Informatique en MP(2)I

Marc de Falco

Introduction à l'analyse des algorithmes

Source de l'image d'en-tête XKCD #1667

■ **Note 1 Roadmap :**
- reprendre les exemples dans les trois langages.
- rajouter plus exercices.

■

Remarque 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.

I État d'un programme

Avant de commencer à raisonner sur les algorithmes, il est nécessaire de préciser la notion d'état qui correspond à un instantané de l'environnement d'exécution d'un programme lorsque son exécution est interrompue. Bien entendu, la description complète d'un tel état ne serait pas forcément pertinent, car cela prendrait en compte l'ensemble de la mémoire. Le plus souvent, on considère uniquement ce qui est important pour ce qu'on étudie.

Ainsi, si on considère le programme suivant :

```
1 int a = 3;
2 int b = 2;
3
```

```
4  a = b;
```

L'état du programme à l'entrée de la ligne 4 est

variable	valeur
a	3
b	2

et l'état à la sortie de la ligne 4 est

a	2
b	2

Parfois, on aura besoin de plus d'information dans l'état comme la position en mémoire de certaines données. Mais assez souvent, pour les algorithmes qui nous intéressent, on pourra adopter un point de vue assez abstrait de l'état d'un algorithme comme étant une fonction partielle des noms vers les valeurs.

Pour la terminaison et la correction, on va considérer des propriétés logiques dépendant de l'état. Par exemple $a \geq 0 \wedge b \geq a$.

II Terminaison

Définition II.1 On dit qu'un algorithme **termine** quand il n'exécute qu'un nombre fini d'étapes sur toute entrée.

Remarque Cela n'empêche pas que ce nombre d'étapes puisse être arbitrairement grand en fonction des entrées.

Un algorithme qui n'utilise ni boucles inconditionnelles ni récursivité termine toujours. Ainsi, la question de la terminaison n'est à considérer que dans ces deux cas.

Considérons par exemple l'algorithme suivant qui, étant donné un entier naturel n strictement positif, inférieur à 2^{30}, détermine le plus petit entier k tel que $n \leq 2^k$:

OCaml
```
let plus_grande_puissance2 n =
    let k = ref 0 in
    let p = ref 1 in
    while !p < n do
        k := !k + 1;
        p := !p * 2
    done;
    !k
```

C
```
int plus_grande_puissance2(int n)
{
    int k = 0;
    int p = 1;

    while (p < n)
    {
        k = k+1;
        p = p*2;
    }

    return k;
}
```

On remarque que p prend successivement pour valeurs toutes les puissances de 2 jusqu'à une éventuelle sortie de boucle. Or, il existe une puissance de 2 supérieure ou égale à n, donc, une fois atteinte, la condition de la boucle `while` n'est plus remplie et l'algorithme termine.

Remarque Prouver la terminaison n'est pas une question facile, elle peut-même être insoluble. Par exemple, si on considère le programme suivant :

```python
def temps_de_vol(a) :
    u = a
    n = 0
    while u != 1 :
        if u % 2 == 0 :
            u = u // 2
        else :
            u = 3*u + 1
        n = n+1
    return n
```

être capable de prouver sa terminaison revient à prouver la conjecture de Collatz (encore dite de Syracuse).
On peut aussi considérer le tri suivant :

```python
def bogo(t) :
    while not est_trie(t) :
        melange(t)
```

Il semble très improbable que cet algorithme ne termine pas, d'ailleurs on peut prouver qu'il termine avec probabilité 1, mais on ne peut pas exclure le cas où il mélange indéfiniment.

II.1 Variant de boucle

Pour prouver la terminaison d'une boucle conditionnelle, on utilise un **variant** de boucle.

Définition II.2 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, la quantité $n - p$ est un variant de boucle :
- Au départ, $n > 0$ et $p = 1$ donc $n - p \geq 0$
- Comme il s'agit d'une différence de deux entiers, c'est un entier. Et tant que la condition de boucle est vérifiée $p < n$ donc $n - p > 0$.
- Lorsqu'on passe d'une itération à la suivante, la quantité passe de $n - p$ à $n - 2p$ or $2p - p > 0$ car $p \geq 1$. Il y a bien une stricte diminution.

Remarque Ici, en sortie de boucle, $n - p \leq 0$. On fait donc bien attention de préciser que la quantité est positive tant que la condition de boucle est vérifiée.

Si on a un variant de boucle qui vaut initialement n avant d'entrer dans la boucle, celle-ci effectue au plus n itérations car le variant diminue au moins de 1 à chaque étape.

Théorème II.1 Si une boucle admet un variant de boucle, elle termine.

II.2 Exemple de la recherche dichotomique

On considère ici la recherche dichotomique dans un tableau trié d'entiers. Étant donné un tableau t de taille $n > 0$ et un entier x dont on cherche à déterminer sa présence dans le tableau entre les indices i et j, on considère l'algorithme suivant :
- Si $i > j$, alors il n'y a pas de sous-tableau et on renvoie `false`.
- Sinon, soit m l'élément d'indice $p = \left\lfloor \frac{i+j}{2} \right\rfloor$.
  - ⋆ Si $x = m$, on renvoie `true`
  - ⋆ Si $x < m$, on continue la recherche dans le sous-tableau des indices i à $p - 1$.
  - ⋆ Si $x > m$, on continue la recherche dans le sous-tableau des indices $p + 1$ à j.

Le programme suivant présente une implémentation de cet algorithme en C.

OCaml
```
exception Trouve of int

let rech_dicho t x =
    let i = ref 0 in
    let j = ref (Array.length t) - 1 in

    try
        while !i <= !j do
            let p = (!i + !j)/2 in
            let m = t.(p) in

            if m = x
            then raise (Trouve p)
            else if x < m
            then j := p-1
            else i := p+1
        done;
        None
    with Trouve i -> Some i
```

C
```
int rech_dicho(int *t, size_t n, int x)
{
    /* renvoie un indice de x
       si x est dans le tableau et -1 sinon */
    int i = 0;
    int j = n-1;

    while (i <= j)
    {
        int p = (i+j)/2;
        int m = t[p];

        if (x == m)
        {
            return p;
        }
        else if (x < m)
        {
            j = p-1;
        }
        else
        {
            i = p+1;
        }
    }

    return -1;
}
```

Remarque On aurait pu écrire `i+(j-i)/2` et non `(i+j)/2` afin d'éviter des erreurs de dépassement dans le calcul de `i+j`.

Ici, la terminaison n'est pas immédiate, on va la prouver à l'aide d'un variant de boucle. On considère ainsi la quantité $d(i,j) = j - i$.

- Comme le tableau est non vide, $d(0, n-1) \geq 0$. Ensuite, la condition de boucle est équivalente à $d(i,j) \geq 0$, donc cette quantité est bien entière et positive à l'entrée de chaque itération.
- Quand on passe à l'itération suivante, on passe
  - ⋆ soit de $d(i,j)$ à $d(i, p-1)$. Or $d(i,p-1) = p - 1 - i = \left\lfloor \frac{i+j}{2} \right\rfloor - 1 - i < \frac{i+j}{2} - i \leq j - i = d(i,j)$.
  - ⋆ soit de $d(i,j)$ à $d(p+1, j)$. Or $d(p+1,j) = j - p - 1 = j - \left\lfloor \frac{i+j}{2} \right\rfloor - 1 < j - \frac{i+j}{2} \leq j - i = d(i,j)$.

  Dans tous les cas, $d(i,j)$ diminue strictement.

Ainsi, il s'agit d'un variant de boucle et l'algorithme termine.

Remarque On remarque que le programme récursif suivant réalise également cet algorithme.

OCaml
```
(* rech_dicho : int array
let rec rech_dicho t i j x
    if i <= j
    then let p = (i+j)/2 i
        let m = t.(p) in
        if x = m
        then Some p
        else if x < m
        then rech_dicho t
        else rech_dicho t
    else None
```

Python

```python
def rech_dicho(t, i, j, x) :
    if i <= j :
        p = (i+j)//2
        m = t[p]
        if x == m :
            return p
        elif x < m :
            return rech_dicho(t, i, p-1, x)
        else :
            return rech_dicho(t, p+1, j, x)
    else :
        return None
```

!langspec(ocaml)(c)(python)

```
Il suffit alors d'appeler `rech_dicho t 0 (Array.length t - 1) x` pour faire une
recherche sur tout le tableau.
```

```
Il suffit alors d'appeler `rech_dicho(t, 0, len(t)-1, x)` pour faire une
recherche sur tout le tableau.
```

Ici, il n'y a pas explicitement de boucle, mais le même principe peut être mis en place pour prouver que le nombre d'appels récursifs est majoré, et donc que toute exécution termine. En effet, la quantité $d(i,j) = j - i$ diminue pour les mêmes raisons à chaque appel récursif et reste entière positive.

II.3 Exemple du tri à bulle

On considère le tri à bulles dans une implémentation naïve. On effectue ainsi une série de balayages d'un tableau : on parcourt le tableau de gauche à droite et si deux éléments consécutifs sont dans le désordre, on les permute. À l'issu d'un tel balayage, si on a effectué au moins un échange, on recommence, sinon on s'arrête, car c'est le signe que le tableau est trié qu'on le prouvera à la fin de ce paragraphe.

On obtient alors le code suivant :

OCaml

C

```c
void swap(int t[], int i, int j)
{
    int temp = t[i];
    t[i] = t[j];
    t[j] = temp;
}

void tri_bulles(int t[], int nb)
{
    bool echange = true;
    while(echange)
    {
        echange = false;
        for (int i = 0; i< nb-1; i++)
        {
            if (t[i] > t[i+1])
            {
                echange  = true;
                swap(t, i, i+1);
            }
        }
    }
}
```

Si la question de la terminaison de la boucle `for` ne se pose pas, celle de la boucle `while` mérite de s'y attarder pour deux raisons :

- premièrement pour la terminaison elle-même
- deuxième pour savoir si on peut majorer le nombre d'itérations, une question qu'on reverra avec la question de la complexité.

On va prouver la terminaison pour un type d'algorithme de tri qui généralise le tri à bulle. Pour cela, on commence par définir la notion d'inversion pour un tableau t de $|t|$ éléments :

Définition II.3 On dit qu'un couple $(i, j) \in [\![0, |t| - 1,]\!]^2$ est une **inversion** pour t si $i < j$ et $t[i] > t[j]$.
On note $Inv(t)$ l'ensemble des inversions de t et $inv(t) = |Inv(t)|$ le nombre d'inversions.
On dira qu'une inversion de la forme $(i, i + 1)$ est une **inversion directe**.

On considère donc un algorithme qui résout une inversion directe dès qu'il y en a au moins une.

Python
```
def tri(t) :
    while existe_inversion_directe(t) :
        i = debut_d_une_inversion_directe(t)
        echange(t, i, i+1)
```

Le tri à bulles est alors une manière de réaliser l'algorithme précédent en effectuant des inversions alors qu'on avance dans le tableau.

Théorème II.2 Pour un tableau t les propositions suivantes sont équivalentes :
1. t est trié dans l'ordre croissant
2. t n'a pas d'inversions
3. t n'a pas d'inversions directes

■ **Preuve**
- $1) \to 3)$ naturellement si t es trié dans l'ordre croissant, $i < i + 1$ entraine $t[i] \leq t[i + 1]$, donc il ne peut il y avoir d'inversions directes.
- $2) \to 1)$ si $i < j$, comme t n'a pas d'inversions, on a $t[i] \not> t[j]$, c'est-à-dire $t[i] \leq t[j]$. Ainsi, le tableau est bien trié dans l'ordre croissant.
- $3) \to 2)$ pour tout $i < |t| - 1$ on a $t[i] \leq t[i + 1]$. Soit $i < j$, on va raisonner par récurrence sur $j - i$ pour montrer que $t[i] \leq t[j]$.
  - ⋆ si $j - i = 1$, on n'a pas d'inversion directe donc $t[i] \leq t[j]$.
  - ⋆ si $j - i > 1$ et que $t[i] \leq t[j - 1]$, car $i < j - 1$ avec $j - 1 - i = (j - i) - 1$, alors $t[i] \leq t[j - 1] \leq t[j]$ car $(j - 1, j)$ n'est pas une inversion.

  Ainsi, dans tous les cas on a bien $t[i] \leq t[j]$. Donc, t est trié par ordre croissant.

■

II.4 Lien avec la récursivité

II.5 Boucles imbriquées

III Correction

Pour parler de correction d'un algorithme, il est nécessaire d'identifier précisément ce qui doit être calculé par l'algorithme. Pour cela, on considère ici informellement des spécifications dépendant des entrées et du résultat de l'algorithme. On verra dans le chapitre sur la logique qu'il s'agit ici de prédicats logiques.

Voici des exemples de spécifications :
- le tableau t en sortie est trié dans l'ordre croissant
- la valeur renvoyée est le plus petit indice de x dans le tableau ou -1 s'il ne le contient pas.

Définition III.1 Un algorithme est **correct** vis-à-vis d'une spécification lorsque quelle que soit l'entrée
- il **termine**
- le résultat renvoyé vérifie la spécification.

On considère également la correction **partielle** en l'absence de terminaison :

Définition III.2 Un algorithme est **partiellement correct** vis-à-vis d'une spécification lorsque quelle que soit l'entrée le résultat renvoyé vérifie la spécification.

III.1 Invariant de boucle

Définition III.3 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 sortie de celle-ci.

Quand la boucle termine, on déduit alors que l'invariant est vérifié en sortie de boucle. On cherche donc un invariant qui permette de garantir la spécification en sortie de boucle.

Remarque Pour les boucles inconditionnelles, il y a une gestion implicite de l'indice de boucle qui va se retrouver dans l'invariant. On peut alors considérer que la sortie de boucle s'effectue après être passé à l'indice suivant.

Dans le cas d'une boucle conditionnelle portant sur la condition P et ayant un invariant de boucle I, en sortie le prédicat $\neg P \wedge I$ (non P et I) sera vérifié.

On peut illustrer cela en reprenant la fonction `plus_grande_puissance2` vue à la partie Terminaison. On considère ici le prédicat $I(k,p) := 2^{k-1} < n$ et $p = 2^k$.

- En entrée de boucle, on a bien $2^{-1} < n$.
- Si le prédicat est vérifié en entrée d'itération. On a alors $2^{k-1} < n$ et comme on est entrée dans cette itération $p = 2^k < n$. Donc en sortie d'itération on aura bien $I(k+1, 2p)$, car $2p = 2^{k+1}$.

Ainsi, ce prédicat est bien un invariant et en sortie de boucle (ce qui arrive nécessairement, car l'algorithme termine), le prédicat $I(k,p)$ signifie que $2^{k-1} < n$ **et la condition de sortie de boucle** qu'on a $n \leq 2^k$.

La valeur renvoyée est bien k tel que $2^{k-1} < n \leq 2^k$ ce qui était la spécification annoncée du programme.

III.2 Exemple du tri par sélection

Le programme suivant présente un algorithme de tri, appelé le *tri par sélection* dont on va analyser la complexité. Il s'agit d'un tri qui repose sur un principe simple, on va chercher le plus petit élément du tableau à trier et le placer à la position courante. On définit ainsi trois fonctions :

- `echange` réalise l'échange de valeurs entre deux cases du tableau
- `indice_minimum` renvoie l'indice de la plus petite valeur entre deux indices donnés
- `tri_par_selection` réalise le tri en parcourant le tableau du premier au dernier indice et en plaçant à la position courante le minimum restant.

OCaml

C

```c
void echange(int *tableau, int i, int j)
{
    int temp = tableau[i];
    tableau[i] = tableau[j];
    tableau[j] = temp;
}

void indice_minimum(int *tableau, int min_indice, int max_indice)
{
    int i = min_indice;

    for (int j = min_indice + 1; j <= max_indice; j++)
    {
        if (tableau[j] < tableau[i])
            i = j;
    }

    return i;
}

void tri_par_selection(int *tableau, int taille)
{
    for (int i = 0; i < taille; i++)
    {
        echange(tableau, i, indice_minimum(tableau, i, taille-1));
    }
}
```

Il n'y a pas de problèmes de terminaison ici, car toutes les boucles sont inconditionnelles. Pour prouver sa correction, on va considérer séparément les deux boucles.

- Boucle dans `indice_minimum` : on va valider l'invariant $I(i,j) := \forall k \in [\![i, j-1]\!]$, tableau$[i] \leq$ tableau$[k]$.
  - ⋆ En entrée de boucle, on a $I(\text{min\_indice}, \text{min\_indice}+1)$ vérifié directement.
  - ⋆ Si en entrée d'itération, $I(i,j)$ est vérifié, ce qui signifie que tableau$[i]$ est plus petit que les valeurs compris entre les indices i et $j-1$. Alors, on distingue deux cas :
    - soit tableau$[j] <$ tableau$[i]$ et alors en sortie i devient $i' = j$. On a alors tableau$[i'] =$ tableau$[j] <$ tableau$[i] \leq$ tableau$[k]$ pour $k \in [\![1, j-1]\!]$. Donc $I(i', j+1)$ est vérifié.

- soit tableau[i] $\leq$ tableau[j] et ainsi on a pu prolonger le prédicat à $I(i, j+1)$.

Ce prédicat est bien un invariant. Ainsi, en sortie de boucle, et donc avant de renvoyer sa valeur, on a bien $I(i, \text{taille})$ donc tableau[i] est la plus petite valeur du tableau.

- Boucle dans `tri_par_selection` : on va valider l'invariant $T(i)$:= le sous-tableau `tableau[0..i-1]` des indices 0 à $i-1$ est trié et ne contient que des valeurs plus petites que celles du sous-tableau `tableau[i..taille-1]`.
  - ⋆ En entrée de boucle, le sous-tableau est vide donc trié.
  - ⋆ Si en entrée d'itération, le prédicat est vérifié. On récupère l'indice j du minimum du sous-tableau `tableau[i..taille-1]` à l'aide la fonction `indice_minimum`, par hypothèse `tableau[j]` est alors supérieur ou égal à chaque élément de `tableau[0..i-1]`, en le plaçant à l'indice i, on a bien `tableau[0..i]` qui est trié et par construction la valeur de `tableau[i]` est inférieure à toutes celels de `tableau[i+1..taille-1]`. On a ainsi $T(i+1)$ vérifié en sortie d'itération.

Ce prédicat est bien un invariant. Ainsi, en sortie de boucle, $T(\text{taille})$ est vérifié : le tableau est trié.

IV Complexité

IV.1 Complexité dans le pire des cas

Considérons un algorithme pour lequel on peut associer à chaque entrée une notion de taille (par exemple le nombre d'éléments d'un tableau). Pour $n \in \mathbb{N}$, on note ainsi I_n l'ensemble des entrées de taille n pour cet algorithme. Pour une entrée e, on note $t(e)$ le temps pris, par exemple en seconde, par l'algorithme sur l'entrée e. De même, on note $s(e)$ l'espace mémoire maximal, par exemple en octets, occupé par l'algorithme au cours de cette exécution **sans compter la taille des entrées**.

Définition IV.1 On appelle :

- **complexité temporelle dans le pire des cas**, la suite $(C_n^t)_{n\in\mathbb{N}}$ telle que pour tout $n \in \mathbb{N}$, $C_n^t = \max_{e\in I_n} t(e)$.
- **complexité spatiale dans le pire des cas**, la suite $(C_n^s)_{n\in\mathbb{N}}$ telle que pour tout $n \in \mathbb{N}$, $C_n^s = \max_{e\in I_n} s(e)$.

Comme on va le voir, calculer explicitement ces suites n'a pas beaucoup d'intérêt tant elles sont dépendantes de la manière dont on mesure le temps et l'espace. Ce qui compte ici, c'est de connaître l'*ordre de grandeur* de ces complexités en fonction de n.

Pour un tableau de taille n, ce programme va effectuer n itérations et sa complexité est ainsi de l'ordre de n. Il est possible d'être très précis en considérant les temps pris

- pour mettre en place l'appel de fonction et le passage des arguments
- par la gestion de l'indice de la boucle for
- pour la comparaison, puis pour l'affectation éventuelle
- pour mettre en place la valeur de retour afin que le résultat soit lu

On peut remarquer que la notion de pire cas dépend de la précision à laquelle on se place. Ici, si on ne s'intéresse qu'à l'ordre de grandeur, tous les tableaux de taille n sont équivalents. Par contre, si on cherche avec précision le pire cas, il est atteint avec un tableau trié par ordre croissant, car c'est le cas qui effectue une affectation à chaque itération.

OCaml

C

```c
int maximum
{
    int M =

    for(int i=
    {
        if (M
            M
    }

    return M
}
```

Python

```python
def maximum(t) :
    m = t[0]
    for i in range(1, len(t)) :
        if m > t[i] :
            m = t[i]
    return m
```

IV.2 Comparer des complexités

Avant de pouvoir comparer les complexités des algorithmes ou des programmes, il est nécessaire de mettre en place des outils pour en parler à la fois avec précision mais également sans rentrer dans des détails d'implémentation non pertinents.

En effet, comparons les deux fonctions suivantes permettant de chercher un élément dans un tableau.

OCaml

```
int recherche(int *tab, int nb,
              int elem)
{
   for(int i=0 ; i<nb ; i++)
      if (elem == tab[i])
         return i ;
   }

   return -1 ;
}
```

C

```
int recherche
              in
{
   int ind =

   for(int i=
   {
      if (ele
         ind
   }

   return in
}
```

Python

```
def recherche(t, elem) :
    for i in range(len(t)) :
        if t[i] == elem :
            return i
    return None
```

```
def recherche(t, elem) :
    ind = None
    for i in range(len(t)) :
        if t[i] == elem :
            ind = i
    return ind
```

La fonction de droite semble moins efficace que celle de gauche, car la seconde sort tout de suite de la fonction dès qu'on a trouvé l'élément alors que la première continue à parcourir `t`.

Mais on doit se poser la question de la pertinence de cette optimisation selon le pire des cas. Ici, le pire des cas correspond à ne pas avoir `elem` dans le tableau, et à ce moment-là les deux fonctions fonctionnent de la même manière.

Remarque En fait, la fonction de droite sera souvent à privilégier, car il est souvent plus facile d'interrompre le flot que de faire en sorte de préserver l'état à la place du `return` pour pouvoir renvoyer la bonne valeur à la fin de la fonction.

De la même manière, il faut déterminer ce que l'on souhaite compter précisément :

- si on s'intéresse au temps mis, certaines opérations prennent moins de temps que d'autre (par exemple une addition par rapport à une multiplication) mais est-ce vraiment important à l'échelle considérée ?
- si on s'intéresse à l'espace mémoire, doit-on considérer la taille précise en octets ou se contenter d'une estimation plus grossière ?

Mis à part dans certains cadres assez spécifiques, on se contente le plus souvent d'un ordre de grandeur pour ces complexités. Pour cela, on utilise des relations de comparaisons de suites et une échelle de grandeur usuelle pour les comparer.

IV.2.i La notation grand O

Définition IV.2 Soit $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ deux suites de nombres réels non nuls, on dit que la suite $(u_n)_n$ est dominée par $(v_n)_n$ lorsque la suite quotient $\left(\frac{u_n}{v_n}\right)_n$ est bornée.

On note alors $u_n = O(v_n)$.

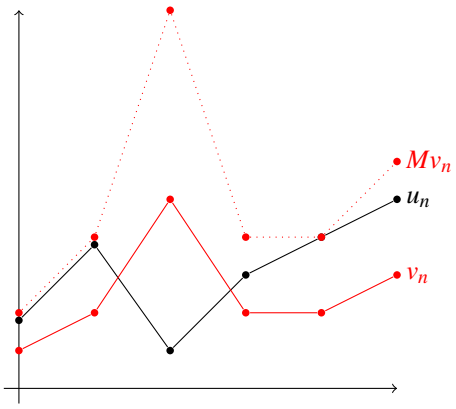
Cette dernière notation se lit *u_n est un grand O de v_n*.

Remarque C'est bien cette locution qu'il faut avoir en tête quand on pense aux grands O et il faut faire attention de ne pas considérer l'égalité en tant que telle sans s'assurer que ce l'on fait est licite. Quand on écrira par la suite $O(v_n)$ on signifiera *n'importe quelle suite qui soit un $O(v_n)$*.

Si $u_n = O(v_n)$, cela signifie qu'il existe un facteur $M > 0$ tel que pour tout entier n, on ait $-M|v_n| \leq u_n \leq M|v_n|$. Les variations de la suite $(u_n)_n$ sont ainsi entièrement contrôlées par les variations de $(v_n)_n$.

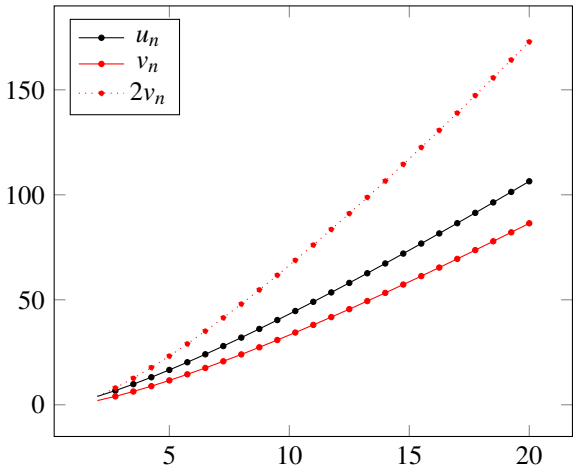
En informatique, on ne considère pour la complexité que des suites positives, ce qui permet de simplifier la relation : si $(u_n)_n, (v_n)_n$ sont des suites de réels strictement positifs, alors $u_n = O(v_n) \iff \exists M > 0, \forall n \in \mathbb{N}, u_n \leq Mv_n$. C'est le cadre dans lequel on se place implicitement dans la suite de ce document.

On peut visualiser graphiquement cette relation :

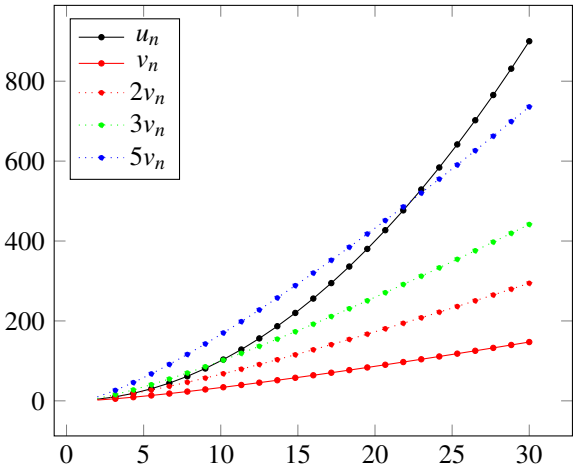


On a $u_n = O(v_n)$ si et seulement s'il est possible de multiplier les ordonnés de chaque point (n, v_n) par une constante afin que ces nouveaux points soient tous au-dessus des points (n, u_n). On peut voir que la courbe déduite des v_n enveloppe, à un facteur près, celle des u_n.
Remarque : *On a relié ici les valeurs des suites pour mieux mettre en valeur cette notion d'enveloppe.*

Cette relation est une notion **asymptotique** : elle n'a d'intérêt que lorsqu'on considère des rangs au voisinage de l'infini. En effet, pour un nombre fini de termes, il est toujours possible de trouver un tel M, mais pour un nombre infini, ce n'est pas le cas.



Ici, on compare asymptotiquement les suites $(u_n)_n$ et $(v_n)_n$ où pour $n \in \mathbb{N}$, $u_n = n + n\log_2 n$ et $v_n = n\log_2 n$. Pour simplifier la visualisation, on a tracé les fonctions correspondantes. On remarque qu'on a bien $n + n\log_2 n = O(n\log_2 n)$.



Par contre, si on compare les suites $(u_n)_n$ et $(v_n)_n$ où pour $n \in \mathbb{N}$, $u_n = n^2$ et $v_n = n\log_2 n$, on remarque que quelle que soit la valeur choisie pour M, il y aura un rang à partir duquel $u_n > Mv_n$.
Ici, $n^2 \neq O(n\log_2 n)$.

Remarque On a ici utilisé le logarithme en base 2, noté $\log_2$, qui est essentiel informatique : si $x = \log_2(n)$ alors $n = 2^x$ où x est un réel. On considère aussi $p = \lceil \log_2(n) \rceil$ qui est le plus petit entier égal ou supérieur à $\log_2(n)$. On parle de **partie entière supérieure** et on a alors $2^{p-1} < n \leq 2^p$. Cet entier p correspond alors au plus petit nombre de chiffre nécessaire pour pouvoir écrire n en binaire. On a $\lceil \log_2(n) \rceil = O(\log_2(n))$ et ainsi, le plus souvent, on ne considère pas la partie entière explicitement. De la même manière, $\log_2(n) = \frac{\ln n}{\ln 2} = O(\ln n)$.

Un cas important de grand O est celui des $O(1)$. Si $u_n = O(1)$, cela signifie que $(u_n)_{n\in\mathbb{N}}$ est une suite bornée.

IV.2.ii Échelle de comparaison

On rappelle les limites obtenues en mathématiques que l'on nomme **croissances comparées** :

$$\forall \alpha, \beta > 0, \lim_{n\to+\infty} \frac{(\ln n)^\alpha}{n^\beta} = 0$$

$$\forall \alpha \in \mathbb{R}, \forall \beta > 1, \lim_{n\to+\infty} \frac{n^\alpha}{\beta^n} = 0$$

Or, si $\frac{u_n}{v_n} \xrightarrow[n\to+\infty]{} 0$ *a fortiori* le quotient est borné et $u_n = O(v_n)$. Ainsi, on a les relations suivantes :

$$\forall \alpha, \beta > 0, (\log_2 n)^\alpha = O(n^\beta)$$

$$\forall \alpha \in \mathbb{R}, \forall \beta > 1, n^\alpha = O(\beta^n)$$

De plus, si $\alpha \geq \beta > 0$, $n^\beta = O(n^\alpha)$, $(\log_2 n)^\beta = O((\log_2 n)^\alpha)$ et $\beta^n = O(\alpha^n)$.

On se ramène souvent à des complexités qui sont des grand O de produits de ces suites.

IV.2.iii Ordre de grandeur et relation Θ

On vient de voir que $\log_2 n = O(n)$, mais on a également $\log_2 = O(n^2)$. Quand on cherche à caractériser la complexité par un grand O, on va souvent chercher le grand O le plus proche de la suite.

Il est possible de définir cela précisément en considérant des suites qui sont chacune des grand O l'une de l'autre.

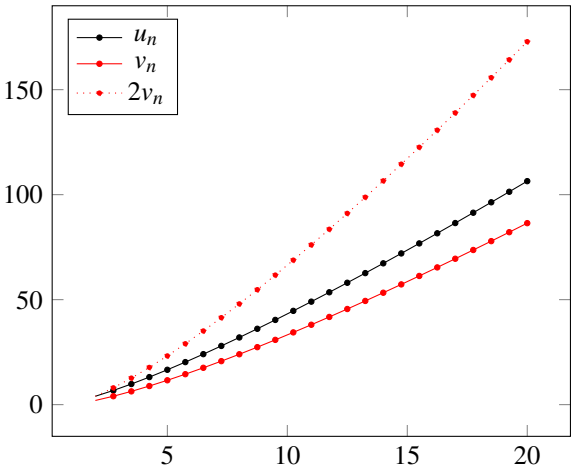
Par exemple, on a vu que $n \log_2 n + n = O(n \log_2 n)$, mais on a également $n \log_2 n = O(n + n \log_2 n)$.

Quand $u_n = O(v_n)$ et $v_n = O(u_n)$, on note $u_n = \Theta(v_n)$ qui est une relation symétrique qui correspond à la notion *avoir le même ordre de grandeur*. Très souvent, lorsque l'on parle de complexité, on utilise des grand O quand, en fait, on exprime des Θ. Par exemple, l'accès à un élément dans un tableau est en $O(1)$ et il ne serait pas précis de dire que c'est en $O(n)$ **même si c'est parfaitement correct**.

On peut visualiser cette relation Θ en considérant qu'il existe ainsi $M, M' > 0$ tels que $u_n \leq M v_n$ et $v_n \leq M' u_n$. Mais on a alors

$$1/M' v_n \leq u_n \leq M v_n$$

Ainsi, $u_n = \Theta(v_n)$ signifie qu'on peut encadrer $(u_n)_n$ entre deux multiples de $(v_n)_n$.



En reprenant la figure précédente, on observe visuellement

$$n \log_2 n \leq n \log_2 n + n \leq 2n \log_2 n$$

Avoir $u_n = \Theta(v_n)$ signifie donc que u_n évolue entre deux guides suivant les variations de v_n.

IV.2.iv Opérations sur les grands O

Si $u_n = O(w_n)$ et $v_n = O(w_n)$ alors $u_n + v_n = O(w_n)$. Ainsi, des grand O de même ordre s'ajoutent.

> **Remarque** Comme on l'a vu précédemment, un grand O n'est pas très précis, et il est possible que par ajout on puisse obtenir un meilleur grand O. Par exemple : $n = O(n)$ et $\log_2 n - n = O(n)$ mais $n + \log_2 n - n = \log_2 n = O(n)$. Comme on ne considère ici que des suites strictement positifs, ce phénomène de compensation n'aura pas lieu.

Si $u_n = O(v_n)$ et w_n est une autre suite de réels strictement positifs, alors $u_n w_n = O(v_n w_n)$. On en déduit ainsi un principe qui nous sera utile par la suite $nO(1) = O(n)$.

IV.3 Complexités en temps classiques

On parle ici de complexité par raccourci pour parler de complexité dans le pire des cas en temps.

IV.3.i Complexité constante

On dit qu'un algorithme a une complexité constante quand $C_n^t = O(1)$. Il existe ainsi une constante M telle que le temps pris par l'algorithme **sur une entrée quelconque** soit inférieur à M.

De nombreuses opérations sont en temps constant sur les structures de données usuelles. Parmi celles-ci, citons-en deux essentielles :

- accéder à une case d'indice quelconque dans un tableau
- accéder à la tête ou à la queue d'une liste chaînée

Les algorithmes ou opérations en temps constant jouent un rôle primordiale dans l'analyse de la complexité d'algorithmes, comme on le verra dans la partie suivante, car elles permettent de se concentrer sur les répétitions de ces opérations pour déterminer la complexité : une boucle qui se répète n fois et n'effectue que des opérations en temps constant dans son corps sera de complexité $nO(1) = O(n)$.

IV.3.ii Complexité linéaire

On dit qu'un algorithme a une complexité linéaire quand $C_n^t = O(n)$.

Cette complexité correspond à un traitement de temps constant sur chaque élément d'une entrée de taille n. C'est le cas de la recherche d'un élément dans un tableau ou de la recherche de son maximum.

Pour la recherche linéaire d'un élément, correspondant par exemple au programme ci-dessous, le pire cas correspond à ne pas avoir `x` dans `tableau` ce qui oblige à effectuer toutes les itérations. On a bien une complexité temporelle en pire cas de $O(n)$.

OCaml

C

```c
int recherche(int *tableau, int taille, int x)
{
    /* renvoie le plus petit indice i tel que tableau[i] = x
       ou -1 si x n'est pas dans le tableau */
    for(int i = 0; i < taille; i++)
    {
        if (tableau[i] == x)
            return i;
    }

    return -1;
}
```

IV.3.iii Complexité quadratique, polynomiale

On dit qu'un algorithme a une complexité quadratique quand $C_n^t = O(n^2)$. Par extension, on dit qu'il a une complexité polynomiale quand il existe $k \in \mathbb{N}$ tel que $C_n^t = O(n^k)$. Par extension, on parle parfois de complexité polynomiale pour des complexité plus précise en $O(n^\alpha)$ où α est un réel strictement positif.

L'exemple classique d'un algorithme quadratique est celui dû à un double parcours d'un tableau. On reprend ici l'algorithme de tri par sélection vu dans la partie Exemple du tri par sélection.

Afin d'analyser sa complexité, on procède fonction par fonction pour un tableau de taille n :

- `echange` est en temps constant. $O(1)$
- `indice_minimum` réalise un parcours du tableau et effectue des opérations en temps constant à chaque étape. La complexité est donc linéaire. $O(n)$
- `tri_par_selection` réalise également un parcours du tableau mais à chaque étape, on appelle `indice_minimum` qui est en $O(n)$, la complexité est donc en $nO(n) = O(n^2)$: elle est quadratique.

IV.3.iv Complexité logarithmique

On dit qu'un algorithme a une complexité logarithmique quand $C_n^t = O(\log_2 n)$.

Pour illustrer cette complexité, on reprend l'algorithme de recherche dichotomique vu dans la partie Exemple de la recherche dichotomique.

Chaque opération effectuée étant en temps constant, la complexité de cet algorithme correspond au nombre d'itérations, soit ici au nombre d'appels récursifs.

Si on considère un sous-tableau de $n = j - i + 1$ éléments lors de l'appel, un appel récursif se fera nécessairement sur un sous-tableau de $\lfloor n/2 \rfloor$ éléments. Ainsi, si $2^{k-1} < n \leq 2^k$, l'algorithme effectue moins de k

itérations. En passant au logarithme, on a donc $k-1 < \log_2 n \leq k$. Donc, le nombre d'itérations est en $O(\log_2 n)$ et c'est ainsi la complexité de l'algorithme.

■ **Note 2** Esquisser dès maintenant le lien entre longueur d'une branche dans un arbre de décision et complexité logarithmique ?

■

IV.3.v Complexité quasi-linéaire

On dit qu'un algorithme a une complexité quasi-linéaire quand $C_n^t = O(n \log_2 n)$. C'est le cas de la plupart des algorithmes efficaces de tri de n éléments. On peut même montrer qu'il s'agit de la complexité optimale.

Comme de nombreux algorithmes commencent par effectuer un tri avant d'effectuer un traitement linéaire, on retrouve des algorithmes quasi-linéaire par simple utilisation de ce tri.

IV.3.vi Complexité exponentielle

On dit qu'un algorithme a une complexité exponentielle quand $C_n^t = O(a^n)$ pour $a > 0$.

Un exemple fondamental d'un tel algorithme est celui de l'énumération de données, par exemple pour chercher une solution par force brute. En effet, il y a 2^n entrées codées sur n bits et un algorithme cherchant une solution ainsi parmi ces entrées aura une complexité en $O(2^n)$.

IV.3.vii Estimation de l'impact des complexités sur le temps

Afin de mesurer l'impact d'une complexité, on va considérer un algorithme qui s'exécute en 1 seconde sur un entrée de taille n, et on va calculer combien de temps prendrait ce même algorithme sur une entrée de taille $10n$.

Pour simplifier, on considère à chaque fois que C_n^t correspond exactement à l'ordre du grand O.

Complexité	Temps pour 10n	Temps pour 100n
1	1s	1s
$\log_2 n$	1,003s	1,007s
n	10s	1m40s
$n \log_2 n$	14,7s	3m13s
n^2	1m40s	2h46m40s
2^n	10^{19} années	10^{289} années.

Remarque Pour déterminer ces valeurs, on a considéré une unité de mesure de 1000ms afin d'en déduire une valeur de n.

Ainsi, si $\log_2 n = 1000$ on a $n = 2^{1000}$. Bien sûr, ici, ce nombre 2^{1000} n'est pas réaliste. Dans un contexte de mémoire finie, une complexité logarithmique est identifiable à une complexité constante. Cela justifie la terminologie quasi-linéaire.

Si $n \log_2 n = 1000$ alors $n \approx 140,2$. Or, $1402 \log_2 1402 \approx 14700ms$.

Si $2^n = 1000$, alors $n \approx 10$. Or $2^{100} \approx 10^{30}$.

IV.4 Calculer des complexités

Deux principes fondamentaux pour calculer des complexités :

- Si on effectue deux passes successives chacune en $O(u_n)$ alors la complexité globale est en $O(u_n)$. Il ne s'agit que de reformuler l'addition des grand O. Quand on a deux passes de complexité différente, il suffit d'utiliser la plus grande complexité. Par exemple, un algorithme qui commence par un tri en $O(n \log_2 n)$ et qui effectue ensuite un traitement en $O(n)$ sera de complexité globale $O(n \log_2 n)$ car le traitement est également en $O(n \log_2 n)$.
- Si on effectue u_n itérations et que chaque itération est en $O(v_n)$ alors l'algorithme a une complexité de $O(u_n v_n)$. Cela permet de compter le nombre de boucles imbriquées et de se contenter de regarder ce qui se passe dans le corps des boucles.

IV.5 Complexité à plusieurs paramètres

Jusqu'ici on a considéré des entrées dépendant d'un unique paramètre n, mais il est possible d'avoir des données dépendant de plusieurs paramètres.

On adapte directement la notation des grands O : si $(u_{n,p})$ et $(v_{n,p})$ sont deux suites de réels non nuls dépendant de deux paramètres, on note toujours $u_{n,p} = O(v_{n,p})$ quand le quotient est borné.

IV.5.i Données multidimensionnelles

Le cas le plus usuel de complexité dépendant de plusieurs paramètres est celui des données multidimensionnelle comme une image.

Si on considère une opération effectuant un traitement en temps constant sur chaque pixel d'une image de $w \times h$ pixels, cette opération aura une complexité en $O(wh)$. On ne peut plus parler de complexité linéaire ou quadratique ici car cela dépend d'une éventuelle relation entre w et h : si on ne travaille que sur des images de taille $1 \times h$ alors la complexité est $O(h)$, mais on ne travaille que sur des images carrées, donc pour lesquelles $w = h$, la complexité est $O(h^2)$.

Plus généralement, si on considère des données organisées dans des tableaux imbriqués, on effectuera un traitement sur chaque donnée à l'aide de boucles imbriquées non conditionnelles. La complexité sera alors celle du corps de boucles multipliée par le produit du nombre d'itérations de chaque boucle.

IV.5.ii Compromis entre paramètres

Dans certains cas, en particulier pour les graphes, on peut effectuer des traitements successifs dont la complexité ne s'exprime pas en fonction du même paramètre. Imaginons par exemple un programme ayant la structure suivante :

OCaml

C
```c
for (int i = 0; i<n; i++)
{
    /* corps de boucle en O(1) */
}

for (int j = 0; j<p; j++)
{
    /* corps de boucle en O(1) */
}
```

La complexité de la première boucle est en $O(n)$ et celle de la deuxième en $O(p)$. La complexité globale est en $O(n+p)$ car $n \leq n+p$ et $p \leq n+p$.

IV.6 Complexité en moyenne

On reprend ici les notations de la partie Complexité dans le pire des cas.

Définition IV.3 Lorsque pour tout $n \in \mathbb{N}$, I_n est fini, on appelle :
- **complexité temporelle en moyenne** la suite $(C_n^{t,m}) = \frac{1}{|I_n|}\sum_{e\in I_n} t(e)$.
- **complexité spatiale en moyenne** la suite $(C_n^{s,m}) = \frac{1}{|I_n|}\sum_{e\in I_n} s(e)$.

On peut étendre cette définition à un cadre infini en considérant une distribution de probabilité sur I_n et T_n la variable aléatoire associée à t sur I_n. Si T_n est d'espérance finie, on pourra parler de complexité en moyenne pour la suite des $E(T_n)$. Concrètement, on considère alors une fonction $p_n : I_n \to [0,1]$ telle que $\sum_{e\in I_n} p(e) = 1$ et, lorsque la somme est définie, on note ainsi

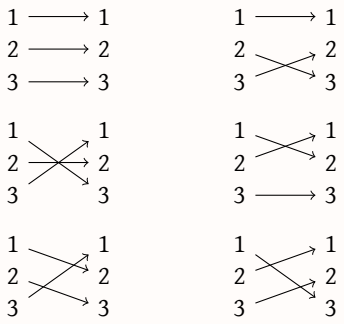
$$C_n^{t,m} = \sum_{e\in I_n} p(e)t(e)$$

$$C_n^{s,m} = \sum_{e\in I_n} p(e)s(e)$$

Un exemple usuel de calcul de complexité en moyenne est celui des tris. En effet, même si les entrées de taille n sont infinies, on peut considérer qu'un tableau de valeurs deux à deux distinctes est l'image par une permutation du tableau triée. Si le tableau est de taille n, on aura ainsi $n!$ permutations ce qui permet, du moment que l'algorithme de tri considéré ne dépend que cette permutation, de calculer la complexité en moyenne sur l'ensemble des permutations.

Remarque Les permutations d'un ensemble sont les applications bijectives de cet ensemble dans lui-même. Si l'ensemble contient n éléments, il y a $n!$ permutations.

Par exemple, les six permutations sur l'ensemble $\{1,2,3\}$ correspondent aux diagrammes sagittaires suivants :



Ces six permutations correspondant elles-mêmes, de gauche à droite et de haut en bas, aux tableaux `{1,2,3}`, `{1,3,2}`, `{3,2,1}`, `{2,1,3}`, `{2,3,1}` et `{3,1,2}`.

IV.6.i Exemple de calcul de complexité temporelle en moyenne

On considère la recherche linéaire vue dans la partie Complexité linéaire. L'ensemble des entrées est ici infini, on va donc supposer pour faire le calcul qu'on ne considère que des tableaux de valeurs deux à deux distinctes et qu'on recherche un élément présent dans le tableau, chaque élément étant équiprobable.

Si on recherche le i-ème élément du tableau, l'algorithme effectue i itérations avant d'y accéder et de renvoyer son indice. Ainsi, le temps pour cet entrée est de iC où C est le coût d'une itération.

La complexité temporelle en moyenne est alors $C_n^{t,m} = \sum_{i=1}^{n} \frac{1}{n} iC = \frac{(n+1)C}{2} = O(n)$. On retrouve ici la même complexité que la complexité dans le pire des cas. La sortie prématurée de la boucle n'a donc aucune influence sur la complexité.

IV.7 Complexité amortie

Dans le cadre de l'étude des structures de données, il est fréquent de considérer non pas la complexité dans le pire des cas d'une opération mais celle d'une succession d'opérations divisée par le nombre d'opérations effectuées. Ainsi, on peut très bien avoir une opération ponctuellement plus coûteuse que les autres, mais en procédant ainsi on lisse le surcoût sur l'ensemble des opérations. On parle alors de **complexité amortie**.

L'étude de la complexité amortie est traitée dans le chapitre TODO.

IV.8 Pertinence de la complexité spatiale

Même si la complexité temporelle est le plus souvent celle qui est importante à calculer, certains algorithmes ont une complexité temporelle faible mais en contrepartie une complexité spatiale élevée. On parle alors de compromis temps-mémoire.

Un exemple classique d'un tel compromis est celui de la programmation dynamique où on passe d'une complexité temporelle exponentielle à une complexité temporelle polynomiale en stockant des valeurs intermédiaires pour ne pas les recalculer. En procédant ainsi, on passe d'une complexité spatiale constante à polynomiale.

Cela est illustré dans le programme suivant qui permet de déterminer le n-ième terme de la suite de Fibonacci, ce qui n'a pas d'intérêt informatique mais est caractéristique de récurrence que l'on résoudra par la programmation dynamique.

OCaml

```ocaml
(* Fibonacci exponentiel *)
let rec fibo n =
    if n = 0
    then 0
    else if n = 1
    then 1
    else fibo (n-1) + fibo (n-2)

(* Fibonacci linéaire *)
let fibo n =
    let prec = Array.make (n+1) 0 in
    prec.(0) <- 0;
    prec.(1) <- 1;
    for i = 2 to n do
        prec.(i) <- prec.(i-1) + prec.(i-2)
    done;
    prec.(n)
```

C

V Exercices

Exercice 1 On considère le programme suivant :

OCaml
```
let multiplication x y c =
    let m = ref 0 in
    let vy = ref y in
    let vx = ref x in
    while !vy <> 0 do
        m := !m + !vx * (!vy mod c);
        vx := !vx * c;
        vy := !vy / c
    done;
    !m
```

C
```
int multiplication(int x, int y, int c)
{
    int m = 0;
    while (y != 0)
    {
        m = m + x * (y % c);
        x = x * c;
        y = y / c;
    }
    return m;
}
```

Python
```
def multiplication(x, y, c) :
    m = 0
    while y  != 0 :
        m = m + x * (y % c)
        x = x * c
        y = y // c
    return m
```

On suppose que $c \geq 2$.

1. Montrer que cet algorithme termine .
2. Montrer que pour $x, y \in \mathbb{N}$, il renvoie le produit xy.
3. Réecrire ce programme en récursif. Peut-on déduire des preuves précédentes que l'algorithme termine et qu'il est correct.

■ **Preuve**

1. Comme $c \geq 2$, on a $y/c < y$ ainsi y est un variant de boucle.
2. On va commencer par noter x_0 et y_0 les valeurs initiales respectives de x et de y. On va également tenir compte du nombre d'itérations, ce qui revient à rajouter un compteur `i` au programme. On va également noter $y_0 = \sum_{j=0}^{n} a_j c^j$.
   - $y = \left\lfloor \frac{y_0}{c^i} \right\rfloor = \sum_{j=i}^{n} a_j c^{j-i}$ et $x = x_0 c^i$ sont des invariants directement validés.
   - On va maintenant prouver l'invariant $I(i, m) := m = x_0 \sum_{j=0}^{i-1} a_j c^j$.
     - ⋆ **Initialisation** avant la première itération, on a $I(0, 0, x_0, y_0)$ qui est vérifié car $0 = 0$.
     - ⋆ **Hérédité** si l'invariant est vérifié au début de la ième itération on a $m + x(y\%c) = m + x_0 a_i c^i = m + x_0 \sum_{j=0}^{i} a_j c^j$ donc $I(i+1, m + x(y\%c))$ est vérifié.

   Ainsi, en sortie de boucle, on a $m = x_0 \sum_{j=0}^{n} a_j c^j = x_0 y_0$.
3. En récursif, cela ne change pas la validité des preuves précédentes mais il faudrait présenter l'invariant différemment.

OCaml
```
let rec multiplication x y c =
    if y = 0
    then 0
    else x * (y mod c) + multiplication (x*c) (y/c) c
```

C
```
int multiplication(int x, int y, int c)
{
  if (y == 0) return 0;
  return x * (y % c) + multiplication(x * c, y / c, c);
}
```

■

Exercice 2 On considère un polynôme $P = \sum_{i=0}^{n} c_i X^i$ représenté comme un tableau de $n+1$ coefficients tel que pour tout $i \in [\![0, n]\!]$, $P[i] = c_i$.

On veut calculer $P(a)$ pour une valeur a. Pour simplifier, on va considérer ici que les valeurs sont toutes entières.

OCaml
```ocaml
let rec horner p a =
    let v = ref 0 in
    for i = Array.length p - 1 downto 0 do
        v := a * !v + p.(i)
    done;
    !v
```

C
```c
int horner(const int *P, int n, int a)
{
    int v = 0;
    for (int i = n; i >= 0; i--)
    {
        v = a * v + P[i];
    }
    return v;
}
```

Python
```python
int horner(P, a) :
    v = 0
    # rappel reversed(P) parcourt P
    # en sens inverse
    for c in reversed(P) :
        v = a * v + c
    return v
```

1. Montrer que ce programme est correct.
2. Combien effectue-t-il de multiplications et d'additions ?
3. Comparer avec l'algorithme obtenu en ajoutant chaque $c_i a^i$ pour i croissant et en maintenant une variable pour a^i.

■ **Preuve**

1. On considère l'invariant $I(v,i) := v = c_{i+1} + c_{i+2}a + \cdots + c_n a^{n-i-1}$
   - **Initialisation** avant la première itération on a $I(0,n)$ vérifié car $v = 0 = c_{n+1}$.
   - **Hérédité** si l'invariant est vérifié en début d'itération, on passe à $I(av + c_i, i-1)$ en fin d'itération or, si $v = c_{i+1} + \cdots + c_n a^{n-i-1}$ on a bien $av + c_i = c_i + c_{i+1}a + \cdots + c_n a^{n-i}$ et donc l'invariant est vérifié en fin d'itération.

   En sortie de boucle, on a alors $I(v,-1)$ donc $v = c_0 + c_1 a + \cdots + c_n a^n = P(a)$.
2. On effectue $n+1$ itérations et à chaque itérations une multiplication et une addition, donc $n+1$ multiplications et $n+1$ additions.
3. Pour cet algorithme, on devrait faire une multiplication de plus à chaque itération pour maintenir a^i.

■

Exercice 3 On considère le problème du drapeau hollandais : étant donné un tableau t et un indice i, on note `p=t[i]`, on cherche à permuter les éléments de `t` de sorte qu'il y ait trois zones dans le tableau : les éléments $< p$, les éléments $= p$ puis les éléments $> p$.

Ainsi, si on considère `t = [5, 2, 3, 5, 1, 4]` et `p=t[2] =3` on pourra obtenir `[2, 1, 3, 5, 5, 4]` à l'issue de cet algorithme.

1. Écrire un programme résolvant ce problème en temps linéaire.
2. Prouver sa correction.

■ **Preuve**

Le programme :

C
```c
void echange(int *t, int i, int j)
{
    int temp = t[i];
    t[i] = t[j];
    t[j] = temp;
}

void drapeau(int *t, int nb, int i)
{
    int v = t[i];
    int l = 0;
    int c = nb-1;
    int r = nb-1;

    while (l <= c)
```

```
    {
        int w = t[l];
        if (w < v)
        {
            l = l+1;
        }
        else if (w == v)
        {
            echange(t, l, c);
            c = c-1;
        }
        else
        {
            echange(t, l, c);
            echange(t, c, r);
            c = c-1;
            r = r-1;
        }
    }
}
```

L'idée de cet algorithme est d'avoir trois indices l, c et r (pour *left*, *center* et *right*) qui délimitent trois zones :

- celle des indices 0 à $l-1$ qui contient des valeurs $< v$
- celle des indices $c+1$ à r qui contient des valeurs égales à v
- celle des indices $r+1$ à $nb-1$ qui contient des valeurs $> v$.

Au départ, ces trois zones sont triviales et au fur à mesure de l'algorithme, elles augmentent jusqu'à couvrir l'ensemble du tableau. Comme soit l croit soit c décroit a chaque itération, on a la distance $c-l$ qui est un variant de boucle et on effectue exactement $|t|$ itérations.

On va montrer que le découpage des zones est effectivement un invariant :

- **Initialisation** au départ, comme on l'a vu, ces trois zones sont vides donc l'invariant est trivialement vérifié
- **Hérédité** si l'invariant est vrai au début d'une itération, on va faire trois cas selon la position de `w` par rapport à `v` :
  - ⋆ Si `w < v` alors la zone des indices de 0 à l contient toujours des valeurs $< v$ et les autres zones n'ont pas bougé, l'invariant est encore vérifié.
  - ⋆ Si `w = v` alors on place cette valeur `w` au début de la zone centrale en faisant l'échange puis en décalant `c` vers la gauche. Les autres zones ne sont pas touchées, donc l'invariant est encore vérifié.
  - ⋆ Si `w > v` alors on veut le placer en tête de la zone de droite, mais l'élément qui s'y trouve est peut-être une valeur de la zone centrale si celle-ci est non triviale. On place alors cette valeur là en début de la zone centrale ce qui revient à la décaler d'un cran vers la gauche. La valeur `w` peut alors etre échangée. La première zone n'a pas bougé et les autres zones vérifient encore les conditions voulues, l'invariant est donc vérifié. !TODO(On devrait être plus précis ici)

On a donc validé l'invariant. En sortie de boucle, on a $x = y + 1 > y$ et donc les trois zones couvrent l'intégralité du tableau.

■