8 Terminaison, correction et complexité d'algorithmes

Nous revenons dans ce chapitre sur ces notions pour les définir plus formellement et pour montrer les techniques classiques de preuves associées à ces notions.

Compétences attendues en fin de chapitre

- Lire, écrire et formaliser la spécification d'un algorithme.
- Montrer la terminaison d'un algorithme itératif par variant de boucle. Montrer la terminaison d'un algorithme récursif par récurrence.
- Montrer la correction d'un algorithme itératif par invariant de boucle. Montrer la correction d'un algorithme récursif par récurrence.
- Identifier les opérations élémentaires ou non élémentaires, notamment lorsque ces opérations concernent les listes.
- Calculer la complexité d'un algorithme itératif ou récursif.

Remarque : face à un algorithme itératif simple, vous devez savoir identifier un variant de boucle sans indication. Vous pourrez être amenés à montrer un invariant de boucle plutôt avec des indications.

I Spécification d'un algorithme

Le but d'un algorithme est toujours d'effectuer certains calculs, de modifier l'état d'un programme ou de produire une sortie. La *spécification d'un algorithme* permet de rassembler ces informations dans une description plus ou moins formelle. Elle se compose de deux parties : la description de l'*entrée* de l'algorithme et la description de sa *sortie*.

- **Entrées :**
  - **Spécification des arguments d'un programme :** on précise le nombre d'arguments d'un algorithme et leur type. Cela correspond en OCaml au type d'une fonction, ça sera aussi le cas en C avec les prototypes de fonction.
  - **Pré-conditions :** des conditions supplémentaires sur les entrées admissibles de l'algorithme et sur son contexte d'exécution. Par exemple, supposer que la liste est triée, supposer que l'entier est positif, supposer l'existence d'un fichier, etc. Sur les entrées ne vérifiant pas les préconditions, l'algorithme pourra avoir un comportement indéfini.

  Les arguments de fonctions vérifiant la spécification des arguments et les préconditions sont appelés des *entrées admissibles*.
- **Sorties :**
  - **Spécification du résultat :** on spécifie la valeur de retour attendue (explicitement, par exemple avec une définition mathématique, ou avec un ensemble de sorties admissibles) et son type attendu/
  - **Effets secondaires de l'algorithme :** on spécifie quels changements son exécution va effectuer sur le contexte dans lequel elle est exécutée. Par exemple, une fonction modifie une variable globale ou la liste en entrée, elle fait planter l'ordinateur, elle envoie un mail à 40 personnes sans les mettre en copie cachée, etc.

Remarque 1 – Sur les préconditions et effets

Si elles ne sont pas précisées, on considère souvent qu'un algorithme n'a pas de précondition et pas d'effet. Par exemple, si une fonction prend une liste en entrée, on s'attend à ce qu'elle ne soit pas modifiée sauf si un effet est spécifié.

Exemple 2 – Spécification d'un algorithme

Voici la spécification de l'algorithme de *recherche dichotomique* :
- **Entrées :** une liste `L` et un élément `e` de même type que les éléments de `L`.
- **Préconditions :** la liste `L` est triée par ordre croissant.
- **Sorties :** un booléen `True` si `e` est dans `L`, `False` sinon.

La spécification d'un algorithme ne spécifie pas une implémentation particulière : plusieurs algorithmes peuvent avoir la même spécification. Par exemple, la fonction de recherche dans une liste classique (en parcourant la liste élément par élément) répond à la spécification de l'Exemple 2.

Remarque 3

Il est courant d'utiliser la syntaxe `assert` pour représenter et vérifier une précondition dans un algorithme. Par exemple, on peut écrire :

Python
```
def fact(n):
    assert n >= 0
    # ...
```

pour écrire une fonction factorielle qui n'est définie que sur les entiers naturels. Si l'assertion est fausse, l'algorithme s'arrête et lève une exception.
Une telle assertion est une bonne idée lorsque l'on programme sur machine, notamment quand on utilise plusieurs fonctions ensemble : une mauvaise programmation de la première pourrait entraîner une erreur d'entrée dans la seconde, ce que l'assertion pourra détecter. Cependant, on évite de les utiliser lorsque la fonction possède une précondition difficile à vérifier en termes de complexité. Par exemple :

Python
```
def recherche_dichotomique(L, e):
    assert est_triee(L)
    # ...
```

est inopportun puisque vérifier qu'une liste est triée s'exécute en $\Theta(|L|)$, ce qui est bien plus coûteux que la recherche dichotomique elle-même.

II Terminaison et correction d'un algorithme

Définition 4 – Terminaison

On dit qu'un algorithme *termine* si son exécution s'arrête après un nombre fini d'opérations pour toute entrée admissible.

Remarque 5 – Sur ce nombre d'opérations

Évidemment, le nombre d'opérations peut dépendre de l'entrée. Ce nombre est en fait l'objet de la Section III sur la complexité.

Exemple 6 – Terminaison d'un algorithme

Voici un algorithme qui ne termine pas :

```python
def boucle_infinie():
    while True:
        print("Je ne termine pas !")
```

En revanche, l'algorithme suivant termine :

```python
# n est un entier positif ou nul
def fonction_qui_termine(n):
    while n > 0:
        print("Je termine !")
        n = n - 1
```

Définition 7 – Correction partielle

Un algorithme est dit *partiellement correct* (par rapport à sa spécification) si pour toute entrée vérifiant les préconditions, soit son exécution termine et est conforme à la spécification [a], soit elle ne termine pas.

a. Ici, j'entends *conforme à la spécification* comme *renvoie une valeur conforme à la spécification et modifie le contexte de l'algorithme conformément à la spécification.*

Définition 8 – Correction totale

Un algorithme est *totalement correct* (par rapport à sa spécification) s'il termine et est partiellement correct, c'est-à-dire pour toute entrée vérifiant les préconditions l'algorithme renvoie un résultat conforme à la spécification. On dit aussi simplement que l'algorithme est *correct*.

Exemple 9 – Exponentiation mal faite

Si l'on écrit l'exponentiation naïve sans réfléchir en OCaml :

```python
def expo_presque_correcte(x, n):
    if n == 1:
        return x
    else:
        return x * expo_presque_correcte(x, n-1)
```

Cette fonction est partiellement correcte pour la spécification usuelle de la puissance sur les entiers positifs :

Exponentiation d'entiers

Entrée : $x, n \in \mathbb{N}$
Sortie : x^n

Mais elle n'est pas totalement correcte, puisque l'appel à `expo_naive(x, 0)` ne termine jamais.

II.A Cas d'un algorithme itératif

Comme dans l'Exemple 6, on voit qu'un algorithme itératif peut ne pas terminer seulement à cause d'une boucle infinie (ou à l'appel d'une fonction ne terminant pas). Ainsi, en face d'un algorithme itératif, on identifie les boucles de notre algorithme et on cherche à montrer qu'elles terminent.

Définition 10 – Variant de boucle

Un *variant de boucle* est une quantité, calculée à partir de l'état du programme, qui est :
- entière ;
- minorée par une constante ;
- strictement décroissante à chaque itération de la boucle.

Cette quantité peut être une variable, une combinaison de variables, une quantité décrivant une propriété d'une liste, etc.

Exemple 11 – Variant de boucle

Pour la fonction `fonction_qui_termine` La quantité `n` est un variant de boucle pour la boucle `while` puisque :
- `n` est un entier, par sa spécification ;
- `n` est minoré par 0, par la condition de boucle ;
- `n` est strictement décroissante à chaque itération de la boucle, par la ligne `n = n - 1`.

Théorème 12 – Terminaison d'une boucle

Une boucle termine si :
- le corps de la boucle termine ;
- et la boucle possède un variant de boucle.

Attention !

Il est important de ne pas oublier la première condition : si le corps de la boucle ne termine pas, la boucle ne termine pas non plus !

Démonstration. Soit x un variant d'une boucle. On note x_0 la valeur de x avant la première itération de la boucle et $x_1, x_2, \ldots$ les valeurs de x après la première, la deuxième, etc. itération de la boucle.
La séquence $x_0, x_1, x_2, \ldots$ est une suite d'entiers naturels minorés et strictement décroissante. Cette suite ne peut donc pas être infinie et le nombre de tours de boucle est donc majoré par le nombre d'éléments de cette suite. Or, le corps de la boucle termine : on exécute la boucle un nombre fini de fois, elle termine donc. □

Remarque 13

Quand la boucle dont on cherche à montrer la terminaison est une boucle `for`, le variant de boucle est souvent évident. Par exemple, dans la boucle `for i in range(n):`, la quantité `n - i` est un variant de boucle.

Exercice 12

Dans la fonction suivante, montrer sa terminaison en identifiant un variant de boucle :

Python
```python
# n est un entier positif ou nul
def log2(n):
    i = 0
    x = n
    while x > 1:
        x = x // 2
        i = i + 1
    return i
```

Correction x est un variant de boucle. En effet :
- x est un entier, puisque initialisé à n qui est un entier et que l'on ne fait que des divisions entières ;
- x est minoré par 1, par la condition de boucle ;
- x est strictement décroissante à chaque itération de la boucle, par la ligne `x = x // 2` et puisque `x >= 1`.

Or, le corps de la boucle termine puisque composée d'un nombre fini d'opérations élémentaires. Ainsi, la boucle `while` termine et donc la fonction termine.

Passons maintenant à la correction partielle : l'outil utilisé pour la montrer est, en présence d'une boucle, la notion d'*invariant de boucle*.

Définition 14 – Invariants de boucle

Un *invariant de boucle* est une propriété sur l'état de l'algorithme tel que :
- au début de l'exécution de la boucle [a], l'invariant est vrai,
- s'il est vrai au début d'un tour de boucle, il reste vrai au début du tour suivant.

a. quand on y rentre pour la première fois

Propriété 15

Si une propriété est un invariant de boucle, elle reste vraie en sortant de la boucle.

Remarque 16

Dans le cas des boucles `for`, on a parfois besoin de considérer un dernier indice de boucle `i = n` pour utiliser cette propriété.

Montrer un invariant de boucle ressemble à une preuve par récurrence sur le nombre de tours de boucle, même si on ne l'exprimera pas comme ça.

Attention !

Comme un variant de boucle ne suffit pas à montrer la terminaison d'un algorithme, un invariant de boucle ne suffit pas à montrer sa correction partielle. En effet, l'invariant de boucle peut nécessiter un raisonnement supplémentaire pour aboutir à la spécification de la sortie de l'algorithme.

Exemple 17 – Tri par sélection

Développons l'exemple du tri par sélection déjà vu ensemble :

Python
```python
def indice_min_tranche(L, i):
    """Renvoie l'indice du minimum de L[i:]"""
    min = i
    for j in range(i+1, len(L)):
        if L[j] < L[min]:
            min = j
    return min
# on suppose cette fonction correcte

def tri_selection(L):
    for i in range(len(L)):
        # On cherche le minimum de L[i:] et on le place en position i
        min = indice_min_tranche(L, i)
        L[i], L[min] = L[min], L[i]
```

Ici, la spécification de l'algorithme de tri est :
- **Entrées :** une liste `L` d'éléments comparables.
- **Sorties :** rien.
- **Effets secondaires :** la liste `L` est une permutation de la liste originale triée par ordre croissant.

Ici, le fait que la liste `L` soit une permutation de la liste originale est évident : la seule instruction modifiant la liste est `L[i], L[min] = L[min], L[i]` qui échange deux éléments de la liste. Ainsi, la liste est toujours une permutation de la liste originale.

Montrons que la propriété suivante est un invariant de boucle :

« La liste `L[:i]` est triée et contient les i plus petits éléments de la liste originale. »

- Au moment de rentrer dans la boucle, on a `i = 0` et donc `L[:i]` est la liste vide. C'est une liste triée contenant les 0 plus petits éléments de la liste originale.
- Supposons l'invariant vrai à un certain tour `i` de la boucle. On a donc `L[:i]` qui contient les i plus petits éléments de la liste originale. On cherche à montrer que l'invariant est vrai au tour suivant, c'est-à-dire au tour `i+1`.
  Après le tour `i`, on a `L[i]` qui contient le minimum de `L[i:]`. Ainsi, `L[i]` est plus petit que tous les éléments de `L[i+1:]`. Or, par hypothèse de récurrence, `L[:i]` contient les i plus petits éléments de la liste originale. Ainsi, `L[:i+1]` contient les $i+1$ plus petits éléments de la liste originale. De plus, elle est triée puisque la tranche `L[:i]` est triée et que `L[i]` est supérieur ou égal à `L[i-1]` par hypothèse.

Ainsi, la propriété est un invariant de boucle. En sortant de la boucle (pour « $i = n$ »), la propriété nous indique donc que la liste `L` est triée. Ainsi, la fonction `tri_selection` est partiellement correcte.
On observe également que la fonction `indice_min_tranche` est constituée d'une boucle `for` dont le corps termine, donc elle termine. Ainsi, la fonction `tri_selection` termine, puisque elle-même constituée d'une boucle `for` dont le corps termine.
On a donc montré que la fonction `tri_selection` est totalement correcte.

II.B Cas d'un algorithme récursif

Il n'y a plus de boucle donc plus de variant ou d'invariants à notre disposition. Cependant, on peut souvent utiliser des *raisonnements par récurrence* pour montrer la terminaison et la correction partielle d'un algorithme récursif.

Exemple 18 – Exponentiation récursive

Fonction puissance

Entrée : $x \in \mathbb{Z}, n \in \mathbb{N}$
Sortie : x^n

Python
```python
def expo(x, n):
    if n == 0:
        return 1
    else:
        return x * expo(x, n-1)
```

Montrons simultanément sa terminaison et correction par récurrence pour $n \in \mathbb{N}$[a] :

Initialisation : Pour $n = 0$, l'appel `expo(x, n)` termine et renvoie 1.
Hérédité : Supposons que pour un certain $n \in \mathbb{N}$, pour tout $x \in \mathbb{R}$ l'appel `expo(x, n)` termine

et renvoie x^n. Alors l'appel `expo(x, n+1)` termine et renvoie :

$$\begin{aligned} \texttt{expo(x, n+1)} &= x \times \texttt{expo(x, n)} \\ &= x \times x^n && \text{par hypothèse de récurrence} \\ &= x^{n+1} \end{aligned}$$

Conclusion : On a montré que pour tout $n \in \mathbb{N}$, pour tout $x \in \mathbb{R}$, l'appel `expo(x, n)` termine et renvoie x^n. Ainsi, la fonction `expo` est correcte.

a. Ici, la propriété précise que l'on montre par récurrence est « pour tout $x \in \mathbb{R}$, l'appel `expo(x, n)` termine et renvoie x^n ».

Des exemples de preuves de terminaison et correction pour des algorithmes itératifs et récursifs feront l'objet d'un TD.

III Complexité d'un algorithme

La première complexité que l'on calcule sur un algorithme, c'est son temps d'exécution. C'est aussi une complexité qui dépend d'une grande variété de facteurs différents : langage d'implémentation, environnement logiciel, versions des programmes, matériel utilisé, etc. Heureusement, quand on réfléchit en termes de complexité asymptotique, il suffit de réfléchir en termes de *nombre d'instructions élémentaires* exécutées. Ainsi, il nous faut :

— estimer le nombre de fois que chaque instruction est exécuté ;
— déterminer quelles instructions sont élémentaires ou pas ;
— estimer la complexité des instructions non élémentaires en terme d'instructions élémentaires qu'elles-mêmes exécutent ;
— sommer toutes ces quantités pour obtenir une estimation du nombre d'instructions élémentaires exécutées.

Définition 19 – Complexité temporelle

La complexité temporelle d'un algorithme est le nombre d'instructions élémentaires exécutées par l'appel à cet algorithme en fonction de ses entrées.

Vous devez savoir identifier les instructions élémentaires et non élémentaires. En voici quelques exemples :

Opérations qui sont élémentaires
— Créer et initialiser une variable ;
— **accéder et modifier un élément d'une liste Python**, récupérer sa longueur avec `len(L)` ;
— exécuter des opérations arithmétiques sur les entiers et les flottants, et les comparer, effectuer des opérations booléennes ;
— effectuer un appel de fonction (attention : on compte séparément le coût de l'appel seul et le coût de l'évaluation de ses arguments éventuels, et on ne compte pas le coût des instructions à l'intérieur de la fonction) ;
— effectuer un test `if` (sauf si l'évaluation de la condition n'est pas élémentaire) ;

Opérations qui ne sont pas élémentaires
— Créer une liste de taille n (se fait en $\mathcal{O}n$, que ce soit par réplication, par compréhension, etc.) ;
— effectuer une opération sur une liste de taille n (par exemple, concaténer deux listes de taille n et m se fait en $\mathcal{O}n + m$) ;
— effectuer une boucle de taille non bornée ;

— etc.

On exprimera toujours cette complexité temporelle avec les notations grand-$\mathcal{O}$. Ainsi, ce n'est pas vraiment le nombre *exact* d'opérations élémentaires qui nous intéressera [i], mais plutôt son ordre de grandeur. Par ailleurs, on exprimera toujours cette complexité en fonction de la *taille de l'entrée*. Par exemple, la complexité d'un algorithme de tri sera exprimée en fonction de la taille de la liste à trier.

On distingue plusieurs façons de calculer cette complexité :

— **Complexité dans le pire des cas :** en fixant la taille de l'entrée, on détermine le maximum de la complexité temporelle sur toutes les entrées de cette taille. **On utilisera presque toujours cette complexité.**
— Complexité dans le meilleur des cas : en fixant la taille de l'entrée, on détermine le minimum de la complexité temporelle sur toutes les entrées de cette taille.
— Complexité en moyenne : en fixant la taille de l'entrée, on détermine la moyenne de la complexité temporelle sur toutes les entrées de cette taille.
— Complexité amortie : on calcule la complexité temporelle sur une suite d'appels à l'algorithme.

III.A Complexité d'un algorithme récursif

Pour calculer la complexité d'un algorithme récursif, on exprimera souvent cette complexité par une relation de récurrence. Par exemple, pour la fonction `expo` de l'Exemple 18, la complexité de la fonction $C(n)$ vérifie :

$$\begin{cases} C(0) = A & \text{puisque l'appel } \texttt{expo(x, 0)} \text{ termine immédiatement après un test,} \\ C(n+1) = C(n) + B & \text{puisque l'appel } \texttt{expo(x, n+1)} \text{ termine après un test et un appel récursif.} \end{cases}$$

où A et B sont des constantes indépendantes de n. On peut alors résoudre cette relation de récurrence pour obtenir $C(n) = A + Bn = \mathcal{O}(n)$.

III.B Complexité d'un algorithme itératif

Déterminer la complexité d'un algorithme itératif est souvent plus simple que pour un algorithme récursif. En effet, il s'agit alors de décomposer la complexité totale de l'algorithme en chacune des opérations effectuées à l'intérieur, éventuellement sous la forme des complexités de chaque tour dans le cas d'une boucle. Déterminons ainsi la complexité de la fonction `tri_selection` de l'Exemple 17 en fonction de n la taille de la liste en entrée :

— La complexité de la fonction `indice_min_tranche` est en $\mathcal{O}(n-i)$ puisqu'elle effectue une boucle `for` de taille $n-i$ contenant un nombre constant d'opérations élémentaires.
— La complexité de la ligne `min = indice_min_tranche(L, i)` est donc en $\mathcal{O}(n-i)$.
— La complexité de la ligne `L[i], L[min] = L[min], L[i]` est en $\mathcal{O}(1)$, puisque composée d'opérations élémentaires (accès et modification de listes).
— La complexité de la boucle `for i in range(len(L))` peut donc s'exprimer comme suis, avec $A, B > 0$ des constantes :

$$\sum_{i=0}^{n-1} A + B(n-i) = nA + B\sum_{i=0}^{n-1}(n-i) = nA + B\sum_{i=0}^{n-1} i = nA + B\frac{n(n-1)}{2} \underline{= \mathcal{O}(n^2)}$$

i. et sur lequel il peut toujours y avoir débat