# Récapitulatif MP2I

## Révisions pour les vacances

Guillaume Rousseau
MP2I Lycée Pierre de Fermat
guillaume.rousseau@ens-lyon.fr

2022-2023

Ce document est un guide de révisions pour les vacances de décembre. Pour chaque chapitre, il fixe les notions importantes à connaître, et les éléments indispensable, à connaître absolument sur le bout des doigts.

On donne ensuite une liste d'algorithmes à connaître aussi comme votre poche. Pour chacun, on donne l'idée ainsi que le pseudo-code. Dans le pseudo-code sont indiqués des invariants de boucle qui permettent de montrer la correction. Entraînez-vous à retrouver les invariants à partir des algorithmes, **mais aussi à faire l'inverse**.

# 1 Chapitre 1 : Premiers programmes en C

## Notions à Connaître

— Bases du C, compilation

— Syntaxe des boucles for / while

— Syntaxe des fonctions

## A savoir faire

**Preuve de terminaison d'un algorithme :**

— Pour une boucle for : termine automatiquement

— Pour une boucle while : soit la transformer en boucle for, soit trouver un **variant de boucle**

Méthode de variant de boucle : Pour montrer qu'une boucle termine, on exhibe une quantité entière, positive, et strictement décroissante à chaque passage de boucle. Cette quantité fait intervenir une où plusieurs variables de l'algorithme.

**Preuve de correction d'un algorithme :**

1. On commence par identifier ce qu'est la correction de l'algorithme, à l'aide d'une formule du style "l'algorithme renvoie ...", ou bien "à la fin de l'algorithme, ...".
2. On identifie un invariant de boucle utile pour la correction. En général, cet invariant ressemble beaucoup à la formule trouvée à l'étape précédente, et la généralise en faisant intervenir les variables qui sont susceptibles de changer à chaque passage de boucle.
3. On montre que la propriété précédente est bien un IdB : Elle est vraie en entrée de boucle, et elle se conserve, i.e. si elle est vraie au début d'un passage, alors elle est vraie à la fin de ce passage.
4. Par propriété, un IdB est vrai en sortie de boucle, et donc on regarde ce que l'on peut dire à ce moment là. Pour cela, on utilise la condition de boucle : pour un while, la condition est fausse lorsque l'on sort. En injectant les nouvelles informations dans l'IdB, on doit pouvoir montrer la correction.

L'invariant choisi à l'étape 2 doit évidemment être pris en pensant aux étapes 3 et 4. Il peut être constitué de plusieurs petites propriétés, par exemple "bla ET bli ET blo" qui exprime plusieurs choses qui restent vraies.

# 2 Chapitre 2 : Encodage des nombres

## Notions à maîtriser

— Écriture en base $B$

— Montrer l'existence de l'écriture en base $B$ de tout nombre

— Unicité de l'écriture si l'on rajoute une condition (longueur fixée, pas de 0 en trop)

## A savoir faire

— Lire et écrire des nombres dans n'importe quelle base

# 3 Chapitre 3 : Pointeurs, tableaux, structures

## Notions à maîtriser

— Pointeurs, tableaux

— Pile et tas

## A savoir faire

— Utilisation de malloc et free

— Types struct en C

— Lecteur et écriture dans des fichiers

— Compilation avec plusieurs fichiers sources

— Exécuter du code à la main en représentant la mémoire

# 4 Chapitre 4 : Complexité

## Notions à maîtriser

— Notations de Landau

## A savoir faire

Analyse de la complexité d'un algorithme

— Boucle for : compter le nombre de passages

— Boucle while : trouver un variant de boucle pour borner le nombre de passages

— Boucles imbriquées, ou appels de fonctions dans une boucle : on somme tous les coûts sur chaque passage.

# 5 Chapitre 5 : Structures de données

## Notions à maîtriser

— Structure de données abstraite / concrète (et la différence entre les deux)

— Pile, file, implémentations par tableaux et par listes chaînées.

— Tableaux redimensionnables

## A savoir faire

— Implémenter une pile ou une file par tableaux et par liste chaînée, et être capable de retrouver le code pour les listes chaînées en faisant des dessins des maillons ajoutés / supprimés.

Vous n'avez pas besoin de savoir faire des preuves de complexité amortie comme celles vues pour les files à deux piles ou bien les tableaux redimensionnables, mais vous devez connaître l'existence de ce type d'analyse de complexité.

# 6 Algorithmes

On présente quelques algorithmes classiques. Pour les algorithmes ayant plusieurs boucles imbriquées, on donne des invariants pour toutes les boucles. Dans une preuve, on peut faire d'abord la preuve d'invariant de la boucle intérieure, puis utiliser ce résultat comme un lemme pour faire la preuve de la boucle extérieure.

Un exemple détaillé de preuve est donné pour le tri par insertion.

## Tri par insertion

Le principe est de construire au début du tableau une **zone triée de plus en plus grande**. A chaque étape, on vient insérer un élément de plus dans la zone triée, en le décalant petit à petit jusqu'à ce qu'il ait atteint sa place.

**Algorithme 1 :** Tri par insertion

**Entrée(s) :** $T$ tableau d'éléments ordonnables, de taille $n \in \mathbb{N}$
**Sortie(s) :** $T$ est trié
// Invariant: $T[0..i[$ est trié
1 **pour** $i = 1$ **à** $n - 1$ **faire**
  // But: décaler $T[i]$ à sa place dans $T[0..i[$
2  $j \leftarrow i$ ;
  // Invariant: $T[0..j-1]$ et $T[j..i]$ sont triés, et $T[j+1] \geq T[j-1]$
3  **tant que** $j > 0$ ***et*** $T[j] < T[j-1]$ **faire**
4  Échanger $T[j]$ et $T[j-1]$;
5  $j \leftarrow j - 1$;
// En sortie, $i = n$ donc $T[0..n[$ est trié, i.e. $T$ est trié.

### Principe

**Terminaison** La boucle intérieure contient un variant de boucle : $j$. En effet, $j$ est entier, diminue strictement de 1 à chaque passage de boucle, et au début de chaque passage , $j > 0$ donc $j \geq 0$ à tout instant.

La boucle extérieure est une boucle pour, donc termine aussi. Finalement, l'algorithme termine.

**Complexité** La boucle intérieure exécute au plus $i$ passages, chacun en temps constant, et la boucle extérieure exécute $n$ passages, faisant varier $i$ de 1 à $n-1$. Donc la complexité est en $\mathcal{O}(\sum_{i=1}^{n-1} i) = \mathcal{O}(n^2)$.

**Correction** Pour faciliter la preuve de correction, on peut réécrire la boucle for avec une boucle while :

```
1 i ← 1;
2 tant que i < n faire
3 |   j ← i ;
4 |   tant que j > 0 et T[j] < T[j − 1] faire
5 |   |   Échanger T[j] et T[j − 1];
6 |   |   j ← j − 1;
7 |   i ← i + 1;
```

Dire que l'algorithme est correct, c'est dire que le tableau $T$ est trié en fin d'exécution. Il y a plusieurs manières d'exprimer cela. Pour cet algorithme, il est plus simple de l'exprimer par :

$$\text{A la fin de l'algorithme}, \forall k \in [\![0, n-2]\!], T[k] \leq T[k+1]$$

Cette propriété va être simple à montrer car le tri par insertion construit à gauche du tableau une partie triée grandissante. En fait, on a comme invariant de boucle :

$$\forall k \in [\![0, i-2]\!], T[k] \leq T[k+1]$$

Autrement dit, les $i-1$ premières cases sont triées entre elles. Afin de montrer cet invariant de boucle, on s'intéresse à la boucle intérieure. Cette boucle effectue l'insertion d'un élément dans la partie triée. Par exemple, si $i = 4$ et qu'on est dans la situation suivante au début d'un passage de la boucle extérieure :

| 3 | 8 | 12 | 14 | 9 | 1 | 13 |
|---|---|---|---|---|---|---|

On veut insérer 9 dans la partie gauche du tableau. Après l'exécution de la boucle intérieure, $T$ sera alors dans l'état suivant :

| 3 | 8 | 9 | 12 | 14 | 1 | 13 |
|---|---|---|---|---|---|---|

On peut donc postuler la propriété suivante pour la boucle intérieure :

Si $T[0..i-1]$ est trié au début de la boucle intérieure, alors $T[0..i]$ est trié après la boucle

Commençons par montrer cette propriété. Quand on insère, on décale la case à insérer à l'intérieur d'un tableau trié. Ainsi, le tableau reste trié si l'on ignore la case en cours d'insertion. On se place donc à un passage quelconque de la boucle extérieure, et on suppose que $T[0..i-1]$ est trié. Montrons l'invariant de boucle suivant :

$$T[0..j-1] \text{ et } T[j..i] \text{ sont triés, et } T[j+1] \geq T[j-1]$$

Montrons cet invariant :

— En entrée de la boucle intérieure, $j = i$, et $T[0..i-1]$ est trié, et $T[i..i]$ ne contient qu'une case donc est trié. Donc la propriété est vraie.

— Supposons la propriété vraie au début d'un passage de la boucle intérieure. Alors, $T[0..j-1]$ et $T[j..i]$ sont triés et $T[j+1] \geq T[j-1]$. Notons $j' = j-1$ la valeur de $j$ en fin de passage et $T'$ l'état de $T$ en fin de passage. $T$ et $T'$ sont égaux, excepté les cases $j$ et $j-1$ qui ont été échangées. De plus, comme on effectue un passage, $T[j] < T[j-1]$. Donc, $T'[j-1] = T[j] < T[j-1] = T'[j]$. Montrons que $T'[0..j'-1 = j-2]$ et $T'[j' = j-1..i]$ sont triés.

  — $T'[0..j-2]$ est trié car les cases $0..j-2$ n'ont pas été modifées et étaient déjà triées.

  — $T'[j-1..i]$ est trié car $T[j..i]$ l'était, donc les cases $j+1..i$ sont toujours triées, et de plus, $T'[j-1] < T'[j] = T[j-1] \leq T[j+1]$, donc tout $T[j..i]$ est trié.

  Enfin, il reste à montrer que $T'[j'+1 = j] \geq T'[j'-1 = j-2]$ On sait que $T'[j] = T[j-1] > T[j-2]$ car $T[0..j-1]$ est trié en début de passage.

  Finalement, la propriété reste vraie en fin de passage.

Le propriété est bien un invariant de boucle. En particulier, en fin de boucle, $j = 0$ ou $T[j-1] \leq T[j]$, et dans tous les cas, on a :

$$T[0..j] \text{ et } T[j..i] \text{ sont triés}$$

Autrement dit, $T[0..i]$ est trié. Donc, la boucle intérieure augmente de 1 la taille de la partie triée que l'on construit. On peut utiliser cette propriété pour montrer l'invariant de boucle suivant pour la boucle extérieure :

$$T[0..i-1] \text{ est trié}$$

En effet :

— En entrée de boucle, $i = 1$ et $T[0..i-1]$ est une case unique, et est donc trié

— Au début d'un passage de boucle, si $T[0..i-1]$ est trié, alors par propriété de la boucle intérieure, en fin de passage $T[0..i = i'-1]$ est trié.

Donc la propriété est un invariant de boucle, et en particulier quand on sort de la boucle extérieure $i = n$ et donc $T[0..n-1]$ est trié : l'algorithme est correct.

## Algorithme d'Euclide

Le principe est d'utiliser les deux formules suivantes du PGCD :

— Pour $x, y \in \mathbb{N}$ avec $(x, y) \neq (0, 0)$, on a $\mathbf{PGCD}(x, y) = \mathbf{PGCD}(y, x \mod y)$

— Pour $x \in \mathbb{N}^*$, $\mathbf{PGCD}(x, 0) = x$

**Algorithme 2 :** Algorithme d'Euclide

**Entrée(s) :** $a \in \mathbb{N}, b \in N$ avec $(a, b) \neq (0, 0)$
**Sortie(s) :** $\mathbf{PGCD}(a, b)$

```
1 x ← a ;
2 y ← b ;
  // Invariant: PGCD(x, y) = PGCD(a, b)
3 tant que y ≠ 0 faire
4 |   t ← x;
5 |   x ← y%x;
6 |   y ← t;
  // A la fin, y = 0 donc x = PGCD(x, y) = PGCD(a, b)
7 retourner x
```

## Exponentiation naïve

Étant donné $a, n \in \mathbb{N}$, on veut calculer $a^n$. La méthode naïve consiste à calculer $a \times a \times \cdots \times a$ en calculant les produits partiels : $1, a, a^2, a^3, \ldots, a^i, \ldots, a^n$.

**Algorithme 3 :** Exponentiation naïve

**Entrée(s) :** $a \in \mathbb{N}, n \in N$
**Sortie(s) :** $a^n$

$i \leftarrow 0$ ;
$res \leftarrow 1$ ;
// Invariant: $res = a^i$
**tant que** $i < n$ **faire**
  $res \leftarrow res \times a$;
  $i \leftarrow i + 1$;
// En sortie, $i = n$ donc $res = a^n$
**retourner** $res$

## Exponentiation rapide

Cet algorithme utilise la formule suivante : pour $a, n \in \mathbb{N}$ :

— Si $n$ est pair, $a^n = (a^2)^{\frac{n}{2}}$

— Si $n$ est impair, $a^n = a \times (a^2)^{\frac{n-1}{2}}$

Le but est donc de diviser $n$ par 2 en faisant sortir des puissances de $a$ lorsque $n$ est impair, jusqu'à avoir $n = 0$. Par exemple :

$$2^1 1 = 2 \times 4^5 = 2 \times 4 \times 16^2 = 2 \times 4 \times 256^1 = 2 \times 4 \times 256 \times 65536^0$$

**Algorithme 4 :** Exponentiation rapide

**Entrée(s) :** $a \in \mathbb{N}, n \in N$
**Sortie(s) :** $a^n$

$N \leftarrow n$ ;
$A \leftarrow a$ ;
$res \leftarrow 1$ ;
// Invariant: $a^n = A^n \times res$
**tant que** $N > 0$ **faire**
  **si** $N\%2 = 1$ **alors**
    $res \leftarrow res \times A$;
  $A \leftarrow A \times A$;
  $N \leftarrow \lfloor \frac{N}{2} \rfloor$;
// En sortie, $N = 0$ donc $a^n = res$
**retourner** $res$

## Lecture depuis la base $B$

Étant donné une base $B$ et un tableau $T$ de $l$ éléments de $[\![0, B-1]\!]$, on souhaite construire l'entier $n$ correspondant à $T$ lu en base $B$. Autrement dit, on doit calculer l'entier :

$$\overline{T[0]\dots T[l-1]}^B = \sum_{i=0}^{l-1} T[l-1-i]B^i$$

Nous allons donc **calculer les sommes partielles successives**. Il sera aussi utile de **garder en mémoire la puissance de $B$ actuelle**.

---

**Entrée(s) :** $T$ tableau de taille $l$, $B$ une base, tels que $T$ ne contient que des éléments de $[\![0, B-1]\!]$
**Sortie(s) :** $\overline{T[0]\dots T[l-1]}^B$

$S \leftarrow 0$;
$pB \leftarrow 1$;
$k \leftarrow 0$;
// Invariant: $pB = B^k$
// Invariant: $S = \sum_{i=0}^{k-1} T[l-1-i]B^i$
**tant que** $k < l$ **faire**
    $S \leftarrow S + T[l-1-k]B^k$;
    $k++$;
// A la fin, $k = l$ donc $S$ vaut exactement le résultat attendu

---

## Indice du maximum d'un tableau

Le principe de l'algorithme proposé est de regarder les éléments un par un, en gardant en mémoire **le plus grand vu jusque là**.

**Algorithme 5 :** Maximum d'un tableau

**Entrée(s) :** $T$ tableau de réels, de taille $n \in \mathbb{N}^*$
**Sortie(s) :** La plus grande valeur contenue dans $T$

$i_m \leftarrow 0$;
$i \leftarrow 1$ ;
// Invariant: $T[i_m]$ est le max de $T[0..i[$
**tant que** $i < n$ **faire**
    **si** $T[i] > T[i_m]$ **alors**
        $i_m \leftarrow i$;
    $i \leftarrow i + 1$;
// En sortie, $i = n$ donc $i_m$ est l'indice du max de $T$
**retourner** $i_m$

## Recherche linéaire dans un tableau

Le principe est le même que l'algorithme de recherche de max : on parcourt chaque case **tant qu'on a pas rencontré** $x$.

**Algorithme 6 :** Rercherche dans un tableau

**Entrée(s) :** $T$ tableau de taille $n \in \mathbb{N}$, $x$ élément à chercher
**Sortie(s) :** "Vrai" si $x$ est dans $T$, "Faux" sinon

```
1 i ← 0 ;
  // x ne figure pas dans T[0..i[
2 tant que i < n et T[i] ≠ x faire
3     i ← i + 1;
  // En sortie, soit i = n, soit T[i] = x. Si i = n, alors x ne figure pas
     dans T, et sinon alors T[i] = x
4 retourner i < n
```

## Recherche par dichotomie

Le principe de cet algorithme est de garder en mémoire **un intervalle de recherche** : au départ cet intervalle est le tableau tout entier, et à chaque étape, on regarde la case dont l'indice est au milieu de l'intervalle, et selon ce qu'on y voit on peut diviser la zone de recherche de moitié.

Attention, le principe de la dichotomie n'est pas "L'élément qu'on cherche est forcément dans l'intervalle", car il se peut que l'élément qu'on cherche n'est même pas dans le tableau. Il est plus correct de dire "**L'élément qu'on cherche ne peut pas être hors de l'intervalle**".

**Algorithme 7 :** Rercherche par dichotomie

**Entrée(s) :** $T$ tableau de taille $n \in \mathbb{N}$ trié, $x$ élément à chercher
**Sortie(s) :** "Vrai" si $x$ est dans $T$, "Faux" sinon

```
1  a ← 0 ;
2  b ← n − 1;
   // Invariant: x n'est ni dans T[0..a[ ni dans T]b..n[
3  tant que b ≥ a faire
4      m ← ⌊(a+b)/2⌋;
5      si T[m] = x alors
6          retourner "Vrai"
7      si T[m] < x alors
8          a ← m + 1 ;
9      si T[m] > x alors
10         b ← m − 1
   // En sortie, a > b donc T[0..a[ et T[b..n[ couvrent tout le tableau, et
      donc x n'est pas dans T.
11 retourner "Faux"
```

## Tri par sélection

On propose une variante de l'algorithme vu en cours, où l'on sélectionne le min à chaque étape. Le principe est donc de chercher l'élément minimal, de le mettre au début, puis de chercher l'élément minimal dans ceux qui restent, de le mettre à la deuxième place, et ainsi de suite.

Ainsi, à chaque étape, **toute la première partie du tableau contient les plus petits éléments**. De plus, l'algorithme ne fera que des échanges et donc **aucun élément ne sera ajouté ou supprimé**.

**Algorithme 8 :** Tri par sélection

**Entrée(s) :** $T$ tableau d'éléments ordonnables, de taille $n \in \mathbb{N}$
**Sortie(s) :** $T$ est trié
// Invariant: $T[0..i[$ est trié, et pour tout $k \in [\![0, i-1]\!]$, pour tout $l \in [\![i, n-1]\!]$, $T[k] < T[l]$
// Invariant: Les éléments de $T$ sont les même qu'en entrée
1 **pour** $i = 0$ **à** $n-1$ **faire**
2  $j \leftarrow$ indice du minimum de $T[i], \dots, T[n-1]$;
3  Échanger $T[i]$ et $T[j]$;

## Tester l'égalité entre deux piles

L'idée est de dépiler les deux piles, jusqu'à trouver des éléments différents, ou bien jusqu'à ce qu'une des deux piles soit vide. **On sauvegarde tous les éléments dépilés dans une pile**, puis avant de renvoyer le résultat, on utilise la sauvegarde pour restaurer les deux piles.

**Algorithme 9 :** Égalité de piles

**Entrée(s) :** $P, Q$ deux piles
**Sortie(s) :** Vrai si $P = Q$, Faux sinon
1 $S \leftarrow$ **pile_vide**()// Sauvegarde
// Invariant :Si l'on renverse $S$ sur $P$, on obtient la pile $P$ de départ. Idem pour $Q$.
2 $egal \leftarrow$ Vrai;
// Invariant: Si $egal$ vaut Faux alors $P \neq Q$
3 **tant que** $P$ ***non vide et*** $Q$ ***non vide et*** $egal =$ ***Vrai*** **faire**
4  $x \leftarrow P$.**depiler**();
5  $y \leftarrow P$.**depiler**();
6  **si** $x = y$ **alors**
7  $S$.**empiler**($x$);
8  **sinon**
9  $egal \leftarrow$ Faux;
10  $P$.**empiler**($x$);
11  $Q$.**empiler**($y$);
// En sortie, on peut renverser $S$ dans $P$ et $Q$ pour les restaurer.
// Avec $egal$ et les tests de pile vide, on peut déterminer l'égalité
12 $res \leftarrow egal$ ET $P$ vide ET $Q$ vide. Renverser $S$ sur $P$ et $Q$;
13 **retourner** $res$

## Mots bien parenthésés

Le principe de cet algorithme est de **garder en mémoire les parenthèses qui n'ont pas encore été fermées dans une pile**. On notera **mot**$(P)$ le mot obtenu en concaténant tous les éléments de $P$ du sommet vers la base.

---

**Entrée(s) :** $s = s_0 s_1 \dots s_{n-1}$ un mot sur {"[", "(", "]", ")"}
**Sortie(s) :** "Oui" si le mot est bien parenthésé, "Non" sinon

```
1  P ← pile_vide();
2  i ← 0;
   // Invariant: mot(P).s_{i+1} ... s_{n-1} est bien parenthésé ssi s est bien
      parenthésé
3  tant que i < n faire
4      si s_i = ( ou s_i = [ alors
5          P.empiler(s_i);
6      sinon
7          si P.est_vide() alors
8              retourner Non ;
9          sinon
10             b ← P.dépiler();
11             si b et s_i ne correspondent pas alors
12                 retourner Non ;
13     i ← i + 1 ;
   // En sortie, i = n donc s est bien parenthésé ssi mot(P) l'est, et comme
      P ne contient que des parenthèses ouvrantes, ssi P est vide.
14 si P.est_vide() alors
15     retourner Oui;
16 sinon
17     retourner Non;
```

---