

Correction des TD d'algorithmique de L2

Philippe Gambette

1^{er} janvier 2008

Table des matières

Table des matières	1
1 Révisions, preuves d'arrêt	3
1.1 Séance 1 (10/10/2007)	3
1.1.1 Algorithme 1	3
1.1.2 Algorithme 2	3
1.1.3 Algorithme 4	3
1.2 Séance 2 (12/10/2007)	5
1.2.1 Algorithme 5	5
1.2.2 Preuve d'arrêt de l'algorithme 6	5
1.2.3 Preuve d'arrêt de l'algorithme 7	5
1.3 Séance 3 (17/10/2007)	5
1.3.1 Preuve d'arrêt de l'algorithme 8	5
1.3.2 Algorithme de calcul des coefficients binomiaux	7
2 Invariants	8
2.1 Séance 4 (19/10/2007)	8
2.1.1 Arrêt et invariants de l'algorithme 15	8
2.1.2 Arrêt et invariants de l'algorithme 13	9
2.2 Séance 5 (24/10/2007)	10
2.2.1 Invariant et correction de l'algorithme 7	10
2.2.2 Invariant et correction de l'algorithme 8	10
2.2.3 Correction de l'algorithme 11	11
2.3 Séance 6 (26/10/2007)	11
2.3.1 Correction de l'algorithme de recherche séquentielle en tableau trié	11
2.3.2 Correction de l'algorithme de multiplication par additions successives	12
2.3.3 Invariant de l'algorithme de recherche dichotomique modifié	13
3 Complexité	14
3.1 Séance 6 (26/10/2007)	14
3.1.1 Complexité de l'algorithme 14	14
4 Listes chaînées	15
4.1 Séance 9 (14/11/2007)	15
4.1.1 Familiarisation avec les listes chaînées	15
4.2 Séance 10 (28/11/2007)	16
4.2.1 Algorithmes sur les listes chaînées	16
4.2.2 Adresse de la dernière cellule d'une liste	16
4.2.3 Premier élément mis à la fin d'une liste	17
4.2.4 Suppression de tous les éléments de valeur donnée d'une liste	17
4.3 Séance 11 (07/12/2007)	17
4.3.1 Suppression de tous les doublons d'une liste	17
4.3.2 Miroir d'une liste	19
4.3.3 Concaténation de deux listes	19
4.3.4 Familiarisation avec les listes doublement chaînées	19
4.4 Exercices non corrigés en TD (28/12/2007)	20
4.4.1 Miroir d'une liste doublement chaînée	20
4.4.2 Concaténation de deux listes doublement chaînées	20

5 Arbres	22
5.1 Séance 12 (12/12/2007)	22
5.1.1 Familiarisation avec les arbres	22
5.1.2 Hauteur d'un arbre binaire	22
5.1.3 Nombre de feuilles d'un arbre binaire	22
5.2 Séance 13 (12/12/2007, 13/12/2007)	23
5.2.1 Intermède complexité	23
5.2.2 Nombre de nœuds de profondeur fixée d'un arbre binaire	23
5.2.3 Effeillage d'un arbre binaire	23
6 Autres structures de données	25
6.1 Séance 14 (14/12/2007)	25
6.1.1 Notation postfixe d'un arbre	25
6.1.2 Pile pour l'évaluation d'une expression postfixe	25
6.1.3 File pour un parcours en largeur d'arbre	26
6.1.4 Taille de l'intersection de deux tableaux d'entiers	26

Avertissement

Ce document contient éventuellement des erreurs volontaires, que j'ai indiquées pendant les séances de TD. Je signale aussi que la pratique de recherche des exercices pendant la séance de TD est indispensable pour s'entraîner, et déconseille cordialement l'utilisation exclusive de ce corrigé (la veille de l'examen par exemple) sans avoir assisté aux séances.

Signalez-moi toute erreur involontaire à gambette@lirmm.fr. Je remercie pour cela (j'espère que la liste s'allongera) :

– Cindy Sartre

Les cours d'algorithmique de L1 sont disponibles à l'adresse <http://www.lirmm.fr/~vilarem/FLIN101/>.

Les indications sur l'utilisation du langage \LaTeX pour que vous puissiez participer à la rédaction de ce corrigé est disponible avec le fichier source de ce document à l'adresse <http://www.lirmm.fr/~gambette/EnsAlgo.php>.

Chapitre 1

Révisions, preuves d'arrêt

1.1 Séance 1 (10/10/2007)

1.1.1 Algorithme 1

```
Données :  $T[1..n]$  tableau d'entiers,  $x$  un entier.  
Résultat :  $PP$  est un des éléments de  $T$  les plus proches de  $x$ , c'est à dire  $PP \in T$  et  $\forall i \in \llbracket 1, n \rrbracket$ ,  
           $|x - PP| \leq |x - T[i]|$ .  
début  
   $PP \leftarrow T[1]$ ;  
  pour tous les  $i$  de 2 à  $n$  faire  
    si  $|x - T[i]| < |x - PP|$  alors  $PP \leftarrow T[i]$ ;  
  fin  
fin
```

Algorithme 1 : PlusProche(**d** $T[1..n]$: tableau d'entiers, **d** x : entier, **r** PP : entier)

1.1.2 Algorithme 2

En considérant que **renvoyer** permet de terminer l'algorithme en arrêtant les boucles en cours :

```
Données :  $A[1..n]$  et  $B[1..n]$  deux tableaux de  $n$  entiers et  $Som$  un entier.  
Résultat : Renvoie Vrai s'il existe  $i, j \in \llbracket 1..n \rrbracket$  tels que  $A[i] + B[j] = Som$ , renvoie Faux sinon.  
début  
  pour tous les  $i$  de 1 à  $n$  faire  
    pour tous les  $j$  de 1 à  $n$  faire  
      si  $A[i] + B[j] = Som$  alors renvoyer Vrai;  
    fin  
  fin  
  renvoyer Faux;  
fin
```

Algorithme 2 : Somme?(**d** $A[1..n]$: tableau d'entiers, **d** $B[1..n]$: tableau d'entiers, **d** Som : entier) : booléen

Alternative "tordue" qui évite la cassure du **renvoyer** en milieu de boucle **pour**, avec un **tant que** :

1.1.3 Algorithme 4

$T[1..n, 1..n]$ est une sorte de matrice de 0 et de 1, $T[i, j]$ représente l'entier à la ligne i et à la colonne j .

Pour la culture...

Remarquons que comme il y a trois boucles (deux **pour** et une **tant que** imbriquées, la complexité de l'algorithme 4 proposé est en $O(n^3)$. On peut en obtenir un en $O(n^2 \log n)$ de la manière suivante : trier les lignes

Données : $A[1..n]$ et $B[1..n]$ deux tableaux de n entiers et Som un entier.
Résultat : Renvoie Vrai s'il existe $i, j \in \llbracket 1, n \rrbracket$ tels que $A[i] + B[j] = Som$, renvoie Faux sinon.

début

```

     $i \leftarrow 1; j \leftarrow 1;$ 
    tant que  $j < n + 1$  et  $A[i] + B[j] \neq Som$  faire
        si  $i = n$  alors
             $j \leftarrow j + 1;$ 
             $i \leftarrow 0;$ 
        sinon
             $i \leftarrow i + 1;$ 
        fin
    fin
    si  $j = n + 1$  alors
        renvoyer Faux;
    sinon
        renvoyer Vrai;
    fin
fin
```

Algorithme 3 : Somme?($\mathbf{d} A[1..n]$: tableau d'entiers, $\mathbf{d} B[1..n]$: tableau d'entiers, $\mathbf{d} Som$: entier) : booléen

Données : $T[1..n, 1..n]$ un tableau de 0 et de 1.
Résultat : Renvoie Vrai si et seulement si T possède deux lignes identiques, c'est à dire ssi $\exists i \neq j \in \llbracket 1, n \rrbracket / \forall k \in \llbracket 1, n \rrbracket, T[i, k] = T[j, k]$.

début

```

    pour tous les  $i$  de 1 à  $n$  faire
        pour tous les  $j$  de 1 à  $n$  faire
             $k \leftarrow 1;$ 
            tant que  $T[i, k] = T[j, k]$  faire
                si  $k = n$  alors
                    renvoyer Vrai;
                sinon
                     $k \leftarrow k + 1;$ 
            fin
        fin
    fin
    renvoyer Faux;
fin
```

Algorithme 4 : LignesEgales?($\mathbf{d} T[1..n, 1..n]$: tableau d'entiers) : booléen

du tableau en $O(n^2 \log n)$ (chaque comparaison de deux lignes se fait en $O(n)$), puis parcourir chaque ligne du tableau pour vérifier si elle est égale à la suivante.

On peut même être encore plus astucieux en utilisant le fait que le tableau contient seulement des 0 et des 1 et en commençant par le trier par un tri par base (tri radix), avant de comparer chaque ligne avec la suivante, pour obtenir une complexité totale en $O(n^2)$.

1.2 Séance 2 (12/10/2007)

1.2.1 Algorithme 5

On n'utilise qu'un seul tableau T en considérant qu'on peut augmenter sa taille de n à $n + 1$.

Données : $T[1..n, 1..n]$ un tableau de 0 et de 1.
Résultat : T modifié de telle sorte que e soit inséré à la bonne place parmi les m premiers éléments (le tableau $T[1..m + 1]$ est donc trié).

```

début
   $i \leftarrow 1$ ;
  tant que  $e < T[i]$  et  $i \leq m$  faire
    |  $i \leftarrow i + 1$ ;
  fin
  /* la  $i$ -ième case et celles à sa droite doivent être décalées d'une case à droite
    pour pouvoir insérer  $e$  en  $i$ -ième position. */
   $j \leftarrow n$ ;
  tant que  $j \geq i$  faire
    |  $T[j + 1] = T[j]$ ;
  fin
   $T[i] = e$ ;
fin

```

Algorithme 5 : InsérerTabTrié(**dr** $T[1..n]$: tableau d'entiers, **d** m : entier, **d** e : entier)

1.2.2 Preuve d'arrêt de l'algorithme 6

Soit n_k la valeur de la variable n à la k -ième itération de la boucle **tant que**. Si $N = 0$ ou $N = 1$ alors l'algorithme s'arrête, considérons donc le cas où $N > 1$. Si N est pair alors on peut vérifier que $n_k = 2^k N$, si N est impair, alors $n_k = 2^{k-1}(N - 1)$. Dans les deux cas la suite (n_k) croît vers $+\infty$ à partir de son deuxième terme si $N \notin \{0, 1\}$, donc la condition d'arrêt du **tant que** n'est pas atteinte, et l'algorithme ne s'arrête pas.

1.2.3 Preuve d'arrêt de l'algorithme 7

Soit n_k la valeur de la variable n à la k -ième itération de la boucle **tant que**. Considérons la suite définie par $u_k = |n - 7|$. (u_k) est à valeurs entières. De plus, si $n_k > 7$ alors $n_{k+1} = n_k - 1 \geq 7$ donc $0 \leq n_{k+1} - 7 \leq n_k - 7$ donc $0 \leq u_k + 1 < u_k$. Si $n_k < 7$ alors $n_{k+1} = n_k + 1 \leq 7$ donc $0 \leq 7 - n_{k+1} - 1 = 7 - n_{k+1} < 7 - n_k$ donc $0 \leq u_k + 1 < u_k$. Donc la suite (u_k) est décroissante strictement, et à valeurs entières. Donc elle sera nulle à un certain rang, la condition d'arrêt de la boucle **tant que** sera alors vérifiée, et l'algorithme s'arrêtera.

Calculer le résultat de l'algorithme 7 sur plusieurs exemples permet de remarquer qu'il calcule $N!$. Nous le prouverons plus loin en utilisant un invariant.

1.3 Séance 3 (17/10/2007)

1.3.1 Preuve d'arrêt de l'algorithme 8

En notant n_k et m_k les valeurs respectives de n et m à la k -ième itération de la boucle **tant que**, on transforme les lignes de l'algorithme faisant intervenir les variables n et m en égalités mathématiques sur les termes des suites (n_k) et (m_k) .

Lignes 3 : $n \leftarrow N$; $m \leftarrow M$; donne : $n_0 = N$, $m_0 = M$.

Lignes 5 et 6 : $m \leftarrow m - n$ donne : $\forall k, m_k > n_k \Rightarrow \begin{cases} m_{k+1} = m_k - n_k \\ n_{k+1} = n_k \end{cases}$

```

Données :  $N, M \in \mathbb{N}^*$ 
Résultat : renvoie  $\text{pgcd}N, M$ .
1 Variables :  $n, m$  entiers
2 début
3    $n \leftarrow N; m \leftarrow M;$ 
4   tant que  $n \neq 0$  et  $m \neq 0$  faire
5     si  $m > n$  alors
6        $m \leftarrow m - n$ 
7     sinon
8        $n \leftarrow n - m$ 
9     fin
10  fin
11  renvoyer  $m + n;$ 
12 fin

```

Algorithme 6 : PGCD(d N, M : entier) : entier

Lignes 7 et 8 : $n \leftarrow n - m$ donne : $\forall k, m_k \leq n_k \Rightarrow \begin{cases} m_{k+1} = m_k \\ n_{k+1} = n_k - m_k \end{cases}$

On peut alors utiliser ces égalités mathématiques pour démontrer les propriétés demandées.

Appelons m_k (respectivement n_k la valeur de la variable m (resp. n) à la k -ième itération de la boucle **tant que** et montrons par récurrence que $\forall k \in \mathbb{N}, m_k \geq 0$.

Initialisation : $m_0 = M \geq 0$

Hérédité : Supposons qu'à un certain rang k $m_k > 0$ et montrons que $m_{k+1} > 0$. Deux cas se présentent :

- si $m_k > n_k$ alors $m_{k+1} = m_k - n_k > 0$,
- sinon, $m_{k+1} = m_k > 0$ par l'hypothèse de récurrence.

La propriété étant initialisée et héréditaire, elle est vraie pour tout k de \mathbb{N} .

De même on démontre que $\forall k \in \mathbb{N}, n_k \geq 0$.

Montrons maintenant que m décroît pendant l'exécution de l'algorithme, en calculant $m_{k+1} - m_k$. Deux cas se présentent :

- si $m_k > n_k$, $m_{k+1} - m_k = m_k - n_k - m_k = -n_k \leq 0$
- sinon, $m_{k+1} - m_k = m_k - m_k = 0 \leq 0$

donc la suite (m_k) est décroissante (pas nécessairement strictement).

De même on peut aussi montrer que (n_k) est décroissante (pas nécessairement strictement).

Les expressions $n - m$ et $m - n$ ne sont pas nécessairement positives, ni monotones, $|n - m|$ n'est pas nécessairement monotone non plus, on peut le prouver en exhibant un contre-exemple. En initialisant avec $N = 10$ et $M = 9$, on obtient :

k	0	1	2
n_k	10	1	1
m_k	9	9	8
$m_k - n_k$	1	\searrow -8	\nearrow -7
$ m_k - n_k $	1	\nearrow 8	\searrow 7

Aucune des expressions proposées pour le moment ne nous permet donc de conclure quant à l'arrêt de l'algorithme 8. La stricte décroissance de la suite à valeurs entières $n + m$ va nous permettre de le prouver.

Fixons $k \in \mathbb{N}$, et notons $u_k = n_k + m_k$. Deux cas se présentent :

- si $m_k > n_k$ alors $u_{k+1} - u_k = m_{k+1} + n_{k+1} - m_k - n_k = m_k - n_k + n_k - m_k - n_k$ donc $u_{k+1} - u_k = -n_k < 0$ puisque $n_k > 0$.
- sinon, $u_{k+1} - u_k = m_{k+1} + n_{k+1} - m_k - n_k = n_k - m_k + m_k - m_k - n_k = -m_k < 0$ puisque $m_k > 0$.

Dans tous les cas, tant que l'algorithme ne s'arrête pas, la suite (u_k) est strictement décroissante, et à valeurs entières. Elle finira donc par atteindre 0 à un certain rang k_f et on aura alors $n_{k_f} = 0$ et $m_{k_f} = 0$, ce qui correspond à la condition d'arrêt de l'algorithme.

On montre similairement que l'expression $m * n$ est aussi décroissante strictement et à valeurs dans \mathbb{N} .

1.3.2 Algorithme de calcul des coefficients binomiaux

Rappel sur les coefficients binomiaux : voir http://fr.wikipedia.org/wiki/Coefficient_binomial

Illustration de l'algorithme 9 : on peut voir se qui se passe sur le triangle de Pascal :

	$p = 0$	1	2	3	4	5
$n = 0$	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

A chaque appel de l'algorithme `coefBin` pour calculer la valeur dans une certaine case (n, p) , celui-ci renvoie 1 s'il est sur un des bords du triangle de Pascal, sinon il appelle récursivement `coefBin` pour calculer les valeurs dans la case située au-dessus, $(n - 1, p)$, et celle au-dessus à gauche, $(n - 1, p - 1)$.

On va de nouveau chercher une expression entière qui décroît à chaque appel récursif pour prouver l'arrêt de l'algorithme. Les données à chaque appel récursif étant n et p qui sont dans \mathbb{N} , on cherche une fonction $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ strictement décroissante.

On pourrait choisir pour f la fonction suivante : $(n, p) \mapsto n$ qui est bien à valeurs entières et strictement décroissante. Le problème est que cela nous permet seulement d'en conclure qu'elle atteindra forcément 0, ce qui donnera $n = 0$, ce qui ne correspond pas à l'arrêt des appels récursifs (qui a lieu quand $p=0$ ou $n=p$). Il faut donc trouver autre chose.

La fonction $(n, p) \mapsto n - p$ est décroissante mais pas strictement, elle ne convient donc pas non plus.

La fonction $(n, p) \mapsto C_n^p = \text{coefBin}(n, p)$ conviendrait, encore faut-il le prouver, ce qui n'est pas complètement évident, et il resterait alors à prouver que quand elle atteint 1 on a bien $p = 0$ ou $n = p$.

Bref, passons à la solution : $f : (n, p) \mapsto n + p$. Effectivement, pour le premier appel récursif, on a $f(n - 1, p - 1) - f(n, p) = n - 1 + p - 1 - (n + p) = -2 < 0$. Pour le second appel récursif on a $f(n - 1, p) - f(n, p) = n - 1 + p - (n + p) = -1 < 0$. Donc dans tous les cas $n + p$ décroît à chaque appel récursif. Etant à valeurs entières, elle atteindra donc forcément 0, ce qui impliquera $p = 0$.

Chapitre 2

Invariants

2.1 Séance 4 (19/10/2007)

Les invariants, à quoi ça sert ?

Ce sont des propriétés mathématiques sur les variables d'un algorithme qui sont valables tout au long de son exécution (à chaque itération d'une boucle, ou à chaque appel récursif). On les utilise pour démontrer sa correction, c'est à dire prouver que l'algorithme renvoie bien le résultat voulu. Pour prouver des invariants, on montre qu'ils sont maintenus de l'étape k à l'étape $k+1$ de l'exécution, plus formellement c'est une démonstration par récurrence qu'il faut effectuer pour montrer que l'invariant est valable à chaque étape.

Rappelons que pour démontrer une égalité $A = B$ (respectivement, une inégalité $A \geq B$) on peut partir un terme et faire des transformations pour arriver à l'autre, ou montrer que $A - B = 0$ (respectivement, $A - B \geq 0$).

2.1.1 Arrêt et invariants de l'algorithme 15

L'algorithme 15 s'arrête car l'expression $M - Z$ est strictement décroissante (à la $k+1$ -ième itération de la boucle on a $M_{k+1} - Z_{k+1} = M_k - Z_k - Z_k - 2 < M_k - Z_k$) et à valeurs entières. Elle deviendra donc forcément négative ce qui correspond à la condition de sortie de la boucle **tant que**, donc l'algorithme s'arrête bien.

Montrons l'invariant $Z = 2I + 1$.

Pour cela appelons Z_k (respectivement I_k) la valeur de la variable Z (respectivement I) à la k -ième boucle **tant que**. Montrons par récurrence sur k que $\forall k \in \mathbb{N}, Z_k = 2I_k + 1$.

Initialisation : au rang 0, $2I_0 + 1 = 0 + 1 = 1 = Z_0$.

Hérédité : soit $k \in \mathbb{N}, 2I_k + 1 = Z_k \Rightarrow 2I_{k+1} + 1 = Z_{k+1}$?

$$\begin{aligned} 2I_{k+1} + 1 - Z_{k+1} &= 2(I_k + 1) + 1 - (Z_k + 2) \text{ (d'après l'algorithme)} \\ &= 2I_k + 2 + 1 - Z_k - 2 \\ &= 2I_k + 1 - Z_k \\ &= 0 \text{ (par l'hypothèse de récurrence)} \end{aligned}$$

Donc la propriété est initialisée et héréditaire, elle est donc vraie pour tout $k \in \mathbb{N}$, et l'invariant $2I + 1 = Z$ est bien démontré.

Montrons l'invariant $M = N - I^2$.

Soit M_k la valeur de la variable M à la k -ième boucle **tant que**. Montrons par récurrence sur k que $\forall k \in \mathbb{N}, M_k = N - I_k^2$.

① au rang 0, $M_0 = N = N - 0^2 = N - I_0^2$.

② soit $k \in \mathbb{N}, M_k = N - I_k^2 \Rightarrow M_{k+1} = N - I_{k+1}^2$?

$$\begin{aligned} M_{k+1} - (N - I_{k+1}^2) &= M_k - Z_k - (N - (I_k + 1)^2) \text{ (algo)} \\ &= M_k - Z_k - N + I_k^2 + 2I_k + 1 \\ &= M_k - (N - I_k^2) - Z_k + 2I_k + 1 \\ &= 0 \text{ (h.r. + question précédente)} \end{aligned}$$

Donc la propriété est initialisée et héréditaire, elle est donc vraie pour tout $k \in \mathbb{N}$, et l'invariant $M = N - I^2$ est bien démontré.

Montrons l'invariant $M \geq 0$. Il n'y a même pas besoin de récurrence : pour $k = 0$, $M_0 = N \geq 0$ et $\forall k \geq 1$, $M_k = M_{k-1} - Z_{k-1} \geq 0$ d'après la condition de la boucle **tant que**, donc on a bien $\forall k \geq 0$, $M_k \geq 0$.

Montrons que $I^2 \leq N < (I+1)^2$. Pour cela montrons qu'en fin d'algorithme on a $N - I^2 \geq 0$ et $(I+1)^2 - N > 0$. $N - I^2 = M \geq 0$ d'après le deuxième et le troisième invariant. $(I+1)^2 - N = I^2 + 2I + 1 - N = 2I + 1 - M = Z - M$ d'après les deux premiers invariants. Or à la fin de l'algorithme la condition d'arrêt de la boucle **tant que** n'est pas vérifiée donc $Z - M > 0$, ce qui fournit bien l'inégalité voulue, et donc la correction de l'algorithme en passant à la racine :

$$I \leq \sqrt{N} < I + 1 \Leftrightarrow I = \lfloor \sqrt{N} \rfloor$$

2.1.2 Arrêt et invariants de l'algorithme 13

L'algorithme 13 s'arrête car la variable C a des valeurs entières et strictement décroissantes, elle deviendra donc négative ou nulle, ce qui correspond à la condition de sortie de boucle **tant que**, et donc l'algorithme s'arrête.

Appelons A_i (respectivement B_i, C_i, Z_i) la valeur de la variable A (respectivement B, C, Z) à la fin de la i -ième itération de la boucle **tant que**. Montrons par récurrence sur i que $\forall i \in \mathbb{N}, A_i = 3(X - C_i) + 1$.

① au rang 0, $A_0 = 1 = 3(X - X) + 1 = 3(X - C_0) + 1$.

② soit $i \in \mathbb{N}$, $A_i = 3(X - C_i) + 1 \Rightarrow A_{i+1} = 3(X - C_{i+1}) + 1$?

$$\begin{aligned} A_{i+1} - 3(X - C_{i+1}) + 1 &= A_i + 3 - 3(X - (C_i - 1)) \quad (\text{algo}) \\ &= A_i + 3 - 3X + 3C_i - 3 \\ &= A_i - 3(X - C_i) \\ &= 0 \quad (\text{h.r.}) \end{aligned}$$

Donc la propriété est initialisée et héréditaire, elle est donc vraie pour tout $i \in \mathbb{N}$, et l'invariant $A_i = 3(X - C_i) + 1$ est bien démontré.

Montrons par récurrence sur i que $\forall i \in \mathbb{N}, B_i = 3(X - C_i)^2$.

① au rang 0, $B_0 = 0 = 3(X - X)^2 = 3(X - C_0)^2$.

② soit $i \in \mathbb{N}$, $B_i = 3(X - C_i)^2 \Rightarrow B_{i+1} = 3(X - C_{i+1})^2$?

$$\begin{aligned} B_{i+1} - 3(X - C_{i+1})^2 &= B_i + 2A_i + 1 - 3(X - (C_i - 1))^2 \quad (\text{algo}) \\ &= B_i + 2A_i + 1 - 3((X - C_i) + 1)^2 \\ &= B_i + 2A_i + 1 - 3(X - C_i)^2 - 6(X - C_i) - 3 \\ &= B_i - 3(X - C_i)^2 + 6(X - C_i) + 2 + 1 - 6(X - C_i) - 3 \quad (\text{question précédente}) \\ &= 0 \quad (\text{h.r.}) \end{aligned}$$

Donc la propriété est initialisée et héréditaire, elle est donc vraie pour tout $i \in \mathbb{N}$, et l'invariant $B_i = 3(X - C_i)^2$ est bien démontré.

Montrons par récurrence sur i que $\forall i \in \mathbb{N}, Z_i = (X - C_i)^3$.

① au rang 0, $Z_0 = 0 = (X - X)^3 = (X - C_0)^3$.

② soit $i \in \mathbb{N}$, $Z_i = (X - C_i)^3 \Rightarrow Z_{i+1} = (X - C_{i+1})^3$?

$$\begin{aligned} Z_{i+1} - (X - C_{i+1})^3 &= Z_i + A_i + B_i - (X - (C_i - 1))^3 \quad (\text{algo}) \\ &= Z_i + A_i + B_i - ((X - C_i) + 1)^3 \\ &= (X - C_i)^3 + 3(X - C_i) + 1 + 3(X - C_i)^2 - ((X - C_i) + 1)^3 \quad (\text{h.r.} + \text{questions précédentes}) \\ &= (X - C_i)^3 + 3(X - C_i)^2 + 3(X - C_i) + 1 - ((X - C_i) + 1)^3 \\ &= 0 \end{aligned}$$

Donc la propriété est initialisée et héréditaire, elle est donc vraie pour tout $i \in \mathbb{N}$, et l'invariant $Z_i = (X - C_i)^3$ est bien démontré.

A la fin de l'algorithme 13, qui renvoie Z , on a $C = 0$, donc d'après l'invariant il calcule $(X - 0)^3$ c'est à dire X^3 .

2.2 Séance 5 (24/10/2007)

2.2.1 Invariant et correction de l'algorithme 7

Pour trouver son invariant on peut essayer d'exécuter l'algorithme 7 sur plusieurs exemples et voir si l'on arrive à trouver une certaine régularité à chaque itération de la boucle **tant que**.

Commençons par un exemple avec $N = 10 > 7$, en appelant n_i (respectivement f_i) la valeur de la variable n à la fin de la i -ième boucle **tant que**.

i	0	1	2	3
n_i	10	9	8	7
f_i	7!	7! × 10	7! × 10 × 9	7! × 10 × 9 × 8

On est tenté de compléter le début de la factorielle qui se construit progressivement pour $f_i : 10, 10 \times 9, 10 \times 9 \times 8$. On remarque que pour obtenir $10!$ à chaque étape, il suffirait de multiplier par $: 9!, 8!, 7!$, ce qui correspond en fait à $n_i!$. Ainsi si à chaque étape on calcule $f_i \times n_i!$, il semble qu'on obtient une quantité constante, égale à $f_0 \times n_0! = 7!N!$.

On peut vérifier sur un exemple que l'invariant semble aussi vérifié pour $N < 7$. Il s'agit maintenant de le démontrer. Montrons donc par récurrence sur i que $\forall i \in \mathbb{N}, n_i!f_i = 7!N!$.

① la propriété est trivialement vraie pour $i = 0$.

② soit $i \in \mathbb{N}, n_i!f_i = 7!N! \Rightarrow f_{i+1} \times n_{i+1}! = 7!N!$? Si $n_i > 7$ alors $n_{i+1}!f_{i+1} = (n_i - 1)n_i!f_i = n_i!f_i = 7!N!$ par l'hypothèse de récurrence. Sinon, $n_{i+1}!f_{i+1} = n_{i+1}! \frac{f_i}{n_{i+1}} = \frac{n_{i+1}!}{n_{i+1}} \frac{7!N!}{n_i!} = 7!N!$.

Ainsi la propriété voulue est bien initialisée et héréditaire, elle est donc vraie sur tout \mathbb{N} . On déduit de cet invariant qu'à la fin de l'algorithme on renvoie $f_i = \frac{7!N!}{n_i!}$ avec $n_i = 7$, donc l'algorithme 7 renvoie $f_i = N!$.

2.2.2 Invariant et correction de l'algorithme 8

Exécutons l'algorithme 8 pour $M = 48$ et $N = 30$, en appelant m_i (respectivement n_i) la valeur de m (respectivement de n) à la i -ième exécution de la boucle **tant que**.

i	m	n	diviseurs communs à m et n
0	48	30	{1, 2, 3, 6}
1	18	30	{1, 2, 3, 6}
2	18	12	{1, 2, 3, 6}
3	6	12	{1, 2, 3, 6}
4	6	6	{1, 2, 3, 6}
5	6	0	{1, 2, 3, 6}

Un invariant semble être que l'ensemble des diviseurs communs à m et n reste constant au cours de l'algorithme, démontrons-le.

Supposons qu'il existe q divisant $m_k = m'q$ et $n_k = n'q$, que $m_k > n_k$, et qu'il y a une $k + 1$ -ième itération de la boucle **tant que**. Alors $m_{k+1} = m_k - n_k = q(m' - n')$. $n_{k+1} = n_k = n'q$ donc q est encore un diviseur commun de m_{k+1} et n_{k+1} . Ainsi on a montré que l'ensemble des diviseurs communs reste constant tout au long de l'algorithme, soit $\{d \in \mathbb{N}/d|m_k \text{ et } d|n_k\}$ constant pour tout k . En particulier en appliquant cet invariant à $k = 0$ et $k = h$, h étant la dernière itération de la boucle **tant que** avant l'arrêt de l'algorithme, on a $\{d \in \mathbb{N}/d|M \text{ et } d|N\} = \{d \in \mathbb{N}/d|m_h \text{ et } d|0\}$. En considérant le maximum de ces ensembles, on obtient $\text{pgcd}(M, N) = \text{pgcd}(m_h, 0) = m_h$, ce qui démontre bien la correction de l'algorithme.

Remarques :

Cet algorithme s'appelle l'algorithme de PGCD par différences successives. On utilise classiquement l'algorithme d'Euclide, par divisions successives, pour calculer le PGCD. Effectivement l'algorithme par différences successives peut avoir une complexité assez mauvaise, regardez par exemple ce qui se passe quand on essaie de calculer $\text{PGCD}(n, 1)$.

2.2.3 Correction de l'algorithme 11

On commence par voir sur un exemple ce que calcule l'algorithme, en appelant i_k (respectivement $r_{1_k}, r_{2_k}, r_{3_k}$) la valeur de la variable i (respectivement r_1, r_2, r_3) à la fin de la k -ième itération de la boucle **tant que**.
Pour $n = 6$:

k	0	1	2	3	4	5	6
r_{3_i}	×	0	1	1	2	3	5
r_{1_i}	0	1	1	2	3	5	8
r_{2_i}	1	1	2	3	5	8	13
i	0	1	2	3	4	5	6

Ceci nous permet de supposer que $r_{1_i} = \text{fib}(i)$, $r_{2_i} = \text{fib}(i+1)$ et $r_{3_i} = \text{fib}(i-1)$. Pour le démontrer, on peut procéder de deux façons :

Première démonstration : une habile récurrence pour r_1 , puis le reste.

Montrons par récurrence sur i que $\forall i \in \mathbb{N}, r_{1_i} = \text{fib}(i)$.

① au rang 1, $r_{1_1} = 1 = \text{fib}(1)$.

② soit $i \in \mathbb{N}, \forall k \leq i, r_{1_k} = \text{fib}(k) \Rightarrow r_{1_{i+1}} = \text{fib}(i+1)$?

$r_{1_{i+1}} = r_{2_i} = r_{3_i} + r_{2_{i-1}} = r_{1_i} + r_{1_{i-1}} = \text{fib}(i+1)$.

Ainsi l'égalité est initialisée et héréditaire, elle est donc vraie pour tout i et donc $r_{1_i} = \text{fib}(i)$.

On peut en déduire les deux autres inégalités : $\forall i \in \mathbb{N}, r_{2_i} = r_{1_{i+1}} = \text{fib}(i+1)$, et $\forall i \in \mathbb{N}^*, r_{3_i} = r_{1_{i-1}} = \text{fib}(i-1)$, et l'égalité est aussi vraie pour $i = 0$.

Seconde démonstration : tout en même temps, c'est plus facile.

Montrons par récurrence sur i que $\forall i \in \mathbb{N}, r_{1_i} = \text{fib}(i)$, $r_{2_i} = \text{fib}(i+1)$, et $r_{3_i} = \text{fib}(i-1)$.

① au rang 1, $r_{1_1} = 1 = \text{fib}(1)$, $r_{2_1} = 1 = \text{fib}(2)$, et $r_{3_1} = 0 = \text{fib}(0)$.

② soit $i \in \mathbb{N}, r_{1_i} = \text{fib}(i)$, $r_{2_i} = \text{fib}(i+1)$, et $r_{3_i} = \text{fib}(i-1) \Rightarrow r_{1_{i+1}} = \text{fib}(i+1)$, $r_{2_{i+1}} = \text{fib}(i+2)$, et $r_{3_{i+1}} = \text{fib}(i)$?

$r_{1_{i+1}} = r_{2_i} = \text{fib}(i+1)$ (par h.r.). $r_{3_{i+1}} = r_{1_i} = \text{fib}(i)$ (par h.r.). $r_{2_{i+1}} = r_{3_{i+1}} + r_{2_i} = \text{fib}(i) + \text{fib}(i+1)$ (par égalité précédente et h.r.) donc $r_{2_{i+1}} = \text{fib}(i+2)$.

Ainsi les trois égalités sont initialisées et héréditaires, donc vraies pour tout i de \mathbb{N} .

En fin d'algorithme $i = n$, il faut donc renvoyer r_1 .

2.3 Séance 6 (26/10/2007)

2.3.1 Correction de l'algorithme de recherche séquentielle en tableau trié

Données : $T[1..N]$: tableau d'entiers trié dans l'ordre croissant, e : entier recherché dans le tableau
Résultat : renvoie **Vrai** si e est un élément du tableau T , **Faux** sinon.
 Variable : i entier.
début
 $i \leftarrow 1$;
 tant que $i \leq N$ et $T[i] < e$ **faire**
 $i \leftarrow i + 1$;
 fin
 renvoyer ($i \leq N$ et $T[i] = e$);
fin

Algorithme 7 : Recherche(d $T[1..N]$: tableau, d e : entier) : booléen

L'invariant doit nous servir à démontrer la correction de l'algorithme, il y a plusieurs possibilités :

- $\forall j \in [1, i-1], T[j] < e$
- $T[i-1] < e$

Démontrons les trois et utilisons-les chacun pour démontrer la correction de l'algorithme. Appelons i_k la valeur de i à la fin de la k -ième itération de la boucle **tant que**.

Invariant 1

Montrons par récurrence sur k que $\forall k \geq 1, \forall j \in [1, i_k - 1], T[j] < e$

Initialisation : au rang $k = 1$, on a exécuté la boucle **tant que** une fois, donc on a bien $T[1] = e$.

Hérédité : supposons qu'à un certain rang $k \geq 1$, $\forall j \in [1, i_k - 1], T[j] < e$, et que la boucle **tant que** s'exécute k fois. Alors au début de la k -ième itération on a bien testé que $T[i_k] < e$, donc en ajoutant cette inégalité à celles de l'hypothèse de récurrence on obtient $\forall k \geq 1, \forall j \in [1, i_k], T[j] < e$.

La propriété étant initialisée et héréditaire, elle est vraie pour tout $k \geq 1$, c'est à dire si la boucle **tant que** s'exécute une fois ou plus.

Prouvons maintenant la correction de l'algorithme. S'il renvoie **Vrai**, alors $i \leq N$ et $T[i] = e$, on a donc bien trouvé une case du tableau, la i -ième, qui contient e . S'il renvoie **Faux**, alors $(i \leq N \text{ et } T[i] = e)$ est faux, donc l'un des deux termes de la conjonction est faux :

- soit $i > N$, alors l'invariant démontré affirme qu'aucune des N cases du tableau ne contient e .
- soit $i \leq N$ et $T[i] \neq e$. L'algorithme s'est arrêté, donc est sorti de la boucle, donc la condition de boucle $T[i] < e$ n'était pas vérifiée, donc $T[i] > e$. Or le tableau contient des entiers rangés par ordre croissant donc e n'est pas contenu dans la i -ième case ni aucune à droite. D'autre part l'invariant nous affirme directement que e n'est contenu dans aucune des $i - 1$ premières cases. Finalement e n'est dans aucune des N cases du tableau.

On a donc bien démontré que l'algorithme est correct.

Invariant 2

Au début k -ième itération de la boucle **tant que** on teste que $T[i_{k-1}] < e$. Or $i_{k-1} = i_k - 1$ d'après l'algorithme, donc on a bien pour tout $k \geq 1$, $T[i_k - 1] < e$.

Prouvons maintenant la correction de l'algorithme. S'il renvoie **Vrai**, alors $i \leq N$ et $T[i] = e$, on a donc bien trouvé une case du tableau, la i -ième, qui contient e . S'il renvoie **Faux**, alors $(i \leq N \text{ et } T[i] = e)$ est faux, donc l'un des deux termes de la conjonction est faux :

- soit $i > N$, alors l'invariant démontré affirme que la $i - 1$ -ième case du tableau contient une valeur inférieure strictement à e , or les cases précédentes contiennent des valeurs strictement inférieures puisque le tableau est rangé dans l'ordre croissant, donc e n'est pas contenu dans le tableau.
- soit $i \leq N$ et $T[i] \neq e$. L'algorithme s'est arrêté, donc est sorti de la boucle, donc la condition de boucle $T[i] < e$ n'était pas vérifiée, donc $T[i] > e$. Or le tableau contient des entiers rangés par ordre croissant donc e n'est pas contenu dans la i -ième case ni aucune à droite. D'autre part l'invariant nous affirme que $T[i - 1] < e$, et le tableau est rangé par ordre croissant, donc e n'est contenu dans aucune des $i - 1$ premières cases. Finalement e n'est dans aucune des N cases du tableau.

On a donc bien démontré que l'algorithme est correct.

Remarquons que l'invariant 2 était plus direct à démontrer (sans récurrence), mais demande quelques lignes supplémentaires pour la preuve de correction. Lorsqu'on demande un invariant il est possible qu'il y ait plusieurs possibilités, il s'agit d'en choisir un qui soit suffisamment simple à démontrer et à utiliser pour la preuve de correction.

Précisons aussi que l'invariant a été démontré dans tous les cas où la boucle **tant que** est exécutée *au moins une fois*. Ceci dit le cas où elle ne s'exécute pas ne pose pas de problème, les preuves de correction restent valides, on peut juste se débarrasser de ce qui suit le « *D'autre part* » (cela concerne les $i - 1$ premières cases qui ne sont alors pas définies puisque $i = 1$).

2.3.2 Correction de l'algorithme de multiplication par additions successives

Données : m, n : entiers
Résultat : renvoie le produit mn .
 Variables : p, i , entiers.
début
 $p \leftarrow 0; i \leftarrow 0;$
 tant que $i < n$ **faire**
 $p \leftarrow p + m; i \leftarrow i + 1;$
 fin
 renvoyer p ;
fin

Algorithme 8 : Produit(m, n : entiers) : entier

L'invariant est $p = mi$. Appelons p_k, i_k , les valeurs de p et i au début de la k -ième boucle **tant que** et démontrons par récurrence sur k que $p_k = mi_k$.

$$\textcircled{1} p_1 = 0 = m \times 0 = mi_0.$$

$$\textcircled{H} p_k = mi_k \Rightarrow p_{k+1} = mi_{k+1} ? p_{k+1} = p_k + m = mi_k + m = m(i_k + 1) = mi_{k+1}$$

Ainsi la propriété est initialisée et héréditaire, donc vraie pour tout k , et l'égalité proposée est bien un invariant de l'algorithme.

2.3.3 Invariant de l'algorithme de recherche dichotomique modifié

La seule modification de l'invariant demandée concerne le cas d'égalité. Dans le cours, l'algorithme renvoyait l'indice de la dernière case contenant e , dans l'exercice on veut récupérer l'indice de la première case contenant e , il suffit donc de remplacer dans l'algorithme « $> e$ » par « $\geq e$ ».

Chapitre 3

Complexité

3.1 Séance 6 (26/10/2007)

3.1.1 Complexité de l'algorithme 14

Comme l'algorithme 14 contient une boucle **tant que**, il s'agit de compter en fonction des données le nombre d'itérations de cette boucle.

Attention, lors du comptage, il faut être prudent, et éventuellement vérifier sur un exemple : si une variable i passe au total par n valeurs différentes par l'affectation $i \rightarrow i + 1$, cela signifie qu'il y a eu $n - 1$ affectations au total.

Pour cet algorithme par exemple, X passe de 0 à B , il y a donc B exécutions de la boucle **tant que**, et $2 + 2B$ affectations, B multiplications et B additions, mais $B + 1$ comparaisons (la dernière comparaison échoue et fait sortir de la boucle).

On peut proposer un algorithme plus efficace de calcul de puissance, c'est l'algorithme d'*exponentiation rapide*¹, qui a une complexité de $O(\log B)$ multiplications au lieu de $O(B)$ pour calculer A^B . Vous pouvez vous amuser à en chercher un invariant pour démontrer sa correction.

```
Données :  $A, B$  : entiers
Résultat : renvoie la puissance  $A^B$ .
Variables :  $p, i, x$ , entiers.
début
   $p \leftarrow 1; x \leftarrow A; i \leftarrow B;$ 
  tant que  $i \neq 0$  faire
    si  $i \bmod 2 = 1$  alors
       $p \leftarrow p \times x;$ 
       $i \leftarrow i - 1;$ 
    fin
     $x \leftarrow x \times x;$ 
     $i \leftarrow i/2;$ 
  fin
  renvoyer  $p;$ 
fin
```

Algorithme 9 : ExponentiationRapide($d A, B$: entiers) : entier

¹ http://fr.wikipedia.org/wiki/Exponentiation_rapide

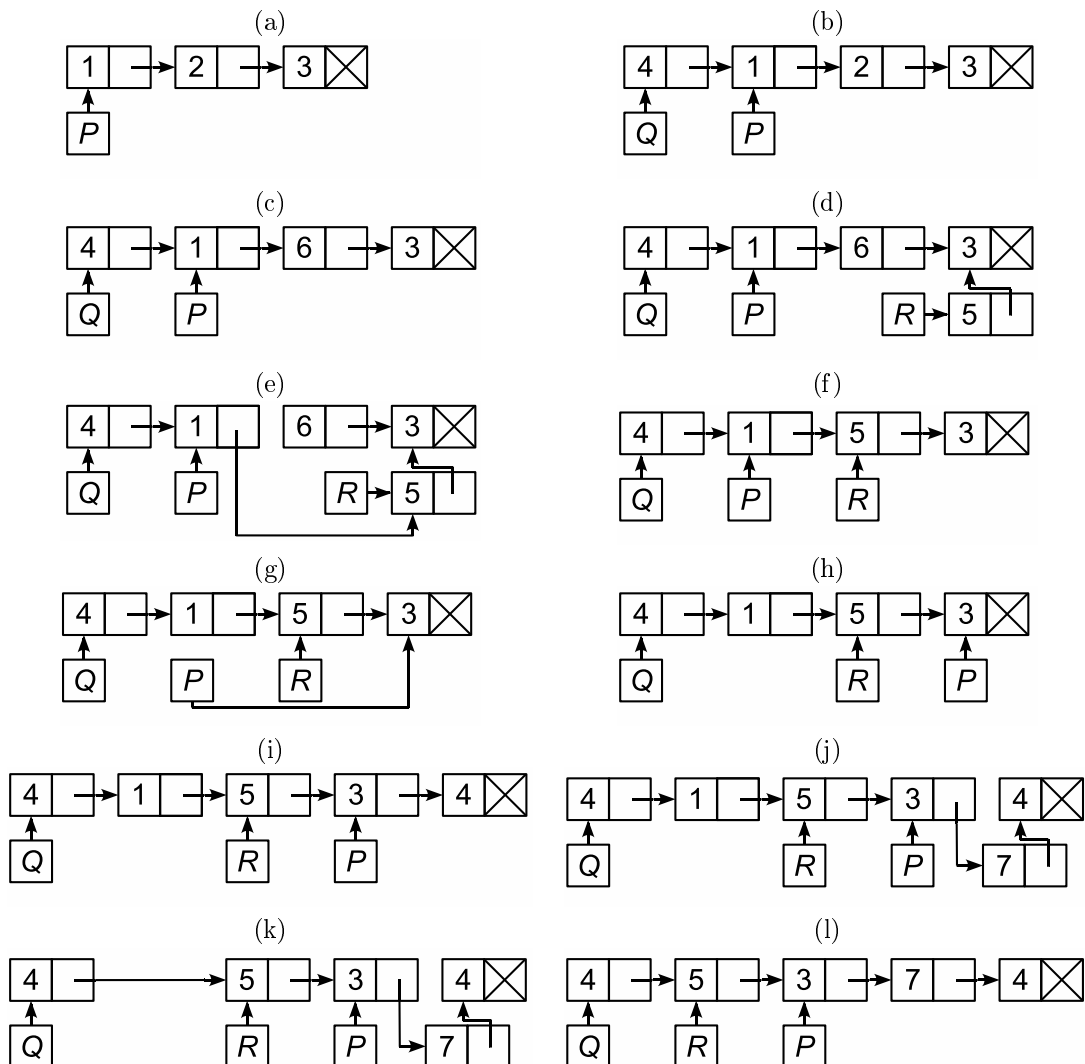
Chapitre 4

Listes chaînées

4.1 Séance 9 (14/11/2007)

4.1.1 Familiarisation avec les listes chaînées

Si vous voulez réfléchir sur des listes chaînées, n'hésitez pas à essayer des idées d'algorithmes, ou exécuter votre algorithme, sur de petits dessins du genre de ceux des figures 4.1.1, en vous rappelant que changer le successeur d'un élément, c'est simplement décoller et recoller ailleurs la flèche qui sort de la case de droite de cet élément.



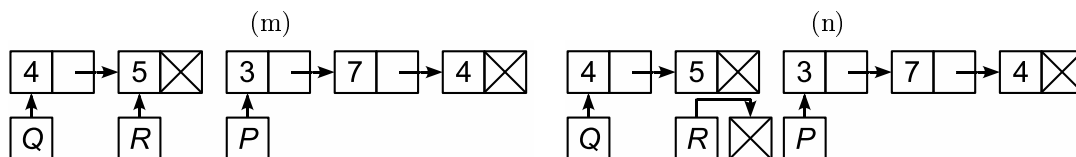


FIG. 4.1 – Utilisations de listes chaînées : après la ligne 1 (a), 2 (b), 3 (c), 4 (d), 5 (e), réorganisation (f), après la ligne 6 (g), réorganisation (h), après la ligne 7 (i), 8 (j), 9 (k), réorganisation (l), après la ligne 10 (m), 11 (n).

4.2 Séance 10 (28/11/2007)

4.2.1 Algorithmes sur les listes chaînées

Les algorithmes qui utilisent une liste chaînée passent la plupart du temps par l'opération consistant à parcourir la liste chaînée, de la même façon que les premiers exercices que nous avons vus sur les tableaux consistent à parcourir le tableau. Pour ces derniers, la structure générale des algorithmes consistait donc à initialiser une variable i indiquant un numéro de case à 1, puis le faire varier à l'aide d'une boucle **pour tout** ou **tant que** jusqu'à un entier n correspondant au nombre de cases du tableau.

Sur les listes chaînées, le parcours peut se faire à l'aide d'une boucle **tant que** comme illustré en figure 10 (on peut aussi effectuer le parcours par une fonction récursive mais il est possible que ce soit plus délicat).

Données : L, \dots : liste simplement chaînée, \dots

Résultat : renvoie un truc après avoir parcouru L .

Variables : Q , liste simplement chaînée.

début

$Q \leftarrow L$;

\dots ;

tant que $Q \neq \text{NULL}$ **faire**

\dots ;

$Q \leftarrow Q \uparrow \text{succ}$;

\dots ;

fin

\dots ;

renvoyer \dots ;

fin

Algorithme 10 : TrucGrâceAUnParcoursDeListe(L : liste simplement chaînée) : truc

Attention à la condition d'arrêt, on pourra préférer s'arrêter après avoir atteint le dernier élément, c'est à dire quand $Q = \text{NULL}$, ou en l'atteignant, c'est à dire quand $Q \uparrow \text{succ} = \text{NULL}$.

4.2.2 Adresse de la dernière cellule d'une liste

Données : L : liste simplement chaînée

Résultat : renvoie l'adresse de la dernière cellule d'une liste

Variables : Q , liste simplement chaînée.

début

$Q \leftarrow L$;

tant que $Q \uparrow \text{succ} \neq \text{NULL}$ **faire**

$Q \leftarrow Q \uparrow \text{succ}$;

fin

renvoyer L ;

fin

Algorithme 11 : DerniereCellule(L : liste simplement chaînée) : liste simplement chaînée

Quand ce n'est pas précisé dans les exercices sur les listes, n est la taille de la liste (son nombre d'éléments).

4.2.3 Premier élément mis à la fin d'une liste

Notons qu'on s'arrête avant d'être arrivé à la fin de la liste, c'est à dire qu'on quitte la boucle dès qu'on atteint le dernier élément.

```
Données :  $L$  : liste simplement chaînée  
Résultat : modifie  $L$  pour déplacer son premier élément en dernière position  
Variable :  $Q$ , liste simplement chaînée.  
début  
   $Q \leftarrow L$ ;  
  si  $Q = NULL$  alors  
    | renvoyer  $NULL$ ;  
  fin  
  tant que  $Q \uparrow succ \neq NULL$  faire  
    |  $Q \leftarrow Q \uparrow succ$ ;  
  fin  
   $Q \uparrow succ \leftarrow \text{créerListe}(L \uparrow \text{info}, NULL)$ ;  
fin
```

Algorithme 12 : PremierALaFin(L : liste simplement chaînée) : liste simplement chaînée

4.2.4 Suppression de tous les éléments de valeur donnée d'une liste

```
Données :  $L$  : liste simplement chaînée,  $e$  : entier  
Résultat : supprime de  $L$  tous les éléments de valeur  $e$   
Variable :  $Q$ , liste simplement chaînée.  
début  
   $Q \leftarrow L$ ;  
  si  $Q = NULL$  alors  
    | renvoyer  $NULL$ ;  
  fin  
  tant que  $Q \uparrow succ \neq NULL$  faire  
    | si  $Q \uparrow \text{info} = e$  alors  
      | Supprimer( $L, Q$ );  
    | fin  
    |  $Q \leftarrow Q \uparrow succ$ ;  
  fin  
fin
```

Algorithme 13 : SupprimeValeur(L : liste simplement chaînée, e : entier)

L'algorithme Supprimer vu en cours a une complexité en $O(n)$, donc SupprimeValeur a une complexité en $O(n^2)$

En utilisant une liste doublement chaînée, l'idée est que la suppression peut se faire en temps constant (comme on a alors accès à l'adresse du dernier élément, il suffit de faire pointer le successeur de cet élément vers le successeur de son successeur). Ainsi la complexité totale de SupprimeValeur en utilisant une liste doublement chaînée serait en $O(n)$.

4.3 Séance 11 (07/12/2007)

4.3.1 Suppression de tous les doublons d'une liste

L'idée de l'algorithme 14 est de stocker la valeur de la case précédente pour la comparer avec la valeur de la case actuelle, mais si elles sont identiques la suppression par l'algorithme Supprimer se fait en $O(n)$, c'est à dire une complexité totale de $O(n^2)$ pour SupprimerDoublons. Il est possible de faire mieux avec l'algorithme 15 qui ne stocke pas la valeur de l'élément précédent, mais carrément l'élément précédent lui-même, permettant d'atteindre une complexité en $O(n)$.

Données : L : liste simplement chaînée

Résultat : Modifie L en supprimant tous ses doublons, c'est à dire ses éléments qui auraient même valeur que leur successeur

Variable : Q , liste simplement chaînée.

```
début
   $Q \leftarrow L$ ;
  si  $L \neq NULL$  alors
    tant que  $Q \uparrow succ \neq NULL$  faire
       $valpred \leftarrow Q \uparrow info$ ;
       $Q \leftarrow Q \uparrow succ$ ;
      si  $Q \uparrow info = valpred$  alors
        Supprimer( $Q, L$ );
      fin
    fin
  fin
fin
```

Algorithme 14 : SupprimeDoublons(L : liste simplement chaînée)

Données : L : liste simplement chaînée

Résultat : Modifie L en supprimant tous ses doublons, c'est à dire ses éléments qui auraient même valeur que leur successeur

Variable : Q , liste simplement chaînée.

```
début
   $Q \leftarrow L$ ;
  si  $L \neq NULL$  alors
    tant que  $Q \uparrow succ \neq NULL$  faire
       $P \leftarrow Q$ ;
       $Q \leftarrow Q \uparrow succ$ ;
      si  $Q \uparrow info = P \uparrow info$  alors
         $P \uparrow succ \leftarrow Q \uparrow succ$ ; //on supprime de  $L$  l'élément pointé par  $Q$ .
      fin
    fin
  fin
fin
```

Algorithme 15 : SupprimeDoublons(L : liste simplement chaînée)

4.3.2 Miroir d'une liste

```
Données :  $L$  : liste simplement chaînée  
Résultat : Renvoie une liste constituée des éléments de  $L$  dans l'ordre inverse  
Variables :  $Q, M$ , listes simplement chaînées.  
début  
   $Q \leftarrow L$ ;  
   $M \leftarrow L$ ;  
  si  $L \neq \text{NULL}$  alors  
     $M \leftarrow \text{créerListe}(L \uparrow \text{info}, \text{NULL})$ ;  
    tant que  $Q \uparrow \text{succ} \neq \text{NULL}$  faire  
       $Q \leftarrow Q \uparrow \text{succ}$ ;  
       $M \leftarrow \text{créerListe}(Q \uparrow \text{info}, M)$ ;  
    fin  
  fin  
  renvoyer  $M$ ;  
fin
```

Algorithme 16 : Miroir(L : liste simplement chaînée) : liste simplement chaînée

On cherche à modifier l'algorithme 16 pour inverser une liste simplement chaînée *en place*, c'est à dire sans créer de nouvelle liste. Pour cela on va faire avancer le long de la liste trois pointeurs sur trois éléments consécutifs de la liste à modifier, pour garder en mémoire leurs adresses et pouvoir faire les modifications.

```
Données :  $L$  : liste simplement chaînée  
Résultat : Modifie  $L$  pour inverser l'ordre de ses éléments  
Variables :  $Q, R, S$ , listes simplement chaînées.  
début  
  si  $L \neq \text{NULL}$  alors  
     $Q \leftarrow L \uparrow \text{succ}$ ;  
     $L \uparrow \text{succ} \leftarrow \text{NULL}$ ;  
    tant que  $Q \neq \text{NULL}$  faire  
       $S \leftarrow Q \uparrow \text{succ}$ ;  
       $Q \uparrow \text{succ} \leftarrow L$ ;  
       $L \leftarrow Q$ ;  
       $Q \leftarrow S$ ;  
    fin  
  fin  
fin
```

Algorithme 17 : Miroir(L : liste simplement chaînée)

4.3.3 Concaténation de deux listes

L'idée est de parcourir la première liste jusqu'à son dernier élément et lui affecter comme successeur le premier élément de la seconde, comme montré dans l'algorithme 18.

4.3.4 Familiarisation avec les listes doublement chaînées

L'idée d'une liste doublement chaînée est que chaque cellule, au lieu de contenir seulement un pointeur vers la cellule suivante, contient aussi un pointeur vers la cellule précédente.

Pour la structure de données à utiliser il y a en fait un détail technique supplémentaire : une sorte de "bloc d'adressage" qui contient l'adresse du premier élément de la liste et l'adresse du dernier élément. Ainsi, si A est une liste doublement chaînée, $A \uparrow \text{info}$ est vide (NULL), $A \uparrow \text{succ}$ désigne son premier élément, $A \uparrow \text{pred}$ désigne son dernier élément, et pour récupérer par exemple la valeur de son premier élément on fait $A \uparrow \text{succ} \uparrow \text{info}$.

Quelles fonctions appliquer pour créer une liste doublement chaînée à 2 éléments, contenant les valeurs 4 et 6? Par exemple on peut procéder comme dans l'algorithme 19.

Données : $L1, L2$: listes simplement chaînées
Résultat : renvoie $L1$ à qui on a concaténé $L2$
 Variable : Q , liste simplement chaînée.

```

début
  | si  $L = NULL$  alors
  | | renvoyer  $L2$ ;
  | fin
  | sinon
  | |  $Q \leftarrow L$ ;
  | | tant que  $Q \uparrow succ \neq NULL$  faire
  | | |  $Q \leftarrow Q \uparrow succ$ ;
  | | fin
  | |  $Q \uparrow succ \leftarrow L2$ ;
  | | renvoyer  $L1$ ;
  | fin
fin

```

Algorithme 18 : Concatène(L : liste simplement chaînée) : liste simplement chaînée

```

 $L \leftarrow \text{créerTriplet}(NULL, NULL, NULL)$ ;
 $Q \leftarrow \text{créerTriplet}(L, 4, L)$ ;
 $L \uparrow succ \leftarrow Q$ ;
 $L \uparrow pred \leftarrow Q$ ;
 $R \leftarrow \text{créerTriplet}(L \uparrow pred, 6, L)$ ;
 $Q \uparrow succ \leftarrow R$ ;
 $L \uparrow succ \leftarrow Q$ ;
 $L \uparrow pred \leftarrow R$ ;

```

Algorithme 19 : Création d'une liste doublement chaînée à 2 éléments contenant 4 et 6.

Sur la liste doublement chaînée ainsi créée, comment supprimer le dernier élément pour obtenir une liste doublement chaînée à 1 élément (contenant donc la valeur 4)? Par exemple on applique l'algorithme 20.

```

 $L \uparrow pred \leftarrow Q$ ;
 $Q \uparrow succ \leftarrow L$ ;

```

Algorithme 20 : Suppression du dernier élément de la liste créée précédemment.

4.4 Exercices non corrigés en TD (28/12/2007)

4.4.1 Miroir d'une liste doublement chaînée

L'idée est de traiter tous les éléments de la liste, y compris le "bloc d'adressage", en inversant leur prédécesseur avec leur successeur. La complexité de l'algorithme 21 est donc en $O(n)$.

4.4.2 Concaténation de deux listes doublement chaînées

L'idée de l'algorithme 22 est de corriger les pointeurs du bloc d'adressage de la première liste, de son dernier élément, ainsi que du premier élément de la seconde liste. Le tout se fait en temps constant, $O(1)$.

Données : L : liste doublement chaînée

Résultat : Renvoie une liste constituée des éléments de L dans l'ordre inverse

Variables : P, Q , éléments d'une liste doublement chaînée.

début

$P \leftarrow L$;

$Q \leftarrow L \uparrow \text{succ}$;

si $Q \neq \text{NULL}$ **alors**

tant que $Q \neq L$ **faire**

 //On traite l'élément P en inversant son successeur stocké dans Q avec son prédécesseur:

$P \uparrow \text{succ} \leftarrow P \uparrow \text{pred}$;

$P \uparrow \text{pred} \leftarrow Q$;

 //On traitera Q , l'ancien successeur de P , à la prochaine étape:

$P \leftarrow Q$;

$Q \leftarrow Q \uparrow \text{succ}$;

fin

 //Il reste à traiter le dernier élément P de la liste, qui a pour successeur $Q = L$

$P \uparrow \text{succ} \leftarrow P \uparrow \text{pred}$;

$P \uparrow \text{pred} \leftarrow Q$;

fin

renvoyer L ;

fin

Algorithme 21 : Miroir(L : liste doublement chaînée) : liste doublement chaînée

Données : $L1, L2$: listes doublement chaînées

Résultat : Renvoie une liste constituée des éléments de $L1$ puis de ceux de $L2$

début

si $L1 \uparrow \text{succ} = \text{NULL}$ **alors**

 | renvoyer $L2$;

fin

sinon si $L2 \uparrow \text{succ} = \text{NULL}$ **alors**

 | renvoyer $L1$;

fin

sinon

$L2 \uparrow \text{succ} \uparrow \text{pred} \leftarrow L1 \uparrow \text{pred}$;

$L1 \uparrow \text{pred} \uparrow \text{succ} \leftarrow L2 \uparrow \text{succ}$;

$L1 \uparrow \text{pred} \leftarrow L2 \uparrow \text{pred}$;

$L1 \uparrow \text{pred} \uparrow \text{succ} \leftarrow L1$;

 renvoyer $L1$;

fin

fin

Algorithme 22 : Concatene($L1, L2$: listes doublement chaînées) : liste doublement chaînée

Chapitre 5

Arbres

5.1 Séance 12 (12/12/2007)

5.1.1 Familiarisation avec les arbres

Un arbre binaire est codé comme une cellule A à trois cases qui représente la racine de l'arbre, contient une valeur $A \uparrow \text{info}$, et un pointeur vers la cellule-racine de son sous-arbre gauche $A \uparrow \text{sag}$, ainsi qu'un pointeur vers la cellule-racine de son sous-arbre droit $A \uparrow \text{sad}$. Les "nœuds au bout des branches" (nœuds de degré ≤ 1) d'un arbre binaire sont appelés feuilles, ils pointent vers NULL à gauche et NULL à droite.

Pour parcourir une liste, on pouvait partir de la tête et suivre les pointeurs vers les successeurs jusqu'à arriver à la fin de la liste et donc un pointeur vers NULL à l'aide d'une boucle **tant que**. Avec un arbre utiliser une boucle **tant que** est beaucoup moins naturel, et on préférera employer des algorithmes récursifs : savoir résoudre un problème sur le sous-arbre gauche et le sous-arbre droit permettre de résoudre le problème sur l'arbre.

Comment créer un arbre de racine étiquetée 2 ayant deux fils étiquetés 1 et 3?
créerArbre(2,créerArbre(1,NULL,NULL),créerArbre(3,NULL,NULL))

5.1.2 Hauteur d'un arbre binaire

Idée : la hauteur d'un arbre binaire est égale à celle du plus haut des sous-arbres gauche et droit de sa racine, augmentée de 1.

Attention à l'initialisation, les nœuds isolés (pointant sur NULL et NULL) sont des arbres de hauteur 0!

Données : A : arbre binaire

Résultat : renvoie la hauteur de l'arbre A

début

```
si  $A = \text{NULL}$  alors
| renvoyer 0;
fin
sinon si  $A \uparrow \text{sag} = \text{NULL}$  et  $A \uparrow \text{sad} = \text{NULL}$  alors
| renvoyer 0;
fin
sinon
| renvoyer  $1 + \max(\text{Hauteur}(A \uparrow \text{sag}), \text{Hauteur}(A \uparrow \text{sad}))$ ;
fin
fin
```

Algorithme 23 : Hauteur(A : arbre binaire) : entier

5.1.3 Nombre de feuilles d'un arbre binaire

Idée : le nombre de feuilles d'un arbre binaire est égale à la somme du nombre de feuilles des sous-arbres gauche et droit de sa racine.

```

Données :  $A$  : arbre binaire
Résultat : nombre de feuilles de l'arbre  $A$ 
début
  | si  $A = NULL$  alors
  |   | renvoyer 0;
  | fin
  | sinon si  $A \uparrow sag = NULL$  et  $A \uparrow sad = NULL$  alors
  |   | renvoyer 1;
  | fin
  | sinon
  |   | renvoyer NombreFeuilles( $A \uparrow sag$ )+NombreFeuilles( $A \uparrow sad$ );
  | fin
fin

```

Algorithme 24 : NombreFeuilles(A : arbre binaire) : entier

5.2 Séance 13 (12/12/2007, 13/12/2007)

5.2.1 Intermède complexité

Attention à ne pas utiliser des fonctions inutiles. Par exemple, si vous parcourez un arbre et qu'à chaque noeud vous appelez une fonction de calcul de hauteur d'arbre, cette dernière étant de complexité linéaire en la taille de l'arbre, votre algorithme global sera déjà au moins de complexité quadratique.

Pour évaluer la complexité d'un algorithme récursif sur un arbre, vous serez peut-être amenés à vous interroger sur son nombre de noeuds n , son nombre d'arêtes a , ou son nombre de feuilles f . Retenez que si un arbre a n feuilles, son nombre de noeuds et son nombre d'arêtes sont en $\Theta(n)$. Détaillons quelques formules qui permettent de le démontrer, ainsi que les petits raisonnements qui permettent de les retrouver.

Tout noeud sauf la racine a un père, on peut donc lui associer l'arête qui vient sur lui. Ainsi, on définit une bijection entre tous les noeuds sauf la racine avec toutes les arêtes. Donc $a = n - 1$. Cette formule est valable pour tout arbre.

Pour la formule sur le nombre de feuilles, on va se restreindre aux arbre strictement binaires, c'est à dire ceux dont aucun noeud interne n'a qu'un fils, en remarquant qu'ils peuvent se construire de la façon suivante. On initialise l'arbre à un noeud isolé (à la fois racine et feuille). Puis on applique la règle suivante : on choisit une feuille, on lui accroche deux feuilles (une "cerise"). Appelons n_k (respectivement a_k, f_k), le nombre de noeuds (respectivement, d'arêtes, de feuilles) après l'ajout de la k -ième cerise. A l'ajout d'une cerise, on enlève une feuille pour en ajouter deux, donc $f_{k+1} = f_k + 1$. De plus on ajoute deux arêtes donc $a_{k+1} = a_k + 2$. Ainsi, comme $a_1 = 2$ et $f_1 = 2$ on peut montrer par récurrence que $a_k = 2k$ et $f_k = k + 1$, donc $a_k = 2f_k - 2$.

Finalement pour tout arbre strictement binaire on a :

$$a = n - 1 = 2f - 2$$

Ainsi, on a bien $a = O(n)$ et $f = O(n)$.

5.2.2 Nombre de nœuds de profondeur fixée d'un arbre binaire

Idée : le nombre de nœuds de profondeur p d'un arbre binaire est égale à la somme du nombre de nœuds de profondeur $p - 1$ dans le sous-arbre gauche de sa racine avec le nombre de nœuds de profondeur $p - 1$ dans le sous-arbre droit de sa racine.

5.2.3 Effeillage d'un arbre binaire

Idée : le nombre de nœuds de profondeur p d'un arbre binaire est égale à la somme du nombre de nœuds de profondeur $p - 1$ dans le sous-arbre gauche de sa racine avec le nombre de nœuds de profondeur $p - 1$ dans le sous-arbre droit de sa racine.

```

Données :  $A$  : arbre binaire,  $p$  : entier
Résultat : renvoie le nombre de nœuds de profondeur  $p$  de l'arbre  $A$ 
début
  | si  $p = 0$  et  $A \neq NULL$  alors
  | | renvoyer 1;
  | fin
  | sinon si  $A = NULL$  alors
  | | renvoyer 0
  | fin
  | sinon
  | | renvoyer NombreNœudsProfondeur( $A \uparrow sag, p - 1$ )+NombreNœudsProfondeur( $A \uparrow sad, p - 1$ );
  | fin
fin

```

Algorithme 25 : NombreNœudsProfondeur(A : arbre binaire, p : entier) : entier

```

Données :  $A$  : arbre binaire
Résultat : modifie  $A$  en l'effeuillant
début
  | si  $A \uparrow sag = NULL$  et  $A \uparrow sad = NULL$  alors
  | |  $A \leftarrow NULL$  // cas où l'arbre de départ est une racine-feuille
  | fin
  | si ( $A \uparrow sag \neq NULL$ ) alors
  | | si ( $A \uparrow sag \uparrow sag = NULL$ ) et ( $A \uparrow sag \uparrow sad = NULL$ ) alors
  | | |  $A \uparrow sag \leftarrow NULL$  // suppression du fils gauche de  $A$ , qui est une feuille;
  | | | fin
  | | | sinon
  | | | | Effeuille( $A \uparrow sag$ );
  | | | | fin
  | | fin
  | | si ( $A \uparrow sad \neq NULL$ ) alors
  | | | si ( $A \uparrow sad \uparrow sag = NULL$ ) et ( $A \uparrow sad \uparrow sad = NULL$ ) alors
  | | | |  $A \uparrow sad \leftarrow NULL$  // suppression du fils droit de  $A$ , qui est une feuille;
  | | | | fin
  | | | | sinon
  | | | | | Effeuille( $A \uparrow sad$ );
  | | | | | fin
  | | | fin
  | | fin
  | fin
fin

```

Algorithme 26 : Effeillage(A : arbre binaire) : entier

Chapitre 6

Autres structures de données

6.1 Séance 14 (14/12/2007)

6.1.1 Notation postfixe d'un arbre

Encore un algorithme récursif assez simple : la notation postfixe d'un arbre, c'est celle de son sous-arbre gauche, suivie de celle du droit, suivie de l'étiquette de sa racine! Attention toutefois, la concaténation se fait en temps linéaire par rapport à la taille de la liste en parallèle. Cette opération peut donc être effectuée un nombre linéaire de fois par rapport à la taille de l'arbre, ce qui donne à l'algorithme 27 une complexité totale quadratique en la taille de l'arbre en entrée.

```
Données : A : arbre
Résultat : Renvoie une liste contenant la notation postfixe correspondant à l'arbre A
début
  si A = NULL alors
    | renvoyer NULL;
  fin
  sinon
    si A ↑ sag = NULL et A ↑ sad = NULL alors
      | renvoyer créerListe(A ↑ info, NULL);
    fin
    sinon si A ↑ sag = NULL alors
      | renvoyer Concaténer(Postfixe(A ↑ sad), créerListe(A ↑ info, NULL));
    fin
    sinon si A ↑ sad = NULL alors
      | renvoyer Concaténer(Postfixe(A ↑ sag), créerListe(A ↑ info, NULL));
    fin
    sinon
      | renvoyer Concaténer(Postfixe(A ↑ sad), Concaténer(Postfixe(A ↑ sad), créerListe(A ↑
      | info, NULL)));
    fin
  fin
fin
```

Algorithme 27 : Postfixe(A : arbre) : liste simplement chaînée

6.1.2 Pile pour l'évaluation d'une expression postfixe

Idée de l'algorithme 28 : on dépile le premier élément de l'expression qui est stockée dans une pile. Si c'est un symbole binaire (de noeud interne), on va récursivement traiter deux fois le reste de la pile, récupérer ainsi deux valeurs et effectuer leur opération par le symbole binaire. Si c'est un entier, on se contente de renvoyer la valeur de l'entier.

```

Données :  $P$  : pile contenant une notation postfixée dont le dernier élément est en haut de la pile
Résultat : Renvoie la valeur de l'expression dont la notation postfixée est stockée dans  $P$ 
Variables :  $R1, R2$  : entiers,  $operation$  : symbole.
début
  si PileVide?( $P$ ) alors
    | renvoyer 0;
  fin
  sinon
     $operation \leftarrow$  sommetPile( $P$ );
    dépiler( $P$ );
    si  $operation = +$  ou  $operation = *$  alors
      |  $R1 \leftarrow$  Evaluer( $P$ ); //  $P$  est alors modifiée!
      |  $R2 \leftarrow$  Evaluer( $P$ );
      | renvoyer  $R1$   $operation$   $R2$ ;
    fin
    sinon
      | renvoyer  $operation$ ; //  $operation$  est alors un entier.
    fin
  fin
fin

```

Algorithme 28 : Evaluer(P : pile) : entier

6.1.3 File pour un parcours en largeur d'arbre

Idée de l'algorithme 29 : ajouter à la file le noeud racine, puis tant qu'il reste des éléments dans la file, enlever le premier élément de la file, et ajouter ses fils en fin de file. Ainsi, tout au long de l'algorithme, la file est constituée d'un ensemble (éventuellement vide) de noeuds d'une certaine profondeur k , suivi d'un ensemble (éventuellement vide) de noeuds de profondeur $k + 1$, ce qui est le principe du *parcours en largeur* (*Breadth-First Search* en anglais). Cet algorithme a une complexité linéaire en la taille de l'arbre à parcourir.

6.1.4 Taille de l'intersection de deux tableaux d'entiers

Dans tous les cas, pour calculer la taille de l'intersection de $T1$ et $T2$, on va mettre dans une liste tout élément apparaissant dans $T1$ et $T2$, puis supprimer les doublons de la liste si besoin, et enfin calculer la taille de la liste.

Idée de l'algorithme 30 : pour chacun des m éléments du premier tableau, on regarde en $O(n)$ s'il est présent parmi les n éléments du second tableau (si oui on l'ajoute à une liste d'éléments de l'intersection des deux tableaux). La complexité totale de cette étape est de $O(mn)$. Il faut alors calculer éliminer les doublons, c'est à dire des éléments égaux du premier tableau qui appartiendraient aussi au second, ou inversement. Pour cela on utilise en particulier l'algorithme 13 pour la fonction **SupprimeValeur**. La complexité de cette étape est quadratique en la taille de la liste d'éléments de l'intersection des deux tableaux, c'est à dire $O(m^2)$. Il reste à calculer la taille de la liste en $O(m)$. Finalement la complexité totale est donc $O(m^2 + mn)$.

Idée de l'algorithme 31 avec un des deux tableaux trié : pour chacun des m éléments du premier tableau, on regarde en $O(\log n)$ s'il est présent parmi les éléments du second tableau qui sont triés (le tri du tableau s'est fait avec une complexité en $O(n \log n)$), par dichotomie. La complexité totale de cette étape est donc en $O(m \log n + n \log n)$, c'est à dire $O(\max(m, n) \log n)$, (et $O(m \log n)$ si on considère que le second tableau est déjà trié) elle est meilleure qu'avec l'idée précédente.

Attention toutefois à la procédure de suppression des doublons que j'ai opportunément omise. Si on la réalise, sa complexité est en $O(m^2)$, ce qui pour répondre précisément à la question posée nous donne une complexité de $O(m^2 + m \log n)$.

Idée de l'algorithme 32 avec les deux tableaux triés : on trie le premier tableau en $O(m \log m)$, le second en $O(n \log n)$, puis on parcourt simultanément les deux tableaux. La complexité totale est en $O(n \log n + m \log m + m + n)$, soit $O(\max(m, n) \log \max(m, n))$, ce qui est encore meilleur que l'idée précédente.

Si en plus on considère que les tableaux sont déjà triés, on obtient alors une complexité en $O(m + n)$.

Données : A : arbre

Résultat : Renvoie une liste chaînée correspondant au parcours en largeur de A

Variables : L , liste chaînée, F , file.

début

$L \leftarrow \text{NULL}$;

$F \leftarrow \text{créerFile}$;

si $A \neq \text{NULL}$ **alors**

ajouterFile(F, A);

tant que *non File Vide?*(F) **faire**

 //On traite le premier élément de la file :

$A \leftarrow \text{têteFile}(F)$;

retirerFile(F);

$L \leftarrow \text{créerListe}(A \uparrow \text{info}, L)$;

 //On ajoute à la fin de la file les fils éventuels, à traiter plus tard :

si $A \uparrow \text{sag} \neq \text{NULL}$ **alors**

 | **ajouterFile**($F, A \uparrow \text{sag}$);

fin

si $A \uparrow \text{sad} \neq \text{NULL}$ **alors**

 | **ajouterFile**($F, A \uparrow \text{sad}$);

fin

fin

fin

 //La liste L a été construite dans le mauvais sens, son premier élément est celui visité en dernier,

 //donc de profondeur maximale dans A :

renvoyer $\text{Miroir}(L)$;

fin

Algorithme 29 : $\text{ParcoursLargeur}(F : \text{file})$: liste simplement chaînée

Données : $T1, T2$: tableaux de respectivement m et n entiers
Résultat : Renvoie le nombre d'entiers présents dans les deux tableaux.
Variables : L , liste chaînée; t , entier.

```

début
   $L \leftarrow \text{NULL}$ ;
  pour  $i$  de 1 à  $m$  faire
     $j \leftarrow 1$ ;
    tant que  $j < n$  et  $T1[i] \neq T2[j]$  faire
       $j \leftarrow j + 1$ ;
    fin
    si  $j < n$  alors
      //  $T1[i] = T2[j]$  donc on ajoute  $T1[i]$  à  $L$  :
       $L \leftarrow \text{créerListe}(T1[i], L)$ ;
    fin
  fin
  //Élimination des doublons de  $L$  :
   $Q \leftarrow L$ ;
  tant que  $Q \neq \text{NULL}$  faire
     $\text{supprimeValeur}(Q \uparrow \text{succ}, Q \uparrow \text{info})$ ;
     $Q \leftarrow Q \uparrow \text{succ}$ ;
  fin
  //Taille de la liste  $L$  :
   $Q \leftarrow L$ ;
   $t \leftarrow 0$ ;
  tant que  $Q \neq \text{NULL}$  faire
     $t \leftarrow t + 1$ ;
     $Q \leftarrow Q \uparrow \text{succ}$ ;
  fin
  renvoyer  $t$ ;
fin

```

Algorithme 30 : TailleIntersectionNaif($T1, T2$: tableaux) : entier

Données : $T1, T2$: tableaux de respectivement m et n entiers, $T2$ est trié
Résultat : Renvoie le nombre d'entiers présents dans les deux tableaux.
Variables : L , liste chaînée; t , entier.

```

début
   $L \leftarrow \text{NULL}$ ;
  pour  $i$  de 1 à  $m$  faire
    si RechercheDicho( $T2, T1[i]$ ) alors
       $L \leftarrow \text{créerListe}(T1[i], L)$ ;
    fin
  fin
  //Élimination des doublons de  $L$  :
   $Q \leftarrow L$ ;
  tant que  $Q \neq \text{NULL}$  faire
     $\text{supprimeValeur}(Q \uparrow \text{succ}, Q \uparrow \text{info})$ ;
     $Q \leftarrow Q \uparrow \text{succ}$ ;
  fin
  //Taille de la liste  $L$  :
   $Q \leftarrow L$ ;
   $t \leftarrow 0$ ;
  tant que  $Q \neq \text{NULL}$  faire
     $t \leftarrow t + 1$ ;
     $Q \leftarrow Q \uparrow \text{succ}$ ;
  fin
  renvoyer  $t$ ;
fin

```

Algorithme 31 : TailleIntersection1Trié($T1, T2$: tableaux) : entier

Données : $T1, T2$: tableaux triés de respectivement m et n entiers

Résultat : Renvoie le nombre d'entiers présents dans les deux tableaux.

Variables : d, t , entier.

début

$i \leftarrow 1$;

$j \leftarrow 1$;

$d \leftarrow 0$;

$t \leftarrow 0$;

tant que $i < m$ et $j < n$ **faire**

si $T1[i] = T2[j]$ **alors**

si $T1[i] \neq d$ **alors**

 // $T1[i]$ n'a pas encore été trouvé dans $T1$ et $T2$:

$t \leftarrow t + 1$;

 // d contient le dernier élément trouvé à la fois dans $T1$ et $T2$:

$d \leftarrow T1[i]$;

fin

$i \leftarrow i + 1$;

$j \leftarrow j + 1$;

fin

sinon si $T1[i] < T2[j]$ **alors**

$i \leftarrow i + 1$;

fin

sinon

$j \leftarrow j + 1$;

fin

fin

 renvoyer t ;

fin

Algorithme 32 : TailleIntersection2Triés($T1, T2$: tableaux) : entier