ITC – cours n°08

Analyse des algorithmes

Ce chapitre présente les trois grands principes qui nous serviront de guide pour analyser les algorithmes et les programmes :
- la terminaison : l'algorithme termine-t-il au bout d'un nombre fini d'étapes quelle que soit l'entrée ?
- la correction : le résultat rendu est-il celui qui était attendu ?
- la complexité : combien de temps prend le programme selon la taille de l'entrée ? Combien d'espace mémoire occupe-t-il ?

Savoir répondre à ces questions, c'est pouvoir prédire, avant d'avoir écrit la moindre ligne de code, ce qui va se passer. Bien entendu, nous nous limiterons à des cas accessibles ; pour la complexité, nous chercherons en général un majorant.

I. État d'un programme

On devra au cours de ce chapitre manipuler des représentations mathématiques de l'état d'un programme : celui-ci correspond, en première approximation[1] à l'ensemble des variables déclarées, leur type et leur valeur à un instant donné. Par exemple, considérons le programme suivant :

```
a, b = 3, 4
c = a - b
a = 2 + c
```

on peut attribuer à chaque variable, si elle est définie, sa valeur en définissant une suite associée ; ainsi a_i correspond à la valeur de a après exécution des instructions de la ligne i. Dans notre cas, on pourra proposer les deux représentations ci-dessous :

E_0	E_1	E_2	E_3
	$a_1 = 3$	$a_2 = 3$	$a_3 = 1$
	$b_1 = 4$	$b_2 = 4$	$b_3 = 4$
		$c_2 = -1$	$c_3 = -1$

var	E_0	E_1	E_2	E_3
a		3	3	1
b		4	4	4
c			-1	-1

On s'intéressera à des propriétés logiques des variables, par exemple $a_4 \leqslant 0$ ou $\forall n \in [\![2;4]\!],\ b_n = 4$.

[1] Qui sera suffisante pour nos besoins

II. Terminaison

2.1. Position du problème

Définition

On dit qu'un algorithme termine s'il effectue un nombre fini d'étapes sur toute entrée. Le nombre d'étapes peut cependant devenir arbitrairement grand selon les entrées.

Dans la pratique, un algorithme s'arrêtera nécessairement s'il n'utilise que des instructions simples (affectations, comparaisons...), des branchements conditionnels (« if ») et des boucles inconditionnelles (boucles « for », qui répètent les opérations sur un nombre fini d'éléments, calculé à l'avance). Les risques de programmes infinis viennent des programmes récursifs et des boucles conditionnelles (« while »). Nous nous intéressons à ce cas pour le moment.

Considérons par exemple la fonction suivante, qui étant donné un entier naturel $n > 0$, détermine le plus petit entier k tel que $n \leqslant 2^k$:

```python
def puissance2_supérieure(n):
    """
    Calcule le plus petit entier k tel que n <= 2**k.
    """
    k, p = 0, 1
    while p < n:  # V = n-p  ; I: 2**(k-1) < n and p == 2**k
        k = k + 1
        p = p*2   # V décroît ici
    return k
```

On comprend qu'un tel programme termine : puisque p augmente à chaque tour de boucle, il finira par être plus grand que n et la boucle s'arrête.

2.2. Formalisation : variants de boucles

Définition – Variant de boucle

Un variant de boucle est une quantité entière positive à l'entrée de chaque itération de la boucle et qui diminue strictement à chaque itération.

Dans l'exemple précédent, on peut étudier la grandeur $V = n - p$, ainsi :
- à l'initialisation, $p = 1$ et $n > 0$ donc $n - p \geqslant 0$;
- si la condition de boucle est vérifiée à une itération donnée, alors par définition $n - p > 0$;
- à la fin de la boucle, la quantité est passée de $V = n - p$ à $V' = n - p' = n - 2p$, or $p \geqslant 1$ donc $V' < V$: elle décroît strictement.

Théorème

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

Notons que si on connaît **à l'avance** un majorant du nombre d'itérations, on peut réécrire le code avec une boucle « for » (quitte à l'arrêter prématurément avec `break`), ce qui résout la question de la terminaison. Cela n'est cependant pas toujours possible.

2.3. Programmes récursifs : variants d'appel

Définition – Variant d'appel

Un variant d'appel est une quantité entière positive à l'entrée de chaque appel de la fonction et qui diminue strictement à chaque appel récursif.

Prenons l'exemple de la factorielle : le variant d'appel est assez naturellement le nombre n passé en paramètre.

```
def fact(n):  # variant d'appel : n
    if n == 0:     # garantit que n reste positif
        return 1
    u = fact(n-1)  # appel récursif : n diminue
    return n*u
```

Théorème

Si un programme récursif admet un variant d'appel, il termine.

III. Correction

3.1. Position du problème

Définition – correction d'un algorithme

Un algorithme est **correct** vis-à-vis d'une spécification lorsque, **quelle que soit l'entrée** :
- il termine
- le résultat renvoyé vérifie la spécification.

Si l'algorithme est tel que sa terminaison n'est pas assurée, mais qu'il renvoie un résultat conforme à la spécification dans les cas où il termine, alors on dit que l'algorithme est **partiellement correct**.

3.2. Formalisation : invariant de boucle

Définition – invariant de boucle

On considère ici une boucle (conditionnelle ou non). Un prédicat est appelé un invariant de boucle lorsque :
- il est vérifié avant d'entrer dans la boucle ;
- s'il est vérifié en entrée d'une itération, il est vérifié en entrée de la suivante.

Cette définition ne dit rien de l'utilité de l'invariant choisi : il faut en choisir un en lien avec les spécifications.

Si on reprend l'exemple de la fonction `puissance2_supérieure` , on peut proposer comme invariant de boucle le prédicat suivant : $I \overset{\text{déf}}{=} \{2^{k-1} < n \text{ et } p = 2^k\}$. En effet :
- **Initialisation** : à l'entrée dans la boucle, $k = 0$ et $p = 1$, donc I est vrai ;
- **Conservation** : si I est vrai à l'entrée de la boucle, alors
  - ▷ $k' = k + 1$ donc $2^{k'-1} = 2^k = p$; or à l'entrée de la boucle $p < n$, donc on maintient $2^{k'-1} < n$ en sortie de boucle ;
  - ▷ $p' = 2p = 2^k \times 2 = 2^{k+1} = 2^{k'}$.

On retrouve donc en sortie de boucle $I' \overset{\text{déf}}{=} \{2^{k'-1} < n \text{ et } p' = 2^{k'}\}$ vraie, donc si $p' < n$, la boucle effectuera un nouveau tour et I sera vraie à l'entrée de cette nouvelle itération : I est bien un invariant de boucle.
- **Terminaison** : il nous permet de vérifier la spécification puisqu'après la dernière boucle, $p > n$ (condition de fin de boucle) et l'invariant assure que $2^{k-1} < n \leqslant p = 2^k$, donc k correspond bien à la spécification.

3.3. Cas récursif : invariant d'appel

Dans un cas récursif, on introduit un **invariant d'appel récursif** : une propriété qui
- est vérifié lorsque la fonction termine dans le cas de base ;
- si elle est vraie après appel récursif sur le sous-problème, est vraie à la fin de la fonction pour le problème général.

Cette approche est finalement une récurrence, ce qui est un des intérêts des approches récursives : les preuves y sont plus simples.

Dans le cas de la factorielle : si on note u_n la valeur retournée par la fonction, alors la propriété $u_n = n!$ est un invariant d'appel puisque :
- dans le cas de base, $u_0 = 1$ et $0! = 1$;
- si après un appel récursif avec le paramètre $n - 1$, $u_{n-1} = (n-1)!$, alors les instructions qui suivent cet appel imposent

$$u_n = n \times u_{n-1} \overset{\text{hyp}}{=} n \times (n-1)! = n!$$

ce qui prouve la récurrence et donc la correction de l'algorithme.

IV. Complexité

4.1. Position du problème

La problématique de la complexité est de savoir combien de temps (complexité temporelle) ou combien de place mémoire (complexité spatiale) sera nécessaire pour l'exécution d'un algorithme. Dans la pratique, il ne s'agit pas de compter la durée de calcul en secondes, mais de caractériser la façon dont cette durée évolue selon la taille des données en entrée. Par exemple, pour un algorithme classique qui compte les occurrences dans un tableau de valeurs :

```
def occurrences(x, tab):
    """
    Compte les occurrences de la valeur x dans le tableau tab
    """
    nb = 0
    for elt in tab:
        if elt == x:
            nb += 1
    return nb
```

on doit parcourir tous les éléments du tableau ; on comprend donc qu'en ordre de grandeur, un tableau de taille 1 000 000 sera mille fois plus long à traiter qu'un tableau de taille 1 000. En revanche, on a besoin d'uniquement deux variables pour faire fonctionner l'algorithme (on met de côté la taille des entrées) : la place mémoire nécessaire pour l'algorithme lui-même est bornée, quelque soit la taille du tableau.

Dans la suite, nous étudierons essentiellement la question de la complexité temporelle.

4.2. Notation $\mathcal{O}$ et ordre de grandeur

Définition – suite dominée

Soient $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ deux suites de nombres réels non nuls, on dit que la suite $(u_n)_{n\in\mathbb{N}}$ est dominée par $(v_n)_{n\in\mathbb{N}}$ lorsque la suite quotient $(u_n/v_n)_{n\in\mathbb{N}}$ est bornée.

On note alors $u_n = \mathcal{O}(v_n)$, qui se lit « u_n est un grand « O » de v_n ».

Dans la pratique qui nous occupe, nous nous intéresserons à des suites strictement positives ; sous cette condition, cela signifie

$$\exists M > 0,\ \forall n\ u_n < M v_n$$

La notion de suite dominée est une notion asymptotique : puisqu'on parle de suite bornée ou non, on s'intéresse implicitement au comportement pour $n \to \infty$. En particulier, les cas où $u_n/v_n \to 0$ sont assez faciles à déterminer, par exemple

$$n = \mathcal{O}(n^2) \quad ; \quad n^\alpha = \mathcal{O}(n^\beta),\ \forall \alpha < \beta \quad ; \quad n^\alpha = \mathcal{O}(\mathrm{e}^n),\ \forall \alpha \quad ; \quad \log_2 n = \mathcal{O}(n)$$

> ℹ On a introduit ici le logarithme en base 2, beaucoup plus courant en informatique que les logarithmes décimal et naturel ; en effet, $y = \log_2 x \iff x = 2^y$, et les puissances de 2 se rencontrent souvent en informatique. Rappelons que $\log_2(x) = \ln x / \ln 2$.

Le cas intéressant en informatique est cependant celui où les deux suites ont le même comportement asymptotique, autrement dit que $u_n/v_n \to \mathrm{C^{te}} \neq 0$, par exemple :

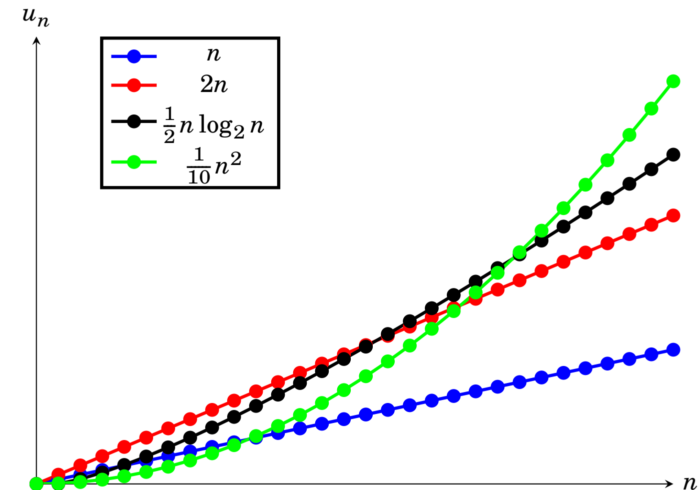
$$Kn^\alpha = \mathcal{O}(n^\alpha),\ \forall \alpha \quad ; \quad \log_2 n = \frac{\ln n}{\ln 2} = \mathcal{O}(\ln n) \quad ; \quad n \ln n + n = \mathcal{O}(n \ln n)$$

Dans ce cas, on a simultanément

$$u_n = \mathcal{O}(v_n) \quad \text{et} \quad v_n = \mathcal{O}(u_n)$$

et on dit que les deux suites sont **du même ordre de grandeur**.

Comparons graphiquement quelques exemples pour expliciter l'idée :



On comprend donc que quel que soit le préfacteur, il existe toujours un rang à partir duquel

- soit deux suites évoluent de façon comparable : elles sont du même ordre de grandeur ;
- soit l'une domine l'autre sans que la réciproque soit vraie : elle augmente alors beaucoup plus vite.

4.3. Application à la complexité

Il est très courant qu'un algorithme aie un temps de calcul dépendant de l'entrée, que l'on notera alors comme une suite C_n (avec n la « taille » de l'entrée, à définir selon le problème) ; par exemple :

```
def indice_première_occurrence(x, tab):
    """
    Si x est présent dans tab : renvoie la position de la 1ere occurrence.
    Sinon : renvoie -1
    """
    n = len(tab)
    for i in range(n):
        if tab[i] == x:
            return i
    return -1
```

Si x se trouve en position 0, l'algorithme prendra un temps constant quelque soit la taille du tableau, mais si celui-ci n'est pas présent, on parcourt tout le tableau, donc $\mathcal{O}(n)$ opérations seront effectuées. On comprend que l'on peut alors calculer plusieurs complexités :

- celle du meilleur cas, ici $C_n = \mathcal{O}(1)$;
- celle du pire cas, ici $C_n = \mathcal{O}(n)$;
- la complexité moyenne, qui demande une analyse probabiliste sur la nature de l'entrée.

Dans le cadre du programme, on se contentera d'estimer les complexités dans le pire cas.

En algorithmique, on cherche à estimer un majorant « représentatif » du temps de calcul, donc on cherchera un **ordre de grandeur**. L'usage en algorithmique est d'utiliser (abusivement) la notation $\mathcal{O}$ pour désigner un ordre de grandeur, et pas n'importe quelle suite dominante.

4.4. Complexités usuelles

a) La complexité constante

On dit qu'un algorithme a une complexité constante lorsque son temps d'exécution ne dépend pas de la taille des données. En général, on les appelle également « opérations élémentaires » : elles sont importantes car elles servent de brique élémentaire aux calculs de complexité. Les opérations de complexité constante usuelles sont toutes les instructions simples du langage Python hors boucles :

- l'affectation `=`
- la comparaison `==`, `!=`, `<` etc...
- les opérations arithmétiques +, −, ∗, /, ∗∗, // et % ;
- les fonctions mathématiques usuelles appliquées à un nombre : $\log_2$, exp ...
- les branchements conditionnels `if`, `elif`, `else`.

Ajoutons à cet ensemble les opérations suivantes sur les listes :

- lecture/affectation de l'élément i : `li[i]` ;
- récupération de la taille : `len(li)` ;
- suppression du dernier élément : `li.pop()` ;
- ajout d'un élément `x` : `li.append(x)`.

ainsi que sur les dictionnaires

- création de clé/lecture/écriture de valeur : `d[clé]` ;
- récupération de la taille : `len(d)` ;
- suppression d'une paire clé-valeur : `del d[clé]` ;
- test d'existence d'une clé : `clé in d`.

`pop(i)` permet de supprimer un élément ailleurs qu'en dernière position, mais cela implique une complexité différente ; c'est pour cela que le programme officiel limite `pop` à son usage en dernière position.

En réalité, `append` n'a pas une complexité constante, mais presque toujours constante, et de temps en temps en $\mathcal{O}(n)$. Le « de temps en temps » est choisi de façon à ce que `append` garde une complexité constante en moyenne, d'où le choix de la considérer toujours constante dans ce cours.

b) La complexité linéaire

On dit qu'un algorithme a une complexité linéaire quand son temps d'exécution est en $\mathcal{O}(n)$. C'est le cas des fonctions `occurrences` et `indice_première_occurrence` (dans le pire cas) précédemment définie.

Démontrons-le pour `indice_première_occurrence` (on note n la taille du tableau en entrée) : le contenu de la boucle « for » ne contient que des opérations élémentaires, donc le temps d'exécution c_i de la i^e boucle est majoré par une constante : $c_i < K$. La boucle est parcourue $n_f \leqslant n$ fois, et n fois exactement dans le pire cas. En ajoutant la durée des opérations élémentaires avant la boucle c_0, on en déduit :

$$C_n = \sum_{i=1}^{n_f} c_i + c_0 \leqslant K n_f + c_0 \leqslant Kn + c_0 = \mathcal{O}(n)$$

C'est également la complexité que l'on retrouve dans les créations de listes et dictionnaires en compréhension

```
li = [i**2 for i in range(n)]
di = {mot: 0 for mot in texte}
```

ainsi que dans le test d'appartenance d'une valeur dans une liste `x in li`.

c) La complexité quadratique / polynomiale

La complexité d'un algorithme est quadratique si elle est en $\mathcal{O}(n^2)$, et par extension polynomiale si elle est en $\mathcal{O}(n^k)$, $k \in \mathbb{N}$. Les cas usuels de complexité quadratique apparaissent lors des parcours de tableaux à deux dimension, ou lors des doubles parcours de tableaux ; par exemple :

```
def plus_petite_différence(tab):
    """
    Renvoie la plus petite différence entre deux éléments
    distincts d'un tableau
    """
    n = len(tab)
    dist = abs(tab[0] - tab[1])
    for i in range(n):
        for j in range(i+1, n):
            if dist > abs(tab[i] - tab[j]):
                dist = abs(tab[i] - tab[j])
    return dist
```

Ici, la petite boucle ne contient que des opérations élémentaires et est répétée exactement $n-i-1$ fois, donc le temps de calcul u_i de cette petite boucle est majoré par $K(n-i-1)$. On en déduit que la grande boucle, répétée pour toutes les valeurs $i \in [\![0; n-1]\!]$ aura pour temps de calcul (on peut tout de suite négliger les opérations élémentaires avant la boucle)

$$C_n = \sum_{i=0}^{n-1} u_i \leqslant K \sum_{i=0}^{n-1} (n-i-1) = K\left(n^2 - \frac{(n-1)n}{2} - n\right) = \mathcal{O}(n^2)$$

d) La complexité logarithmique, quasi-linéaire

On dit que la complexité d'un algorithme est logarithmique si $C_n = \mathcal{O}(\log_2 n)$, et quasi-linéaire si $C_n = \mathcal{O}(n \log_2 n)$.

Reprenons pour exemple la fonction `puissance2_supérieure` et considérons la valeur de n comme paramètre de complexité. La boucle « while » fait exactement k tours et ne contient que des opérations élémentaires, donc le temps d'exécution de la fonction est $C_n = Ak$ avec A une constante ; il reste donc à relier k et n. Par définition du contrat de fonction,

$$2^k \leqslant n \leqslant 2^{k+1} \quad \Rightarrow \quad k \leqslant \log_2 n \leqslant k+1 \quad \Rightarrow \quad \mathcal{O}(k) = \mathcal{O}(\log_2 n)$$

On en déduit finalement que la complexité $C_n = \mathcal{O}(\log_2 n)$ est logarithmique.

e) La complexité exponentielle / factorielle

La complexité est exponentielle si $C_n = \mathcal{O}(a^n)$ pour $a > 0$, et factorielle si $C_n = \mathcal{O}(n!)$. Citons par exemple l'énumération de toutes les permutations d'une liste de taille n.

f) Importance de la complexité

Afin de mesurer l'impact d'une complexité, on va considérer un algorithme qui s'exécute en une seconde sur une entrée de taille $n = 10$, et on va calculer combien de temps prendrait ce même algorithme sur une entrée de taille 100 ou 1000 :

Complexité	Temps pour $n = 100$	Temps pour $n = 1000$
1	1 s	1 s
$\log_2 n$	2 s	3 s
n	10 s	1 min 40 s
$n \log_2 n$	20 s	5 min
n^2	1 min 40 s	2 h 46 min 37 s
2^n	$\sim 10^{19}$ années	$\sim 10^{290}$ années
$n!$	$\sim 10^{144}$ années	$\sim 10^{2553}$ années

4.5. Complexité spatiale

Proposons une fonction pour calculer l'ensemble des termes de la suite de Syracuse jusqu'au rang n :

```
def syracuse(n, u0):
    u = [u0]
    for i in range(n):
        u1 = u[-1]
        if u1%2 == 0:
            u.append(u1//2)
        else:
            u.append(3*u1 + 1)
    return u
```

On constate que la liste va finalement avoir pour taille $n+1$: la place prise en mémoire par le programme sera donc proportionnelle à n. On dit alors que la complexité spatiale est en $\mathcal{O}(n)$.

V. Exemple d'analyse complète : le tri par insertion

5.1. Présentation

On considère un algorithme de tri de données : le tri par insertion, dont le principe est le suivant :

- les $i-1$ premières cases étant triées, on détermine la position à laquelle insérer le i^e élément afin de trier le tableau tout en décalant les cases correspondantes ;
- on effectue l'opération précédente pour chaque case i.

On appelle cet algorithme le « tri par insertion » car à chaque passage, on insère un nouvel élément dans un ensemble trié. On le décrit souvent comme le tri naturel du joueur de cartes. Par exemple, pour un tableau contenant 3-2-1-5-4-2, on aura l'éxécution suivante :

1er passage	3	**[2]**	1	5	4	2
2e passage	2	3	**[1]**	5	4	2
3e passage	1	2	3	**[5]**	4	2
4e passage	1	2	3	5	**[4]**	2
5e passage	1	2	3	4	5	**[2]**
6e passage	1	2	2	3	4	5

où la case entourée en gras est celle insérée. On propose l'implémentation suivante :

```python
def tri_insertion(tab):
    """
    Procédure qui trie en place un tableau de nombres fourni.
    """
    n = len(tab)
    for i in range(n):  # insertion de i
        j, x = i, tab[i]
        while j > 0 and tab[j-1] > x:  # on décale les cases
            tab[j] = tab[j-1]
            j = =-1
        tab[j] = x  # on insère la nouvelle valeur
```

5.2. Terminaison

La boucle for se termine, puisqu'elle s'exécute exactement n fois. La boucle while admet pour variant j, qui est bien décroissante dans la boucle et nécessairement positif étant donné la condition de boucle : on en déduit que la boucle termine.

5.3. Correction

Dans toute la suite, la notation $t_{[i:j]}$ représente, comme en Python, le sous-tableau des éléments de i inclus à j exclus. Les notations primées représentent les variables en fin de boucle.

Examinons d'abord la boucle « while » interne, pour laquelle on a pour précondition la propriété :

$$I \overset{\text{déf}}{=} \{t_{[0:i]} \text{ est trié}\}$$

On considère la propriété suivante : à chaque tour, $t_{[0:j]}$ et $t_{[j+1:i+1]}$ sont triés :

$$J \overset{\text{déf}}{=} \{t_{[0:j]} \text{ trié} \quad \text{et} \quad t_{[j+1:i+1]} \text{ trié} \quad \text{et} \quad x < t_{j-1}\}$$

- **Initialisation** : cette propriété est vraie avant d'entrer dans la boucle, puisque au début $j = i$, $t_{[0:j]}$ est donc trié par précondition, $t_{[j+1:i+1]}$ est vide donc trié, et $x < t_j$ sinon on ne rentre pas dans la boucle ;
- **Conservation** : le fait de décaler t_{j-1} en t_j ne change pas la propriété de tri des sous-tableaux, et si la boucle refait un tour alors $x < t_{j'-1}$. J est donc un invariant ;
- **Terminaison** : la condition de sortie nous amène alors à

$$t_{[0:j]} \text{ trié} \quad \text{et} \quad t_{[j+1:i+1]} \text{ trié} \quad \text{et} \quad t_{j-1} \leqslant x < t_{j+1}$$

donc en insérant x (valeur qui était en position i au début de la boucle while) en position t_j, on obtient un ensemble $t_{[0:i+1]}$ trié.

Considérons maintenant la boucle for externe et la propriété I :

- **Initialisation** : avant d'entrer dans la boucle, $i = 0$ donc le sous-tableau $t_{[0:i]}$ est vide, donc trié ;
- **Conservation** : si I est vraie avant d'entrer dans la boucle, alors $t_{[0:i+1]}$ est trié en sortie d'après l'analyse précédente ; incrémenter i restaure ainsi la propriété I ; on en déduit que I est une propriété invariante ;
- **Terminaison** : en fin de boucle, $i = n$ et on peut donc conclure grâce à I que **le tableau est trié**.

5.4. Complexité

La boucle while interne effectue entre 1 et kn opérations, le meilleur cas étant celui de t_i déjà en bonne position et le pire étant celui où t_i est le minorant de $t_{[0:i+1]}$. Cet ensemble étant répété n fois, on obtient une complexité en $\mathcal{O}(n^2)$ dans le pire cas, et en $\mathcal{O}(n)$ dans le meilleur cas (tableau déjà trié). Dans le cas moyen, la complexité reste en $\mathcal{O}(n^2)$.

> Le tri par insertion est le tri le plus efficace sur des petits tableaux, ou sur des tableaux déjà presque triés.

VI. Exemple d'analyse complète : le tri fusion

Reprenons le tri fusion proposé comme ci-dessous :

```python
def fusion(t1, t2):
    i1, i2, n1, n2 = 0, 0, len(t1), len(t2)
    tab = [0] * (n1+n2)      # tableau de retour
    t1.append(float('inf'))  # sentinelle
    t2.append(float('inf'))  # sentinelle
    for j in range(n1+n2):
        if t2[i2] < t1[i1]:
            tab[j], i2 = t2[i2], i2+1
        else:
            tab[j], i1 = t1[i1], i1+1
    return tab

def tri_fusion(tab):
    n = len(tab)
    if n < 2:  # condition d'arrêt
        return tab
    t1 = tri_fusion(tab[:n//2])
    t2 = tri_fusion(tab[n//2:])
    return fusion(t1, t2)
```

6.1. Terminaison

La fonction `fusion` ne contient pas de boucle while ni d'appel récursif : elle termine donc.

La fonction `tri_fusion` fait des appels récursifs uniquement si la taille du tableau est $n \geqslant 2$, et dans ce cas chaque appel récursif reçoit un tableau deux fois plus petit : $n' = n/2 < n$: on en déduit que n est un variant d'appel, donc que la fonction récursive `tri_fusion` termine.

6.2. Correction

a) Correction de `fusion`

On rappelle que t_1 et t_2 sont triés en entrée par précondition. On propose les propriétés invariantes suivantes pour la boucle :

$$I_1 = \{t_{[0:j]} \text{ contient les } j \text{ plus petits éléments de } t_1 \text{ et } t_2 \text{ en ordre trié}\}$$

$$I_2 = \{t_{1[i_1]} \text{ et } t_{2[i_2]} \text{ sont les plus petits éléments de } t_1 \text{ et } t_2 \text{ non copiés dans } t\}$$

Vérifions qu'il s'agit bien d'invariants de boucle :

- **Initialisation** : avant l'entrée dans la boucle, $j = 0$ donc $t_{[0:j]}$ est vide, ce qui assure I_1 ; aucun élément n'est copié et t_1 et t_2 sont triés, donc I_2 est aussi vérifiée ;
- **Conservation** : supposons maintenant qu'au début d'un nouveau tour de boucle $t_{1[i_1]} < t_{2[i_2]}$, alors $t_{1[i_1]}$ est le plus petit élément qui n'a pas été copié dans *tab* ;
  - ▷ en copiant $t_{1[i_1]}$ dans $t_{[j]}$, on s'assure que $t_{[0:j]}$ contient les $j+1$ plus petits éléments de t_1 et t_2 en ordre trié ;
  - ▷ en incrémentant i_1 et avec l'hypothèse de t_1 trié, on restaure l'invariant I_2 ;
  - ▷ à la fin de la boucle, j augmente, ce qui restaure l'invariant I_1.

  La démonstration est identique si $t_{2[i_2]} < t_{1[i_1]}$.
- **Terminaison** : à la fin de la boucle, $j = n$, donc l'invariant I_1 assure que t contient les éléments de t_1 et t_2 triés, et I_2 assure qu'il ne reste pas d'élément à copier puisque seules restent les sentinelles.

Ainsi on peut conclure que si t_1 et t_2 sont triés en entrée, `fusion` renvoie un tableau contenant leurs éléments en ordre trié.

b) Correction de `tri_fusion`

Considérons la propriété suivante :

$$I = \{t \text{ est trié}\}$$

Alors :

- dans le cas de base, t contient 0 ou 1 élément, il est donc trié et I est vraie ;
- si on conduit des appels récursifs, alors supposons que I soit vraie pour t_1 et pour t_2 ; puisque t_1 et t_2 contiennent les éléments de t, l'analyse précédente de la fonction `fusion` nous permet de conclure que t contient toujours les mêmes éléments, mais en ordre trié.

Nous avons ainsi prouvé la correction du tri fusion.

6.3. Complexité

Puisque `fusion` effectue n tours de boucle, son temps d'exécution s'écrit $D_n = an$; on en déduit la relation de récurrence suivante pour le temps d'exécution de `fusion` :

$$C_n = 2C_{n/2} + an \quad \Rightarrow \quad \frac{C_n}{n} = \frac{C_{n/2}}{n/2} + a$$

supposons (sans perte de généralité) que $n = 2^k$; dans ce cas on pose $w_k = C_n/n$ et la relation précédente se réécrit

$$w_k = w_{k-1} + a \quad \Rightarrow \quad w_k = ak + w_0 \quad \Rightarrow \quad C_n = akn + w_0$$

et puisque $k = \log_2 n$,

$$C_n = \mathcal{O}(n \log_2 n)$$

VII. Exercices

1 – On considère la fonction ci-dessous qui, étant donné un entier $n > 1$, cherche un entier $p \leqslant n$, $p \neq 1$, qui divise n :

```
def diviseur(n):
    p = 2
    while n%p != 0:
        p += 1
    return p
```

Montrer que l'algorithme termine, puis donner sa complexité dans le pire cas.

2 – On considère la fonction ci-dessous qui calcule le nombre de chiffres dans l'écriture décimale d'un nombre

```
def nombre_chiffres(x):
    nb_dec = 1
    y = x  # on évite de modifier le paramètre x (mauvaise pratique)
    while y > 9:
        y = y // 10
        nb_dec += 1
    return nb_dec
```

Montrer que l'algorithme termine, puis qu'il est correct. Estimer sa complexité.

3 – On considère l'algorithme d'Euclide pour déterminer un PGCD de deux nombres non nuls :

```
def pgcd(a, b):
    x, y = a, b  # on évite de modifier les paramètres a, b
    while y != 0:
        x, y = y, x%y
    return x
```

Montrer que l'algorithme termine.

4 – Écrire une fonction avec une boucle « while » qui prend en paramètre une liste `tab` de nombres entiers et un nombre `x` et renvoie `True` si `x` est contenu dans `tab`, `False` sinon. Montrer que cette fonction termine, puis estimer sa complexité dans le pire cas.

5 – On souhaite écrire une fonction qui prend en paramètre un tableau de nombres (liste Python) et qui renvoie la somme des éléments du tableau.

***5.1* –** Écrire une telle fonction (avec sa docstring).

***5.2* –** Proposer un jeu de tests.

***5.3* –** Montrer la correction de l'algorithme.

***5.4* –** Calculer la complexité.

6 – On étudie une autre méthode de tri d'un tableau : le tri par sélection.

***6.1* –** Écrire une fonction `indice_min` qui prend en paramètre un tableau et renvoie l'indice donnant la position du plus petit. Calculer sa complexité.

Le tri par sélection consiste à parcourir le tableau et, en étant à une position i :

- identifier la position du plus petit élément dans l'intervalle $[\![i; n-1]\!]$;
- le permuter avec celui en position i.

***6.2* –** Écrire une fonction `tri_sélection` qui prend en paramètre un tableau et applique en place l'algorithme.

***6.3* –** Calculer la complexité de cet algorithme de tri.

7 – Prouver le tri rapide.