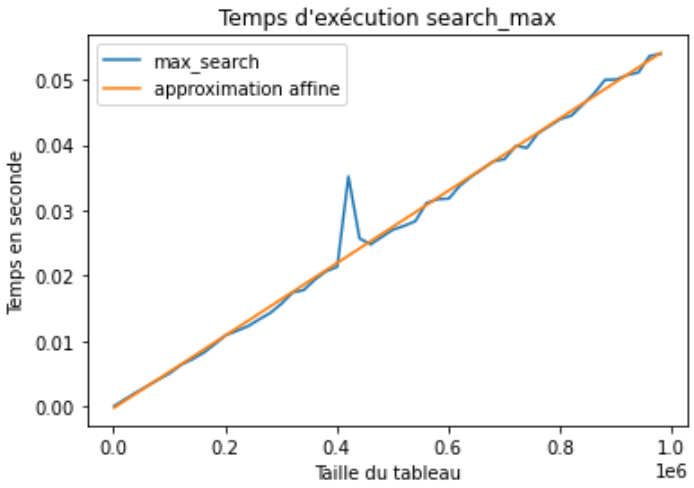


FIGURE 1.2 – Temps d'exécution de search_max en fonction de la longueur du tableau.

Recherche du maximum dans un tableau 2d

L'algorithme suivant recherche le maximum et la position du maximum d'un tableau de nombres à n lignes et n colonnes.

Algorithm 14: Search_max_mat(t)

		Coût	nb fois
1	**def** *search_max_max*(t)**:**		
2	$n \leftarrow$ **len**(t)	c_1	1
3	$i_{max} \leftarrow 0$	c_2	1
4	$j_{max} \leftarrow t[0]$	c_3	1
5	$t_{max} \leftarrow t[0][0]$	c_4	1
6	**for** $i \leftarrow 0$ **to** $n-1$ **do**	c_5	n
7	**for** $j \leftarrow 0$ **to** $n-1$ **do**	c_6	n^2
8	**if** $t_{max} < t[i][j]$ **then**	c_7	n^2
9	$i_{max} = i$	c_8	nb
10	$j_{max} = j$	c_9	nb
11	$t_{max} = t[i]$	c_{10}	nb
12	**return** max, i_{max}	c_{11}	1

Notons encore $T(n)$ le temps d'exécution de cet algorithme en fonction de n.

Le test à la ligne 8 est effectué n^2 fois, mais suivant son résultat les lignes 12 à 14 le sont ou non. Notons nb le nombre de fois ou ces lignes sont exécutée lors lors de l'appel à `search_max_mat` sur un tableau donnée. Alors $0 \leq nb \leq n^2$.

Cette fois-ci, si l'on additionne le coût de toutes lignes on obtient qu'il existe des nombres $\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_3, \gamma_3 \in \mathbb{R}$ tels que :

$$\alpha_1 \cdot n^2 + \beta_1 \cdot n + \gamma_1 \leq T(n) \leq \alpha_2 \cdot n^2 + \beta_2 \cdot n + \gamma_2.$$

Encore une fois vérifions ce calcul par une simulation. La figure (1.3) représente le temps d'exécution de la fonction `search_max_mat` en fonction de la taille du tableau, ainsi que deux approximations polynomiales de degré deux.

Un algorithme de tri.

Algorithm 15: Insertion Sort(t)

		Coût	nb fois
1	**def** *insertion_sort*(t)**:**		
2	$n \leftarrow len(t)$	c_1	1
3	**for** $i \leftarrow 1$ **to** $n-1$ **do**	c_2	$n-1$
4	$key \leftarrow t[i]$	c_3	$n-1$
5	$j \leftarrow i-1$	c_4	$n-1$
6	**while** $j \geq 0$ **and** $t[j] > key$ **do**	c_5	$\sum_{i=1}^{n-1}(t_i+1)$
7	$t[j+1] \leftarrow t[j]$	c_6	$\sum_{i=1}^{n-1} t_i$
8	$j \leftarrow j-1$	c_7	$\sum_{i=1}^{n-1} t_i$
9	$t[j+1] \leftarrow key$	c_8	$n-1$

Ici `t_i` est le nombre de fois que le test de la boucle `while` sera effectuée en fonction de `i`.
Le test est toujours exécuté une fois de plus que le corps de la boucle.

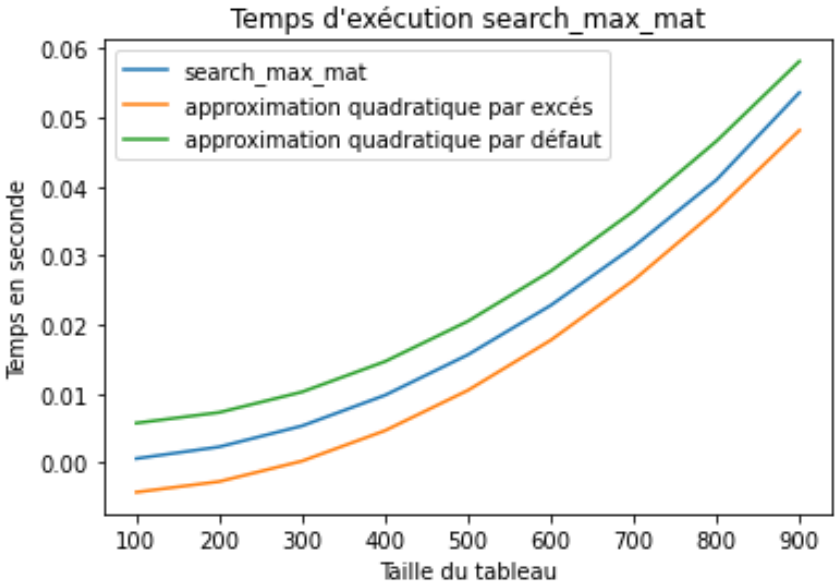


FIGURE 1.3 – Temps d'exécution de search_max_mat en fonction de la taille du tableau.

Algorithm 16: Insertion Sort(t)

		Coût	nb fois
1	$n \leftarrow len(t)$	c_1	1
2	**for** $i \leftarrow 1$ **to** $n-1$ **do**	c_2	$n-1$
3	$\quad key \leftarrow t[i]$	c_3	$n-1$
4	$\quad j \leftarrow i-1$	c_4	$n-1$
5	$\quad$ **while** $j \geq 0$ **and** $t[j] > key$ **do**	c_5	$\sum_{i=1}^{n-1}(t_i+1)$
6	$\quad\quad t[j+1] \leftarrow t[j]$	c_6	$\sum_{i=1}^{n-1} t_i$
7	$\quad\quad j \leftarrow j-1$	c_7	$\sum_{i=1}^{n-1} t_i$
8	$\quad t[j+1] \leftarrow key$	c_8	$n-1$

Si $T(n)$ est le temps d'exécution de l'algorithme en fonction de la taille n du tableau à trier, alors :

$$T(n) = c_1 + c_2 n + (c_3 + c_4)(n-1) + c_5 \sum_{i=1}^{n-1}(t_i+1) + (c_6 + c_7)\sum_{i=1}^{n-1} t_i + c_8(n-1).$$

Les t_i peuvent être déterminer dans certains cas :

— Le meilleur cas : Si t est déjà trié alors la condition $t[j] \leq key$ est toujours vérifiée et $t_i = 0$. Donc T est linéaire :

$$T(n) = \alpha \cdot n + \beta,$$

où α et β sont des fonctions des coefficients c_i.

— Le pire cas : Si t est trié dans l'ordre décroissant alors il faudra comparer $t[i]$ avec tous les éléments de $t[0:i]$, donc $t_i = i$. Après calculs on trouve que T est quadratique :

$$T(n) = \alpha \cdot n^2 + \beta \cdot n + \gamma,$$

où α, β et γ sont des fonctions des coefficients c_i.

— Le cas général : On ne sait pas déterminer t_i, mais logiquement il se situe entre les deux cas précédents.

— Le cas moyen : Si on suppose que les entrées de t ont été tirées au hasard (suivant une loi uniforme), alors pour chaque clef key la moitié du tableau lui est supérieur et l'autre moitié inférieur en moyenne,

donc $t_i = \frac{i}{2}$, après calculs on trouve encore que T est quadratique :

$$T(n) = \alpha' \cdot n^2 + \beta' \cdot n + \gamma',$$

où α', β' et γ' sont des fonctions des coefficients c_i.

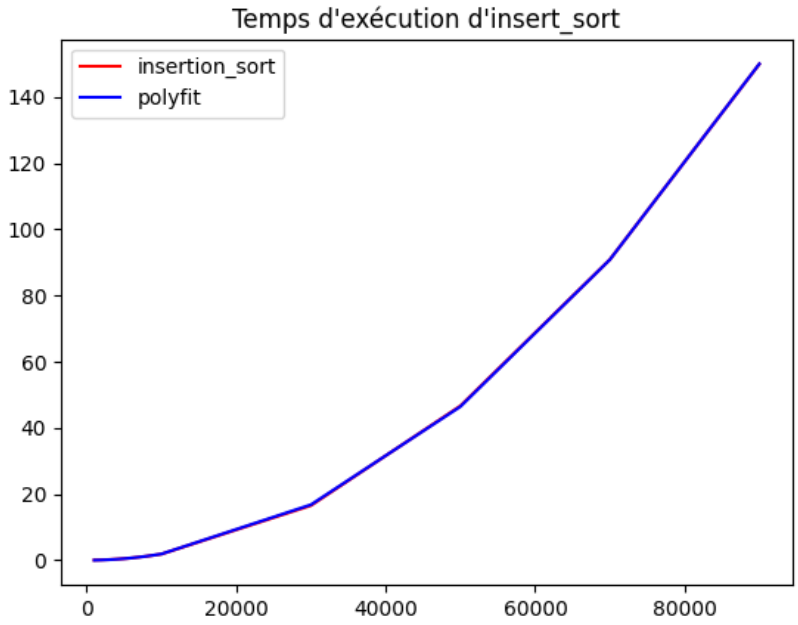


FIGURE 1.4 – Temps d'exécution de insertion_sort en fonction de la longueur de la liste.

```
tests = []
x1 = range(10**3,10**4,10**3)
x2 = range(10**4,10**5, 20*10**3)
x = list(x1)+list(x2)
for k in x:
    tests.append([randint(0,200) for __ in range(k)])
y = [insertion_sort(t) for t in tests]
p = np.polyfit(x,y,deg=2)
print(p)
```

```
[ 1.84576333e-08  6.56322398e-06 -4.69599035e-02]
```

Donc on trouve que sur ma machine on a :

$$T(n) \approx \underbrace{1.84576333e-08}_{\alpha} \cdot n^2 + \underbrace{6.56322398e-06}_{\beta} \cdot n + \underbrace{-4.69599035e-02}_{\gamma}.$$

On peut se servir de l'approximation polynomiale fournie par `polyfit` pour prédire le temps nécessaire au tri d'une liste sur ma machine :

```
>>> np.polyval(p,10**6)/3600
5.128930436001011 #5 heures
>>> np.polyval(p,10**9)/3600/24/365
585.2879207477561 # 600 ans...
>>> np.polyval(p,10**10)/3600/24/365
58528.773344260975 # 60 000 ans
```

Évidemment les constantes α, β et γ que nous avons trouvés dépendent de ma machine et du jeu de tests pseudo-aléatoires. Donc, il ne faut pas tenir compte des constantes et ne considérer que le comportement asymptotique de $T(n)$. La complexité d'un algorithme lui est intrinsèque. S'il est quadratique sur une machine il le sera sur toutes les machines, seuls les coefficients vont changer.

1.2.3 Comportement asymptotique

Étant donné une fonction $g : \mathbb{N} \to \mathbb{R}$, on considère les différentes classes de fonctions suivantes :
— celles dont la croissance asymptotique est dominée par celle de g :

$$\mathrm{O}(g(n)) = \{f(n) \,|\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, |\, f(n) \,| \leq cg(n)\}$$

— celles dont la croissance asymptotique domine celle de g :

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$

— celles qui ont une croissance asymptotique équivalente à celle de g :

$$\Theta(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{l} \exists c_1 > 0, \exists c_2 > 0, \exists n_0 \in \mathbb{N} \\ \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \end{array} \right\}$$

Avec ces notations on a : $\mathrm{O}(g(n)) \cap \Omega(g(n)) = \Theta(g(n))$.

Par exemple, nous avons montré que pour le tri par insertion :
— dans le pire cas $T(n) = \Theta(n^2)$
— dans le meilleur cas $T(n) = \Theta(n)$
— dans le cas moyen $T(n) = \Theta(n^2)$
Donc la complexité du tri par insertion est quelque part entre $\Omega(n)$ et $\mathrm{O}(n^2)$.

Le plus souvent on ne s'intéresse qu'à la complexité au pire d'un algorithme et on dira donc par exemple que celle du tri par insertion est en $\mathrm{O}(n^2)$.

Calculs avec ces notations :

Algorithm 17: Insertion Sort(t)

1	$n \leftarrow len(t)$	$\mathrm{O}(1)$	+
2	**for** $i \leftarrow 1$ **to** $n-1$ **do**	$\Theta(n)$	+
3	$\quad key \leftarrow t[i]$	$\Theta(n)$	+
4	$\quad j \leftarrow i-1$	$\Theta(n)$	+
5	$\quad$ **while** $j \geq 0$ **and** $t[j] > key$ **do**	$\mathrm{O}(n^2)$	+
6	$\quad\quad t[j+1] \leftarrow t[j]$	$\mathrm{O}(n^2)$	+
7	$\quad\quad j \leftarrow j-1$	$\mathrm{O}(n^2)$	+
8	$\quad t[j+1] \leftarrow key$	$\Theta(n)$	
		$\overline{\mathrm{O}(n^2)}$	

1.2.4 Trois exemples de calculs

Recherche dans une tableau trié

La fonction suivante prend comme arguments un tableau de nombres triés dans l'ordre croissant, et un nombre x, et elle retourne, s'il existe l'unique indice i tel que $t[i] \leq x < t[i+1]$. L'algorithme utilisé est celui de la recherche dichotomique.

Algorithm 18: *dichotomic_search*(x, t)

```
1  def dichotomic_search(x, t):
2      ℓ ← 0                                          Θ(1)          +
3      r ← len(t) − 1                                 Θ(1)          +
4      if x < t[ℓ] or x > t[r] then                   Θ(1)          +
5          return false                               O(1)          +
6      else                                           O(1)          +
7          while ℓ + 1 < r do                         Θ(log2(n))    +
8              k ← (ℓ + r)//2                         Θ(log2(n))    +
9              if t[k] ≤ x then                       Θ(log2(n))    +
10                 l ← k                              O(log2(n))    +
11             else
12                 r ← k                              O(log2(n))    +
13     return ℓ                                       O(1)
                                                      ―――――――――
                                                      O(log2(n))
```

Notons nb le nombre de fois que la ligne 7 est exécutée.

Pour déterminer nb, il faut se demander combien de fois la condition $1 < r - \ell$ sera vérifiée au cours de l'exécution de l'algorithme. Avant de rentrer dans la boucle pour la première fois, $\ell = 0$ et $r = n - 1$, donc $r - \ell = n - 1$, donc on ne rentre dans la boucle que si le tableau à au moins deux cases. Après un tour de boucle $r - \ell = n - 1 - \lfloor (n-1)/2 \rfloor$ ou $r - \ell = \lfloor (n-1)/2 \rfloor$.

Pour simplifier, faisons donc l'hypothèse que $n = 2^p + 1$. Dans cas on voit facilement qu'après i tours de boucles $r - \ell = 2^{p-i}$. Donc $r - \ell > 1$ tant que $i < p$, autrement dit $nb = \log_2(n)$.

Dans le cas général, pour n assez grand on peut toujours trouver un $p \in \mathbb{N}$, tel que $2^p + 1 \leq n < 2^{p+1} + 1$, par croissance de la fonction T on en déduit que dans tous les cas la ligne 7 sera exécutée $\Theta(\log_2(n))$ fois. Comme on ne tient pas compte des constantes on peut écrire $T(n) = \Theta(\ln(n))$.

Donc au meilleur cas $T(n) = \Theta(1)$, et au pire cas, comme dans le cas moyen, $T(n) = \Theta(\ln(n))$.

On dira que la recherche dichotomique a une complexité logarithmique.

Recherche de l'existence d'un mot dans une chaîne

L'algorithme suivant teste l'existence du mot `word` dans la chaîne de caractères `string`.

Algorithm 19: *search_word*$(string, word)$

1	**def** *search_word*$(string, word)$**:**		
2	$n \leftarrow$ **len**$(string)$	$\Theta(1)$	$+$
3	$p \leftarrow$ **len**$(word)$	$\Theta(1)$	$+$
4	**if** $p > n$ **then**	$\Theta(1)$	$+$
5	**return false**	$O(1)$	$+$
6	**else**		
7	$pos \leftarrow 0$	$O(1)$	$+$
8	**while** $pos + p < n + 1$ **and** $string[pos..pos + p] \neq word$ **do**	$O(pn)$	$+$
9	$pos \leftarrow pos + 1$	$O(pn)$	$+$
10	**if** $pos + p == n + 1$ **then**	$\Theta(1)$	$+$
11	**return false**	$O(1)$	$+$
12	**else**		
13	**return** pos	$O(1)$	
		$\overline{\mathrm{O}(np)}$	

Notons $T(n, p)$ la complexité au pire de `search_word` en fonction de la longueur n de la chaîne `string` et de la longueur p du mot `word`.

Le seul point à éclaircir est le coût de la ligne 8. Le test $string[pos..pos + p] \neq word$ coûte exactement p comparaisons, soit un $\Theta(p)$, et l'inégalité $pos + p < n + 1$ se fait au pire $n - p$ fois. Au final la ligne 8 s'exécute au pire $p(n - p)$ fois. Donc son coût est en $O(p(n - p)) = O(pn)$.

On a donc $T(n, p) = O(pn)$.

Tri récurcif

Voici un autre algorithme de tri :

Algorithm 20: Merge Sort

Data: t un tableau, i premier indice, k dernier indice
Result: t trié dans l'ordre croissant

```
1 def merge_sort(t, i, k):
2     if k − i > 1 then
3         j ← ⌊(i+k)/2⌋
4         merge_sort(t, i, j)
5         merge_sort(t, j, k)
6         merge(t, i, j, k)
```

Nous étudierons cet algorithme plus en détail plus tard. Mais il est assez simple de comprendre son principe. On découpe le tableau en deux parties à peu prêt de même taille, on trie chaque sous tableau, puis on fusionne les deux tableaux triés en un tableau trié grâce à la fonction `merge`.

Regardons un exemple :

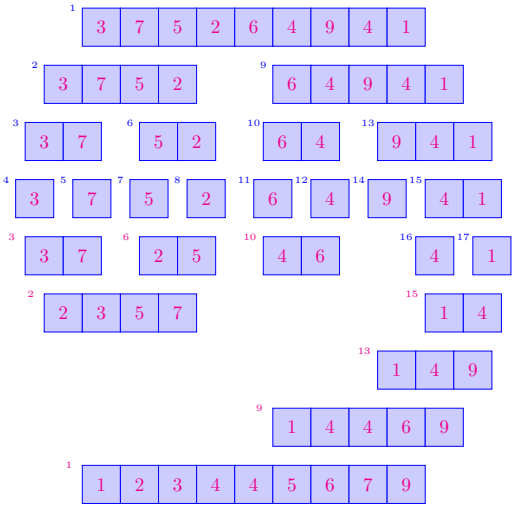


FIGURE 1.5 – Illustration d'un exemple de tri.

Notons $T(n)$ le temps d'exécution de cet algorithme sur un tableau de taille n. On admet [i] que la fonction *merge* a une complexité en $\Theta(n)$.

Algorithm 21: Merge Sort

		$T(1)$	$T(n)$
1	**def** *merge_sort(t, i, k)***:**		
2	**if** $k - i > 1$ **then**	$\Theta(1)$	$\Theta(1)$
3	$j \leftarrow \lfloor \frac{i+k}{2} \rfloor$		$\Theta(1)$
4	$merge\_sort(t, i, j)$		$\Theta(\lfloor n/2 \rfloor)$
5	$merge\_sort(t, j+1, k)$		$\Theta(\lceil n/2 \rceil)$
6	$merge(t, i, j, k)$		$\Theta(n)$

On a donc

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{sinon} \end{cases}$$

Comment résoudre : $T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{sinon} \end{cases}$?

Écrivons que pour $n > 1$ il existe un $c > 0$:

$$\begin{aligned} T(n) &= 2T(n/2) + cn = 2\left(2T(n/4) + c\frac{n}{2}\right) + cn \\ &= 2^2 T(n/4) + 2cn = 2^2\left(2T(n/8) + c\frac{n}{4}\right) + 2cn \\ &= 2^3 T(n/8) + 3cn \\ &= \dots \\ &= 2^k T(n/2^k) + kcn \end{aligned}$$

Donc, tand que $n/2^k \leq 1$ on a :

$$T(n) = 2^k T(n/2^k) + kcn$$

i. Nous le prouverons plus tard.

et on s'arrête lorsque $n/2^k = 1$, soit lorsque $k = \log_2(n)$, et dans ce cas $T(1) = \Theta(1)$.
Donc :

$$T(n) = nT(1) + cn\log_2(n) = n\Theta(1) + \Theta(n\log_2(n)) = \Theta(n\log n).$$

Finalement on trouve que $T(n) = \Theta(n\log(n))$, donc est semi-logarithmique.

1.2.5 Taille du problème et opérations fondamentales

Pour décrire la complexité du tri pas insertion nous l'avons « naturellement » exprimer en fonction de la taille du tableau t à trier.

En pratique, il faudra toujours correctement définir ce que l'on considère comme « taille du problème », par exemple :

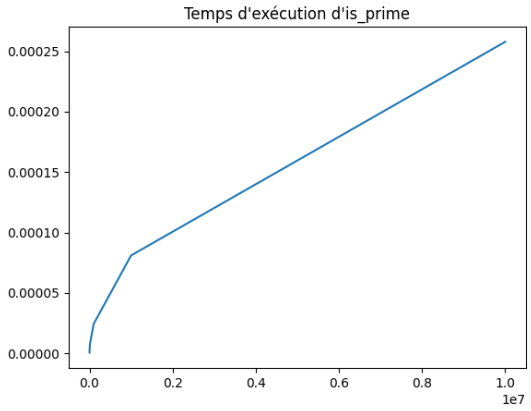
— pour une liste la taille est son nombre d'éléments ;
— pour un entier n, on peut prendre ce nombre n, ou encore le nombre de bits nécessaires à son écriture en binaire.
— pour graphe on pourra prendre deux variables : le nombre de sommets et le nombre d'arrêtes.

Un mauvais choix

Par exemple, la complexité au pire de l'algorithme suivant est en $O(\sqrt{n})$.

```
def is_prime(n: int)->bool:
    rep = True
    if n in [0,1] or (n > 2 and n % 2 == 0):
        rep = False
    k = 3
    while rep and k**2 <= n:
        if n % k == 0:
            rep = False
        k += 2
    return rep
```

Ce qui semble se vérifier d'après la figure suivante.



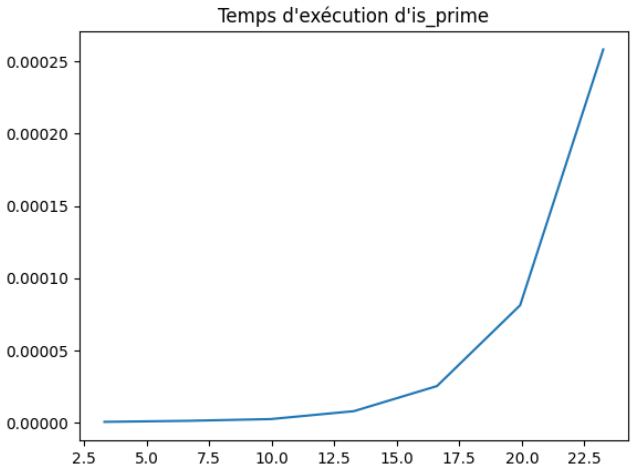
Pourtant d'après wikipédia : Le test de primalité AKS est un algorithme de preuve de primalité déterministe et généraliste publié le 6 août 2002 par trois scientifiques indiens nommés Manindra Agrawal, Neeraj Kayal et Nitin Saxena (A.K.S). Ce test est le premier en mesure de déterminer la primalité d'un nombre dans un

temps polynomial. Ce test a été publié dans un article scientifique intitulé « PRIMES is in P ». Cet article leur a valu le prestigieux prix Gödel 20063.

Qu'est-ce que ça veut dire ? Simplement, que l'on s'est trompé dans le choix de la taille du problème...

Notre algorithme fait des opérations arithmétique (division euclidienne, produit, etc...) sur des nombres entiers codés en binaire. Le temps d'exécution d'une opération va donc être proportionnelle au nombre de chiffres dans l'écriture binaire du nombre que l'on traite.

La bonne complexité est donc en $\sqrt{n} = \sqrt{2}^{\log_2(n)}$, qui est exponentielle...



Pour mesurer la complexité en temps on choisit en général une, ou des, opération(s) fondamentale(s) pour l'algorithme que l'on souhaite évaluer. Et on ne tient compte que de ces opérations dans la calcul de la complexité.

Le choix est bon si le nombre total d'opérations est proportionnel au nombre d'opérations fondamentales.

Algorithme	Opérations fondamentales
Arithmétique sur des entiers	Opération binaire
Multiplication de matrices	Addition et multiplication des coefficients
Trier un tableau	Comparaison et affectation
Recherche du max	Comparaison
Recherche de la mediane	Comparaison

1.2.6 Temps d'exécution

Sur la base de 10^9 opérations élémentaires à la seconde voici les temps d'exécutions pour différentes valeurs de la taille du problème :

	Temps d'exécution					
	$O(\ln(n))$	$O(n)$	$O(n\ln(n))$	$O(n^2)$	$O(n^3)$	$O(2^n)$
10^2	7 ns	100 ns	$0,7$ μs	10 μs	1 ms	4×10^{13} années
10^3	10 ns	1 μs	10 μs	1 ms	1 s	10^{292} années
10^4	13 ns	10 μs	133 μs	100 ms	17 s	
10^5	17 ns	100 μs	2 ms	10 s	$11,6$ jours	
10^6	20 ns	1 ms	20 ms	17 mn	32 années	

Il apparaît clairement qu'un algorithme dont la complexité est plus que quadratique n'est pas utilisable pour des valeurs de $n \geq 10^5$.

Donnons pour l'exemple les complexités de quelques algorithmes :

Complexité de quelques algorithmes	
Calcul de la moyenne ou de la variance d'une liste de taile n	$\Theta(n)$
Recherche du maximum dans une liste de taille n	$\Theta(n)$
Recherche d'un élément dans une liste de taille n	$\Theta(n)$
Recherche dichotomique dans une liste triée de taille n	$O(\ln(n))$
Recherche d'un mot de longueur p dans une chaîne de caractères de longueur n	$O(np)$
Multiplication de matrices $A \in \mathcal{M}_{n,p}(\mathbb{K})$ et $B \in \mathcal{M}_{p,q}(\mathbb{K})$	$O(npq)$
Multiplication de matrices carrées de taille n	$O(n^3)$
Méthode de Gauss : Triangulation	$O(n^3)$
Méthode de Gauss : Remontée	$O(n^2)$
Méthode d'Euler avec $n = \dfrac{b-a}{n}$	$O(n)$
Tri par insertion avec n la longueur du tableau	$O(n^2)$
Tri fusion avec n la longueur du tableau	$\Theta(n\ln(n))$

1.3 Exercices

Exercice 1.3.

1. Écrire un algorithme naïf qui détermine le plus grand élément d'un tableau de nombres entiers.
2. Combien de comparaisons effectue votre algorithme ? Peut-on faire mieux ? Quelle est sa complexité ?
3. Écrire une algorithme tout aussi naïf qui détermine le minimum et le maximum d'un tableau de nombres.
4. Combien de comparaisons effectue votre algorithme ? Quelle est la complexité de votre algorithme ?
5. On peut faire mieux en remarquant que l'on peut faire seulement trois comparaisons plutôt que quatre lors du parcours de la liste par pas de 2, pour déterminer le minimum et maximum.
6. Confrontons la théorie à la réalité. Tester l'implémentation de vos deux codes sur des listes aléatoires, et mesurer le temps d'execution des deux implémentations en fonction de la longueur de la liste pour des valeurs de 10 à 2×10^5 par pas de 10^4. Que constatez-vous ?

1.4 Corrections des exercices

Corrigé 10 1. À la sortie $r = a^n$.

2. La valeur de $p \geq 0$ est entière et décroît à chaque itération de la boucle, donc l'algorithme se termine. Considérons la propriété P : « $A^p \times r = a^n$ ».
   — Initialisation : Avant de rentrer dans la boucle $A^p \times r = a^n \times 1 = a^n$.
   — Conservation : Notons A', r' et p' les valeurs de A, r et p après une itération.
   Si p est pair alors $r' = r$, $p' = p/2$ et $A' = A^2$ alors :
   $$A'^{p'} \times r' = (A^2)^{p/2} \times r = A^p \times r = a^n.$$
   Si p est impair alors $r' = A \times r$, $p' = p - 1$ et $A' = A$ alors :
   $$A'^{p'} \times r' = A^{p-1} \times A \times r = A^p \times r = a^n.$$
   — Terminaison Lorsque $p = 0$, $A^0 \times r = a^n \Rightarrow r = a^n$.

Corrigé 9 La boucle principale est une boucle `for` qui se termine, la quantité `i` est un variant de boucle pour la boucle interne `while`.
Considérons la propriété P : « le sous-tableau `t[0:i]` est constitué des éléments initialement dans `t[0:i]` mais triés. »

1. Initialisation : Avant de rentrer dans la boucle `for`, `i=1` et `t[0:i]=t[0]` qui contient `t[0]` et est trié.
2. Conservation : Si P est vraie avant la prochaine itération. Alors les éléments de `t[0:i]` sont ceux initialement dans le sous-tableau `t[0:i]` et ils sont triés. La boucle `while` va déplacer les éléments `t[i-1]`, `t[i-2]` etc vers la droite jusqu'à trouver la place que `key=t[i]` doit occuper pour que le tableau `t[0:i+1]` soit trié, ou alors on ne rentre pas dans la boucle et `key=t[i]` est laissé à sa place. Donc P est vraie.
3. Terminaison : la boucle `for` s'arrête lorsque `i=n`, en substituant `n` à la place de `i` dans P, on obtient que le tableau `t[0:n]=t` est trié.

Corrigé 1.3 1. Un algorithme naïf :

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

2. L'algorithme effectue $n - 1$ comparaison, et sa complexité est en $\Theta(n)$. On ne peut pas faire mieux ni pour la nombre de comparaison, ni pour la complexité, simplement parce que pour déterminer le maximum il faut nécessairement lire les n cases du tableau au moins une fois et faire au moins $n - 1$ comparaisons.
3. Un algorithme naïf :

```
def minmax(t):
    n = len(t)
    m, M = t[0], t[0]
    for i in range(1, n):
        if t[i] > M: M = t[i]
        if t[i] < m: m = t[i]
    return m, M
```

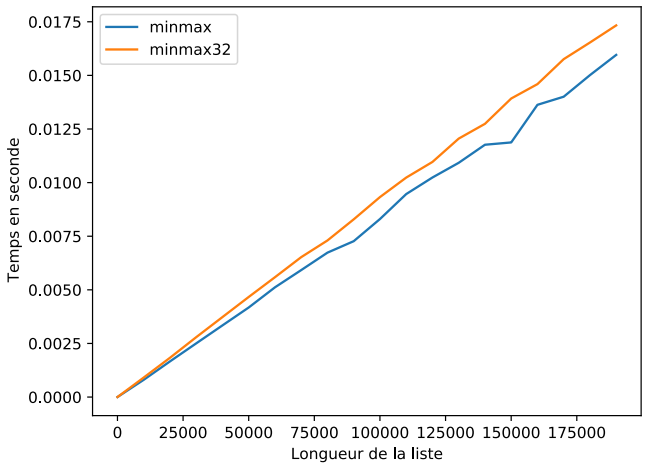


FIGURE 1.6 – Comparaison des deux algorithmes min-max.

4. L'algorithme effectue $2(n-1)$ comparaisons, 2 par chacune des entrées du tableau d'indice $1 \leq i \leq n-1$. Sa complexité est en $\Theta(n)$.
5. Un algorithme moins naïf :

```
def minmax32(t):
    n = len(t)
    m, M = t[0], t[1]
    for i in range(1,n//2,1):
        a = t[2*i]
        b = t[2*i+1]
        if a < b:
            if a < m: m = a
            if b > M: M = b
        else:
            if b < m: m = b
            if a > M: M = a
    return m, M
```

6. L'algorithme effectue maintenant 3 comparaisons pour deux valeurs du tableau, ce qui fait que le nombres de comparaisons est de l'ordre de $\frac{3}{2}n$, ce qui est moins que l'algorithme naïf. Sa complexité est toujours en $\Theta(n)$.
7. Si la complexité des deux algorithmes est la même on s'attend à ce que le second soit plus rapide mais il n'en est rien. C'est même le contraire. La raison est que nous avons introduit un branchement conditionnel qui est répété un très grand nombre de fois, et que les processeurs n'aiment pas attendre... Ceux qui veulent en savoir plus peuvent se renseigner sur les