CHAPITRE 1

AFFECTATIONS, VARIABLES, OBJETS

1.1 ÉLÉMENT DE SYNTAXE

§1 Expressions

Définition 1

Une expression est un élément de syntaxe[a] formé d'une combinaison de valeur (littéraux), variables et opérateurs.
Pour simplifier, on peut dire qu'une **expression** est une phrase qui peut être **évaluée**. Une expression retourne donc une **valeur** d'un certain **type**.

[a] c'est-à-dire conforme à la grammaire

Exemple 2

Une valeur est elle même une expression, de même qu'une variable.

```
1 217
2 x
3 3 + 5 * y
```

Dans la syntaxe d'un langage de programmation, on parle souvent

- Les valeurs ou littéraux. Ce sont les données manipulées par un algorithme ou un programme. Un **littéral**, est une valeur que l'on écrit de manière conventionnelle et dont la valeur est «évidente» (`123`, `4.3`, `"Bonjour"`). À l'opposé des **variables**, un littéral ne change pas de valeur. On parle parfois de *constante explicite* ou *constante pure*.
- D'identifiants, noms, références, label ou variables.

- D'opérateurs et d'opérandes. Un opérateur permet de former des expressions composées. Par exemple, si vE_1 et E_2 sont deux expressions, alors la construction $E_1 + E_2$ est aussi une expression où $+$ est un opérateur et E_1 et E_2 les opérandes.

```
The following tokens are operators:

+       -       *       **      /       //      %      @
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=
```

- De délimiteurs.

```
The following tokens serve as delimiters in the grammar:

(       )       [       ]       {       }
,       :       .       ;       @       =       ->
+=      -=      *=      /=      //=     %=
&=      |=      ^=      >>=     <<=     **=

The period can also occur in floating-point and imaginary
literals. A sequence of three periods has a special meaning
as an ellipsis literal. The second half of the list, the
augmented assignment operators, serve lexically as delimiters,
but also perform an operation.
```

§2 Instructions

Définition 3

Une **instruction** est un ordre de modification de l'état courant de l'exécution d'un programme.

Dans un langage évolué, la notion d'instruction est très large. Dire qu'une **instruction** est un ordre donné à l'ordinateur reste une définition acceptable.

Exemple 4

- L'affectation `x = 3`,
- L'exécution séquentielle (exprimée par un passage à la ligne en Python),
- La sélection (branchement `if ...`),
- Les boucles (`while ...`, `for ...`),
- L'affichage `print("Bonjour")`,

Remarque

La frontière entre instructions et expressions n'est pas aussi stricte qu'il n'y parait. Certaines instructions sont aussi des expressions…

§3 Objets et types

En python, tout est **objet**. Lors de la création d'un nouvel objet, son type est défini en fonction de l'objet crée. Plus précisément, `Python` est un langage orienté-objet: on dit que les entités créées sont des objets d'une certaine **classe**.

L'instruction `python`

```
type(x)
```

où x est un objet, permet de connaître la classe de `x`. Les objets les plus habituellement manipulés sont

Python 3 Shell

```
>>> type(5)
<class 'int'>
>>> type(2.3)
<class 'float'>
>>> type(5 > 3)
<class 'bool'>
>>> type([23, 5, 22, 1, 9, 10])
<class 'list'>
>>> type("PTSI 2")
<class 'str'>
>>> type((23, 5, 22, 1, 9, 10))
<class 'tuple'>
>>>
```

La notion de type est une conséquence du codage binaire, par exemple, l'octet 01010011 peut à la fois désigner l'entier 83 ou la lettre S (ou autre).

Une objet a

- une identité (avec CPython, c'est tout simplement l'adresse mémoire),
- un type ($\approx$ classe),
- une valeur (on peut donc évaluer un objet).

`http://docs.python.org/3/reference/datamodel.html#objects-values-and-types`

1.2 OBJETS ET AFFECTATIONS

§1 Références et affectations

L'exemple suivant illustre l'exécution d'un programme simple manipulant des «variables»

Python 3 Shell

```
>>> x = 3
>>> y = 12
>>> z = x
>>> x = 2 * x
>>> y = 2 * y
>>> z = x * y
>>> x, y, z
(6, 24, 144)
>>>
```

Le programmeur utilise 3 variables qu'il a nommées `x`, `y` et `z`. Chaque variable est une **référence** à une adresse mémoire où est stocké la valeur correspondante.

Test 5

Quelles seront les valeurs des variables `a`, `b`, `c` après l'exécution de chacune des instructions

```
1 a = 5
2 b = 3
3 c = a + b
4 a = 2
5 c = b - a
```

Test 6

Qu'obtiendra-t-on dans les variables `a` et `b`, après exécution des instructions suivantes ?

```
1 a = 5
2 b = a + 4
3 a = a + 1
4 b = a - 4
```

Test 7

1. Qu'obtiendra-t-on dans les variables `a` et `b`, après exécution des instructions suivantes ?

```
1 a = 5
2 b = 7
3 a = b
4 b = a
```

2. Qu'obtiendra-t-on dans les variables `a` et `b`, après exécution des instructions suivantes ?

```
a = 5
b = 7
b = a
a = b
```

Test 8 Comment échanger les valeurs des deux variables `a` et `b` ?

§2 Identifiants et mots clefs

Les **identifiants** (ou **noms**) peuvent être choisis assez librement. Retenons quelques principes:

- Le premier caractère peut être une lettre minuscule ou majuscule.
- Les caractères suivants sont des lettres minuscules ou majuscules, des chiffres, ou des underscore _.
- Éviter les accents, `Python` s'en accommode très bien, les outils tiers souvent bien moins…
- Certains mots sont réservés, et donc interdit comme identifiants

```
The following identifiers are used as reserved words,
or keywords of the language, and cannot be used as
ordinary identifiers.
They must be spelled exactly as written here:

False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

§3 Sémantique précise de l'affectation

En python, il est crucial de bien comprendre le fonctionnement de l'affectation `a = b`:

1. Si `b` est déjà l'identifiant d'un objet, recopier l'identité de l'objet dans `a`.
2. Sinon, évaluer l'expression `b` et créer un objet python ayant cette valeur (et ce type) et faire pointer `a` vers cet objet.

Dans tout les cas, on affecte un nom à un objet existant.

Voici quelques conséquences de ce comportement. Dans la pratique, il faut distinguer le tas d'exécution (qui contient les objets) et les espaces de noms. Voir `http://pythontutor.com`!

Python 3 Shell

```
>>> a = "Bonjour Bob"
>>> b = "Bonjour Bob"
>>> a, b
('Bonjour Bob', 'Bonjour Bob')
>>> id(a), id(b)
(140367134555184, 140367134555248)
>>> a == b
True
>>> a is b
False
>>>
```

Python 3 Shell

```
>>> a = "Bonjour Bob"
>>> b = a
>>> a, b
('Bonjour Bob', 'Bonjour Bob')
>>> id(a), id(b)
(140367134555312, 140367134555312)
>>> a == b
True
>>> a is b
True
>>>
```

Si deux noms font référence à un même objet, cela ne lie pas magiquement les deux noms. Réaffecter un nom ne réaffecte pas le second.

Python 3 Shell

```
>>> x = 23580
>>> y = x
>>> x = 2
```

```
>>> x, y
(2, 23580)
```

Lorsque l'on exécute `y = x`, cela ne signifie pas qu'il seront égaux pour toujours. Réaffecter `x` laisse `y` seul (imaginez le bazar dans le cas contraire).

Python 3 Shell

```
>>> x = "Bonjour"
>>> x = "Bob"
```

`Python` garde le nombre de références associées à un objets, et détruit automatiquement cet objets lorsqu'aucun nom ne s'y réfère. C'est ce que l'on appelle le **ramasse miettes**. Cela signifie que l'on a pas besoin de supprimer d'objets, il disparaissent tout seuls lorsque l'on a plus besoin d'eux.

Python 3 Shell

```
>>> nombres = [1, 2, 3]
>>> autres = nombres
```

Lorsqu'une valeurs possède plusieurs noms, il est facile de se perdre et de penser comme deux noms et deux valeurs. NON! On a deux noms mais une seule valeur.

Après le code précédent, nous avons une liste, référencée par deux noms, ce qui peut entrainer quelques surprises.

Python 3 Shell

```
>>> nombres = [1, 2, 3]
>>> autres = nombres
>>> nombres
[1, 2, 3]
>>> autres
[1, 2, 3]
>>> nombres[1] = 333
>>> nombres
[1, 333, 3]
>>> autres
[1, 333, 3]
```

Notons bien la différence avec le cas suivant, nous avons deux listes, référencées chacune par un nom.

Python 3 Shell

```
>>> nombres = [1, 2, 3]
>>> autres = [1, 2, 3]
>>> nombres == autres
True
>>> nombres is autres
False
```

```
>>> nombres[1] = 333
>>> nombres
[1, 333, 3]
>>> autres
[1, 2, 3]
```

Les éléments d'une liste sont eux même des références

Python 3 Shell

```
>>> nombres = [1, 2, 3]
>>> x = nombres[1]
```

Python est un langage typée dynamiquement, cela signifie que les nom n'ont pas de type. Tout nom peut faire référence à n'importe quelle valeur.

```
1 x = 12
2 x = "Bonjour"
3 x = [1, 2, 3]
4 x[1] = "Deux"
```

Un nom peut faire référence à un entier, puis une chaine de caractères, puis à une fonction et enfin à un module.

Ce comportement est à éviter pour des questions de lisibilité, mais le langage Python s'en moque.

Cela permet de comprendre pourquoi les instructions suivantes permettent d'«échanger deux variables». On utilise pour cela un nouveau type d'objet, les tuples.

Python 3 Shell

```
>>> a = 12
>>> b = 34
>>> a, b = b, a  #  lire: (a, b) = (b, a)
>>> a
34
>>> b
12
```

1.3 FONCTIONS

§1 Paramètres formels ou effectifs

Définition 9

Les paramètres figurant dans l'en-tête d'une fonction se nomment des **paramètres formels**. Ceux fournis lors de l'appel de la fonction se nomment **paramètres effectifs**.

§2 Espace de noms (namespace), variable locale

Le passage de paramètre suit le même fonctionnement qu'une affectation: le paramètre effectif est soit une référence à un objet existant soit une expression dont la valeur donne un objet; le paramètre formel est une **variable locale** qui lors de l'appel recevra une référence à cet objet.

Le passage d'argument est une affectation.

```
1 def expo(x):
2     y = 2**x
3     return y
```

Python 3 Shell

```
>>> nombre = 17
>>> expo(nombre)
131072
>>> nombre
17
>>> expo(8)
256
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

Python 3 Shell

```
>>> x = 23
>>> y = 5
>>> expo(8)
256
>>> x
23
>>> y
5
>>>
```

§3 Exécution d'une fonction

Dans la suite, nous parlerons de la fonction appelante pour identifier la fonction qui effectue l'appel à la fonction invoquée; cette dernière sera identifiée comme étant la fonction appelée.

Lorsqu'une fonction est appelée par une autre fonction, la séquence du traitement est la suivante :

1. Création de l'espace de nom associé à l'instance de la fonction appelée : les **variables locales** sont crées.
2. Passage des arguments (initialisation des variables locales) entre la fonction appelante et la fonction appelée. *Même processus que l'affectation*.
3. Exécution de la fonction appelée.
4. Terminaison de la fonction (`return` explicite ou implicite (`return None`).
5. Destruction de l'espace de nom associé à l'instance de la fonction.

Exemple 10

```
1 def multipliepremier(s, val):
2     s[0] = s[0] * val
3     return None
```

Python 3 Shell

```
>>> a = [1, 2, 3]
>>> multipliepremier(a, 8)
>>> a
[8, 2, 3]
>>>
```

§4 Portée des variables, variables globales

```
1 def expo(x):
2     """Fonction qui calcule l'exponentielle de base a de x"""
3     y = a**x
4     return y
```

Python 3 Shell

```
>>> expo(8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "code.py", line 12, in expo
    y = a**x
NameError: name 'a' is not defined
```

Python 3 Shell

```
>>> a = 2
>>> expo(8)
256
>>> a = 3
>>> expo(8)
6561
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
>>>
```

- Toute variable affectée est par défaut *locale*.
- Si Python a besoin de la valeur d'une variable `x`, il va chercher le nom `x`
  - dans l'espace local des variables ;
  - si `x` ne s'y trouve pas, dans l'espace de nom *global* ;
  - si `x` ne s'y trouve pas non plus, il renvoie un message d'erreur.
- Dans le cadre d'une structuration avec plusieurs niveaux de fonctions, si la `fonction_2` définit la variable `x` et est appelée par `fonction_1`, alors la `fonction_1` ne regardera pas dans l'espace des noms local de `fonction_2` pour savoir si `x` est définie.

L'utilisation de variables globales pourrait correspondre à la définition des constantes du problème. Cela peut avoir du sens pour des constantes physique (accélération de la gravité, permittivité du vide,...).

Néanmoins, il vaut mieux ne *jamais* avoir recourt aux variables globales (non constantes).

Il est préférable d'ajouter un paramètre à votre fonction.

```python
def expo(a, x):
    y = a**x
    return y
```

Si vous tenez aux variables globales, il vaut mieux les déclarer dans le corps de votre fonction:

```python
def expo(x):
    global a
    y = a**x
    return y
```

- Les noms n'ont pas de type,
- les valeurs n'ont pas de portée.

CHAPITRE 3

LOGIQUE, INVARIANTS ET VÉRIFICATION D'ALGORITHMES

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> Edsger W. Dijkstra

> One does not need to give a formal proof of an obviously correct program; but one needs a thorough understanding of formal proof methods to know when correctness is obvious.
>
> John C. Reynold

Problème:

- Comment concevoir un programme afin d'avoir une *vision* aussi *claire* que possible des situations dynamiques qu'il engendre lors de son exécution ?
- L'objectif de la programmation est une *lisibilité* aussi claire que possible du programme, afin de pouvoir le communiquer entre «programmeurs» et surtout de pouvoir fournir une démonstration plus ou moins rigoureuse de sa validité (faire la preuve qu'il fournira les résultats désirés).

Comment atteindre au mieux ces objectifs? Avec des techniques adéquates de programmation.

3.1 SPÉCIFICATION

Définition 1

On appelle **spécification** d'un programme un quadruplet formé de

- une désignation et un typage des données,
- une désignation et un typage des résultats,
- une **pré condition** définissant des contraintes sur les données restreignant les valeurs admises par le typage.
- une **post condition** définissant le résultat en fonction des données.

Exemple 2

Voici un modèle pour le problème de calcul de la racine carrée entière par défaut.

- Donnée: `n` entier (nombre dont on veut calculer la racine).
- Résultat: `r` entier (racine carrée entière par défaut de `n`).
- Pré condition: $n \geq 0$.
- Post condition:
$$r^2 \leq n < (r+1)^2.$$

Définition 3

Un programme est **totalement correct** s'il calcule le résultat attendu et termine.

Définition 4

Un programme est **partiellement correct** s'il calcule le résultat attendu sous l'hypothèse que toutes ses itérations se terminent.

3.2 EXÉCUTION SYMBOLIQUE

§1 Triplet de Hoare

Définition 5

L'exécution d'un programme A sur une valeur symbolique de données définies par l'expression p qui donne une valeur symbolique résultat définie par l'expression q est notée

$$\{\ p\ \}\quad A\quad \{\ q\ \}.$$

Cette notation, appelée **triplet de Hoare**, signifie que si on exécute le programme A sur n'importe quelle données vérifiant l'assertion $\{\ p\ \}$, alors on obtient une valeur vérifiant l'assertion $\{\ q\ \}$ si le programme termine.

Exemple 6

```
{n == n0}    n = n - 1    {n == n0 - 1}
```

Test 7

Le triplet de Hoare suivant est-il valide?

```
{n > 0}    n = n + 1    {n >= 0}
```

Exercice 1. Déterminer les triplet de Hoare valides. On supposera que `i`, `j` et `N` sont des entiers.

1. { $i == 1$ } `j = i` { $i == j == 1$ }.
2. { $i == 1$ } `i = j` { $i == j == 1$ }.
3. { $0 \leq i < N$ } `i = i + 1` { $0 < i \leq N$ }.
4. { `True` } `i = j + 1` { $i < j$ }.
5. { $i == 1$ } `i = 0` { `True` }.
6. { $i == 0$ } `i = 1` { `False` }.
7. { `False` } `i = 1` { $i == 0$ }.

§2 Règle d'affectation

Théorème 8

Règle d'affectation simple
Étant donnée une assertion Q

$$\{ Q[V \mapsto E] \} \quad V = E \quad \{ Q \}$$

est un triplet de Hoare valide.

Pour l'instant, considérons que $Q[V \mapsto E]$ signifie «l'assertion Q dans laquelle nous avons substituer E à chaque occurrence de V».

Exercice 2. Déterminer la plus faible précondition P pour que les triplets suivants soient valides dans l'axiomatique de Hoare. Toutes les variable sont supposées de type `int`.

1. `{P} x = 0 {x == 0}`
2. `{P} x = 0 {x >= 0}`
3. `{P} x = 0 {x > 0}`
4. `{P} x = 0 {y > 0}`
5. `{P} x = x + 1 {x == 1}`
6. `{P} x = x + 1 {x > 0}`
7. `{P} x = x + 1 {x == y}`
8. `{P} x = x - 1 {x == y - 1}`

§3 Implications

Théorème 9

On suppose

$$\{ P \} \quad \mathtt{A} \quad \{ Q \}.$$

Si de plus, $P' \implies P$ *et* $Q \implies Q'$, *alors*

$$\{ P' \} \quad \mathtt{A} \quad \{ Q' \}.$$

On dit que l'on peut **renforcer** la pré condition ou **relacher** la post condition.

Exemple 10

On a

```
{n - 1 >= 0}    n = n - 1    {n >= 0}
```

Si n est un entier, alors on a l'implication

$$n > 0 \implies n - 1 \geq 0$$

et donc

```
{n > 0}    n = n - 1    {n >= 0}
```

Exercice 3. Les affirmation de correction suivantes sont-elles valides?

1. `{n < 0} n = n * n {n > 0}`
2. `{n != 0} n = n * n {n > 0}`
3. `{True} n = n * n {n >= 0}`
4. `{False} n = n * n {n < 0}`

§4 Composition séquentielle

Théorème 11

Si $\{ P \} C_0 \{ Q \}$ *et* $\{ Q \} C_1 \{ R \}$ *alors*

$$\{ P \} \quad C_0; C_1 \quad \{ R \}$$

Exemple 12

Montrer la validité du code suivant

```
1 # x == x0 et y == y0
2 z = x
3 x = y
4 y = z
5 # x == y0 et y == x0
```

Exercice 4. Vérifier la validité de

```
# i ≥ 0 et y == x^i
y = y * x
i += 1
# i ≥ 0 et y == x^i
```

Exercice 5. Vérifier la validité de

```
# k > 0 et f == Fib(k) et g == Fib(k − 1)
t = f + g
g = f
f = t
# k >= 0 et f == Fib(k + 1) et g = Fib(k)
```

où $Fib(n)$ désigne le terme d'indice n de la suite de Fibonacci, définie par $Fib(0) = 0$, $Fib(1) = 1$ et pour tout $n > 1$, $Fib(n) = Fib(n-1) + Fib(n-2)$.

Exercice 6. Quelle pré condition P permet au code suivant d'être correct?

```
# P
t = f + g
g = f
f = t
k = k + 1
# k > 0 et f == Fib(k) et g == Fib(k − 1)
```

Exercice 7. Les implémentation alternative des spécification de l'exercice **5** sont-elles valides?

1.

```
# k > 0 et f == Fib(k) et g == Fib(k − 1)
t = f
f = f + g
g = t
# k >= 0 et f == Fib(k + 1) et g = Fib(k)
```

2.

```
# k > 0 et f == Fib(k) et g == Fib(k − 1)
f = f + g
g = f - g
# k >= 0 et f == Fib(k + 1) et g = Fib(k)
```

Exercice 8. Les assertions de corrections suivantes sont elles correctes? Si oui, en donner une preuve; si non, donner la plus faible pré condition nécessaire pour avoir un code correct.

```
# x == x0 et y == y0
x = x - y
y = x + y
x = y - x
# x == y0 et y == x0
```

§5 Conditionnelle

Théorème 13

On suppose

$$\{ P \text{ et } B \}\, C \,\{ Q \} \qquad \text{et} \qquad P \text{ et } (\text{non } B) \implies Q.$$

Alors

$$\{ P \} \quad \boxed{\texttt{if B:}\ \texttt{C}} \quad \{ Q \}$$

Ou plus généralement,

Théorème 14

On suppose

$$\{ P \text{ et } B \}\, C_0 \,\{ Q \} \qquad \text{et} \qquad \{ P \text{ et } (\text{non } B) \}\, C_1 \,\{ Q \}.$$

Alors

$$\{ P \} \quad \boxed{\begin{array}{l}\texttt{if B:}\\ \quad \texttt{C\_0}\\ \texttt{else:}\\ \quad \texttt{C\_1}\end{array}} \quad \{ Q \}$$

On a des résultat analogue avec l'utilisation de `elif`, qui n'est qu'une concaténation d'un bloc `else` avec un bloc `if`.

On peut présenter le résultat sous cette forme.

```
# {P}
if B:
    # {P et B}
    C_0
    # {Q}
else:
    # {P et non B}
    C_1
    # {Q}
# {Q}
```

Exemple 15

Nous souhaitons écrire une fonction qui affecte à la variable `m` le plus grand entier parmi `x` et `y`.

- `x`, `y`, `m` sont des entiers.
- Pré condition: aucune
- Post condition: $m = \max(x, y)$, c'est-à-dire

$$m \geq x \text{ et } m \geq y \text{ et } (m = x \text{ ou } m = y).$$

On propose le code suivant

```
1 if x > y:
2     m = x
3 else:
4     m = y
```

Nous devons gérer les deux cas, c'est-à-dire démontrer

1.
```
# {True and x > y}
    m = x
# {m >= x and m >= y and (m == x or m == y)}
```

2.
```
# {True and (not x > y)}
    m = y
# {m >= x and m >= y and (m == x or m == y)}
```

3.3 Preuve des segments itératifs

§1 Invariant de boucle

À première vue, il peut sembler difficile de traiter le cas des boucle. On pourrait croire qu'il faudrait considérer chaque itération séparement!

En pratique, la clef pour raisonner sur un programme avec une boucle while est d'exhiber une assertion bien choisie, appelée **invariant de boucle**. Intuitivement, un invariant de boucle décrit «l'état intermédiaire» d'une boucle dont on a déjà parlé. On doit toujours avoir conscience d'un invariant en écrivant une boucle, même s'il n'est pas totalement formalisé.

Une fois bien compris le concept d'invariant et comment les obtenir, les invariant ne servent pas seulement à *vérifier* le code, mais permettent d'*écrire* un code correct par construction!

Théorème 16

On suppose

$$\{ I \text{ et } B \} \quad C \quad \{ I \}$$

Alors

$\{ I \}$ `while B: C` $\{ I \text{ et non } B \}$

On peut présenter le résultat sous cette forme:

```
# {I}
while B:
    # {I and B}
    C
    # {I}
# {I and not B}
```

Remarquons que l'invariant apparait quatre fois:

- L'assertion I doit être *préservée* par chaque exécution du code `C`, le corps de la boucle, lorsque `B`, la condition de boucle, est vraie. Ainsi, I est à la fois une pré condition et une post condition pour le corps de la boucle!

  Remarquons toutefois que I peut être faux *temporairement* durant l'exécution de `C`.

- L'assertion I est la pré condition de la boucle, et doit donc être vraie *avant* l'exécution de la boucle. En pratique, l'invariant découle souvent des spécifications, avec l'ajout éventuel d'un pré traitement des données.

- L'assertion I et la négation de B sont alors vraies *après* l'exécution de la boucle, indépendamment du nombre d'exécution du corps de la boucle (éventuellement aucune).

Dans de nombreux exemples, l'assertion «I et non B» implique la post condition spécifiée. Parfois, il est nécessaire d'ajouter un post traitement pour finaliser le programme.

Exemple 17

- `x` et `y` sont des flottants. `i` et `n` sont des entiers.
- Pré condition: aucune.
- Post condition: $y = x^n$.

On complétera le code suivant:

```
#

while i != n:
    y = y * x
    i = i + 1
# y == x^n
```

§2 Terminaison d'une boucle

Dans l'exemple précédent,

```
i = 0
y = 1
while i != n:
    # Invariant: i ≥ 0 et y == x^i
    y = y * x
    i = i + 1
# y == x^n
```

l'exécution pour $n < 0$ ne termine pas. Si nous modifions la pré condition en $n \geq 0$, alors il est facile de vérifier que le code précédent termine. La variable `i` est initialisée à 0, et sa valeur est incrémentée (de 1) à chaque itération. La condition de boucle `i != n`, avec $n \geq 0$, finira par devenir fausse et la boucle terminera. On en déduit que le code est correct pour tout $n \geq 0$.

Dans d'autres exemples, il est plus dur de vérifier la terminaison de l'algorithme. On utilise pour cela une **fonction de terminaison** ou **variant de boucle**.

Définition 18

Un **variant** de boucle est une expression *entière* N telle que, pour chaque exécution du corps de la boucle,

- $N > 0$ initialement,
- N décroit *strictement* durant l'exécution.

En pratique, on ajoute une condition correspondant à $N > 0$ dans l'invariant de boucle.

Exemple 19

Ajoutons la condition `i <= n` à l'invariant précédent (l'inégalité large n'est pas une erreur). Le nouvel invariant devient

$$0 \leq i \leq n \quad \text{et} \quad y = x^i.$$

Dans cet exemple, le variant de boucle est $n - i$. Il est facile de voir que lorsque ce nouvel invariant est vrai ainsi que la condition de boucle ($i \neq n$) alors $n - i > 0$. De plus, chaque exécution du corps de boucle incrément `i` de 1, et fait donc décroitre strictement $n - i$.

§3 Exercices

Exercice 9. Utiliser l'invariant proposé pour prouver que, après exécution de ce code, la variable `f` a pour valeur $Fib(n)$.

```
f, g, k = 1, 0, 1
while k != n:
    # Invar (k > 0 et f == Fib(k) et g == Fib(k - 1))
    t = f + g
    g = f
    f = t
    k = k + 1
```

Exercice 10. Utiliser l'invariant proposé pour prouver que cette fonction est (partiellement) correcte.

```python
def racine_int(n):
    """Retourne la racine carrée entière par défaut d'un entier.

    :n: nombre dont on veut calculer la racine (entier >= 0)
    :returns: r : racine carrée entière par défaut (entier >= 0)

    Pré condition : n >= 0
    Post condition : r ** 2 <= n et n < (r + 1) ** 2
    """
    r = 0
    while (r + 1) ** 2 <= n:
        # Invariant: r ** 2 <= n
        r = r + 1
    return r
```

Exercice 11. Effectuer la preuve totale de l'algorithme suivant, qui affecte la somme des éléments de $b_1, \ldots, b_{10}$ à `s`.

```python
i, s = 10, 0
while i != 0:
    s = s + b[i]
    i = i - 1
```

Precondition	`b` est une liste de nombre de longueur >10
Invariant	$P : 0 \leq i \leq 10$ et $s = \sum_{i+1 \leq k \leq 10} b[k]$
Variant	$t : i$
Postcondition	$R : s = \sum_{1 \leq k \leq 10} b[k]$

Exercice 12. Effectuer la preuve totale de l'algorithme suivant qui affecte à i la plus grande puissance de 2 inférieure à n.

```python
i = 1
while 2 * i <= n:
    i = 2 * i
```

Precondition	$0 < n$
Invariant	$P : 0 < i \leq n$ et $\exists p, i = 2^p$
Variant	$t : n - i$
Postcondition	$R : 0 < i \leq n < 2 * i$ et $\exists p, i = 2^p$

Exercice 13. Le n-ième nombre de Fibonacci f_n ($n > 0$) est défini par $f_0 = 0$, $f_1 = 1$ et $f_n = f_{n-1} + f_{n-2}$ pour $n > 1$. Effectuer la preuve totale de l'algorithme suivant qui calcule le n-ième nombre de Fibonacci.

```
1 i, a, b = 1, 1, 0
2 while i < n:
3     i = i + 1
4     a, b = a + b, a
```

Precondition	$n > 0$
Invariant	$P : 0 \leq i \leq n$ et $a = f_i$ et $b = f_{i-1}$
Variant	$t : n - i$
Postcondition	$R : a = f_n$

Exercice 14. Effectuer la preuve totale de l'algorithme suivant qui calcule le quotient q et le reste r dans la division euclidienne de x par y.

```
1 q, r = 0, x
2 while r >= y:
3     r = r - y
4     q = q + 1
```

Precondition	$x \geq 0$ et $0 < y$
Invariant	$P : 0 \leq r$ et $0 < y$ et $q * y + r = x$
Variant	$t : r$
Postcondition	$R :$ (compléter)

Exercice 15. Effectuer la preuve totale de l'algorithme suivant qui détermine la valeur maximale dans une liste `A`.

```
1 def maximum(A):
2     n = len(A)
3     m = A[0]
4     for i in range(1, n):
5         if A[i] > m:
6             m = A[i]
7     return m
```

CHAPITRE 4

COMPLEXITÉ

Montrer qu'une version d'un programme est plus efficace qu'une autre n'est, à priori, pas aisé. La succession ou l'imbrication des tests et des boucles et la multitude des possibilités fait qu'il n'est généralement pas possible ou raisonnable d'évaluer l'efficacité de l'algorithme pour chaque cas possible.

Dans ce chapitre, nous donnerons un aperçu des techniques et méthodes d'estimation de l'efficacité d'un algorithme. En particulier nous définirons les notations $\mathcal{O}, \Theta, \Omega$ qui permettront de classer les algorithmes selon leur type d'efficacité sans devoir détailler le nombre exact d'instructions exécutées par l'algorithme.

Généralement, il existe plusieurs méthodes pour résoudre un même problème. Il faut alors en choisir une et concevoir un algorithme pour cette méthode de telle sorte que cet algorithme satisfasse le mieux possible aux exigences. Parmi les critères de sélection d'une méthode et en conséquence d'un algorithme, deux critères prédominent : la *simplicité* et l'*efficacité* de cet algorithme.

Si parfois ces critères vont de paire, bien souvent ils sont contradictoires ; un algorithme efficace est, en effet, bien souvent compliqué et fait appel à des méthodes fort élaborées. Le concepteur doit alors choisir entre la simplicité et l'efficacité.

Si l'algorithme devra être mis en œuvre sur un nombre limité de données qui ne demanderont qu'un nombre limité de calculs, cet algorithme doit de préférence être simple. En effet un algorithme simple est plus facile à concevoir et a moins de chance d'être erroné que ne le sera un algorithme complexe.

Si, par contre, un algorithme est fréquemment exécuté pour une masse importante de données, il doit être le plus efficace possible.

Au lieu de parler de l'efficacité d'un algorithme, on parlera généralement de la notion opposée, à savoir, de la complexité d'un algorithme.

La complexité d'un algorithme ou d'un programme peut être mesurée de diverses façons. Généralement, le temps d'exécution est la mesure principale de la complexité d'un algorithme. D'autres mesures sont possibles dont :

- la quantité d'espace mémoire occupée par le programme et en particulier par ses variables ;
- la quantité d'espace disque nécessaire pour que le programme s'exécute ;
- la quantité d'information qui doit être transférée (par lecture ou écriture) entre le programme et les disques ou entre le programme et des serveurs externes via un réseau;
- ...

Nous n'allons pas considérer de programme qui échange un grand nombre d'informations avec des disques ou un autre ordinateur.

En général nous analyserons le temps d'exécution d'un programme pour évaluer sa complexité. La mesure de l'espace mémoire occupé par les variables sera un autre critère qui pourra être utilisé ici.

De façon générale, la complexité d'un algorithme pour un ensemble de ressources donné est une mesure de la quantité de ces ressources utilisées par cet algorithme.

Le critère que nous utiliserons dans la suite est un nombre d'actions élémentaires exécutées. Nous prendrons soin, avant toute chose, de préciser le type d'actions prises en compte et si nous voulons calculer une complexité minimale, moyenne ou maximale.

Nous verrons que, généralement, la complexité d'un algorithme est exprimée par une fonction qui dépend de la taille du problème à résoudre.

4.1 Notion de complexité

§1 Motivation

Problème du tri
Tri d'un tableau de 10 millions de nombres

- Tri insertion $c_1 n^2$, $c_1 = 2$, sur ordinateur rapide, 10^{10} ips
- Tri fusion $c_2 n \lg n$, $c_2 = 50$, sur ordinateur lent, 10^7 ips.

Tri insertion : 20000 secondes (plus de 5.5 heures). Tri fusion : 1163 secondes (moins de 20 minutes).

§2 Exemple: la recherche du minimum

Le travail nécessaire pour qu'un programme s'exécute dépend de la 'taille' du jeu de données fourni.

Par exemple, si nous analysons la fonction `indice_min`

```
1 def indice_min(A):
2     n = len(A)
3     argmin = 0  # Indice du candidat
4     for i in range(n):
5         if A[i] < A[argmin]:
6             argmin = i
7     return argmin
```

qui recherche l'indice de l'élément de valeur minimale dans un tableau `A`. Il est intuitivement normal que cet algorithme prenne un temps proportionnel à `len(A)`. On note ce temps $T(len(A))$.

Dénotons `n = len(A)`. L'entier n est la longueur de la liste et donc par extension, également la taille du problème.

Redonnons ici la partie traitement de cet algorithme en ayant eu soin de transformer[1] le *for* en *while* pour mieux détailler chaque étape d'exécution de l'algorithme. Nous supposons `n = len(A)` connu dans le programme :

```
1 n = len(A)
2 argmin = 0
3 i = 0
4 while i < n:
5     if A[i] < A[argmin]:
6         argmin = i
7     i = i + 1
```

Pour obtenir des résultats qui restent valables quel que soit l'ordinateur utilisé, il faut réaliser des calculs simples donnant une approximation dans une unité qui soit indépendante du processeur utilisé.

On peut, par exemple, faire l'approximation que

- chaque assignation se fait en temps constant C_a,
- chaque branchement se fait en temps constant C_b,
- chaque comparaison se fait en temps constant C_c,
- chaque incrémentation se fait en temps constant C_i,
- chaque accès à un élément d'un tableau par son indice se fait en temps constant C_t,

Avec nos hypothèses, les instructions des lignes 1, 2 prennent chacune C_a unité de temps d'exécution.

La boucle `while` de la ligne 3 à 6 s'exécute $n-1$ fois, mais

- le test `i < n` s'effectue n fois. Il y a donc nC_c unités de temps pour la comparaison.
- Le test `A[i] < A[argmin]` de la ligne 4 prend $2C_t + C_c$ unités; il est effectué $n-1$ fois et donc la ligne 4 utilisera au total $(2C_t + C_c)n$ unités de temps.
- L'assignation de la ligne 5 prend une C_a unités. Elle n'est effectuée que lorsque le test du `if` est vérifié.

  Si l'on ne désire pas uniquement faire une seule mesure sur un jeu de données bien précis, il faut donc expliciter si l'on désire connaître le temps d'exécution dans *le meilleur des cas*, T_{min}, *le pire des cas*, T_{max}, ou *en moyenne*, T_{moyen}. Ici,

  - T_{min} est donné dans le cas où le test du `if` n'est jamais vérifié, ce qui signifie que le minimum se trouve à la première composante. La ligne 5 n'est alors jamais exécutée.
  - T_{max} est donné dans le cas où le test du `if` est à chaque fois vérifié, ce qui signifie que la liste est dans l'ordre strictement décroissant. La ligne 5 est exécutée $n-1$ fois pour un coût de $(n-1)C_a$.

[1] le code n'est pas tout à fait équivalent

- La ligne 6 est exécutée $n-1$ fois pour un coût de $(n-1)C_i$.
- Le nombre de branchements est difficile à évaluer… Comptons nC_b pour le `while` et nC_b pour le `if`…

Finalement, nous obtenons

$$\begin{aligned} T_{max}(n) &= 2C_a + nC_c + (2C_t + C_c)n + (n-1)C_a + (n-1)C_i + 2nC_b \\ &= \left(C_a + 2C_c + C_i + 2C_t + 2C_b\right) n + C_a - C_i. \end{aligned}$$

$$\begin{aligned} T_{min}(n) &= 2C_a + nC_c + (2C_t + C_c)n + 0C_a + (n-1)C_i + 2nC_b \\ &= \left(2C_c + C_i + 2C_t + 2C_b\right) n + 2C_a - C_i. \end{aligned}$$

C'est peut lisible, mais retenons

$$T_{min}(n) = An + B \qquad\qquad T_{max}(n) = Cn + D.$$

§3 Exemple: recherche de la position d'un élément

```
1 n = len(A)
2 i = 0
3 while i < n and A[i] != elem:
4     i = i + 1
```

$$T_{min} = C_a + C_c + C_t + C_c + C_{and} = \text{constante}$$

$$T_{max} = \cdots = An + B.$$

4.2 Notation asymptotique

§1 Notation $\mathcal{O}, \Omega, \Theta$

T_{min} T_{max} Notation $\Theta(n)$.

4.3 Modèle de calcul

1. Unité.
2. Séquence.
3. if
4. boucles while
5. boucles for
6. appel de fonction.

Complexité des opérations sur les listes

D'après `https://wiki.python.org/moin/TimeComplexity`

Opération	Cas moyen	Pire cas	Pire cas amorti
Copy	$O(n)$	$O(n)$	$O(n)$
append	$O(1)$	$O(n)$	$O(1)$
insert	$O(n)$	$O(n)$	$O(n)$
get item	$O(1)$	$O(1)$	$O(1)$
set item	$O(1)$	$O(1)$	$O(1)$
delete item	$O(n)$	$O(n)$	$O(n)$
iteration	$O(n)$	$O(n)$	$O(n)$
x in s	$O(n)$	$O(n)$	$O(n)$
min(s), max(s)	$O(n)$	$O(n)$	$O(n)$
len(s)	$O(1)$	$O(1)$	$O(1)$

Exemples

Exemple 1

```python
def signal_carre(t):
    if t % 2 <= 1:
        return -1
    else:
        return 1
```

Exemple 2

```python
def fibonacci(n):
    u, v = 0, 1
    for i in range(n):
        t = u + v
        u = v
        v = t
    return u
```

Exemple 3

```python
def maximum(liste):
    n = len(liste)
    candidat = liste[0]
    for k in range(n):
        if liste[k] > maxval:
            candidat = liste[k]
    return candidat
```

Exemple 4

```python
def position(liste, elem):
    n = len(liste)
    k = 0
    while k < n and elem != liste[k]:
        k += 1

    if k == n:
        return None
    else:
        return k
```

Exemple 5

```python
def est_croissante(liste):
    n = len(liste)
    k = 1
    while k < n and liste[k - 1] <= liste[k]:
        k += 1
    return k == n
```

Exemple 6

```python
def somme_double(f, p, q):
    acc = 0
    for i in range(p):
        for j in range(q):
            acc = acc + f(i, j)
    return acc


def somme_tsup(f, n):
    acc = 0
    for i in range(1, n):
        for j in range(i, n):
            acc = acc + f(i, j)
    return acc
```

Exemple 7

```python
def produit2(A, B):
    tA = taille(A)
    tB = taille(B)
    C = matrice(tA[0], tB[1])
    for i in range(tA[0]):
        for j in range(tB[1]):
            for k in range(tA[1]):
                C[i][j] += A[i][k] * B[k][j]
    return C
```

EXERCICES

Dans chacun des exercices, les calculs de complexité se réfèrent à un modèle de machine à accès aléatoire (RAM), à processeur unique avec exécution séquentielle.

Exercice 1. Quelle valeur est retournée par la fonction suivante? Évaluer la complexité en temps en fonction de n.

```
def alpha(n):
    r = 0
    for i in range(1, n):
        for j in range(i+1, n+1):
            for k in range(1, j + 1):
                r = r + 1
    return r
```

Exercice 2. Quelle valeur est retournée par la fonction suivante? Évaluer la complexité en temps en fonction de n.

```
def bravo(n):
    r = 0
    for i in range(1, n + 1):
        for j in range(1, i + 1):
            for k in range(j, i + j + 1):
                r = r + 1
    return r
```

Exercice 3. Quelle valeur est retournée par la fonction suivante? Évaluer la complexité en temps en fonction de n.

```
def charlie(n):
    r = 0
    for i in range(1, n + 1):
        for j in range(1, i + 1):
            for k in range(j, i + j + 1):
                for l in range(1, i + j - k + 1):
                    r = r + 1
    return r
```

Exercice 4. Quelle valeur est retournée par la fonction suivante? Évaluer la complexité en temps en fonction de n.

```
def delta(n):
    r = 0
    for i in range(1, n + 1):
        for j in range(i + 1, n + 1):
            for k in range(i + j - 1, n + 1):
                r = r + 1
    return r
```

Exercice 5. On considère les quatre algorithmes suivants.

```python
# Code A
i = 1
j = 1
while i <= m and j <= n:
    i = i + 1
    j = j + 1
```

```python
# Code B
i = 1
j = 1
while j <= n:
    if i <= m:
        i = i + 1
    else:
        j = j + 1
        i = 1
```

```python
# Code C
i = 1
j = 1
while j <= n:
    if i <= m:
        i = i + 1
    else:
        j = j + 1
```

```python
# Code D
i = 1
j = 1
while i <= m or j <= n:
    i = i + 1
    j = j + 1
```

Déterminez la complexité en temps des algorithmes suivants parmi ces possibilités

1. $\Theta(\max(m, n))$
2. $\Theta(\min(m, n))$
3. $\Theta(m + n)$
4. $\Theta(mn)$

Dans chaque cas, m et n sont des entiers.

Exercice 6. Estimer la complexité en temps de chacun de ces fragments de code.

1.
```python
sum = 0
n = N
while n > 0:
    for i in range(n):
        sum += 1
    n = n // 2
```

2.
```python
sum = 0
i = 1
while i < N:
    for j in range(i):
        sum += 1
    i = i * 2
```

3.
```python
sum = 0
i = 1
while i < N:
    j = 0
    while j < N:
        sum += 1
        j = j + 1
    i = i * 2
```

Exercice 7. Soit P un polynôme et x un réel. Calculer la valeur de P en x n'est pas gratuit! En général (c'est-à-dire si aucun terme n'est nul), l'expression

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d$$

Comporte $d+1$ termes et son calcul nécessite donc d additions. Le calcul de $a_i x^i$ nécessite de même i multiplications, ce qui donne au total $0 + 1 + \cdots + d = \dfrac{d(d+1)}{2}$ multiplications. Mais on peut faire mieux, c'est le schéma de Horner.

```
k, s = d, a[d]
while k > 0:
    k, s = k - 1, s * x + a[k-1]
```

1. Effectuer la preuve totale de l'algorithme de Horner. On donne

$$\begin{array}{ll} \mathit{Precondition} & 0 \leq d \\ \mathit{Invariant} & P : 0 \leq k \leq d \text{ et } s * x^k = \displaystyle\sum_{i;k\leq i\leq d} a_i * x^i \\ \mathit{Variant} & t : k \\ \mathit{Postcondition} & R : s = \displaystyle\sum_{i;0\leq i\leq d} a_i * x^i \end{array}$$

2. Calculer la complexité en temps et en espace du schéma de Horner.
3. Comparer avec l'algorithme naïf.

Remarque En plus pythonesque :

```
s = 0
for c in reversed(a):
    s = s * x + c
```

CHAPITRE 5

RÉCURSIVITÉ

5.1 FONCTIONS RÉCURSIVES

§1 Premier exemple

Définition 1

On dit qu'une fonction f est récursive si son exécution peut provoquer un ou plusieurs appels de f elle-même.

Exemple 2

On peut programmer un algorithme d'exponentiation rapide de manière récursive.
Par exemple, on souhaite calculer a^{34} en utilisant uniquement les opérations arithmétiques élémentaires $(+, -, \times, \div)$.

- Pour calculer a^{34}, on calcule $a^{17} * a^{17}$.
- Pour calculer a^{17}, on calcule $a^8 * a^8 * a$.
- Pour calculer a^8, on calcule $a^4 * a^4$.
- Pour calculer a^4, on calcule $a^2 * a^2$.
- Pour calculer a^2, on calcule $a * a$.

Ce calcul requière 6 multiplications au lieu de 33.

```
1 def expo(a, n):
2     """Retourne a ** n"""
3     if n == 0:
```

```
        return 1
    else:
        q = n // 2
        p = expo(a, q)
        p = p * p
        if n % 2 == 1:
            p = p * a
        return p
```

L'algorithme précédent montre bien les deux étapes d'un algorithme récursif. Pour résoudre un problème X appliqué à un objet N, on dégage les deux cas suivants:

- Les cas récursifs : on identifie le même problème, mais appliqué à un objet M de «taille» inférieure et dont la résolution permet de résoudre le problème initial.
- Les cas d'arrêts ou cas limites : on identifie des cas où le problème X peut se résoudre «facilement» sans être ramené à un sous-problème.

Dans le problème du calcul de puissance, pour calculer a^{34} il suffit de calculer $p = a^{17}$ et d'effectuer la multiplication $p * p$ (cas récursif). Pour le calcul de a^0, on ne peut décomposer le problème. Mais dans ce cas, le calcul de a^0 peut se faire directement.

La présence d'un cas d'arrêt et sa bonne gestion est indispensable sinon l'algorithme récursif risque de ne pas terminer.

§2 Gestion de la mémoire à l'exécution

Lors de son exécution, un programme Python gère des espace de noms (namespace). Un espace de nom est un **mappage** (mapping) de noms vers des objets.

En pratique, la gestion des objets et des espaces de nom en mémoire implique deux espaces mémoire:

1. La **pile d'exécution** (**runtime stack**) qui va contenir l'espace de nom global et les espaces de noms locaux;
2. Le **tas d'exécution** (**runtime heap**) qui contient les objets.

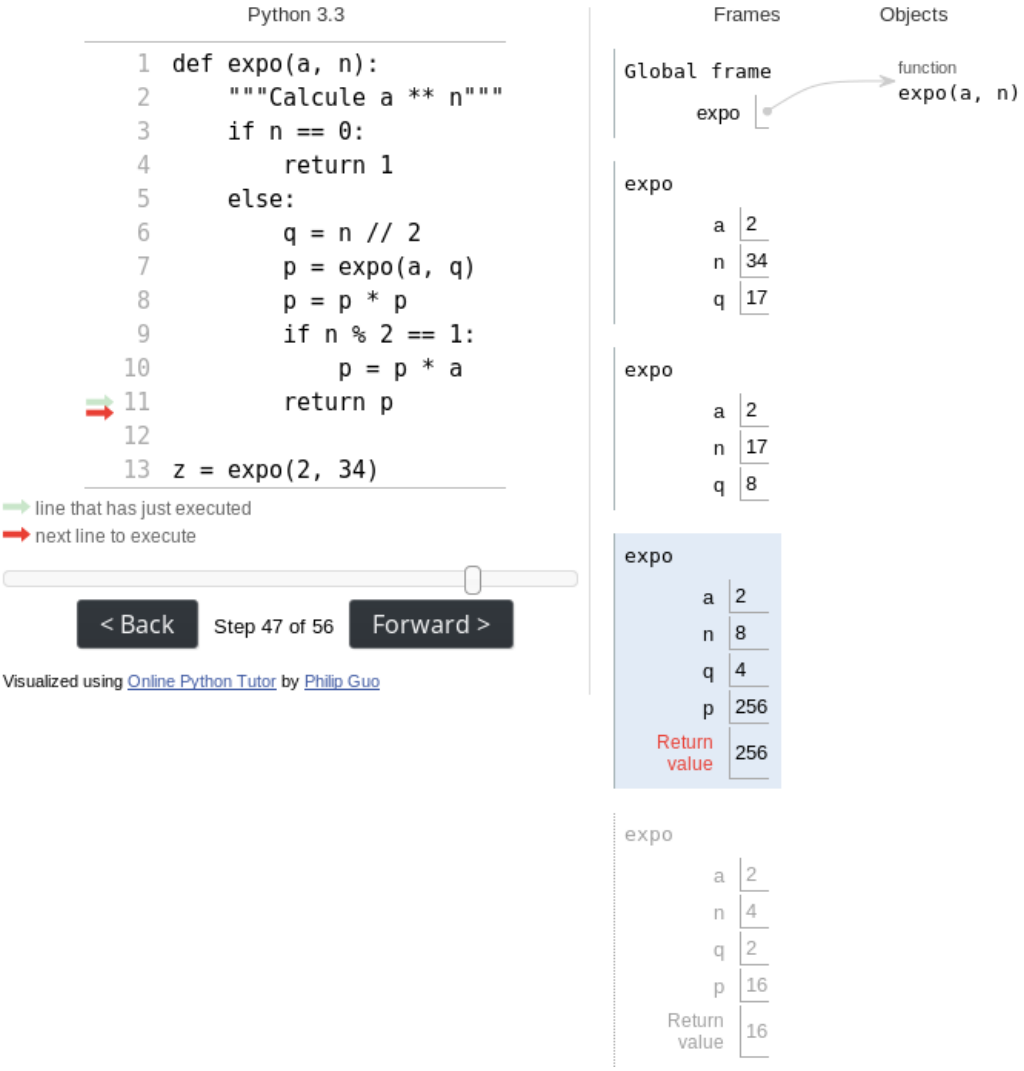
À chaque fois qu'une instance de fonction `truc()` est appelée, une trame associée à cette instance est crée et mise comme un élément supplémentaire de la pile d'exécution.

Cette trame contient en particulier, l'espace de nom local à cette instance. Cette trame sur la pile d'exécution continuera a exister jusqu'à la fin de l'exécution de cette instance de `truc()`. Ensuite (au moment du `return`), la trame est enlevée de la pile d'exécution et on revient à l'instance appelant dont la trame se trouve maintenant au sommet de la pile d'exécution.

Python limite, arbitrairement, le nombre d'appels imbriqués à 1000.

Python 3 Shell

```
>>> expo(2, 2**1000)

...
```



VisualisationavecPythonTutor

```
  File "expo_rapide.py", line 11, in expo
    p = expo(a, q)
  File "expo_rapide.py", line 11, in expo
    p = expo(a, q)
  File "expo_rapide.py", line 7, in expo
    if n == 0:
RuntimeError: maximum recursion depth exceeded in comparison
>>>
```

Même si cette limitation peut paraître basse, elle est suffisante pour des algorithmes comme l'exponentiation rapide qui effectuent $\lceil \log_2 n \rceil$ appels récursifs.

Cette limite peut être augmentée avec le code suivant[1]

```python
import sys
sys.setrecursionlimit(100000)
```

§3 Dangers de la récursivité

Prenons l'exemple de la suite de Fibonacci définie par

$$\begin{cases} F_0 &= 0 \\ F_1 &= 1 \\ \forall n \geq 2, F_n &= F_{n-1} + F_{n-2} \end{cases}$$

La mise en œuvre récursive naïve qui suit la définition de la suite de Fibonacci est immédiate

```python
def fibo(n):
    """Retourne le terme de rang n de la suite de Fibonacci"""
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibo(n - 1) + fibo(n - 2)
```

On vérifie facilement que cette fonction termine et est correcte.

- On a $\mathcal{A} = \mathbb{N}$.
- On utilise l'ensemble $\mathbb{N}$ munie de la relation $\leq$ usuelle avec $\varphi(n) = n$.
- Le seul cas de base est $n = 0$, auquel on peut ajouter le «cas limite» $n = 1$.
- Si $n \in 0, 1$ (cas limites), alors l'appel `fibo(n)` termine (au plus quatre lignes exécutées) et retourne une valeur correcte (`0` ou `1`).
- Si $n \geq 2$. L'appel `fibo(n)` fait apparaitre deux appels récursifs `fibo(n - 1)` et `fibo(n - 2)`. On a bien

  $$\varphi(n-1) = n-1 < \varphi(n) = n \quad \text{et} \quad \varphi(n-2) = n-2 < \varphi(n) = n.$$

  En supposant que ces deux appels terminent et retourne une valeur correcte (c'est-à-dire F_{n-1} et F_{n-2}), alors l'appel `fibo(n)` termine et retourne

[1] c'est apparemment plus compliqué sous windows…

`fibo(n - 1) + fibo(n - 2),`

c'est-à-dire $F_{n-1} + F_{n-2} = F_n$.

Par conséquent, l'appel `fibo(n)` termine et est correct pour tout $n \in \mathbb{N}$.

Ce n'est cependant pas une façon judicieuse de calculer la suite de Fibonacci,...

Python 3 Shell

```
>>> fibo(0)
0
>>> fibo(1)
1
>>> fibo(2)
1
>>> fibo(5)
5
>>> fibo(8)
21
>>> fibo(20)
6765
>>> fibo(30)
832040
>>> fibo(35)
9227465
>>> fibo(40)  # ZZZzzzz...
```

Voici une version modifiée avec

- une variable globale `nb_appels` qui compte le nombre d'appels à la fonction `fibo`
- Un `print` qui affiche les paramètre d'appels.

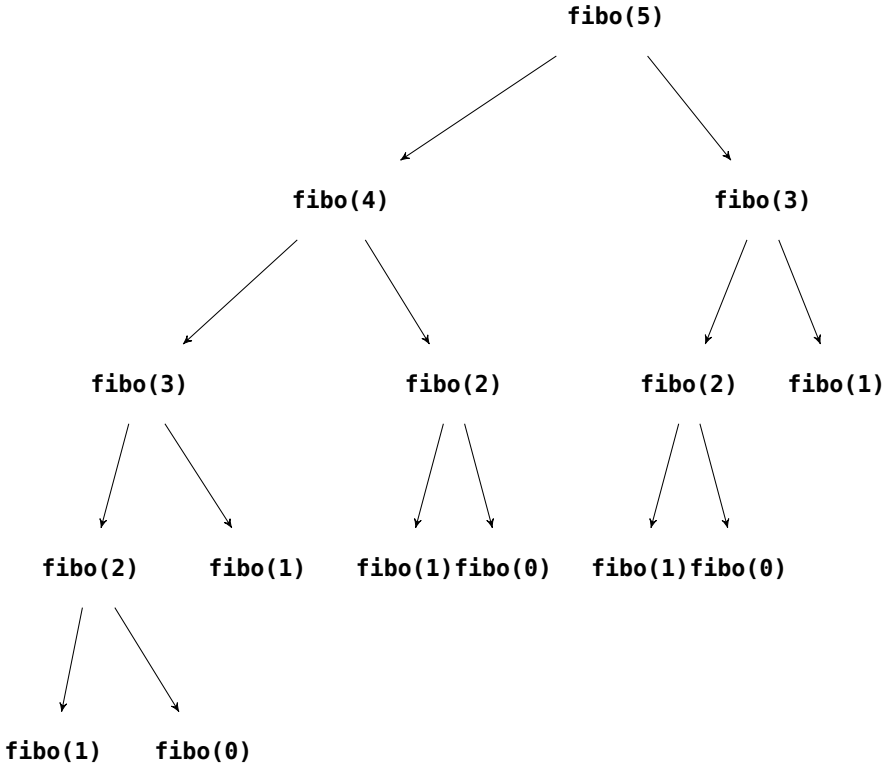
```
1  def fibo(n):
2      """Retourne le terme de rang n de la suite de Fibonacci"""
3      global nb_appels
4
5      nb_appels += 1
6      print("Appels n°", nb_appels, "avec n=", n)
7      if n == 0:
8          return 0
9      elif n == 1:
10         return 1
11     else:
12         return fibo(n - 1) + fibo(n - 2)
```

Python 3 Shell

```
>>> nb_appels = 0
>>> fibo(5)
Appels n° 1 avec n= 5
```

```
Appels n° 2 avec n= 4
Appels n° 3 avec n= 3
Appels n° 4 avec n= 2
Appels n° 5 avec n= 1
Appels n° 6 avec n= 0
Appels n° 7 avec n= 1
Appels n° 8 avec n= 2
Appels n° 9 avec n= 1
Appels n° 10 avec n= 0
Appels n° 11 avec n= 3
Appels n° 12 avec n= 2
Appels n° 13 avec n= 1
Appels n° 14 avec n= 0
Appels n° 15 avec n= 1
5
>>>
```

Avec `fibo(35)`, il y a 29860703 appels !



5.2 Terminaison et correction d'une fonction récursive

La preuve d'un algorithme consiste à vérifier que cet algorithme termine et renvoie le bon résultat. On distingue donc dans la preuve deux étapes : la preuve de terminaison et la preuve de correction.

§1 Un peu de théorie

Bien souvent, les valeurs successives associés à un des paramètre forment une suite strictement monotone qui finit par atteindre une des valeurs de base, ce qui assure la terminaison de la fonction récursives.

Définition 3

Soit $\preceq$ une relation d'ordre partielle ou totale sur un ensemble non vide E. On dit que $(E, \preceq)$ est un ensemble est **bien fondé** s'il n'existe pas de suite (infinie) strictement décroissante d'éléments de cet ensembles.

Un même ensemble peut en général être ordonné de plusieurs façons (être muni de plusieurs relations d'ordre). C'est en particulier le cas de l'ensemble $\mathbb{N} \times \mathbb{N}$ des couples d'entiers naturels qui possède plusieurs relations utiles.

Exemple 4

Regardons quelques exemples d'ensembles bien fondés.

- L'ensemble $\mathbb{N}$ muni de l'ordre usuel $\leq$ est bien fondé. Par contre $\mathbb{Z}$ ne l'est pas car il n'a en particulier pas d'élément minimal.
- L'ensemble $\mathbb{N}^2$ muni de l'ordre produit défini par

$$(a, b) \preceq (c, d) \iff a \leq c \text{ et } b \leq d.$$

  est bien fondé.
- L'ensemble $\mathbb{N}^2$ muni de l'ordre lexicographique défini par

$$(a, b) \preceq (c, d) \iff a < c \text{ ou } (a = c \text{ et } b \leq d).$$

  L'ordre lexicographique, généralisé à des n-uplets de lettres de longueur quelconque, est celui utilisé pour classer les mots dans un dictionnaire.

On peut alors énoncer le principe d'induction.

Théorème 5

Soit $(E, \preceq)$ un ensemble bien fondé. Si un prédicat $\mathcal{P}$ sur E vérifie

$$\forall x \in E, (\forall y \prec x, \mathcal{P}(y)) \implies \mathcal{P}(x).$$

Alors, pour tout $x \in E$, $\mathcal{P}(x)$.

Dans le cas de $(\mathbb{N}, \leq)$, c'est le principe de récurrence avec prédécesseurs.

Définition 6

Soit $\preceq$ une relation d'ordre partielle ou totale sur un ensemble non vide E. Soit $A \subset E$ et $m \in A$. On dit que m est un élément minimal de E si le seul élément x de E vérifiant

$x \leq m$ est $x = m$, c'est-à-dire s'il n'existe aucun élément de A strictement plus petit que m.

Remarque Si l'ordre est total, la notion d'élément minimal coïncide avec celle de minimum.

Proposition 7 *Un ordre $\preceq$ sur un ensemble E est bien fondé si, et seulement si toute partie non vide de E admet au moins un élément minimal.*

Exemple 8

- Pour $(\mathbb{N}, \leq)$, il n'y a qu'un seul élément minimal, 0, c'est un minimum.
- Pour $\mathbb{N}^2$ muni de l'ordre produit ou de l'ordre lexicographique, il n'y a qu'un seul élément minimal, $(0,0)$ et c'est un minimum.
- $\mathbb{N}^2 \setminus \{ (0,0) \}$ muni de l'ordre produit a deux éléments minimaux $(1,0)$ et $(0,1)$.
- $\mathbb{N}^2 \setminus \{ (0,0) \}$ muni de l'ordre lexicographique, il n'y a qu'un seul élément minimal, $(0,1)$ et c'est un minimum.

Remarque Si M désigne l'ensemble des éléments minimaux alors la condition d'induction se décompose

- $\forall x \in M, \mathcal{P}(x)$,
- $\forall x \in E \setminus M, (\forall y \prec x, \mathcal{P}(y)) \implies \mathcal{P}(x)$.

Cette forme est plus proche du traitement habituel des fonctions récursives en informatique.

§2 Preuve de terminaison

Dans ce qui suite, on considère une fonction récursive f dont l'ensemble des arguments sera noté $\mathcal{A}$. Pour prouver la terminaison, on considère une fonction

$$\varphi : \mathcal{A} \to E$$

à valeurs dans un ensemble bien fondé $(E, \preceq)$.

Remarque Si les arguments sont déjà ordonnés de manière naturelle, la fonction φ est généralement l'identité. Sinon, il convient en général de prendre un indicateur de la taille de l'argument: longueur de la liste, taille ou hauteur d'un arbre…

La propriété d'induction permet de montrer le théorème suivant de terminaison.

Théorème 9 *Avec les notations précédentes, supposons que pour tout $x \in \mathcal{A}$,*

- *le calcul de $f(x)$ n'admet qu'un nombre fini d'appels d'arguments $\left(y_i\right)_{i=1\ldots N}$,*
- *pour tout $i = 1 \ldots N$, $\varphi(y_i)$ est strictement inférieur à $\varphi(x)$,*
- *$f(x)$ termine sous l'hypothèse que tous les appels $f(y_i)$ terminent.*

Alors, $f(x)$ termine pour tout argument $x \in \mathcal{A}$.

Méthode Pour une fonction récursive f dont l'ensemble des arguments sera noté $\mathcal{A}$, il est utile d'introduire

- $\mathcal{B}$ la partie de $\mathcal{A}$ constituée des **cas de base**, c'est-à-dire $b \in \mathcal{B}$ si, et seulement si $\varphi(b)$ est un élément minimal de $\varphi(\mathcal{A})$.
- $\mathcal{L}$ la partie de $\mathcal{A}$ constituée des **cas limite**, c'est-à-dire $x \in \mathcal{L}$ si, et seulement si $f(x)$ n'effectue pas d'appel récursifs.

Pour appliquer le théorème précédent, on vérifie alors que

- $f(x)$ termine pour tout $x \in \mathcal{L}$,
- Le calcul de $f(x)$ fait apparaitre des appels récursifs $f(y_i)$ en nombre fini et tels que $\varphi(y_i) \prec \varphi(x)$.

On a alors $\mathcal{B} \subset \mathcal{L}$ et on peut alors conclure que $f(x)$ termine pour tout x dans $\mathcal{A}$.

Exemple 10

La fonction `expo(a, n)` termine. On choisit $\varphi(a, n) = n$ et $E = \mathbb{N}$ muni de la relation $\leq$ usuelle. Ici[a] $\mathcal{A} = \mathbb{R} \times \mathbb{N}$, $\varphi(\mathcal{A}) = \mathbb{N}$. Il y a donc une infinité de cas de base de la forme $(a, 0)$ avec $a \in \mathbb{R}$: ce seront nos cas limite.

- L'appel `expo(a, 0)` termine pour tout flottant `a` (deux lignes sont exécutées).
- Les cas récursifs `expo(a, n)` avec $n \neq 0$ font apparaître qu'un seul appel récursif `expo(a, q)` avec `q < n` (ici `q == n // 2` et `n > 0`). C'est-à-dire $\varphi(a, q) < \varphi(a, n)$. Les autres lignes ne posent pas de problème de terminaison.

Par conséquent, ce programme termine bien pour tout argument $(a, n) \in \mathbb{R} \times \mathbb{N}$.

[a] Il y a bien sûr une limitation machine dû à la représentation des nombres flottants.

§3 Preuve de correction

De manière analogue à la preuve de terminaison, on peut prouver le résultat de correction suivant.

Théorème 11

Avec les notations du théorème de terminaison. On suppose que pour tout $x \in \mathcal{A}$,

- *le calcul de $f(x)$ n'admet qu'un nombre fini d'appels d'arguments $\left(y_i\right)_{i=1\ldots N}$,*
- *pour tout $i = 1 \ldots N$, $\varphi(y_i)$ est strictement inférieur à $\varphi(x)$,*
- *L'appel $f(x)$ donne le résultat correct sous l'hypothèse que tous les appels $f(y_i)$ donnent le résultat correct.*

Alors, pour tout $x \in \mathcal{A}$, $f(x)$ donne le résultat correct.

Méthode

Par commodité, on peut encore distinguer cas limites et cas récursifs dans les hypothèse du théorème. On montrer que

1. L'appel $f(x)$ donne un résultat correct pour tout $x \in \mathcal{L}$,
2. L'appel $f(x)$ fait apparaitre des appels récursifs $f(y_i)$ en nombre fini tels que $\varphi(y_i) \prec \varphi(x)$.

3. L'appel $f(x)$ donne le résultat correct sous l'hypothèse que tous les appels $f(y_i)$ donnent le résultat correct.

On peut alors conclure que $f(x)$ donne un résultat correct pour tout x dans $\mathcal{A}$.

Exemple 12

On reprend l'exemple pour l'appel `expo(a, n)`.

- L'appel `expo(a, 0)` retourne `1`, c'est-à-dire a^0 pour tout flottant `a`.
- Pour l'appel `expo(a, n)` avec `n != 0`. On suppose que le seul appel récursif `expo(a, q)` retourne correctement a^q. On peut alors écrire

```
q = n // 2
# q == n // 2
p = expo(a, q)
# q == n // 2 et p == a ** q
# donc
# q == n // 2 et p * p == (a ** q) * (a ** q)
# équivaut à
# q == n // 2 et p * p == a ** (2 * q)
p = p * p
# q == n // 2 et p == a ** (2 * q)
# Si n % 2 != 1 alors n == 2 * q et p == a ** n
# Sinon
if n % 2 == 1:
    # n % 2 == 1 et q = n // 2 et p == a ** (2 * q)
    # donc
    # 2 * q == n - 1 et p == a ** (n - 1)
    # donc
    # p * a == a ** n
    p = p * a
    # p == a ** n
# p == a ** n
```

Ainsi, l'appel `expo(a, n)` retourne `p`, c'est-à-dire a^n.

Par conséquent, l'appel `expo(a, n)` retourne a^n pour tout $(a, n) \in \mathbb{R} \times \mathbb{N}$.

CHAPITRE

6

DIVISER POUR RÉGNER

6.1 L'APPROCHE DIVISER-POUR-RÉGNER

Le paradigme **diviser-pour-régner** implique trois étapes à chaque niveau de la récursivité:

- *Diviser* le problème en un certain nombre de sous-problèmes qui sont des instances plus petites du même problème.
- *Régner* sur les sous-problèmes en les résolvant de manière récursive. Si la taille d'un sous-problème est suffisament réduite, on peut toutefois le résoudre directement.
- *Combiner* les solutions des sous-problèmes pour produire la solution du problème originel.

Quand les sous-problèmes sont suffisamment grands pour être résolus de manière récursive, nous parlons de **cas récursifs**. Quand les sous-problèmes deviennent suffisamment petits pour que nous le fassions plus de récursivité, nous disons que la récursivité «se termine» et que nous sommes arrivés au **cas de base**. Il peut arriver que, en plus des sous-problèmes qui sont des instances plus petites du même problème, nous ayons à résoudre des sous-problèmes qui ne sont pas tout à fait les mêmes que le problème original. Nous intégrerons la résolution de tels sous-problèmes à la phase «combiner».

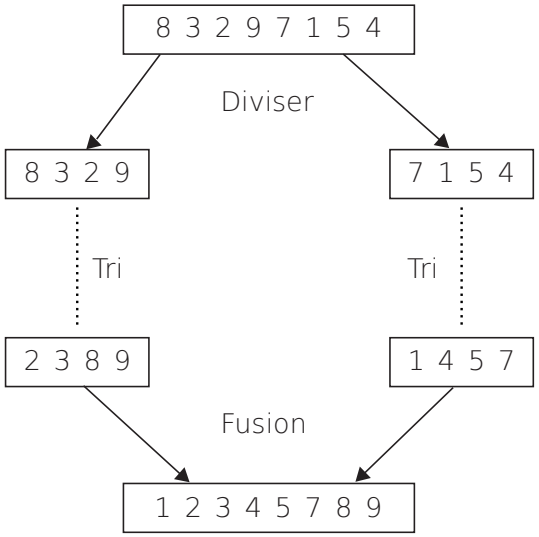
6.2 TRI PAR FUSION

§1 Présentation

L'algorithme du **tri par fusion** suit fidèlement la méthodologie diviser-pour-régner. Intuitivement, il agit de la manière suivante:

- *Diviser* : Diviser la suite de n éléments à trier en deux sous-suites de $n/2$ éléments (à 1 près) chacune.
- *Régner* : Trier les deux sous-suites de manière récursive en utilisant le tri par fusion.
- *Combiner*: Fusionner les deux sous-suites triées pour produire la réponse triée.

La récursivité s'arrête quand la la séquence à trier a une longueur 1, auquel cas il n'y a plus rien à faire puisqu'une suite de longueur 1 est déjà triée.

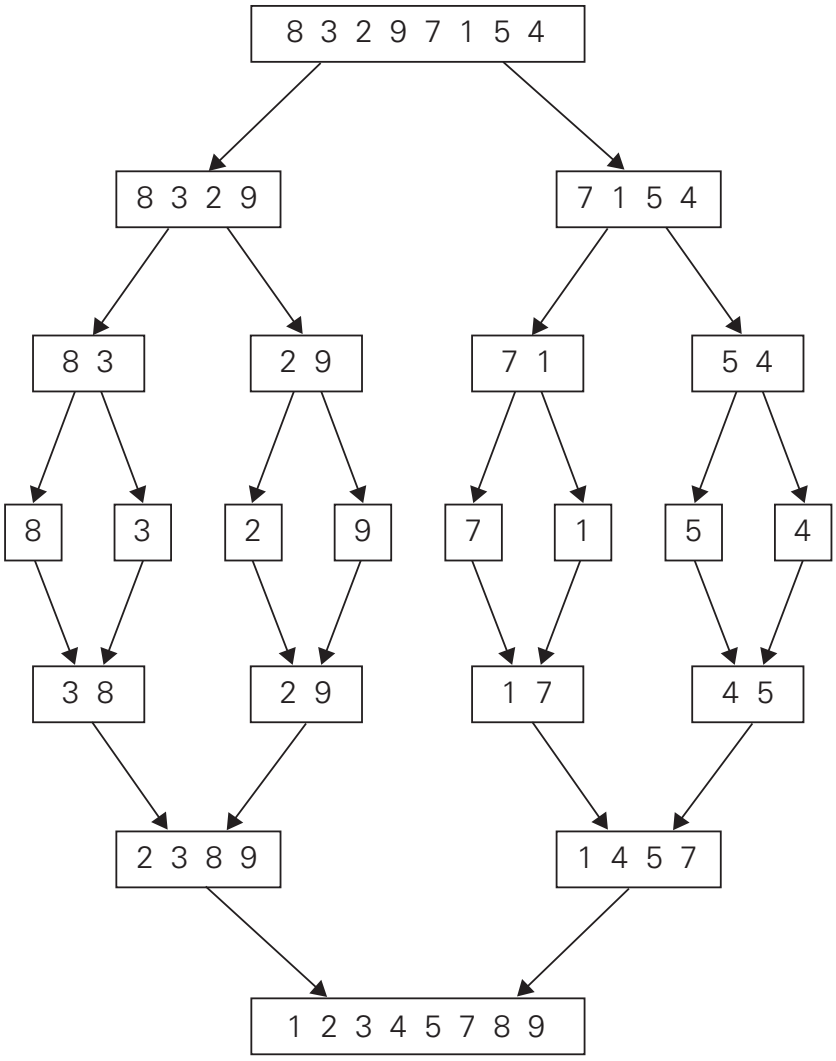


Quelques problèmes à régler.

- Que faire si A a un nombre impair d'éléments ?
- Comment passer les arguments ?
- Comment gérer les cas de bases ?

La difficulté principale est la fonction `Fusion(U, V)`, spécifiée par:

- Pré condition: `U` et `V` sont deux listes triées.
- Résultat: `L` est une permutation triée de `U + V` (concaténation de `U` et `V`).

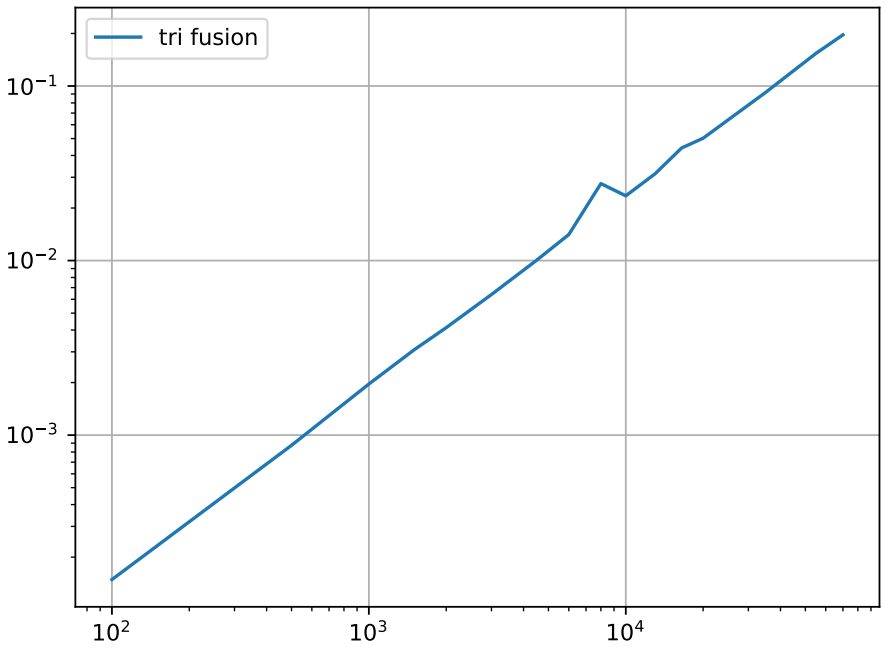

8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

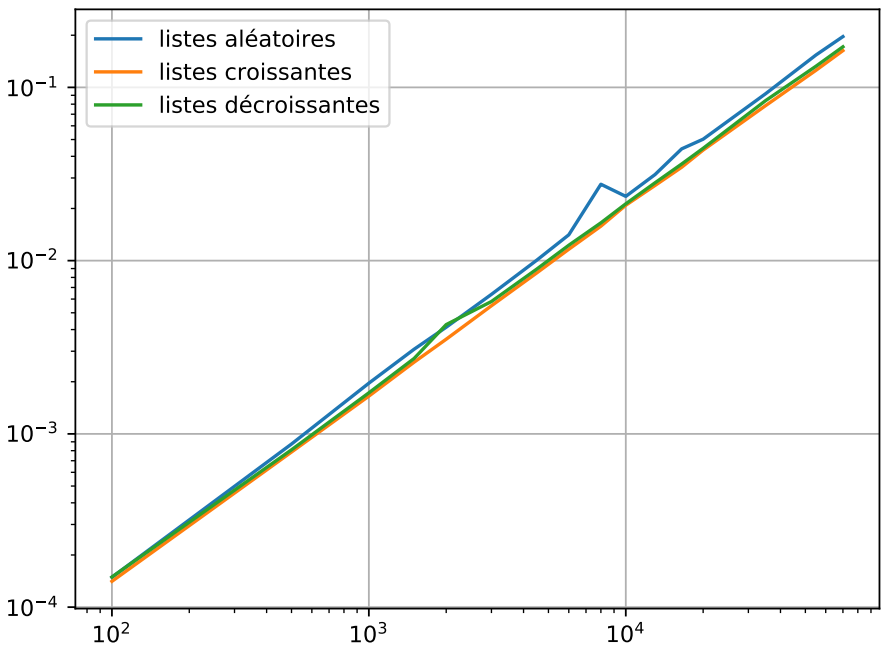
§2 Implémentation

```python
def fusion(U, V):
    n = len(U) + len(V)
    L = [None] * n
    i = 0
    j = 0
    for k in range(n):
        if i >= len(U):
            L[k] = V[j]
            j += 1
        elif j >= len(V):
            L[k] = U[i]
            i += 1
        elif U[i] <= V[j]:
            L[k] = U[i]
            i += 1
        else:
            L[k] = V[j]
            j += 1
    return L
```

```python
def tri_fusion(A):
    if len(A) <= 1:
        return A
    else:
        p = len(A) // 2
        U = tri_fusion(A[:p])
        V = tri_fusion(A[p:])
        return fusion(U, V)
```

§3 Complexités empiriques





§4 Étude du tri fusion

$$T(n) = \begin{cases} \Theta(1) & : n \leq 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & : n > 1. \end{cases}$$

qui a un comportement analogue à la relation de récurrence

$$T(n) = \begin{cases} c & : n = 1 \\ 2T(n/2) + cn & : n > 1. \end{cases}$$

On trouve

Complexité en temps pour le tri fusion

$$T(n) = \Theta(n \lg n).$$

Complexité en espace $S(n)$

$$\begin{aligned} S(n) &\leq kn + c + S(n/2) \\ &\leq kn + k\frac{n}{2} + k\frac{n}{4} + \cdots + k + c + \cdots + c \leq 2kn + c \lg n \end{aligned}$$

et $S(n) \geq \frac{k}{4} n$

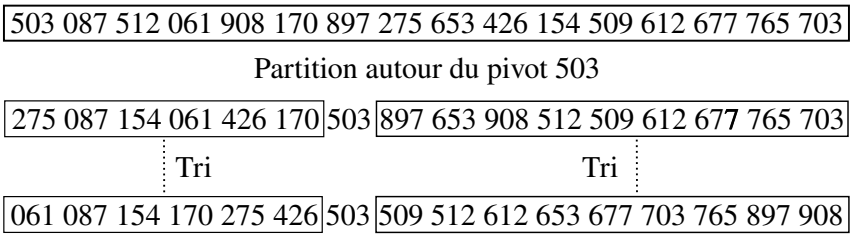
Complexité en espace pour le tri fusion

$$S(n) = \Theta(n).$$

6.3 TRI RAPIDE

§1 Présentation

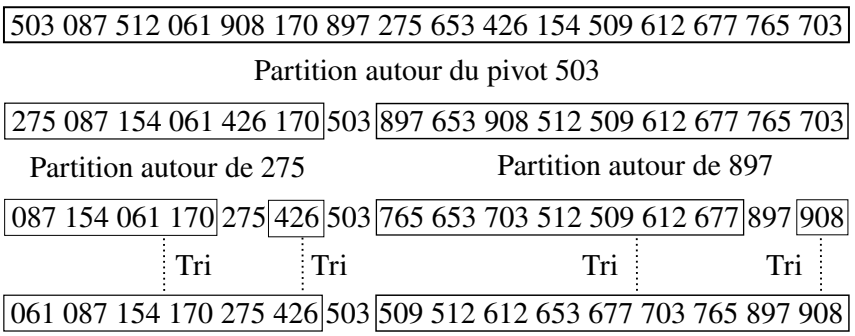
- *Diviser*: Le tableau `A[p:r]` est partitionné (réarrangé) en deux sous-tableaux (éventuelement vides) `A[p:q]` et `A[q + 1:r]` tels que chaque élément de `A[p:q]` soit inférieur ou égal à `A[q]` qui, lui-même, est inférieur ou égal à chaque élément de `A[q + 1:r]`. L'indice `q` est calculé dans le cadre de cette procédure de partitionnement.
- *Régner* : Les deux sous-tableaux `A[p:q]` et `A[q + 1:r]` sont triés par des appels récursifs au tri rapide.
- *Combiner*: Comme les sous-tableaux sont déjà triés, aucun travail n'est nécessaire pour les combiner: le tableau `A[p:r]` tout entier est maintenant trié.



- Il est facile d'effectuer une opération de partition si l'on s'autorise une complexité mémoire en $O(n)$. Mais ce n'est plus un «tri rapide».

```
pivot = A[0]
for i in range(1, len(A))
    x = A[i]
    if x <= pivot:
        U.append(x)
    else:
        V.append(x)
tableau_partitionné = U + [pivot] + V
```

- Partition *en place*: Exemple du TP 505.



- Il est donc nécessaire de revoir les spécifications des fonctions `partition` et `tri_rapide` afin qu'elle acceptent des bornes inférieures et supérieures.

- `partition_bornes(A, p, q)`

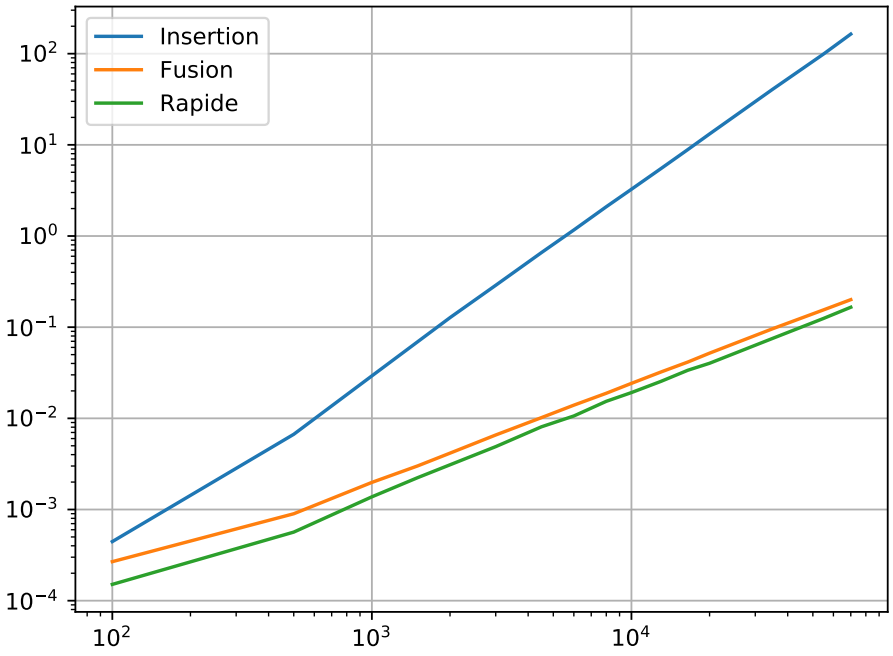
- `tri_rapide_bornes(A, p, q)`

§2 Implémentation

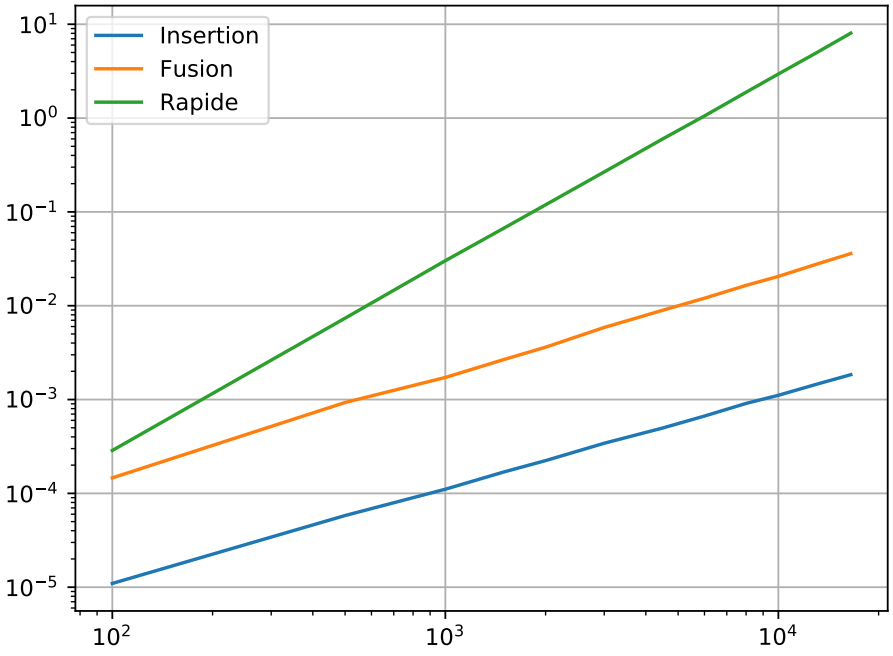
```python
def partition_bornes(A, p, q):
    pivot = A[p]
    i = j = p + 1
    while j < q:
        # pivot | < pivot | >= pivot | ???
        #   p   |         |i         |j
        if A[j] < pivot:
            A[j], A[i] = A[i], A[j]
            i = i + 1
        j = j + 1

    A[p], A[i - 1] = A[i - 1], A[p]
    return i - 1
```

```python
def tri_rapide_bornes(A, p, q):
    if q - p >= 2:  # q - p est le nombre d'éléments dans A[p:q]
        i = partition_bornes(A, p, q)
        tri_rapide_bornes(A, p, i)
        tri_rapide_bornes(A, i + 1, q)


def tri_rapide(A):
    tri_rapide_bornes(A, 0, len(A))
```

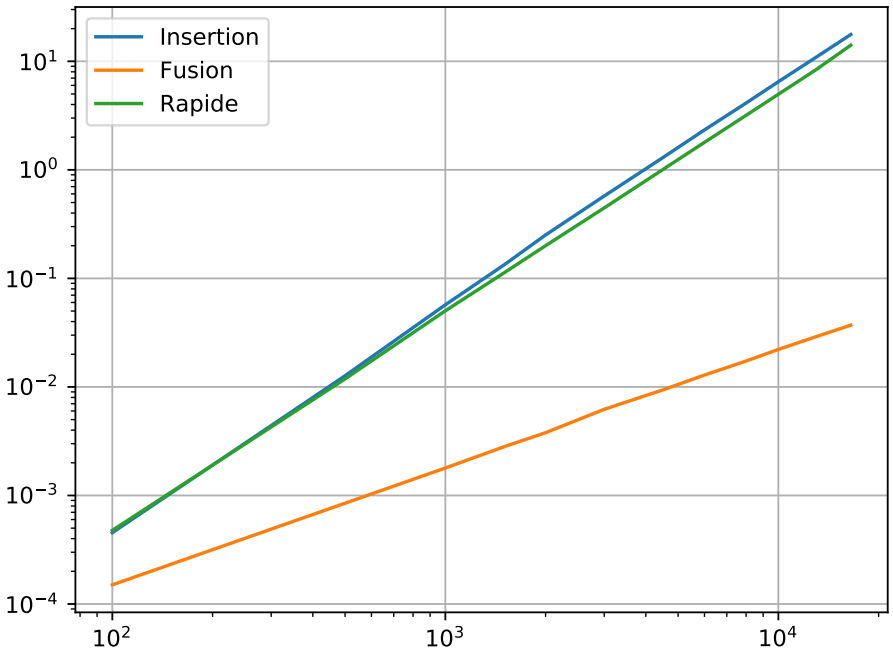
§3 Complexités empiriques



(Listes aléatoires de flottants)



(Listes croissante de flottants)



(Listes décroissante de flottants)

§4 Étude du tri rapide

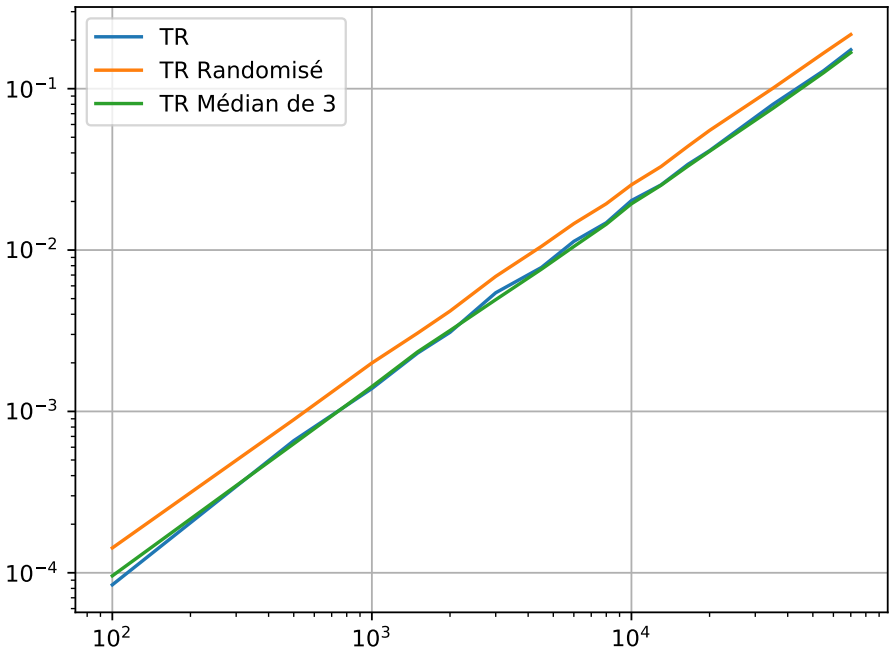
§5 Variations

Tri rapide randomisé

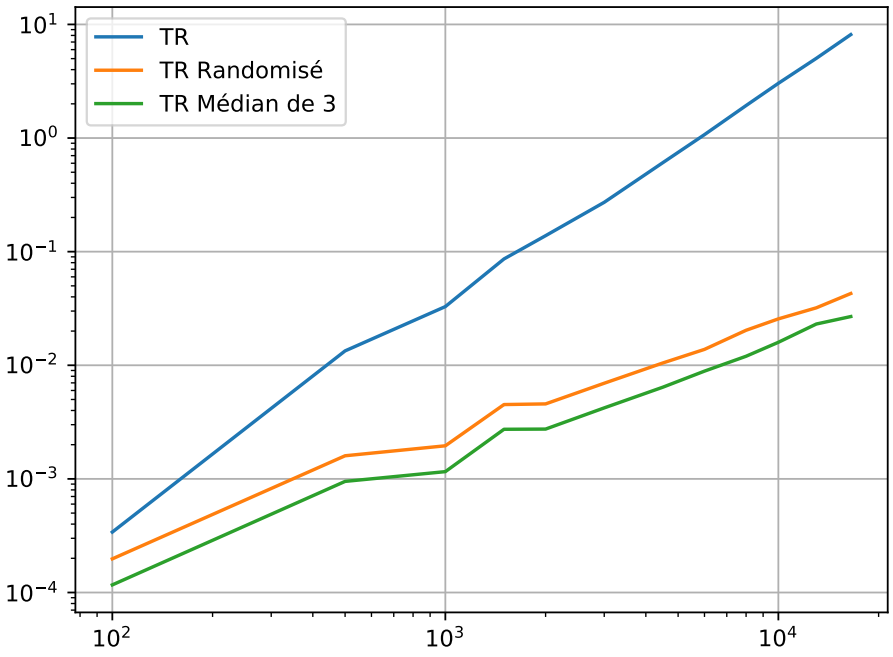
```python
def tri_rapide_randomise_bornes(A, p, q):
    # q - p est le nombre d'éléments dans A[p:q]
    if q - p >= 2:

        r = random.randrange(p, q)
        A[p], A[r] = A[r], A[p]

        i = partition_bornes(A, p, q)
        tri_rapide_randomise_bornes(A, p, i)
        tri_rapide_randomise_bornes(A, i + 1, q)


def tri_rapide_randomise(A):
    tri_rapide_randomise_bornes(A, 0, len(A))
```

Tri rapide avec pivot médian parmi 3 valeurs

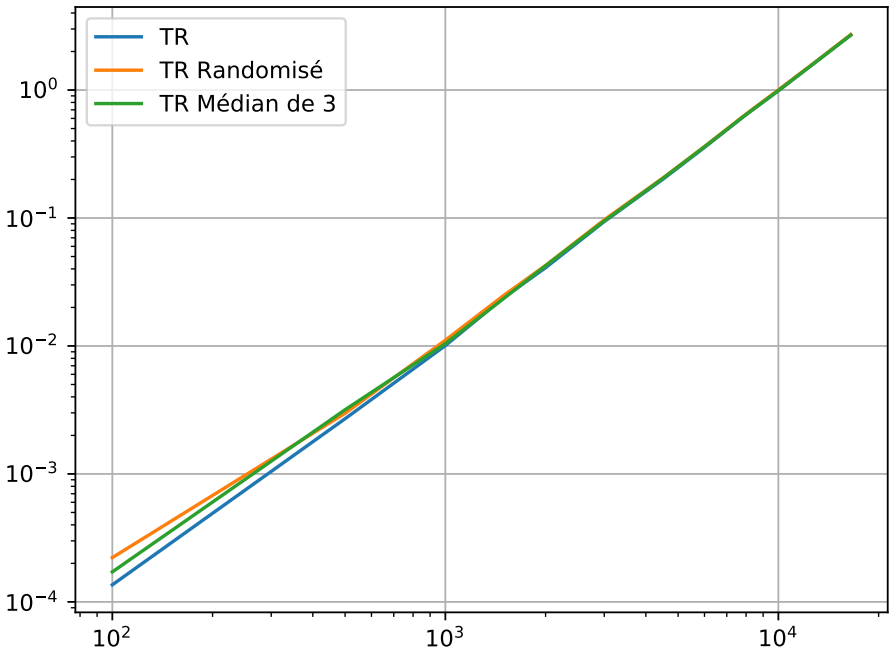
```python
def tri_rapide_med3_bornes(A, p, q):

    if q - p >= 3:  # Ainsi p < r < q - 1
        r = (p + q) // 2
        # On peut trouver le médian avec au plus 3 comparaisons,
        # Mais simplifions...
        if A[p] < A[r] <= A[q - 1] or A[p] > A[r] >= A[q - 1]:
            A[p], A[r] = A[r], A[p]

        elif A[p] < A[q - 1] <= A[r] or A[p] > A[q - 1] >= A[r]:
            A[p], A[q - 1] = A[q - 1], A[p]

    if q - p >= 2:
        i = partition_bornes(A, p, q)
        tri_rapide_med3_bornes(A, p, i)
        tri_rapide_med3_bornes(A, i + 1, q)


def tri_rapide_med3(A):
    tri_rapide_med3_bornes(A, 0, len(A))
```



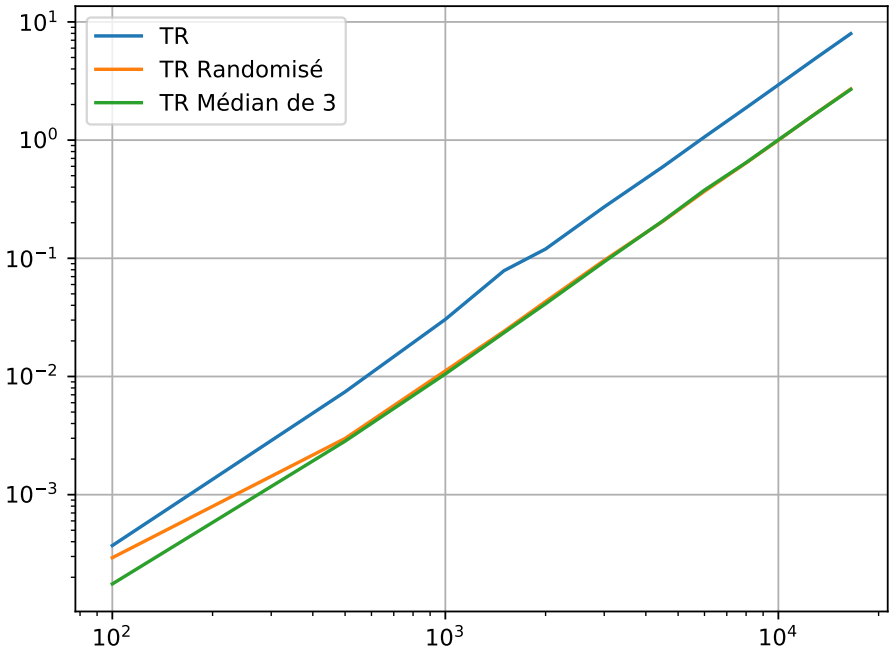
(Listes de n entiers aléatoires «distincts»)



(Listes triée de n entiers «distincts»)



(Listes n entiers prenants 3 valeurs)



(Listes triée de n entiers prenants 3 valeurs)

§6 Application au calcul de la médiane

```python
def selection(A, p, q, i):
    r = partition_bornes(A, p, q)
    if r == i:
        return A[r]
    elif r > i:
        return selection(A, p, r, i)
    else:
        return selection(A, r + 1, q, i)


def rang(A, i):
    B = list(A)
    return selection(B, 0, len(B), i)


def median(A):
    return rang(A, len(A) // 2)
```

6.4 En guise de conclusion

	Tri par insertion	Tri fusion	Tri rapide	Tri rapide randomisé
T_{min}	$\Theta(n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$ (espéré).
T_{max}	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n^2)$	$\Theta(n \lg n)$ (espéré).
T_{moy}	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$ (espéré).
Complexité en espace	$\Theta(1)$	$\Theta(n)$	$\mathcal{O}(n)$ $\Omega(\lg n)$	$O(\lg n)$ (espéré)
Sur place	Oui	Non	Oui ou presque	Oui ou presque
Stable	Oui	Oui	Non	Non

6.5 Complément

Soit $T : \mathbb{N}^\star \to \mathbb{R}_+^\star$ telle que

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n) \quad [n \to +\infty].$$

Alors $T(n) = \mathcal{O}(n \lg n)$.

Démonstration. Montrons par récurrence sur $n \geq n_0$, que $T(n) \leq cn \lg n$ où c est une constante à déterminer. Ici $\lg n$ désigne $\log_2 n = \ln n / \ln 2$, mais on peut aussi utiliser le logarithme népérien.

L'hypothèse sur $T(n)$ peut se réécrire

$$\forall n \geq n_0, T(n) \leq T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + kn.$$

où $g(n) = \mathcal{O}(n)$, c'est-à-dire qu'il existe $k > 0$ et $n_0 \in \mathbb{N}$ tels que

$$\forall n \geq n_0, g(n) \leq kn.$$

On a donc

$$\forall n \geq n_0, T(n) \leq T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + kn.$$

Quitte à changer, n_0, on peut supposer $n_0 \geq 2$. On peut alors choisir $c \geq 1$ (en fait $c \geq k + 1$) tel que

$$\forall n \in [\![n_0, 2n_0]\!], T(n) \leq cn \lg n$$

Remarque Par exemple

$$c = \max\left\{ k+1, \frac{T(n_0)}{n_0 \lg n_0}, \frac{T(n_0+1)}{(n_0+1)\lg(n_0+1)}, \dots, \frac{T(2n_0)}{2n_0 \lg(2n_0)} \right\}.$$

où lg désigne le logarithme de base 2.

Pour $n \geq n_0$, définissons le prédicat $R(n)$ par «$T(n) \leq cn \lg n$» de sorte que $R(n_0)$, $R(n_0 + 1)$,…, $R(2n_0)$ sont vrais par définition de c.

Soit $n \geq 2n_0$ tel que $R(n_0)$, $R(n_0 + 1)$, … , $R(n)$, alors

$$n_0 \leq \left\lfloor \frac{2n_0+1}{2} \right\rfloor \leq \left\lfloor \frac{n+1}{2} \right\rfloor \leq \frac{n+1}{2} \leq \left\lceil \frac{n+1}{2} \right\rceil \leq \frac{n+2}{2} \leq n$$

En particulier, $R\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right)$ et $R\left(\left\lceil \frac{n+1}{2} \right\rceil\right)$ sont vraies et donc

$$\begin{aligned}
T(n+1) &= T\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n+1}{2} \right\rceil\right) + g(n+1) \\
&\leq c\left\lfloor \frac{n+1}{2} \right\rfloor \lg\left\lfloor \frac{n+1}{2} \right\rfloor + c\left\lceil \frac{n+1}{2} \right\rceil \lg\left\lceil \frac{n+1}{2} \right\rceil + k(n+1) && \because R\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) \\
&\leq 2c\frac{n+1}{2} \lg \frac{n+2}{2} + k(n+1) \\
&= (n+1)(c\lg(n+2) - c\lg 2 + k) \\
&= \dots \text{...À modifier} \\
&\leq (n+1)(c\lg(n+1) + 1 - c + k) \\
&\leq c(n+1)\lg(n+1) && \text{en choisissant } c \geq k+1 \text{ au départ.}
\end{aligned}$$

D'où $R(n + 1)$.

Conclusion

Par récurrence, on a pour $n \geq n_0$, $T(n) \leq cn \lg n$, d'où

$$T(n) = \mathcal{O}(n \lg n).$$

∎

CHAPITRE

7

GRAPHES

7.1 VOCABULAIRE ET NOTATION

7.2 REPRÉSENTATION

CHAPITRE 8

LES NOMBRES ET LEURS REPRÉSENTATIONS

Exercice 1.

- $401302_{\text{cinq}} = \_\_\_\_\_{}_{\text{dix}}$
- $2341_{\text{cinq}} = \_\_\_\_{}_{\text{dix}}$
- $944_{\text{dix}} = \_\_\_\_\_{}_{\text{cinq}}$
- $289_{\text{dix}} = \_\_\_\_{}_{\text{cinq}}$

Exercice 2.

1. Convertir de la base deux à la base dix.

(a) 0101	(c) 1011	(e) 10000
(b) 1001	(d) 0110	(f) 10010

2. Convertir de la base dix à la base deux.

(a) 6	(c) 11	(e) 27
(b) 13	(d) 18	(f) 4

Exercice 3. Pour multiplier par dix un entier naturel exprimé en base dix, il suffit d'ajouter un 0 à sa droite, par exemple $12 \times 10 = 120$. Quelle est l'opération équivalente pour les entiers naturels exprimés en base deux ?

Exprimer en base deux les nombres 3, 6, 12 pour illustrer cette remarque.

Exercice 4.

1. Trouver la représentation en base seize des nombres 6725_{dix} et 18379_{dix}.

2. Trouver la représentation en base dix des nombres $ABCD_{\text{hex}}$ et $281EF_{\text{hex}}$.

Exercice 5.

1. Convertir de la base deux à la base dix.

(a) 101010
(b) 100001
(c) 10111
(d) 0110
(e) 11111

2. Convertir de la base dix à la base deux.

(a) 32
(b) 64
(c) 96
(d) 15
(e) 27

Exercice 6.

1. Déterminer l'écriture en base dix des entiers donnés en notation en complément à deux sur 5 bits.

(a) 00011
(b) 01111
(c) 11100
(d) 11010
(e) 00000
(f) 10000

2. Déterminer l'écriture en notation en complément à deux sur 8 bits des entiers donnés en base dix.

(a) 6
(b) -6
(c) -17
(d) 13
(e) -1
(f) 0

3. Supposons que les mots suivant représentent des entiers codés en notation en complément à deux sur 8 bits. Déterminer la représentation de leurs opposés.

(a) 00000001
(b) 01010101
(c) 11111100
(d) 11111110
(e) 00000000
(f) 01111111

4. Supposons qu'un ordinateur stocke les nombres en notation en complément à deux. Quels sont les plus petits et plus grands nombres représentables si le codage

(a) est sur quatre bit ?
(b) est sur six bit ?
(c) est sur huit bit ?

5. Dans chaque question, chaque mot représente un entier en notation en complément à deux sur quatre bits.. Effectuer les additions demandées. Ensuite, vérifier le résultat en convertissant le problème et votre réponse en base dix.

(a) $0101 + 0010$
(b) $0011 + 0001$
(c) $0101 + 1010$
(d) $1110 + 0011$
(e) $1010 + 1110$

CHAPITRE

8

NOMBRES FLOTTANTS

8.1 REPRÉSENTATION DES FLOTTANTS

8.2 Représentation des flottants sur des mots de taille fixe

§1 Encodage

Un nombre flottant est formé de trois éléments : la mantisse, l'exposant et le signe.

Exemple 1

En binary32, on utilise un bit de signe, 8 bits pour l'exposant, et 23 bits pour la partie fractionnaire. Pour l'exposant, on utilise un biais de $+127$.

$$\begin{array}{ccccc} 1 & & 11000110 & & 10010011110000111000000 \\ s & & e & & f \\ \downarrow & & \downarrow & & \downarrow \\ (-1)^s & \times & 2^{e-b} & \times & 1.f \\ (-1)^1 & \times & 2^{198-127} & \times & (1.10010011110000111000000)_{\text{bin}} \end{array}$$

On a

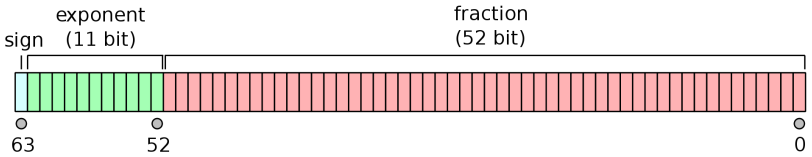
$$\begin{aligned} (1.10010011110000111000000)_{\text{bin}} = & \\ & 1+\frac{1}{2}+\frac{1}{2^4}+\frac{1}{2^7}+\frac{1}{2^8}+\frac{1}{2^9}+\frac{1}{2^{10}}+\frac{1}{2^{15}}+\frac{1}{2^{16}}+\frac{1}{2^{17}} \\ & = (11001001111000011)_{\text{bin}} \times 2^{-17} = 206727 \times 2^{-17} \\ & = (11001001111000011000000)_{\text{bin}} \times 2^{-22} = 13230528 \times 2^{-22}. \end{aligned}$$

Le nombre encodé est donc

$$-2^{54} \times 206727 \approx -3.7 \times 10^{21}.$$

Pour un nombre flottant encodé en `binary64` (ou double précision):

- Le bit de poids fort est le bit de signe,
- Les 11 bits suivants représentent l'exposant biaisé (ou décalé) en base 2. Pour les nombres normalisés, le décalage de l'exposant est $+1023$,
- Les 52 bits suivants représentent la partie fractionnaire en base 2.



Exercice 1. Trouver le nombre à virgule représenté par le mot (encodage binary64)

1. 11000100 01101001 00111100 00111000 00000000 00000000 00000000 00000000
2. 00010000 00111101 00111001 01011000 00000000 00000000 00000000 00000000

Exercice 2. Comment est représenté le nombre à virgule 2^{-1022} (qui est égal à $2.225... \times 10^{-308}$) dans l'encodage binary64.

Exercice 3. Comment est représenté l'entier 7? Et le flotant 7.0 ?

Exercice 4. À combien de décimales environ correspondent 52 chiffres en binaire après la virgule ?

Exercice 5. Quelle précision perd-on si on divise par deux un nombre à virgule avant de le remultiplier par deux ?

§2 Norme IEEE-754-2008

Aucune connaissance liée à la norme IEEE-754 n'est au programme

IEEE 754-2008 Ancien nom	binary32 Single precision	binary64 Double precision
taille w	32	64
précision p	24	53
partie fractionnaire	23	52
exposant	8	11
b, biais	127	1023
e_{max}	$+127$	$+1023$
e_{min}	-126	-1022
plus grand nombre	$2^{128} - 2^{104} \approx 3.403 \times 10^{38}$	$2^{1024} - 2^{971} \approx 1.798 \times 10^{308}$
plus petit nombre normalisé	$2^{-126} \approx 1.175 \times 10^{-38}$	$2^{-1022} \approx 2.225 \times 10^{-308}$
plus petit nombre dénormalisé	$2^{-149} \approx 1.401 \times 10^{-45}$	$2^{-1074} \approx 4.941 \times 10^{-324}$
Chiffres significatifs	environ 7	environ 16

	Signe	Exposant	«Fraction»	Valeur représentée
Normalisés	s	$e_{min} \leq e \leq e_{max}$	$f \geq 0$	$(-1)^s(1.f) \times 2^e$.
Dénormalisés	s	$e = e_{min} - 1$	$f > 0$	$(-1)^s(0.f) \times 2^{e_{min}}$.
Zéro signé	s	$e = e_{min} - 1$	$f = 0$	± 0
Infinis	s	$e = e_{max} + 1$	$f = 0$	$\pm\infty$
Not a number	s	$e = e_{max} + 1$	$f > 0$	NaN

Remarque

- $e = e_{max} + 1$ correspond à un l'exposant biaisé 111 … 111.
- $e = e_{min} - 1$ correspond à un l'exposant biaisé 000 … 000.

8.3 Précision des calculs en flottants

Exemples de choses fausses que vous pensiez vraies

Affirmation	Contre-exemple
$x+1 \neq x$	$x = 2^{53}$
$x+(y+z) = (x+y)+z$	$x = -1, y = 1, z = 2^{-53}$
$x \neq 0 \implies x^2 \neq 0$	$x = 2^{-600}$
$x \neq 0 \implies x/2 \neq 0$	$x = 2^{-1074}$
$x = x$	$x = NaN$
$x \neq \infty$	$x = \pm\infty$
$x < y \implies \exists z, x < z < y$	$x = 0, y = 2^{-1074}$

Phénomène d'élimination (cancellation)

Lors de la soustraction de deux nombres très proches.

```
  1.10010010000111111011011
- 1.10010010000111111000000
----------------------------
  0.00000000000000000011011
```

Après renormalisation la significande est 1.1011000000000000000000000. Si les opérande sont eux-mêmes des résultats de calculs avec des erreurs d'arrondi, les 0 ajouté à droite (en rouge) sont faux.

Phénomène d'absorption

Lors de l'addition de deux nombres ayant des ordres de grandeur très différents, on peut «perdre» toute l'information du plus petit des deux nombres.

```
 1.10010010000111100 011001
+                   1.10010000100111011000000
-------------------------------------------
 1.10010010000111101 111101
```

Comparaison à zéro

Les vecteurs $(3,1)$ et $(0.3, 0.1)$ sont-ils colinéaires? Calculons le déterminant $\begin{vmatrix} 3 & 0.3 \\ 1 & 0.1 \end{vmatrix}$.

Python 3 Shell

```
>>> 3 * 0.1 - 0.3 * 1
5.551115123125783e-17
```

Conclusion

- Les tests d'égalité sur les flottants ce font à ϵ près.
- L'addition et la multiplication sont commutatives (et encore, on vous cache des choses), mais ne sont pas associatives !
- Pour améliorer la précision dans les additions, on utilise le principe de la photo de classes dans les additions (les petits devant).
- Éviter d'additionner ou de soustraire deux nombres d'ordres de grandeur très différents.
- Éviter de soustraire deux nombres presque égaux.