Analyse d'Algorithmes

I Algorithmes

Historiquement, le terme *algorithme* vient du *calcul arithmétique*. C'est en effet une latinisation du nom du scientifique *al-Khwarizmi* auteur d'un livre sur la numérotation indienne. C'est d'ailleurs d'un autre livre de ce même auteur que vient l'origine du mot *algèbre*.

Le terme peut aujourd'hui désigner n'importe quelle *séquence d'instructions non ambigüe*, se généralisant ainsi à l'idée d'avoir une *méthode systématique* pour résoudre un problème, qu'il soit un calcul mathématique ou tâche répétitive. Ainsi une recette de cuisine est un exemple classique d'*algorithme* qui doit être réalisé par un cuisinier (humain ou machine) pour transformer une liste d'ingrédient en un plat.

On choisit dans ce cours de restreindre et formaliser la définition d'un algorithme autour de

- la machine de calcul qui nous intéresse : l'ordinateur *théorique* défini plus tôt.
- le langage dans lequel les instructions sont écrites : Python
- le but de son écriture : la résolution d'un problème algorithmique.

1.1 Problème algorithmique

❦ Définition. *Problème algorithmique*

Un problème algorithmique est un objet abstrait défini par :

- la description d'un ensemble d'entrées possibles. Les entrées sont des *objets* de notre modèle de calcul.
- la description de la sortie attendue pour chaque entrée. Les sorties sont aussi des *objets* de notre modèle de calcul.

Ces descriptions donnent pour chaque entrée possible :

- le *type* de l'objet,
- la contrainte qu'il soit satisfaire pour faire partie de l'ensemble d'entrées

et pour la sortie :

- le *type* de l'objet,
- la contrainte qu'il doit satisfaire en fonction de l'entrée pour répondre au problème.

⚛ Exemples.

Voici quelques problèmes :

Euclide

Entrées : a et b deux entiers positifs tels que $a < b$.
Sortie : un entier égal à PGCD(a, b)

Max

Entrées : `lst` une liste d'entiers positifs
Sortie : un entier égal à `max(lst)`

Recherche linéaire

Entrées : `lst` une liste d'objets.
x un objet.
Sortie : -1 si $x \notin$ `lst`
un entier i tel que `lst`$[i] = x$ sinon

Suffixes

Entrées : `txt` et `s` deux chaînes de caractère.
Sortie : une liste de chaînes de caractère contenant tous les mots dans `txt` qui ont `s` comme suffixe.

1.2 Algorithme

❦ Définition. *Algorithme*

Un algorithme est un objet abstrait défini par :
— sa spécification : un problème algorithmique qu'il a vocation à résoudre
— son corps : une séquence finie d'instructions agissant sur notre ordinateur *théorique*.

❦ Définition. *Implémentation d'un Algorithme*

L'*implémentation* d'un algorithme est l'écriture d'une *fonction Python* réalisant, dans le même ordre, les mêmes instructions que l'algorithme.

Un algorithme est donc l'abstraction qui précède l'écriture ou l'analyse d'une fonction Python dans le but de la résolution d'un problème algorithmique. Cette abstraction est souvent donnée sous la forme :

— d'une description en français des différentes étapes, éludant souvent les instructions précises que l'ordinateur exécutera ;
— d'un pseudo code (instructions écrites dans un mélange de symboles formels et de descriptions en langage naturel)
— par son implémentation…

⚠ Bon à savoir (HP).

La *programmation* consiste à *implémenter* un algorithme, *i.e.* à l'écrire dans un langage compréhensible par la machine. Un même algorithme peut-être implémenté dans des langages différents, sur des machines différentes. Mais aussi selon différents *paradigmes* auxquels sont associées différentes méthodes de programmation.

Les deux paradigmes utilisés dans ce cours sont :

▷ Programmation *Impérative*
  — Programmation Structurée vs Programmation Spaghetti
  — Programmation Procédurale (fonctions sans retour, avec effet de bords)
  — Élément de base : *les boucles*.

— Méthode de prédilection : **itérative**

▷ Programmation *Fonctionnelle*

— Élément de base : *les fonctions*.

— Méthode de prédilection : **récursive**

Il existe aussi d'autres paradigmes :

— Programmation Orientée Objet (POO)

— Programmation Logique

— Programmation Concurrente,

— …

⚛ Exemples.

Méthode de Héron

$$\forall n \in \mathbb{N} \quad x_{n+1}^a = \frac{x_n^a + \frac{a}{x_n^a}}{2}, \; x_0^a = b$$

Algorithme `heron_imperatif`

entrées : a, b et n entiers, $a > 0$, $b > 0$, $n \geq 0$
sorties/effets : un entier égal à x_n^a

```
1. r ← b
2. pour i ← 0 à n faire
3.     r ← (r + a/r) / 2
4. fin pour
5. renvoyer r
```

Algorithme `heron_recursif`

entrées : a, b et n entiers, $a > 0$, $b > 0$, $n \geq 0$
sorties/effets : un entier égal à x_n^a

```
1. si n = 0 alors
2.     renvoyer b
3. fin si
4. r ← heron(a, b, n − 1)
5. renvoyer (r + a/r) / 2
```

1.3 Spécification

La spécification d'un algorithme est le problème algorithme que celui-ci a vocation à résoudre. Cette section fixe un vocabulaire pour désigner différentes parties de la spécification.

Algorithme Max encadré

entrées :
- `li` une liste d'entiers positifs
- a et b sont deux entiers positifs tels que $a < b$.

sorties/effets : un entier $m = max(\{x, x \in \texttt{li} \text{ et } a \leq x < b\})$ avec, par convention, $max(\varnothing) = -1$.

```
1. m ← −1
2. pour x ∈ li faire
3.     si a ≤ x < b et x > m alors
4.         m ← x
5.     fin si
6. fin pour
7. renvoyer m
```

La spécification d'une fonction en Python et la spécification de l'algorithme qu'elle implémente.

Elle est souvent donnée en commentaire dans une chaîne de caractère en première ligne de la fonction. C'est ce que l'on nomme en Python la *docstring*.

Exemples.

```
def max_encadre(li, a, b):
    """ Entrées :
    ◦ li est une liste d'entiers positifs.
    ◦ a et b sont deux entiers positifs tels que a < b.
    Sortie : un entier m = max({x, x ∈ li et a ≤ x < b})
        avec, par convention, max(∅) = −1.
    """
    m = -1
    for x in li:
        if a <= x < b and x > m:
            m = x
    return m
```

Définition. *Signature*

La signature d'un algorithme est la donnée du :

1. nom de l'algorithme
2. du type des entrées possibles de l'algorithme, *i.e.* de sa spécification.
3. du type de la sortie de l'algorithme, *i.e.* de sa spécification.

La signature d'une fonction en Python et la signature de l'algorithme qu'elle implémente.

Elle peut être décrite en langage naturel, incluse dans la spécification ou donnée par des *annotations de type*.

Les annotations de types sont des commentaires particuliers donnés lors de la définition de la fonction permettant de spécifier le type de chaque entrée et le type de la sortie de la fonction.

Exemples.

```
def max_encadre(li: list[int], a: int, b: int) -> int:
    ...
```

La signature de la fonction peut alors être donnée sous la forme :

`max_encadre(li: list[int], a: int, b: int) -> int`.

❦ Définition. *Préconditions*

Les *préconditions* sur les entrées d'un algorithme sont les contraintes imposées sur chaque entrée par la spécification.

⚛ Exemples.

Les préconditions de `Max encadré` sont donc :

— pour tout `x ∈ li`, `x ⩾ 0`
— `0 ⩽ a < b`

❧ ❧ ❧

❦ Définition. *Postcondition*

La *postcondition* sur la sortie d'un algorithme est la contrainte vérifiée par la sortie.

La postcondition est donc la description du résultat attendu pour chaque entrée.

⚛ Exemples.

La postcondition de `Max encadré` est donc $m = max(\{x, x \in \texttt{li} \text{ et } a \leq x < b\})$, avec, par convention, $max(\emptyset) = -1$.

❧ ❧ ❧

Ensembles, la signature, les préconditions et la postcondition constituent la spécification.

1.4 Taille des entrées

Dans le cadre spécifique de l'analyse d'algorithme, il est souvent nécessaire de définir la *taille* des entrées d'un algorithme. Il n'y a pas de définition universelle de taille, et des définitions différentes de la taille d'un même type d'objet peuvent être données dans le cadre d'algorithmes différents.

❦ Définition. *Taille des entrée*

La *taille des entrées* d'un algorithme est définie par une fonction associant à chaque entrée de l'ensemble des entrées possibles de l'algorithme un entier naturel.

⚛ Exemples.

— La taille d'un entier est le plus souvent cet entier ou une expression entière et positive utilisant cet entier
— La taille d'une collection est le plus souvent la longueur de cette séquence. Elle peut aussi être la somme des tailles de ses éléments, ou le maximum des tailles, …
— La taille de plusieurs entrées peut être représentée par la somme des tailles de chaque entrée, par la taille d'une seule de ces entrées ou par une expression utilisant la taille de chaque entrée (par exemple une combinaison linéaire).

II Résolution de problèmes

2.1 Terminaison

❦ Définition. *Terminaison*

Un algorithme *s'arrête* sur une entrée si et seulement si la séquence d'instructions qui sera alors exécutée est *finie*, *i.e.* elle atteint une instruction de *retour* (`return` en Python) ou la fin du corps de l'algorithme (ce qui équivaut à un retour de `None` en Python).

Un algorithme *termine* si et seulement s'il s'arrête sur toutes les entrées possibles décrites par sa spécification.

⚛ Exemples.

L'algorithme $\mathcal{A}_0$ termine, mais pas l'algorithme $\mathcal{A}_1$.

Algorithme $\mathcal{A}_0$

entrées : n un entier positif
sorties/effets : n^2
1. $r \leftarrow 0$
2. **pour** i allant de 0 à n exclus **faire**
3. $\quad r \leftarrow r + n$
4. **fin pour**
5. **renvoyer** r

Algorithme $\mathcal{A}_1$

entrées : a, b et k trois entiers
sorties/effets : $\{a + i\,k, i \in \mathbb{N} \text{ et } a + i\,k < b\}$
1. $v \leftarrow a$
2. E $\leftarrow \varnothing$
3. **tant que** $v < b$ **faire**
4. $\quad$ Ajouter v dans E
5. $\quad v \leftarrow v + k$
6. **fin tant que**
7. **renvoyer** E

En effet $\mathcal{A}_1$ ne s'arrête pas sur les entrées $a = 0$, $b = -10$, $k = 1$.

2.2 Correction

❦ Définition. *Correction*

Un algorithme est *partiellement correct* si et seulement si pour chaque entrée possible décrite par sa spécification, si l'algorithme s'arrête alors la sortie obtenue suite à l'exécution vérifie la postcondition.

Un algorithme est *correct* s'il est partiellement correct et termine.

On résume souvent la correction à cette phrase : *pour chaque entrée vérifiant les préconditions la sortie existe et vérifie la postcondition.*

⚛ Exemples.

En reprenant les algorithmes précédents. L'algorithme $\mathcal{A}_0$ est correct et l'algorithme $\mathcal{A}_1$ est partiellement correct.

🐀🐀🐀

⚛ Exemples.

Algorithme A_2

entrées : n un entier positif.
sorties/effets : $\lfloor \sqrt{n} \rfloor$

```
1. r ← 1
2. tant que r² < n faire
3.     r ← r + 1
4. fin tant que
5. renvoyer r − 1
```

$\mathcal{A}_2$ n'est pas partiellement correct, car il s'arrête et renvoie 1 sur l'entrée 4 alors que la postcondition spécifie 2 comme sortie.

🐀🐀🐀

2.3 Résolution

❦ Définition. *Solution d'un problème algorithmique*

Un algorithme *résout* un problème algorithmique, P, si :

1. P est la spécification de l'algorithme.
2. l'algorithme est correct.

⚠ Bon à savoir (HP).

On dit qu'un problème P est *décidable* s'il existe une solution de P, *i.e.* s'il existe un algorithme qui résout P.

Tous les problèmes algorithmiques ne sont pas décidables. C'est l'une des contributions majeure d'Alan Turing, qui a prouvé que le problème de l'arrêt, consistant à prendre en entrée le code source d'une fonction Python (par exemple) et de renvoyer un booléen vrai si et seulement si l'algorithme implémenté termine, n'est pas décidable.

🐀🐀🐀

III Preuves

3.1 Méthodologie Générale

Pour prouver la terminaison et la correction partielle d'un algorithme, il faut donc prouver des propriétés universelles. On considère une entrée quelconque de l'ensemble possible des entrées et on montre que la séquence des instructions qui sera exécutée :

— atteint la fin du corps l'algorithme (ou un retour placé au milieu du code). On obtient alors la terminaison.
— si elle est finie (l'algorithme s'arrête sur l'entrée), la séquence modifie la mémoire de telle manière à ce que, à la fin, la postcondition soit vérifiée.

La correction se montre donc en prouvant la terminaison, et la correction partielle. Les deux preuves peuvent être faîtes conjointement : pour chaque entrée possible, on montre que l'exécution s'arrête *et* que la postcondition est vérifiée.

Terminaison

Le flux d'exécution allant par définition d'une instruction à la suivante et le corps d'un algorithme étant fini, il atteindra trivialement la fin du corps de l'algorithme s'il ne revient jamais en arrière. Seules les boucles et les appels à d'autres algorithmes permettent d'interrompre la progression de la séquence des instructions vers la fin.

Parmi les boucles, les parcours de séquence (boucles **`for`**) ne peuvent revenir en arrière qu'un nombre *fini* de fois, égal à la longueur de la séquence. Pour les appels à d'autres algorithmes, hormis le cas des appels *récursifs*, l'appel termine si et seulement l'algorithme appelé s'arrête sur l'entrée donnée, c'est le cas *a fortiori* si l'algorithme termine.

Ainsi reste le cas des boucles **tant que** (**`while`**) et des appels récursifs.

❦ Théorème.

> Un algorithme *termine* si et seulement si chaque boucle *tant que* et chaque appel récursif *s'arrêtent dans toutes les configurations initiales* de la mémoire rendues possibles par les préconditions.

Les boucles **tant que** peuvent par ailleurs se trouver dans le corps d'autres structures, telles que des conditionnelles, des parcours de séquence ou d'autres boucles **tant que**. Par exemple si la boucle se trouve au sein d'un parcours de séquence, il faut bien montrer qu'elle s'arrête *à chaque itération du parcours*, *i.e.* quelque soit les configurations de variables produites par chaque itération.

Pour ces cas-là, la terminaison doit être prouvée rigoureusement. La méthode est décrite dans les sections Variants et Algorithmes récursifs.

Correction

⚠ Bon à savoir (HP).

Il existe un cadre formel, la logique de Hoare, pour prouver la correction d'un algorithme. C'est de ce cadre formel, hors programme, qui justifie les méthodes que nous exposons ensuite.

🐾 🐾 🐾

La preuve de correction partielle pose comme première hypothèse les *préconditions* et doit assurer la postcondition à la fin du corps complet de l'algorithme. On peut séparer le corps en ses différents *blocs* d'instructions, organisés suivant les *structures de contrôle*. La preuve exploite alors la propriété suivante pour deux blocs B1 et B2 consécutifs dans le corps de l'algorithme : si

— j'ai un état H de la mémoire avant B1
— si je peux déduire un état E1 de la mémoire après B1 en supposant H
— si je peux déduire un état E2 de la mémoire après B2 en supposant E1

alors : je peux déduire un état E2 de la mémoire après B1 et B2 en supposant H.

⚛ Exemples.

On considère les instructions suivantes. On suppose que x est défini dans la mémoire et qu'il est strictement plus grand que 2 (H).

```
#** B1 **#
y = x - 2
z = x * y
#** B2 **#
u = z % y
if y > 0 :
    u = u + y
else:
    u = -1
```

On peut déduire de B1 d'après H que $y = x-2 > 0$ et $z = x(x-2)$ (E1). On peut déduire de B2 d'après E1 que $u = x-2$. Donc on peut déduire de ce code, d'après H que $y = x-2 > 0, z = x(x-2)$ et $u = x-2$

🐾 🐾 🐾

La preuve se fait donc bloc par bloc de manière que les conclusions d'un bloc puissent être considérée comme les hypothèses du suivant et que se faisant, on avance vers la déduction de la postcondition. On s'intéresse donc au différents blocs.

Affectations et calculs.

Comme l'exemple le montre, on peut déduire des affectations que la mémoire évoluera de telle manière à ce que chaque nom soit associé au dernier résultat de l'expression qui lui a été affecté.

Les appels récursifs mis à part, supposons qu'un calcul fasse appel à un autre algorithme B, partiellement correct, sur une entrée vérifiant les préconditions de B et tel que B s'arrête sur cette entrée. Alors l'appel a pour retour ou effet de bord, le retour ou l'effet de bord décrit par la postcondition de B. Et ceci en vertu de la définition de la correction partielle.

Structure de contrôle conditionnelle.

Face à une structure conditionnelle de la forme :

```
if condition:
    # B1
else:
    # B2
```

en supposant une hypothèse H sur la mémoire, on peut exploiter la propriété suivante : si

— en supposant H et la véracité de la condition, on peut déduire un état E1 de la mémoire après B1

— en supposant H et la fausseté de la condition, on peut déduire un état E2 de la mémoire après B2

alors on peut déduire de H que

$$\texttt{condition} \implies \text{E1 et } \neg\texttt{condition} \implies \text{E2}$$

Boucles et appels récursifs.

Face à une boucle quelconque ou un appel récursif, il n'est pas trivial de savoir que déduire du bloc, ou prouver que cette déduction est correcte. La méthode est décrite dans les sections Invariants et Algorithmes récursifs.

3.2 Invariants

Dans le cadre de la preuve de terminaison comme dans celle de la correction partielle, les boucles restent l'obstacle à l'application d'une méthodologie automatisée. Les invariants de boucles permettent d'expliciter des propriétés sur les variables de la mémoire tout au long de la répétition des calculs. Propriétés qui peuvent être démontrées, puis utiliser dans le cadre d'une preuve de terminaison ou de correction.

Définition. *Invariant de boucle*

Un *invariant* de boucle est une *propriété logique sur les variables de l'algorithme* qui reste vrai après une itération quelconque de la boucle, *i.e.* s'il est vrai avant la vérification de la condition de boucle et si cette condition est aussi vrai, il sera vrai après l'exécution de tout le corps de la boucle.

Une boucle peut admettre plusieurs invariants, et *a fortiori* la conjonction de plusieurs invariant comme invariant. Elle admet par ailleurs toutes les *tautologie* comme invariants… Il s'agira de découvrir les invariants *intéressants* à chaque nouvelle démonstration.

Exemples.

Dans l'algorithme $\mathcal{A}_0$, la boucle :

1. **pour** i allant de 0 à n exclus **faire**
2. $\quad r \leftarrow r + n$
3. **fin pour**

a pour invariants, entre autre :

— $0 \leq i \leq n$; c'est un invariant classique des parcours d'entiers.
— $n >= 0$; c'est la précondition sur n qui ne change jamais car n n'est jamais modifié)
— $r = n*i$; c'est l'invariant au cœur de l'algorithme, puisqu'il nous donne l'évolution de r à chaque itération et ainsi nous permettra de conclure, lorsque i vaut n, que $r = n^2$.
— $r >= 0$; déductible par exemple des invariants précédents.

Dans l'algorithme $\mathcal{A}_1$, la boucle :

1. **tant que** $i < b$ **faire**
2. Ajouter i dans E
3. $i \leftarrow i + k$
4. **fin tant que**

a pour invariants, entre autre :

— $v - a \equiv 0\,[k]$
— $\mathrm{E} = \{a + i\,k, i \in \{0, \cdots, \frac{v-a}{k} - 1\}\}$
— $\forall x \in \mathrm{E}, x < b$

❧ ❧ ❧

Preuve

Pour prouver que I est un invariant d'une boucle de la forme :

```
while condition:
    # Bloc B
```

on montre qu'en prenant comme hypothèse « I *et* `condition` » comme hypothèse, on peut déduire I du bloc B.

Si on considère un parcours sous la forme :

```
for x in S:
    # Bloc B
```

on la traitera comme équivalente à la boucle :

```
ix = 0
while ix < len(S):
    x = S[ix]
    # Bloc B
    ix = ix + 1
```

Ainsi, on pourra considérer qu'il existe `ix` dans l'environnement, qui est nul en entrée du bloc de la boucle, et que l'hypothèse en entrée du bloc B « I, x = `S[ix]` *et* `ix < len(S)` ». On montre alors qu'on peut déduire I du bloc constitué du bloc B *et* de l'instruction `ix = ix + 1`.

Dans le cas particulier d'un parcours des entiers :

```
for i in range(a,b,p)
    # Bloc B
```

on simplifiera l'équivalence à la boucle :

```
i = a
while i < b:
    # Bloc B
    i = i + p
```

Ainsi, on pourra considérer qu'il existe `i` dans l'environnement, qui est égal à a en entrée du bloc de la boucle, et que l'hypothèse en entrée du bloc B « I *et* `i` $< b$ ». On montre alors qu'on peut déduire I du bloc constitué du bloc B *et* de l'instruction `i = i + p`.

Utilisation dans les preuves de correction partielle

❦ Théorème.

> Soit B une boucle dont la condition et C et I un invariant de B. Alors si on peut déduire du bloc B, en prenant I comme hypothèse, la propriété I *et* $\neg$C à la sortie de la boucle.

La preuve de correction partielle s'articule donc autour de la découverte d'invariant *intéressants* qui une fois démontrés peuvent permettre par l'application de la méthodologie générale, bloc d'instruction par bloc d'instruction, de déduire des préconditions, la postcondition.

⚛ Exemples.

Algorithme $\mathcal{E}$

entrées : a et b deux entiers positifs non nuls.
sorties/effets : $p = \text{PGCD}(a, b)$

```
1.  p ← a
2.  q ← b
3.  tant que p ≠ q faire
4.      si p > q alors
5.          p ← p − q
6.      sinon
7.          q ← q − p
8.      fin si
9.  fin tant que
10. renvoyer p
```

L'algorithme d'Euclide si dessus est partiellement correct. On suppose que l'algorithme s'arrête sur deux entiers positifs non nuls a et b.

1. On déduit du premier block (l.1 et l.2) que $p = a$ et $q = b$.
2. On démontre l'invariant de boucle I : $\text{PGCD}(a, b) = \text{PGCD}(p, q)$. Supposons I et $p \neq q$. Si $p > q$ alors la nouvelle valeur associée à p sera $p - q$ et par propriété du PGCD, $\text{PGCD}(p - q, q) = \text{PGCD}(p, q) = \text{PGCD}(a, b)$. Sinon, comme $p \neq q$, $p < q$ et symétriquement $\text{PGCD}(p, q - p) = \text{PGCD}(p, q) = \text{PGCD}(a, b)$. I est préservé.
3. Avant la boucle et d'après les déductions de (1), I est vrai. Comme I est un invariant, on peut déduire qu'à la sortie de la boucle, avant exécution de la ligne 10, $\text{PGCD}(a, b) = \text{PGCD}(p, q)$

et $p = q$ (négation de la condition de boucle). Donc que $p = \text{PGCD}(a, b)$. La postcondition est vérifiée.

🐭🐭🐭

⚛ Exemples.

Pour une liste d'entiers de longueur n, on note dans la suite $\Sigma(l) = \sum_{i=0}^{n-1} l[i]$.

Algorithme $\mathcal{E}$

entrées : l une liste d'entiers
sorties/effets : $m = \Sigma(l)$
1. $s \leftarrow 0$
2. **pour** $x \in l$ **faire**
3. $\quad s \leftarrow s + x$
4. **fin pour**
5. **renvoyer** s

L'algorithme de somme d'une liste si dessus est partiellement correct. On suppose que l'algorithme s'arrête sur une liste l de n entiers positifs.

1. On déduit du premier block (l.1) que $s = 0$
2. On démontre l'invariant de boucle I : $s = \Sigma(l[0 : i_x])$ avec i_x l'indice de x dans l et $l[0 : i_x]$ la liste des i_x premiers éléments de l. Supposons I et $i_x < n$. Après l'exécution du corps de la boucle la nouvelle valeur de i_x sera $i'_x = i_x + 1$ (*c.f.* boucle équivalente plus haut). Aussi la nouvelle valeurs de s sera $s' = s + x = s + l[i_x]$. D'après I on aura donc $s' = \Sigma(l[0 : i_x]) + l[i_x] = \Sigma(l[0 : i_x + 1]) = \Sigma(l[0 : i'_x])$. I reste vérifier après l'exécution du corps de la boucle.
3. Avant la boucle et d'après les déductions de (1), $s = 0 = \Sigma(l[0 : 0])$ car $i_x = 0$ au départ de la boucle. Donc I est vrai. Comme I est un invariant, on peut déduire qu'à la sortie de la boucle, avant exécution de la ligne 5, $s = \Sigma(l[0 : i_x])$ et $i_x >= n$ (négation de la condition de boucle). Donc que $s = \Sigma(l)$. La postcondition est vérifiée.

🐭🐭🐭

3.3 Variants

Comme déjà évoqué, la preuve de terminaison passe par la preuve de l'arrêt de chaque boucle **tant que** (`while`). Pour ce faire, on doit pouvoir déduire d'une hypothèse H sur les variables de l'algorithme que la condition de boucle sera fausse après un nombre fini d'exécutions du corps de boucle.

❦ Définition. *Variant de boucle*

Un *variant de boucle* est une *expression entière des variables de l'algorithme* qui reste *positive* et *décroît* après une itération quelconque de la boucle, *i.e.* si elle est égale à $n \in \mathbb{N}$ avant la vérification de la condition de boucle et si cette condition est aussi vrai, elle sera égale, après l'exécution complète du corps de la boucle, à $n' \in \mathrm{N}$ tel que $n' < n$.

❦ Théorème.

Soit une boucle B et V un variant de B. Si on a démontré que V s'évalue en un entier naturel un début de B (avant la première vérification de condition de boucle), alors B s'arrête.

En effet, supposons que B ne s'arrête pas, on peut alors définir la suite $v = (v_i)_{i\in\mathbb{N}}$ des valeurs du variant avant chaque itération de la boucle. v serait alors une suite d'entiers naturels strictement décroissante.

Preuve

La découverte d'un variant permet donc de montrer l'arrêt d'une boucle et donc celle d'un algorithme. La méthodologie consiste à démontrer d'abord que V est bien un variant, puis de vérifier que V est bien entier au début de la boucle.

Une fois V exhibé, pour montrer que V est un variant d'une boucle de la forme :

```
while condition:
    # Bloc B
```

on montre, en prenant comme hypothèse que V s'évalue en $n \in \mathbb{N}$ et que `condition` est vrai, que V s'évalue après l'exécution du bloc B en $n' \in \mathbb{N}$ avec $n' < n$. On peut, pour se faire, s'aider d'un ou plusieurs invariants de boucle déjà démontrés. On n'oubliera pas de montrer, comme c'est le cas pour les boucles imbriquées, que l'exécution du bloc B s'arrête.

Puis, on conclut en montrant qu'avant la boucle, V s'évalue bien en un entier positif et que tous les invariants utilisés sont bien vérifiés.

En pratique, on fera attention à ne pas changer le type associer à une variable, ce qui permettra d'admettre les invariants qui consistent à affirmer qu'une variable reste du même type. Ce sera en particulier important en Python pour les variables entières qu'on évitera d'associer par erreur à des flottants en utilisant des opérations flottantes (la division `/`).

⚛ Exemples.

On prend l'algorithme de fusion de deux listes triées. Dans la suite on note $|l|$ la longueur d'une liste l.

Algorithme $\mathcal{F}$

entrées : l_1 et l_2 deux listes d'entiers triés dans l'ordre croissant
sorties/effets : l une version triée de $l_1 + l_2$

```
1.  i_1 ← 0, i_2 ← 0
2.  l ← [] /* Boucle B */
3.  tant que i_1 < |l_1| et i_2 < |l_2| faire
4.    si l_1[i_1] < l_2[i_2] (*) alors
5.      Ajouter l_1[i_1] à la fin de l
6.      i_1 ← i_1 + 1
7.    sinon
8.      Ajouter l_2[i_2] à la fin de l
9.      i_2 ← i_2 + 2
10.   fin si
11. fin tant que
12. si i_1 = |l_1| alors
13.   renvoyer l + l_2
14. sinon
15.   renvoyer l + l_1
16. fin si
```

#+end_export

On montre que V défini par $|l_1| + |l_2| - (i_1 + i_2)$ est un variant de la boucle B. Supposons pour se faire que V s'évalue en un entier $m \in \mathbb{N}$ à l'entrée d'une itération de la boucle B. Supposons de plus que $i_1 < |l_1|$ et $i_2 < |l_2|$. On peut déjà en déduire que $m > 0$. Si $(*)$ est vrai, alors la nouvelle valeur de i_1, noté i_1', sera augmenté de 1. Sinon ce sera la nouvelle valeur de i_2, noté i_2', qui le sera. Dans tous les cas, $(i_1' + i_2')$ sera augmenté de 1. La nouvelle valeur de V sera donc égale à $m - 1$. Or $m - 1 \geq 0$ et $m - 1 < m$. V est bien un variant de la boucle B.

Par ailleurs, avant la première itération de boucle i_1 et i_2 sont nuls et V s'évalue en la somme des tailles des deux listes, donc en un entier naturel par définition. On en déduit que la boucle B s'arrête. C'est la seule boucle de l'algorithme, il termine donc.

🐾 🐾 🐾

⚛ Exemples.

On reprend l'algorithme d'Euclide.

Algorithme $\mathcal{E}$

entrées : a et b deux entiers positifs non nuls.
sorties/effets : $p = \text{PGCD}(a, b)$

1. $p \leftarrow a$
2. $q \leftarrow b$ /* *Boucle* B */
3. **tant que** $p \neq q$ **faire**
4. **si** $p > q$ **alors**
5. $p \leftarrow p - q$
6. **sinon**
7. $q \leftarrow q - p$
8. **fin si**
9. **fin tant que**
10. **renvoyer** p

#+end_export

On montre que V défini par $p+q$ est un variant de la boucle B. Supposons pour se faire que V s'évalue en un entier $n \in \mathbb{N}$ à l'entrée d'une itération de la boucle B. Supposons de plus que $p \neq q$. Si $p > q$, la nouvelle valeur de p après exécution du corps de boucle, noté p', sera $p - q$. Le variant sera alors évalué après exécution du corps de boucle à q. Il nous faut montrer que $q \geq 0$ et $q < p + q$. Pour cela il faut montrer que $q \in \mathbb{N}^*(\mathrm{I}_1)$ et $p \in \mathbb{N}^*(\mathrm{I}_2)$ sont des invariants de la boucle B. Supposons cette preuve acquise, on peut alors conclure que V est un variant de B.

Par ailleurs, on peut déduire des préconditions et des deux premières lignes de l'algorithme que $q \in \mathbb{N}^*$ et $p \in \mathbb{N}^*$ avant la première itération de boucle, donc que $\mathrm{V} \in \mathbb{N}$ et que (I_1) et (I_2) sont vérifiés. Donc que la boucle B s'arrête. C'est la seule boucle de l'algorithme, il termine donc. On avait montré plus haut qu'il était partiellement correct, il est donc correct et résout le problème du calcul du PGCD.

🦋 🦋 🦋

⚛ Exemples.

On considère l'algorithme implémenté par la fonction Python suivante :

```
def f(l) :
    """
␣␣␣␣Entrée :␣une␣liste␣d'entiers␣positifs ,␣l ,␣de␣longueur␣n
␣␣␣␣Sortie :␣une␣chaîne␣de␣caractère␣de␣la␣forme␣"|m_0|m_1|...|m_{n-1}|"
␣␣␣␣␣␣␣␣avec␣chaque␣m_i une alternance de '*' et '-' de longueur l[i].
    """
    s = '|'
    y = 0
    for x in l:
        y = x
        while y != 0: # Boucle B
            y = y - 1
            if y % 2 == 0:
                s += '*'
            else:
```

```
            s += '-'
        s += '|'
    return s
```

Pour montrer sa terminaison, il faut montrer l'arrêt de la boucle `B` dans toute configuration possible. On montre d'abord que V défini simplement par l'entier associé à y est un variant de la boucle `B`. Pour ce faire, on suppose que y est un entier et que $y \neq 0$. Après l'exécution du corps de boucle la nouvelle valeur de y, noté y', est égale à $y - 1$ qui est bien positif et strictement plus petit que y. Donc V reste positif et décroît strictement. C'est un variant de la boucle `B`. De plus on admet l'invariant « $x \geq 0$ »pour le parcours de l qu'on peut prouver d'après les préconditions et en remarquant que l n'est jamais modifiée. Ainsi à chaque itération du parcours, `B` s'arrête, et l'algorithme termine.

3.4 Algorithmes récursifs

Terminaison

Correction

Théorème.

On considère un algorithme récursif muni d'une taille sur ses entrées. On suppose que les appels récursifs se font *toujours sur des entrées de tailles strictement plus petites*. Alors

1. si l'algorithme termine sur les entrées de taille nulles et si, en supposant que *tous* les appels récursifs terminent, on montre qu'un algorithme termine sur des entrées quelconques de taille non nulle, alors l'algorithme termine.
2. si l'algorithme est partiellement correct sur les entrées de taille nulles et si, en supposant que *tous* les appels récursifs sont partiellement corrects, on montre qu'un algorithme est partiellement correct sur des entrées quelconques de taille non nulle, alors l'algorithme est partiellement correct.

La notion de taille des entrées, et la diminution de la taille sur chaque appel, permet une preuve par récurrence qui est résumée dans le théorème précédent.

Pour ce qui est de la méthodologie, on montrera le plus souvent la terminaison et la correction dans le même temps, en commençant par définir, si ce n'est pas donné, la taille des entrées de l'algorithme. On traitera le cas des entrées de taille nulles, qui ne font donc normalement pas d'appels récursifs, puis on montrera simultanément pour les entrées de taille non nulle :

— que tous les appels récursifs se font toujours sur des entrées de taille plus petite ;
— que si les appels récursifs terminent, l'appel termine ;
— que si les appels récursifs sont partiellement corrects, l'appel est correct.

Exemples.

On prend l'algorithme de tri fusion d'une liste d'entiers. Dans la suite, on note $|l|$ la longueur d'une liste l, qui fera office de taille de l'entrée.

Algorithme $\mathcal{T}$

entrées : l liste d'entiers.
sorties/effets : l' une version triée de l
1. **si** |l| = 0 **alors**
2. **renvoyer** []
3. **fin si**
4. $m \leftarrow \lfloor \frac{|l|}{2} \rfloor$
5. $l_1 \leftarrow \mathcal{T}(l[:m])$
6. $l_2 \leftarrow \mathcal{T}(l[m:])$
7. **renvoyer** $\mathcal{F}(l_1, l_2)$

Dans le cas d'une liste vide, la fonction est trivialement correcte. Sinon, on remarque que $m < |l|$ et que chaque appel à $\mathcal{T}$ se fait donc sur une liste de taille strictement inférieure à la taille de l. Si on suppose que $\mathcal{T}$ est correcte sur $l[:m]$ et $l[m:]$, alors trivialement $\mathcal{T}$ termine sur l (il n'y a pas de boucle), et par correction de l'algorithme $\mathcal{F}$, l_1 et l_2 étant des copies triées de $l[:m]$ et $l[m:]$ par hypothèse, $\mathcal{T}$ renvoie une copie triée de $l[:m] + l[m:]$, *i.e.* une copie triée de l.

$\mathcal{T}$ est correcte. #+end_export

IV Complexité

Dans toute cette section, les algorithmes considérés seront toujours supposés corrects.

L'analyse de la *complexité* d'un algorithme est une évaluation formelle de sa consommation d'une *ressource*. Selon la ressource concernée, différentes complexités peuvent être définies.

En informatique les deux ressources principales sont :

— le temps, lié au nombre de calculs que doit réaliser le microprocesseur, on parle alors de *complexité temporelle*.
— l'espace, lié au nombre de case occupées dans la mémoire vive, on parle alors de *complexité spatiale*.

4.1 Opération élémentaire et espace élémentaire

Dans le modèle de l'ordinateur utilisé dans ce cours, nous devons déterminer ce qui correspond à *une* opération et à *un* espace mémoire pour pouvoir formellement évaluer leur nombre.

Définition. *Opération élémentaire*

Les opérations élémentaires de notre modèle sont :
— le passage d'une instruction à une autre de l'algorithme ;
— les affectations et lectures de variables ;
— les créations d'objets simples : entiers, flottants, booléens, `None` ;
— les opérations numériques sur les entiers et flottants, *à l'exception* de la puissance `**` ;

— les comparaisons numériques sur les entiers et flottants ;
— les opérations booléennes ;
— la création d'une séquence vide ;
— l'appel de la fonction **`len`** sur un conteneur ;
— l'ajout et la suppression à la fin d'une liste ;
— l'ajout, la mise à jour et l'appartenance dans un dictionnaire.

❦ Définition. *Espace élémentaire*

Les objets suivants occupent un espace élémentaire :
— les variables ;
— les objets simples : entiers, flottants, booléens, `None` ;
— les conteneurs vides ;

Les algorithmes de cette section étant supposés corrects et donc, en particulier, terminant sur chaque entrée possible, on remarquera alors que pour toute entrée possible de l'algorithme, il réalise un nombre fini d'opérations élémentaires et occupe un nombre fini d'espaces élémentaires.

4.2 Analyse temporelle de la complexité

❦ Définition. *Complexité temporelle*

La *complexité temporelle* d'un algorithme est la fonction qui associe à chaque entrée possible d'un algorithme le nombre d'opérations élémentaires qu'il réalise lors de son exécution sur cette entrée.

Une fois définie une taille sur les entrées de l'algorithme, on s'intéresse à des définitions dérivées de complexités temporelles en fonction de la taille d'une ou plusieurs entrées de l'algorithme.

❦ Définition. *Complexité temporelle au pire cas*

La *complexité temporelle* d'un algorithme *au pire cas* en fonction de la taille de ses entrées est la fonction qui associe à chaque taille possible, `n`, le plus grand nombre d'opérations élémentaires réalisés sur l'ensemble des exécutions sur les entrées de taille `n`.

On peut faire dépendre cette fonction de la taille de plusieurs entrées, par exemple la complexité temporelle d'un algorithme au pire cas en fonction de la taille de deux entrées `n` et `m`, est une fonction à deux variables qui donne le plus grand nombre d'opérations élémentaires réalisés sur l'ensemble des exécutions sur les entrées dont la première est de taille `n` et la seconde de taille `m`.

⚠ Bon à savoir (HP).

Il y a d'autres complexités dérivées possibles :

— la complexité temporelle *au meilleur cas* prendra le plus petit nombre d'opérations élémentaires réalisées par les exécutions sur les entrées de taille n.
— la complexité temporelle *en moyenne* prendra la moyenne des opérations élémentaires réalisées par les exécutions sur les entrées de taille n. On travaille alors en termes de probabilité.

🐭🐭🐭

L'analyse de complexité est dans les faits une analyse *comparative* visant à la fois à sélectionner les meilleurs algorithmes et à estimer l'ordre de grandeur du temps d'exécution de leur implémentation en le comparant à celui d'algorithmes similaires. Cette comparaison se fait le plus souvent *asymptotiquement*, en faisant tendre la taille des entrées vers l'infini.

🐭 Définition. *Complexité temporelle asymptotique au pire cas*

La *complexité temporelle asymptotique* d'un algorithme *au pire cas* en fonction de la taille de ses entrées est l'ensemble, exprimé en notation de Landau, auquel appartient la complexité temporelle au pire cas de l'algorithme.

On donnera une expression en *grand O* de la complexité et on dira pour simplifier que l'algorithme a une *complexité temporelle en* $O(g)$.

⚠ Bon à savoir (HP).

Il y a d'autres expressions utiles parmi les notations de Landau. En effet, le *grand O* permet d'avoir une *majoration* (domination) de la complexité temporelle au pire cas. Mais cette majoration peut être excessive. Le *grand Omega* (Ω) est défini à l'inverse comme une *minoration* de la complexité temporelle. Le *grand Theta* (Θ) est obtenu lorsque la fonction de complexité temporelle au pire cas est majorée (O) et minoré (Ω) par une même fonction g et donne la meilleure évaluation de la complexité.

🐭🐭🐭

4.3 Calculs de la complexité temporelle asymptotique au pire cas

La complexité temporelle au pire cas peut-être calculée de manière méthodique par analyse de la séquence d'instructions de l'algorithme. On considérera que chaque partie *contribue* à la complexité en *apportant un* O. Et on développe les règles de calculs à appliquer.

Appels de fonction

Le nombre d'opérations élémentaires au pire cas d'un appel à un autre algorithme est par définition donné par la valeur de la complexité temporelle au pire cas de cet algorithme. Dans l'analyse asymptotique, il contribue alors du O auquel appartient cette complexité.

Instructions

Le nombre d'opérations élémentaires au pire cas réalisées par une instruction est la somme des opérations élémentaires qui constituent cette instruction. Dans l'analyse asymptotique, on sommera les

O apportés par chaque opération, ce qui équivaut à garder le plus grand d'entre eux.

⚛ Exemples.

Dans l'instruction `x = len(l) * 50 + h.pop()` on décompte 2 lectures, une affectation, un calcul de longueur, une suppression à la fin d'une liste, 2 opérations numériques, une création d'objet, ce qui fait 8 opérations élémentaire. La longueur de `l` et `h` n'ont aucune influence sur ce nombre et le nombre d'opérations élémentaires de cette instruction est donc asymptotiquement un O(1).

🐁🐁🐁

Blocs séquentiels

Le nombre d'opérations élémentaires au pire cas réalisées par deux blocs séquentiels d'instructions est la somme du nombre d'opérations réalisés par chaque bloc. Dans l'analyse asymptotique, on sommera les O apportés par chaque bloc, ce qui équivaut à garder le plus grand d'entre eux.

⚛ Exemples.

Supposons un algorithme de la forme

```
def f(n):
    # B1
    ...
    # B2
    ...
```

Si le bloc `B1` réalise $O(n^2)$ opérations élémentaires et le bloc `B2` en réalise $O(2^n)$, la complexité temporelle asymptotique au pire cas de `f` est en $O(2^n)$.

🐁🐁🐁

Conditionnelles

On suppose des instructions de la forme

```
if C1:
    # B1
    ...
elif C2:
    #B2
    ...
...
else:
    #Bn
    ...
```

Le nombre d'opérations élémentaires au pire cas réalisées par cette conditionnelle est le nombre d'opérations obtenu par l'exécution du bloc avec le pire nombre d'opérations. Chaque bloc `BX` et

chaque condition `CX` produit un nombre d'opérations β_X et γ_X au pire cas, dépendant de la taille des entrées. Le bloc X ne sera exécuté qu'après vérification de *toutes* les conditions `C1, C2, ..., CX` (avec `Cn` qui ne contera que comme une opération élémentaire). Le nombre d'opérations au pire cas est donc le maximum du nombre d'opérations parmi les $\beta_k + \sum_{i=1}^{k} \gamma_i$. Dans l'analyse asymptotique, on sommera les O apportés par les conditions au O apporté par le bloc, puis on prendra le plus grand O, ce qui revient à prendre le plus grand O parmi ceux des conditions et des blocs.

⚛ Exemples.

Supposons un algorithme de la forme

```
def f(n):
    if C1 :
        # B1
        ...
    else:
        # B2
        ...
```

Si la condition `C1` réalise O(1) opérations élémentaires, le bloc `B1` réalise $O(n^2)$ opérations élémentaires et le bloc `B2` en réalise $O(n)$, la complexité temporelle asymptotique au pire cas de `f` est en $O(n^2)$, le pire cas étant le cas de l'exécution du bloc `B1`.

🐾🐾🐾

⚛ Exemples.

Supposons un algorithme de la forme

```
def f(n):
    if C1 :
        # B1
        ...
    else:
        # B2
        ...
```

Si la condition `C1` réalise $O(n)$ opérations élémentaires, le bloc `B1` réalise O(1) opérations élémentaires et le bloc `B2` en réalise O(1), la complexité temporelle asymptotique au pire cas de `f` est en $O(n)$, identique dans le cas de l'exécution de `B1` ou `B2`.

🐾🐾🐾

⚛ Exemples.

Supposons un algorithme de la forme

```
def f(n):
    if C1 :
        # B1
        ...
```

```
    elif C2:
        # B2
        ...
    else:
        # B3
        ...
```

Si la condition `C1` réalise O(1), la condition `C2` réalise O(n) opérations élémentaires, le bloc `B1` réalise O(1) opérations élémentaires, le bloc `B2` en réalise O(1) et le bloc `B3` en réalise O(1), la complexité temporelle asymptotique au pire cas de `f` est en O(n), identique dans le cas de l'exécution de `B2` ou `B3`.

Boucles

On suppose des instructions de la forme

```
while C:
    # B
    ...
```

Le nombre d'opérations élémentaires au pire cas réalisées par cette boucle est la somme du nombre d'opérations réalisée par le calcul de la condition `C` avec le nombre obtenu par les exécutions répétées du bloc `B`. On note β_i le nombre d'opérations élémentaires réalisées par l'exécution du bloc `B` à l'itération i de la boucle. Ce nombre dépend de la taille des entrées. Si la boucle réalise k itération, le nombre d'opérations élémentaires au pire cas est alors $\sum_{i=0}^{k} \beta_i$. Si les β_i sont tous majorés par une même fonction g en la taille des entrées, on peut en conclure que le nombre d'opérations élémentaires au pire cas réalisées par cette boucle est majoré par kg. Dans l'analyse asymptotique, il arrive souvent *abusivement* de sommer les O apportés par chaque itération. Il est important que la constante cachée dans ce O soit indépendante de l'itération. Il faut donc faire dépendre le O de toutes les quantités qui varient à chaque itération. Le calcul de la somme doit aboutir à un O qui est indépendant de ces quantités.

Une boucle de la forme

```
for x in S:
    # B
    ...
```

a une complexité équivalente à une boucle de la forme :

```
i = 0
while i < len(S):
    x = S[i]
    # B
    ...
    i += 1
```

et donc produit un nombre d'opérations élémentaires en $\sum_{i=0}^{\text{len(S)}} \beta_i$.

Exemples.

Supposons un algorithme de la forme

```
def f(l):
    for x in l :
        # B
        ...
```

On note `n = len(l)`. Si le bloc `B` réalise moins de Kn opérations élémentaires à chaque itération, K étant indépendant de l'itération (le bloc `B` apporte O(n)), la complexité finale est en O(n^2)

Exemples.

Supposons un algorithme de la forme

```
def f(l):
     i = 0
     while i*i < len(l)
        # B
        ...
        i += 1
```

On note `n = len(l)`. Si le bloc `B` réalise moins de Ki^2 opérations élémentaires à chaque itération, K étant indépendant de l'itération (le bloc `B` apporte O(i)), la complexité finale est en $O(\sum_{i=0}^{\lfloor\sqrt{n}\rfloor} i^2) = O(\frac{\sqrt{n}(\sqrt{n}+1)(2\sqrt{n}+1)}{6}) = O(n\sqrt{n})$.

4.4 Appels récursifs

D'après les règles précédentes, les appels récursifs dans un algorithme apportent le nombre d'opérations élémentaires au pire cas de cet appel. L'expression de la complexité temporelle de l'algorithme dépend alors d'elle-même. On obtient une expression par récurrence de la complexité. On peut *réinjecter* l'égalité dans l'expression pour conjecturer une majoration de la complexité, puis la prouver par récurrence.

Exemples.

Supposons un algorithme de la forme

```
def f(n):
    # B
    ...
    return f(n-1)
```

On note $\gamma(n)$ la complexité temporelle au pire cas de f.

1. On suppose que `B` apporte O(1) opérations élémentaires, alors $\gamma(n) = \gamma(n-1) + O(1)$ et on peut montrer que $\gamma(n) = O(n)$.

2. On suppose que `B` apporte $O(n)$ opérations élémentaires, alors $\gamma(n) = \gamma(n-1) + O(n)$ et on peut montrer que $\gamma(n) = O(n^2)$.

🐀🐀🐀

⚛ Exemples.

Supposons un algorithme de la forme

```
def f(n):
    # B
    ...
    return f(n//2)
```

On note $\gamma(n)$ la complexité temporelle au pire cas de f.

1. On suppose que `B` apporte $O(1)$ opérations élémentaires, alors $\gamma(n) = \gamma(\lfloor \frac{n}{2} \rfloor) + O(1)$ et on peut montrer que $\gamma(n) = O(\log(n))$.
2. On suppose que `B` apporte $O(n)$ opérations élémentaires, alors $\gamma(n) = \gamma(\lfloor \frac{n}{2} \rfloor) + O(n)$ et on peut montrer que $\gamma(n) = O(n)$.

🐀🐀🐀

⚛ Exemples.

Supposons un algorithme de la forme

```
def f(l):
    n = len(l)
    # B
    ...
    return f(l[:n//2]) + f(l[n//2:])
```

On note $\gamma(n)$ la complexité temporelle au pire cas de f.

1. On suppose que `B` apporte $O(1)$ opérations élémentaires, alors $\gamma(n) = \gamma(\lfloor \frac{n}{2} \rfloor) + \gamma(\lceil \frac{n}{2} \rceil) + O(1)$ et on peut montrer que $\gamma(n) = O(n)$.
2. On suppose que `B` apporte $O(n)$ opérations élémentaires, alors $\gamma(n) = \gamma(\lfloor \frac{n}{2} \rfloor) + \gamma(\lceil \frac{n}{2} \rceil) + O(n)$ et on peut montrer que $\gamma(n) = O(n \log(n))$

🐀🐀🐀

4.5 Catégories et ordre de grandeurs des complexités temporelles

Les complexités les plus fréquentes sont représentées dans le tableau suivant.

	10	100	1 000	10 000	1 000 000	10^9
$\Theta(\log n)$	1ns	10ns	10ns	10ns	10ns	100ns
$\Theta(\sqrt{n})$	1ns	10ns	100ns	100ns	1mics	1ms
$\Theta(n)$	10ns	100ns	1mics	10mics	1ms	1s
$\Theta(n \log n)$	10ns	100ns	10mics	100mics	10ms	10s
$\Theta(n^2)$	100ns	10mics	1ms	100ms	10 min	10 ans
$\Theta(n^3)$	1mics	1ms	1s	10 min	10 ans	∞
$\Theta(2^n)$	1mics	∞	∞	∞	∞	∞

On distingues les complexités :

— logarithmiques (O(ln(n)))
— polynomiales (O(n^α)), avec en particulier la complexité linéaire ($\alpha = 1$) et quadratique ($alpha = 2$).
— exponentielles (O(q^n) avec $q > 1$)

4.6 Analyse spatiale de la complexité

Les méthodes d'analyse spatiale de la complexité ne sont pas au programme. On donne les définitions et quelques exemples de calcul du nombre d'espaces élémentaires occupés par un algorithme.

❦ Définition. *Espace occupé par les conteneurs*

Les conteneurs en Python (liste, tuples, chaînes, dictionnaires) occupent un nombre de cases élémentaires dépendant de leur contenu. Dans notre modèle, on supposera que les conteneurs de n objets, occupent la somme des espaces de chaque objet qu'elle contient et un espace élémentaire supplémentaire. C'est cet espace qui implique qu'un conteneur vide occupe tout de même un espace.

On notera qu'un caractère dans une chaîne est considéré comme occupant un espace.

Un dictionnaire contient des associations, il faut donc considérer dans son contenu, à la fois l'espace occupé par la clé, et celui occupé par la valeur.

❦ Définition. *Complexité spatiale*

La *complexité spatiale* d'un algorithme est la fonction qui associe à chaque entrée possible d'un algorithme le nombre d'espaces élémentaires occupés tout au long de son exécution sur cette entrée.

Tout comme pour la complexité temporelle, une fois définie une taille sur les entrées de l'algorithme, on s'intéresse à des définitions dérivées de complexités spatiales en fonction de la taille d'une ou plusieurs entrées de l'algorithme.

❦ Définition. *Complexité spatiale au pire cas*

La *complexité spatiale* d'un algorithme *au pire cas* en fonction de la taille de ses entrées est la fonction qui associe à chaque taille possible, `n`, le plus grand nombre d'espaces élémentaires occupés sur l'ensemble des exécutions sur les entrées de taille `n`.

On peut, comme pour la complexité temporelle, faire dépendre cette fonction de la taille de plusieurs entrées.

❦ Définition. *Complexité spatiale asymptotique au pire cas*

La *complexité spatiale asymptotique* d'un algorithme *au pire cas* en fonction de la taille de ses entrées est l'ensemble, exprimé en notation de Landau, auquel appartient la complexité spatiale au pire cas de l'algorithme.

On donnera une expression en *grand O* de la complexité et on dira pour simplifier que l'algorithme a une *complexité temporelle en* $O(g)$.

⚛ Exemples.

Les algorithmes classiques ne créant pas d'objets conteneurs en python (recherche linéaire, calcul de maximum, de somme, fenêtre glissante, recherche dichotomique, …) ont une complexité spatiale constante, qui ne dépend pas de la taille de leurs entrées. La complexité spatiale asymptotique au pire cas est donc en $O(1)$.

❧ ❧ ❧

⚛ Exemples.

L'algorithme de `fusion` décrit plus haut rempli une liste. En plus des objets utilisés occupant un seul espace élémentaire, en nombre constant, chaque itération de la boucle ajoute un élément dans la liste et donc occupe un espace élémentaire de plus. La taille de la liste est au plus égale à la somme de la taille des deux listes en entrées. On en conclu que l'algorithme a une complexité spatiale asymptotique au pire cas en $O(|l_1| + |l_2|)$

❧ ❧ ❧