Correction des TD d'algorithmique de L2

U.E. FLIN 301 - 2008/2009

Philippe Gambette

5 juin 2009

Table des matières

Table des matières 1

1 Révisions, preuves d'arrêt 3
1.1 Séance 1 (29/09/2008) . . . 3
1.1.1 Algorithme 1 - PlusProche . . . 3
1.1.2 Algorithme 2 - Somme? . . . 4
1.2 Séance 2 (01/10/2008) . . . 4
1.2.1 Méthodologie . . . 4
1.2.2 Algorithme 3 - SSTableauVide . . . 4
1.2.3 Algorithme 4 - LignesEgales? . . . 4
1.2.4 Algorithme 5 - InsérerTabTrié . . . 6
1.3 Séance 3 (02/10/2008 ou 21/10/2008) . . . 6
1.3.1 Méthodologie . . . 6
1.3.2 Preuve d'arrêt de l'algorithme 6 . . . 7
1.3.3 Preuve d'arrêt de l'algorithme 7 . . . 7
1.3.4 Preuve d'arrêt de l'algorithme 8 . . . 7

2 Invariants 9
2.1 Séance 4 (08/10/2008) . . . 9
2.1.1 Les invariants, à quoi ça sert ? . . . 9
2.1.2 Invariants de l'algorithme 15 . . . 9
2.1.3 Invariants de l'algorithme 13 . . . 10
2.2 Séance 5 (13/10/2008) . . . 11
2.2.1 Invariant et correction de l'algorithme 8 . . . 11
2.3 Séance 6 (15/10/2008) . . . 12
2.3.1 Correction de l'algorithme 11 . . . 12
2.3.2 Correction de l'algorithme de recherche séquentielle en tableau trié . . . 12

3 Complexité d'algorithmes 14
3.1 Séance 7 (20/10/2008) . . . 14
3.1.1 Méthodologie . . . 14
3.1.2 Complexité de l'algorithme 15 (exercice 1) . . . 14
3.1.3 Complexité de l'algorithme 16' (exercice 3 complexité) . . . 14
3.2 Séance 8 (22/10/2008) . . . 15
3.2.1 Arrêt et complexité de l'algorithme 12 (exercice 2 complexité) . . . 15
3.2.2 Arrêt et complexité de l'algorithme 13 (exercice 2 invariants) . . . 15
3.2.3 Arrêt et complexité de l'algorithme 9 (exercice 3 arrêt) . . . 15
3.2.4 Complexité de la recherche dichotomique (exercice 3 algorithme et preuves) . . . 16
3.3 Séance 9 (03/11/2008) . . . 16
3.3.1 Algorithme indicePrédécesseur (exercice 4 algorithme et preuve) . . . 16
3.3.2 Test de tri d'un tableau (exercice 4 complexité) . . . 16
3.4 Séance 10 (05/11/2008) . . . 17
3.4.1 Algorithme d'évaluation d'un polynôme (exercice 5 complexité) . . . 17
3.4.2 Algorithme de plus long préfixe suffixe (exercice 7 complexité) . . . 17
3.4.3 Exponentiation rapide (exercice 6 complexité) . . . 18
3.5 Séance 11 (12/11/2008) . . . 18
3.5.1 Exponentiation rapide - version récursive (exercice 6 complexité) . . . 18

3.5.2 Problème SomMax (complexité) . . . 18

4 Listes chaînées **22**
4.1 Séance 12 (17/11/2008) . . . 22
4.1.1 Algorithmes sur les listes chaînées . . . 22
4.1.2 Familiarisation avec les listes chaînées . . . 22
4.1.3 Adresse de la dernière cellule d'une liste . . . 22
4.2 Séance 13 (19/11/2008) . . . 23
4.2.1 Premier élément mis à la fin d'une liste . . . 23
4.2.2 Suppression de tous les éléments de valeur donnée d'une liste . . . 23
4.3 Séance 14 (26/11/2008) . . . 26
4.3.1 Suppression de doublons consécutifs d'une liste . . . 26
4.3.2 Concaténation de deux listes simplement chaînées . . . 26
4.3.3 Concaténation de deux listes doublement chaînées . . . 26

5 Arbres **28**
5.1 Fin de la séance 14 (26/11/2008) . . . 28
5.1.1 Familiarisation avec les arbres . . . 28
5.2 Séance 15 (03/12/2008) . . . 28
5.2.1 Hauteur d'un arbre binaire . . . 28
5.2.2 Nombre de feuilles d'un arbre binaire . . . 28
5.2.3 Effeuillage d'un arbre binaire . . . 29

6 Tris **30**
6.1 Séance 16 (10/12/2008) . . . 30
6.1.1 Taille de l'intersection de deux tableaux d'entiers . . . 30
6.1.2 Test de l'existence de doublons . . . 32
6.2 Séance 17 (17/12/2008) . . . 32
6.2.1 Tri par fusion . . . 32
6.2.2 Tri de notes . . . 32
6.2.3 Tri par base . . . 32

Avertissement

Ce document peut contenir des erreurs, indiquez-les moi à `gambette@lirmm.fr` (ceci pourrait vous permettre de gagner un bonus pour la note de participation). 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.

Les cours d'algorithmique de L1 sont disponibles sur l'espace pédagogique (unité FLIN 101).
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 (29/09/2008)

1.1.1 Algorithme 1 - PlusProche

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 $\forall i \in [\![1, n]\!]$, $|x - PP| \leq |x - T[i]|$.

```
début
    PP ← T[1];
    pour tous les i de 2 à n faire
        si |x − T[i]| < |x − PP| alors PP ← T[i];
    fin
fin
```

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

On pouvait aussi procéder en utilisant une boucle *Tant que*. Dans ce cas, attention à bien initialiser la variable de boucle, et l'incrémenter à chaque itération.

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 $\forall i \in [\![1, n]\!]$, $|x - PP| \leq |x - T[i]|$.

```
début
    PP ← T[1];
    i ← 2;
    tant que i ≤ n faire
        si |x − T[i]| < |x − PP| alors PP ← T[i];
        i ← i + 1;
    fin
fin
```

Algorithme 2 : PlusProche(**d** $T[1..n]$: tableau d'entiers, **d** x : entier, **r** PP : entier)

Pensez à vérifier votre algorithme sur un exemple si vous avez le temps. Vous devez aussi être capable de le décrire en quelques phrases, par exemple pour celui-là : on parcourt les cases du tableau en gardant en mémoire à chaque étape le meilleur élément trouvé jusqu'à maintenant pour le comparer avec l'élément de la case courante.

Remarquez aussi que la condition $|x - T[i] < |x - PP|$ assure qu'on renverra le premier élément du tableau qui répond à la question. On aurait pu renvoyer le dernier élément du tableau qui répond à la question en remplaçant cette condition par $|x - T[i] \leq |x - PP|$.

1.1.2 Algorithme 2 - Somme?

En considérant que **renvoyer** permet de terminer l'algorithme en arrêtant les boucles en cours, une solution est proposée dans l'algorithme 3

```
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 ∈ [1..n] 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 3 : Somme?(**d** $A[1..n]$: tableau d'entiers, **d** $B[1..n]$: tableau d'entiers, **d** Som : entier) : booléen

1.2 Séance 2 (01/10/2008)

1.2.1 Méthodologie

On vous demande un algorithme que vous ne savez pas comment écrire ? Prenez un exemple et tentez de voir intuitivement dessus comment résoudre le problème. Tentez alors de généraliser ces opérations pour obtenir les principes généraux de votre algorithme. Puis traduisez ces principes en "pseudo-code", le langage utilisé pour décrire un algorithme.

Si besoin, quand vous cherchez les principes de base de votre algorithme, décomposez le problème en sous-problèmes que vous traitez indépendamment, comme pour l'algorithme *LignesEgales?*.

1.2.2 Algorithme 3 - SSTableauVide

Faire un dessin de ce tableau à double entrée de booléens (c'est à dire un matrice booléenne) peut être utile pour visualiser le problème demandé. On indiquera sur le tableau les valeurs l_1, l_2, c_1, c_2 (c_k indique bien sûr un numéro de colonne, et l_k un numéro de ligne) ainsi que la progression des variables qui permettent de parcourir le sous-tableau (i et j dans l'algorithme 4).

```
Données : T[1..n, 1..n] un tableau de booléens, l1, l2, c1, c2 quatre entiers tels que 1 ≤ l1 ≤ l2 ≤ n et
          1 ≤ c1 ≤ c2 ≤ n.
Résultat : Renvoie Vrai si le sous-tableau T[l1..l2, c1..c2] est vide (toutes les cases sont à Faux), renvoie
           Faux sinon.
début
|  pour tous les i de l1 à l2 faire
|  |  pour tous les j de c1 à c2 faire
|  |  |  si T[i, j] alors renvoyer Faux;
|  |  fin
|  fin
|  renvoyer Vrai;
fin
```

Algorithme 4 : SousTabVide?(**d** $T[1..n, 1..n]$: tableau d'entiers, **d** l_1, l_2, c_1, c_2 : entiers) : booléen

1.2.3 Algorithme 4 - LignesEgales?

$T[1..n, 1..n]$ est un tableau à double entrée rempli de Vrai ou Faux, $T[i, j]$ représente l'entier à la ligne i et à la colonne j.

Si vous ne voyez pas comment procéder, vous pouvez décomposer votre algorithme LignesEgales? en :

- un algorithme **CoupleLignesEgales?** (algorithme 5) qui teste si deux lignes données, la i-ième et la j-ième, sont égales.
- un algorithme **LignesEgales1?** (algorithme 6) qui, pour tout couple de lignes différentes, appelle l'algorithme **CoupleLignesEgales?**.

```
Données : T[1..n, 1..n] un tableau de 0 et de 1, i et j deux entiers tels que 1 ≤ i ≤ n et 1 ≤ j ≤ n
Résultat : Renvoie Vrai si et seulement si les lignes i et j de T sont identiques, c'est à dire ssi
           ∀k ∈ ⟦1, n⟧, T[i, k] = T[j, k].
début
    tant que k ≤ n et T[i, k] = T[j, k] faire
        si k = n alors
            renvoyer Vrai;
        sinon
            k ← k + 1;
        fin
    fin
    renvoyer Faux;
fin
```

Algorithme 5 : CoupleLignesEgales?(**d** $T[1..n, 1..n]$: tableau d'entiers, i : entier, j : entier) : booléen

Remarquez que la boucle **Tant que** de l'algorithme **CoupleLignesEgales?** permet de l'arrêter dès qu'on s'aperçoit que la k-ième case est différente dans les lignes i et j, contrairement à la solution moins élégante mais plus simple qui aurait consisté à utiliser une boucle **Pour** : pour k de 1 à n, si $T[i, k] = T[j, k]\ldots$

```
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
           ∃i ≠ j ∈ ⟦1, n⟧/ ∀k ∈ ⟦1, n⟧, T[i, k] = T[j, k].
début
    pour tous les i de 1 à n − 1 faire
        pour tous les j de i + 1 à n faire
            si CoupleLignesEgales?(T, i, j) alors
                renvoyer Vrai;
            fin
        fin
    fin
    renvoyer Faux;
fin
```

Algorithme 6 : LignesEgales1?(**d** $T[1..n, 1..n]$: tableau d'entiers) : booléen

Remarquez dans l'algorithme **LignesEgales1?** les valeurs parcourues par les variables i et j : ceci permet en fait d'atteindre toutes les valeurs des couples (i, j) tels que $i < j$. On évite donc de tester si la ligne $j = 1$ est égale à la ligne $i = 2$ après avoir testé si la ligne $i = 1$ est égale à la ligne $j = 2$. Cela permet donc de faire en gros deux fois moins d'étapes que la solution plus simple mais moins élégante suivante : pour tout i de 1 à n, pour tout j de 1 à n, si $i \neq j$ alors ...

Une fois que vous avez trouvé les deux algorithmes permettant de résoudre chaque partie décomposée du problème, vous pouvez les réunir dans l'algorithme 7, en faisant bien attention au fait que **renvoyer** stoppe l'algorithme (et donc il ne faut pas réécrire dans **LignesEgales?** le "**renvoyer** `Faux`" de l'algorithme **CoupleLignesEgales?**).

Pour approfondir : 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 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 représentant `Faux` par 0 et `Vrai` par 1 (ainsi le tableau T contient seulement des 0 et des 1) et en commençant par trier T par un tri par base (aussi appelé tri radix, voir algorithme 41), avant de comparer chaque ligne avec la suivante, pour obtenir une complexité totale en $O(n^2)$.

```
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
           ∃i ≠ j ∈ ⟦1, n⟧/ ∀k ∈ ⟦1, n⟧, T[i, k] = T[j, k].
début
    pour tous les i de 1 à n − 1 faire
        pour tous les j de i + 1 à n faire
            k ← 1;
            tant que k ≤ n et T[i, k] = T[j, k] faire
                si k = n alors
                    renvoyer Vrai;
                sinon
                    k ← k + 1;
                fin
            fin
        fin
    fin
    renvoyer Faux;
fin
```

Algorithme 7 : LignesEgales?(**d** $T[1..n, 1..n]$: tableau d'entiers) : booléen

1.2.4 Algorithme 5 - InsérerTabTrié

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 trié d'entiers.
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 ← 1;
    tant que e > T[i] et i ≤ m faire
        i ← 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 ← n;
    tant que j ≥ i faire
        T[j + 1] = T[j];
    fin
    T[i] = e;
fin
```

Algorithme 8 : InsérerTabTrié(**dr** $T[1..n]$: tableau d'entiers, **d** m : entier, **d** e : entier)

1.3 Séance 3 (02/10/2008 ou 21/10/2008)

1.3.1 Méthodologie

Quand on vous demande de prouver l'arrêt d'un algorithme, vous pouvez avoir l'impression qu'on peut le justifier en décrivant la partie de l'algorithme qui concerne l'arrêt. Le problème est que vous paraphrasez généralement l'algorithme avec force périphrases. Il est impossible de voir si vous avez vraiment compris dans le cas où vos explications seraient embrouillées.

Bref, pour éviter cela, il existe une méthode formelle que vous devez appliquer. Il s'agit de construire (ou d'identifier) une expression calculée à partir des variables de l'algorithme, qui décroît strictement tout au long de l'algorithme, et qui est à valeurs entières et positives. Cette expression n'apparaît pas nécessairement parmi celles calculées dans l'algorithme, il vous faudra parfois jongler avec les variables et des additions, soustractions, multiplications, fractions, valeurs absolues, etc., pour construire cette expression.

Une fois que vous avez trouvé une bonne expression candidate. Appelons u_k la valeur de cette expression à

la k-ième itération de l'algorithme. Il faut montrer qu'elle vérifie bien les propriétés demandées. Le fait qu'elle est à valeurs dans $\mathbb{N}$ découle le plus souvent directement de la définition. Pour la décroissance stricte, il s'agit de montrer que si la $k+1$-ième itération de l'algorithme existe, alors $u_{k+1} < u_k$, ce que l'on fait :

- soit en montrant $u_{k+1} - u_k < 0$
- soit, si $u_k > 0$, en montrant que $u_{k+1}/u_k < 1$, ce qui peut être plus facile dans certains cas (si u_k est défini avec des multiplications ou divisions par exemple).

Si u_k est bien une expression à valeurs entières positives qui décroît strictement à chaque pas (= intération) de l'algorithme, on est certain que l'algorithme ne peut avoir un nombre de pas illimité, c'est à dire qu'il a un nombre de pas limité, et donc qu'il s'arrête.

Pour finir, remarquons que pour désigner l'expression strictement décroissante à valeurs entières, nous utilisons le formalisme des suites. La première étape de votre preuve d'arrêt (de même que les preuves d'invariants du chapitre suivant) sera donc toujours de déduire de l'algorithme des relations sur la valeur des variables au $k+1$-ème pas en fonction du k-ième. Pour toute variable x, vous appelez donc x_k sa valeur à la fin du k-ième pas. Puis vous utiliserez les affectations de l'algorithme pour exprimer la relation entre x_k+1 et x_k. Par exemple si à chaque pas, la seule instruction concernant x est : `si x>0 alors x←x-1`, vous traduisez et obtenez cette propriété sur la suite $(x_k)_{k\geq 0}$:

- si $x_k > 0$ alors $x_{k+1} = x_k - 1$
- sinon, $x_{k+1} = x_k$ (ne pas oublier ça !)

1.3.2 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.

1.3.3 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 $0 \leq 7 - n_k - 1 = 7 - n_{k+1} < 7 - n_k$ donc $0 \leq u_k + 1 < u_k$. Donc la suite (u_k) est décroissante strictement, et à valeurs entières. Donc elle ne peut prendre qu'un nombre fini de valeurs, donc l'algorithme s'arrête.

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.

1.3.4 Preuve d'arrêt de l'algorithme 8

```
   Données : N, M ∈ ℕ*
   Résultat : renvoie pgcdN, M.
 1 Variables : n, m entiers
 2 début
 3 |  n ← N; m ← M;
 4 |  tant que n ≠ 0 et m ≠ 0 faire
 5 |  |  si m > n alors
 6 |  |  |  m ← m − n
 7 |  |  sinon
 8 |  |  |  n ← n − m
 9 |  |  fin
10 |  fin
11 |  renvoyer m + n;
12 fin
```

Algorithme 9 : PGCD(**d** N, M : entier) : entier

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 : $\forall k, m_k > n_k \Rightarrow \begin{cases} m_{k+1} = m_k - n_k \\ n_{k+1} = n_k \end{cases}$

`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 $\forall k, m_k \geq 0$.

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 entier.

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

Montrons maintenant que m décroit 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 ne peut donc prendre qu'un nombre fini de valeurs, donc l'algorithme s'arrêtera.

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

Chapitre 2

Invariants

2.1 Séance 4 (08/10/2008)

2.1.1 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.

Précisons qu'une démonstration par récurrence permet de montrer qu'une propriété $P(k)$ est valable pour tout k de $\mathbb{N}$. Or si l'on parle de la k-ième itération, ou boucle, d'un algorithme, on n'est pas sûr qu'elle existe, l'algorithme s'est peut-être arrêté avant d'arriver à sa k-ième boucle. Bref, pour être totalement rigoureux, il faudrait dire que la propriété $P(k)$ est valable pour tout k de $\mathbb{N}$ tel que la k-ième itération de l'algorithme est effectuée. Nous abrégerons (et vous êtes autorisés, et même encouragés, pour ne pas perdre de temps et surcharger vos copies, à le faire) en "valable pour tout k de $\mathbb{N}$".

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, $A - B \geq 0$). C'est même très fortement conseillé pour éviter des rédactions douteuses voire complètement fausses...

2.1.2 Invariants de l'algorithme 15

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 intialisée 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, $M_0 = N = N - 0^2 = N - I_0^2$.

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

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

Donc la propriété est intialisée 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 :

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

2.1.3 Invariants de l'algorithme 13

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

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

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

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

Donc la propriété est intialisée 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 $\forall i \in \mathbb{N}, B_i = 3(X - C_i)^2$.

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

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

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

Donc la propriété est intialisée 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 $\forall i \in \mathbb{N}, Z_i = (X - C_i)^3$.

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

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

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

Donc la propriété est intialisée 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 (13/10/2008)

2.2.1 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 fin de 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.

Appelons $D_k = \{d \in \mathbb{N} / d | m_k \text{ et } d | n_k\}$ l'ensemble des diviseurs communs aux valeurs des variables m et n à la fin de la k-ième itération de la boucle **tant que**. On cherche donc à montrer que $\forall k \in \mathbb{N}$, $D_k = D_{k+1}$. Rappelons que pour montrer une égalité d'ensemble, il faut montrer que tout élément de l'un est dans l'autre, et vice versa.

Supposons qu'il existe $q \in D_k$, c'est à dire q vérifiant : $\exists m', n' \in \mathbb{N}$ tels que $m_k = m'q$ et $n_k = n'q$. Si $m_k > n_k$, 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}. On a le même résultat si $m_k \leq n_k$. Donc dans tous les cas, $q \in D_{k+1}$

Inversement, considérons $q \in D_{k+1}$. Alors $\exists m', n' \in \mathbb{N}$ tels que $m_{k+1} = m'q$ et $n_{k+1} = n'q$. Si $m_k > n_k$, alors $n'q = n_{k+1} = n_k$ donc n_k est aussi divisible par q. De plus, $m'q = m_{k+1} = m_k - n_k$ donc $m_k = m'q + n_k = m'q + n'q = (m' + n')q$ donc m_k est aussi divisible par q. La preuve fonctionne de façon symétrique pour le cas $m_k < n_k$. Finalement, dans tous les cas, $q \in D_k$.

Ainsi on a montré que l'ensemble des diviseurs communs reste constant tout au long de l'algorithme, soit D_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 PGCD$(n, 1)$.

2.3 Séance 6 (15/10/2008)

2.3.1 Correction de l'algorithme 11

Rappelons que la suite de Fibonacci est définie par récurrence de la façon suivante :
- fibo(0)=0
- fibo(1)=0
- fibo(i)=fibo($i-1$)+fibo($i-2$)

On commence par voir sur un exemple ce que calcule l'algorithme, en appelant i_k (respectivement $r1_k$, $r2_k$, $r3_k$) 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
$r3_i$	×	0	1	1	2	3	5
$r1_i$	0	1	1	2	3	5	8
$r2_i$	1	1	2	3	5	8	13
i	0	1	2	3	4	5	6

Ceci nous permet de supposer que $r1_i = \text{fibo}(i)$, $r2_i = \text{fibo}(i+1)$ et $r3_i = \text{fibo}(i-1)$. 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 $\forall i \in \mathbb{N}, r1_i = \text{fibo}(i)$.

Ⓘ au rang 1, $r1_1 = 1 = \text{fibo}(1)$.

Ⓗ soit $i \in \mathbb{N}$, $\forall k \leq i, r1_k = \text{fibo}(k) \Rightarrow r1_{i+1} = \text{fibo}(i+1)$?

$r1_{i+1} = r2_i = r3_i + r2_{i-1} = r1_i + r1_{i-1} = \text{fibo}(i+1)$.

Ainsi l'égalité est initialisée et héréditaire, elle est donc vraie pour tout i et donc $r1_i = \text{fibo}(i)$.

On peut en déduire les deux autres inégalités : $\forall i \in \mathbb{N}, r2_i = r1_{i+1} = \text{fibo}(i+1)$, et $\forall i \in \mathbb{N}^*, r3_i = r1_{i-1} = \text{fibo}(i-1)$, 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 $\forall i \in \mathbb{N}, r1_i = \text{fibo}(i)$, $r2_i = \text{fibo}(i+1)$, et $r3_i = \text{fibo}(i-1)$.

Ⓘ au rang 1, $r1_1 = 1 = \text{fibo}(1)$, $r2_1 = 1 = \text{fibo}(2)$, et $r3_1 = 0 = \text{fibo}(0)$.

Ⓗ soit $i \in \mathbb{N}$, $r1_i = \text{fibo}(i)$, $r2_i = \text{fibo}(i+1)$, et $r3_i = \text{fibo}(i-1) \Rightarrow r1_{i+1} = \text{fibo}(i+1)$, $r2_{i+1} = \text{fibo}(i+2)$, et $r3_{i+1} = \text{fibo}(i)$?

$r1_{i+1} = r2_i = \text{fibo}(i+1)$ (par h.r.). $r3_{i+1} = r1_i = \text{fibo}(i)$ (par h.r.). $r2_{i+1} = r3_{i+1} + r2_i = \text{fibo}(i)+ \text{fibo}(i+1)$ (par égalité précédente et h.r.) donc $r2_{i+1} = \text{fibo}(i+2)$.

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.

2.3.2 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 ← 1;
  tant que i ≤ N et T[i] < e faire
    i ← i + 1;
  fin
  renvoyer (i ≤ N et T[i] = e);
fin
```

Algorithme 10 : 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'hyptohèse de récurrence on obtient $\forall k \geq 1, \forall j \in [1, i_k], T[j] < e$.

La propriété étant initalisée 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 ($i \leq N$ 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 ($i \leq N$ 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 débarasser de ce qui suit le « *D'autre part* » (cela concerne les $i - 1$ premières cases qui ne sont alors pas définies puisque $i = 1$).

Chapitre 3

Complexité d'algorithmes

3.1 Séance 7 (20/10/2008)

3.1.1 Méthodologie

La complexité d'un algorithme permet d'exprimer formellement sa rapidité en fonction de la taille des données en entrée. Pour l'évaluer finement, on compte précisément le nombre d'opérations. 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 $i \to i+1$, cela signifie qu'il y a eu $n-1$ affectations au total.

Le plus souvent on se contente d'ordres de grandeurs avec la notation de Landau : $O(\dots)$. Cette notation a une définition mathématique à base de limites, en pratique il faut retenir qu'elle correspond au **terme dominant** de l'expression du nombre d'opérations en fonction de la taille des données en entrée. Si un algorithme effectue $5n^3 + 2n$ opérations pour une entrée de taille n, le terme dominant est n^3, on dit donc qu'il y a $O(n^3)$ opérations, et que l'algorithme a une complexité cubique. Pour $O(n)$, la complexité est linéaire, $O(n^2)$, la complexité est quadratique, $O(\log n)$, la complexité est logarithmique.

Si l'algorithme fait $\sqrt{n}+\log_{10} n$ étapes dans le pire des cas, quel est l'ordre de grandeur de sa complexité ? Le terme dominant est $\sqrt{n}$ (pour avoir une intuition à ce sujet, comparer $\sqrt{1000000} = 1000$ à $\log_{10} 1000000 = 6$), donc c'est $O(\sqrt{n})$.

3.1.2 Complexité de l'algorithme 15 (exercice 1)

Comme l'algorithme 15 contient une boucle **tant que**, il s'agit de compter en fonction des données le nombre d'itérations de cette boucle.

Pour cet algorithme, 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).

Pour approfondir : 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.

3.1.3 Complexité de l'algorithme 16' (exercice 3 complexité)

Il s'agit de compter le nombre d'itérations de chacune des boucles "pour". La première est faite N fois. Celle à l'intérieur est effectuée 1 fois pour $I = 1$, 2 fois pour $I = 2$, ... N fois pour $I = N$.

Finalement, le nombre total d'exécutions de la ligne "$R \leftarrow R+1$;" est donc $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$ (pour calculer, utiliser le fait que c'est la série d'une suite arithmétique, ou bien retrouver la formule à l'aide d'un dessin). Il y a donc $\frac{N(N+1)}{2} + 1$ affectations dans les lignes 1 et 2, et $N(N+1) + 1 + N = (N+1)^2$ en comptant les affectations "chachées" dans les boucles **Pour**. La complexité en temps de l'algorithme est donc en $O(N^2)$, c'est à dire quadratique.

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

```
Données : A, B : entiers
Résultat : renvoie la puissance A^B.
Variables : p, i, x, entiers.
début
  p ← 1; x ← A; i ← B;
  tant que i ≠ 0 faire
    si i modulo 2 = 1 alors
      p ← p × x;
      i ← i − 1;
    fin
    x ← x × x;
    i ← i/2;
  fin
  renvoyer p;
fin
```

Algorithme 11 : ExponentiationRapide(d A, B : entiers) : entier

3.2 Séance 8 (22/10/2008)

3.2.1 Arrêt et complexité de l'algorithme 12 (exercice 2 complexité)

Soit M_k (resp. Z_k) la valeur de M (resp. Z) après la k-ième itération de la boucle **Tant que**. $M_{k+1} - M_k = -Z_k < 0$ donc (M_k) est une suite strictement décroissante à valeurs entières, donc elle ne peut prendre qu'un nombre fini de valeurs, et donc l'algorithme s'arrête.

On pouvait aussi considérer la suite $u_k = M_k - Z_k$ pour obtenir une suite strictement décroissante et à valeurs entières.

Comme I est initialisé à 0, incrémenté de 1 à chaque itération de la boucle **Tant que**, et qu'en fin d'algorithme il vaut $\lfloor\sqrt{I}\rfloor$, l'algorithme a une complexité en $O(\sqrt{I})$.

3.2.2 Arrêt et complexité de l'algorithme 13 (exercice 2 invariants)

Soit C_k la valeur de C après la k-ième itération de la boucle **Tant que**. $C_{k+1} - C_k = -1 < 0$ donc (C_k) est une suite strictement décroissante à valeurs entières, donc elle ne peut prendre qu'un nombre fini de valeurs, et donc l'algorithme s'arrête.

Comme C est initialisé à X, décrémenté de 1 à chaque itération de la boucle **Tant que**, et qu'en fin d'algorithme il vaut 0, l'algorithme a une complexité en $O(X)$.

3.2.3 Arrêt et complexité de l'algorithme 9 (exercice 3 arrêt)

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, $(n-1, p-1)$.

La valeur de n décroît strictement à chaque appel récursif et elle est entière. Donc dans chaque branche de l'arbre (binaire) d'exécution de l'algorithme, n ne peut prendre qu'un nombre fini de valeurs. Ainsi, toutes les branches de cet arbre binaire ont une profondeur finie donc l'arbre d'exécution est fini, donc il n'y a qu'un nombre fini d'appels récursifs, et donc l'algorithme s'arrête.

3.2.4 Complexité de la recherche dichotomique (exercice 3 algorithme et preuves)

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, c'est à dire qu'on veut faire un léger changement d'invariant, il suffit donc de remplacer dans l'algorithme « $> e$ » par « $\geq e$ ».

La complexité de cet algorithme de recherche dichotomique est logarithmique. Pour s'en convaincre, on peut représenter l'algorithme par un arbre de recherche : à chaque appel de l'algorithme, on coupe l'intervalle de recherche en deux, puis on recherche récursivement soit dans la partie de gauche, soit dans celle de droite. L'arbre est donc binaire (chaque étape donne naissance à deux étapes possibles). Si n est une puissance de 2, alors il a n feuilles, sa profondeur est donc $\log_2(n)$. Or la complexité de l'algorithme, le nombre d'étapes, correspond justement à la profondeur de l'arbre, et chaque étape se fait en temps constant, la complexité en temps est donc en $O(\log n)$.

Si n n'est pas une puissance de 2, on considère le nombre n' qui est la puissance de 2 juste supérieure à n (c'est à dire $n' = 2^{\lfloor \log_2(n) \rfloor + 1}$), et l'arbre d'exécution a donc au plus n' feuilles, et la complexité de l'algorithme de recherche dichotomique est en $O(\log_2 n')$, c'est à dire en $O(\log n)$.

3.3 Séance 9 (03/11/2008)

3.3.1 Algorithme indicePrédécesseur (exercice 4 algorithme et preuve)

En pratique, pour écrire ce genre d'algorithmes, on écrit la structure générale permettant d'effectuer la recherche dichotomique, puis on vérifie et on ajuste les paramètres des tests, la condition d'arrêt, et les mises à jour des variables. Pour ce faire, il est utile de représenter l'invariant qui indique la situation à laquelle on veut aboutir, et d'en déduire ces différents paramètres à ajuster.

```
Données : T[1..n] : un tableau d'entiers trié par ordre croissant, x : un entier
Résultat : renvoie 0 si tous les éléments de T sont supérieurs ou égaux à x, sinon renvoie le plus grand
           indice i ∈ [1..n] tel que T[i] < x.
Variables : Deb, Fin, Mil, entiers.
début
  | Deb ← 1; Fin ← n;
  | tant que Deb ≤ Fin faire
  |   | Mil ← (Deb + Fin) div 2;
  |   | si T[Mil] < x alors
  |   |   | Deb ← Mil + 1;
  |   | sinon
  |   |   | Fin ← Mil − 1;
  |   | fin
  | fin
  | renvoyer Fin;
fin
```

Algorithme 12 : indicePred(d $T[1..n]$: tableau d'entiers, d x : entier) : entier

La formule suivante, qui peut aussi être représentée sous forme d'un schéma sur le dessin du tableau trié, est un invariant : $\forall i \in [1..Deb[, \forall j \in]Fin..n], T[i] < x \leq T[j]$.

La complexité est bien sûr logarithmique.

3.3.2 Test de tri d'un tableau (exercice 4 complexité)

Le nombre minimum d'itérations, 1, est atteint si $T[1] > T[2]$, le nombre maximum, $N - 1$, est atteint si T est trié. Dans le pire des cas, on passe N fois par la ligne 1 (puisque I passe de 1 à $N + 1$). Comme la condition d'arrêt est une conjonction "A et B", à sa dernière itération, A étant évalué faux, "A et B" est nécessairement faux, donc B n'est pas testé. Il y a donc finalement $N - 1$ comparaisons de type $T[I] \leq T[I + 1]$ dans le pire des cas. L'algorithme a donc une complexité en $O(N)$.

3.4 Séance 10 (05/11/2008)

3.4.1 Algorithme d'évaluation d'un polynôme (exercice 5 complexité)

Naïvement, on est tenté de calculer X^i pour tout i et de le multiplier par a_i. Si l'on calcule naïvement X^i, on le fait en $O(i)$ et la complexité de l'algorithme d'évaluation du polynôme en X est donc quadratique. Si on calcule X^i avec l'algorithme d'exponentiation rapide présenté dans l'exercice 6 qui suit, on le fait en $O(\log i)$, soit une complexité golable $O(n \log n)$.

On peut faire mieux et atteindre une complexité linéaire en stockant dans une variable R les valeurs de X^i calculées et en les utilisant pour calculer X^{i+1}.

Données : X : un entier, $a[0..n]$: un tableau de $n+1$ entiers correspondant aux coefficients du polynôme.
Résultat : renvoie la valeur $R = \sum_{i=0}^{n} a_i X^i$ du polynôme en X.
Variables : R, P, des entiers.
début
  $R \leftarrow 0; P \leftarrow 1;$
  pour tous les i *de* 0 *à* n **faire**
    $R \leftarrow R + a[i] \times P;$
    $P \leftarrow P \times X;$
  fin
  renvoyer R;
fin

Algorithme 13 : EvaluationPolynome(d X : entier, d $a[0..n]$: tableau d'entiers) : entier

3.4.2 Algorithme de plus long préfixe suffixe (exercice 7 complexité)

Commençons par une version où on commence par écrire le sous-problème consistant à tester l'égalité du préfixe et du suffixe d'une certaine taille.

Données : $T[1..n]$: un tableau de n entiers, k : un entier
Résultat : renvoie VRAI si le préfixe et le suffixe de T de longueur k sont égaux
Variable : i, un entier.
début
  pour tous les i *de 1 à* k **faire**
    si $T[i] \neq T[n-k+i]$ **alors renvoyer** FAUX;
  fin
  renvoyer VRAI;
fin

Algorithme 14 : SuffixePrefixeEgaux(d $T[1..n]$: tableau d'entiers, d k : entier) : booléen

Données : $T[1..n]$: un tableau de n entiers
Résultat : renvoie la taille du plus long préfixe égal au suffixe de T et différent de T.
Variable : k, un entier.
début
  $k \leftarrow n;$
  tant que $k \geq 0$ *et non(SuffixePrefixeEgaux(*T, k*))* **faire**
    $k \leftarrow k - 1;$
  fin
  renvoyer k;
fin

Algorithme 15 : PlusLgPrefSuffDecomposé(d $T[1..n]$: tableau d'entiers) : booléen

Et maintenant une version en un seul algorithme :

```
Données : T[1..n] : un tableau de n entiers
Résultat : renvoie la taille du plus long préfixe égal au suffixe de T et différent de T.
Variables : k, i, des entiers, et contient, un booléen.
début
  | k ← n; contient ← Faux;
  | tant que k > 0 et non(contient) faire
  |   | k ← k − 1;
  |   | contient ← Vrai;
  |   | pour tous les i de 1 à k faire
  |   |   | si T[i] ≠ T[n − k + i] alors contient ← Faux;
  |   | fin
  | fin
  | renvoyer k;
fin
```

Algorithme 16 : PlusLgPrefSuff(d $T[1..n]$: tableau d'entiers) : booléen

3.4.3 Exponentiation rapide (exercice 6 complexité)

Il suffit de considérer l'algorithme de multiplication russe vu en cours et de remplacer les additions par des multiplications et les initialisations à 0 par des initialisations à 1.

Pour comprendre l'algorithme, rien de mieux que de voir son fonctionnement sur un exemple : pour calculer rapidement X^{16} en faisant le moins de multiplications possible, l'idée est de le décomposer en : $X^{16} = (X^8)^2 = ((X^4)^2)^2 = (((X^2)^2)^2)^2$, soit simplement 4 multiplications (en commençant le calcul par l'intérieur des parenthèses) ! Si l'exposant n'est pas une puissance de 2, on s'adapte facilement, par exemple 2^{20} se décompose en : $(X^{10})^2 = ((X^5)^2 \times X)^2 = (((X^2)^2 \times X)^2 \times X)^2$, soit seulement 6 multiplications !

Dans tous les cas, si k est une puissance de 2, on a besoin de $\log_2 k$ multiplications (qui sont des élévations au carré). Sinon, si k est compris entre 2^i et $2^{i+1} - 1$, on a besoin de $i = \lfloor \log_2 n \rfloor$ élévations au carré, et au plus autant de multiplications par X. Dans tous les cas, au total, il y a $\Theta(\log n)$ opérations.

3.5 Séance 11 (12/11/2008)

3.5.1 Exponentiation rapide - version récursive (exercice 6 complexité)

```
Données : A, B : deux entiers
Résultat : renvoie A^B
Variable : p, un entier.
début
  | si B = 0 alors
  |   | renvoyer 1;
  | sinon
  |   | p ← ExponentiationRapide(A,B div 2);
  |   | si B mod 2=0 alors
  |   |   | renvoyer p × p;
  |   | sinon
  |   |   | renvoyer p × p × A;
  |   | fin
  | fin
fin
```

Algorithme 17 : ExponentiationRapide(A,B : entiers) : entier

3.5.2 Problème SomMax (complexité)

(a) La complexité de l'algorithme proposé est en $O(n^3)$.

(b) Pour passer à une complexité quadratique il suffit de mettre à jour SomMax quand on fait varier i et j.

(c) On peut passer à $O(n \log n)$ avec une approche "Diviser et Résoudre". L'idée est que le sous-tableau de somme maximale se trouvera soit dans la première moitié, soit dans la seconde, soit il empiètera sur les deux et

Données : $T[1..n]$: un tableau de n entiers relatifs
Résultat : renvoie 0 si tous les éléments de T sont négatifs, la somme maximale des éléments, parmi tous ses sous-tableaux sinon, c'est à dire $\max_{a \leq b \in [1..n]}(0, \sum_{i=a}^{b} T[i])$.
Variables : *somme*, un entier.
début
  $SomMax \leftarrow 0$;
  pour tous les *i de 1 à n* **faire**
    $somme \leftarrow 0$;
    pour tous les *j de i à n* **faire**
      $somme \leftarrow somme + T[j]$;
      $SomMax \leftarrow max(somme, SomMax)$;
    fin
  fin
  renvoyer max(0,$SomMax$);
fin

Algorithme 18 : SomMax(d $T[1..n]$: tableau d'entiers) : entier

a donc la somme maximale en partant du milieu dans le sous-tableau gauche, et somme maximale en partant du milieu dans le sous-tableau droit, ce qui permet de le trouver en $O(n)$.

(d) Pour obtenir un algorithme linéaire, et donc optimal, considérons qu'on a les deux invariants proposés pour un certain k, et regardons ce qui se passe à la case $k+1$. On appelle T_k le sous-tableau de somme maximale de $T[1..k]$, et T'_k le sous-tableau de somme maximale se terminant à la case k de $T[1..k]$. Ainsi au k-ième pas, $SomMax$ est la somme des éléments de T_k et $SomMaxDroite$ est la somme des éléments de T'_k.

L'algorithme 20 peut être raccourci pour donner l'algorithme 21 (qui semble un peu trop "magique", ce qui explique la présence de la version longue, qui est en outre plus facile à modifier pour renvoyer les indices de début et de fin du tableau de somme maximale).

Données : $T[1..n]$: un tableau de n entiers relatifs, d, f : deux entiers, les indices de début et fin du sous-tableau considéré
Résultat : renvoie 0 si tous les éléments de T sont négatifs, la somme maximale des éléments, parmi tous ses sous-tableaux sinon, c'est à dire $\max\limits_{a \leq b \in [1..n]}(0, \sum\limits_{i=a}^{b} T[i])$.
Variables : *sommeGauche*, *sommeDroite* : deux entiers.
début
  si $d{=}f$ **alors**
    renvoyer $\max(0, T[d])$
  sinon
    $sommeGauche \leftarrow 0$;
    $somme \leftarrow 0$;
    pour tous les i *de* $\lfloor \frac{d+f}{2} \rfloor$ *à* d *par pas de -1* **faire**
      $somme \leftarrow somme + T[i]$;
      $sommeGauche \leftarrow max(somme, sommeGauche)$;
    fin
    $sommeDroite \leftarrow 0$;
    $somme \leftarrow 0$;
    pour tous les i *de* $\lfloor \frac{d+f}{2} \rfloor + 1$ *à* f **faire**
      $somme \leftarrow somme + T[i]$;
      $sommeDroite \leftarrow max(somme, sommeDroite)$;
    fin
    renvoyer
    $\max\Big(0, sommeGauche + sommeDroite, \text{RecSomMax}(T, d, \lfloor \frac{d+f}{2} \rfloor), \text{RecSomMax}(T, \lfloor \frac{d+f}{2} \rfloor + 1, f)\Big)$;
  fin
fin

Algorithme 19 : RecSomMax(d $T[1..n]$: tableau d'entiers, d d : entier, d f : entier): entier

Données : $T[1..n]$: un tableau de n entiers relatifs, d, f : deux entiers, les indices de début et fin du sous-tableau considéré
Résultat : renvoie 0 si tous les éléments de T sont négatifs, la somme maximale des éléments, parmi tous ses sous-tableaux sinon, c'est à dire $\max_{a \leq b \in [1..n]}(0, \sum_{i=a}^{b} T[i])$.
Variables : $SomMax$, $SomMaxDroite$: deux entiers.

```
début
  SomMax ← T[1]; SomMaxDroite ← T[1];
  pour tous les k de 2 à n faire
    si SomMaxDroite ≥ 0 alors
      si SomMaxDroite + T[k] > SomMax alors
        //Le nouveau tableau optimal T_k est T'_{k-1} + la k-ième case :
        SomMax ← SomMaxDroite + T[k];
        SomMaxDroite ← SomMaxDroite + T[k];
      sinon
        SomMaxDroite ← SomMaxDroite + T[k];
      fin
    sinon
      //SomMaxDroite < 0
      si T[k] > SomMax alors
        //Le nouveau tableau optimal T_k est uniquement la k-ième case :
        SomMax ← T[k];
        SomMaxDroite ← T[k];
      sinon
        SomMaxDroite ← T[k];
      fin
    fin
  fin
  renvoyer max(SomMax,0);
fin
```

Algorithme 20 : SomMax4Long(d $T[1..n]$: tableau d'entiers, d d : entier, d f : entier): entier

Données : $T[1..n]$: un tableau de n entiers relatifs, d, f : deux entiers, les indices de début et fin du sous-tableau considéré
Résultat : renvoie 0 si tous les éléments de T sont négatifs, la somme maximale des éléments, parmi tous ses sous-tableaux sinon, c'est à dire $\max_{a \leq b \in [1..n]}(0, \sum_{i=a}^{b} T[i])$.
Variables : $SomMax$, $SomMaxDroite$: deux entiers.

```
début
  SomMax ← T[1]; SomMaxDroite ← T[1];
  pour tous les k de 2 à n faire
    SomMaxDroite ← max(SomMaxDroite + T[k], T[k]);
    SomMax ← max(SomMaxDroite, SomMax);
  fin
  renvoyer max(SomMax,0);
fin
```

Algorithme 21 : SomMax4Court(d $T[1..n]$: tableau d'entiers, d d : entier, d f : entier): entier

Chapitre 4

Listes chaînées

4.1 Séance 12 (17/11/2008)

4.1.1 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 22 (on peut aussi effectuer le parcours par une fonction récursive mais il est possible que ce soit plus délicat).

```
Données : L, ... : liste simplement chaînée, ...
Résultat : renvoie un truc après avoir parcouru L.
Variables : Q, liste simplement chaînée.
début
  | Q ← L;
  | ...;
  | tant que Q ≠ NULL faire
  |   | ...;
  |   | Q ← Q ↑ succ;
  |   | ...;
  | fin
  | ...;
  | renvoyer ...;
fin
```

Algorithme 22 : 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 = NULL, ou en l'atteignant, c'est à dire quand $Q \uparrow$ succ = NULL. Dans le cas où vous choisissez cette dernière solution, assurez-vous que Q n'est pas vide.

4.1.2 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, 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.

4.1.3 Adresse de la dernière cellule d'une liste

L'algorithme 23 a une complexité linéaire puisqu'on doit parcourir toute la liste pour atteindre sa dernière cellule (alors qu'avec une liste doublement chaînée on peut accéder à l'adresse de la dernière cellule de la liste en temps constant).

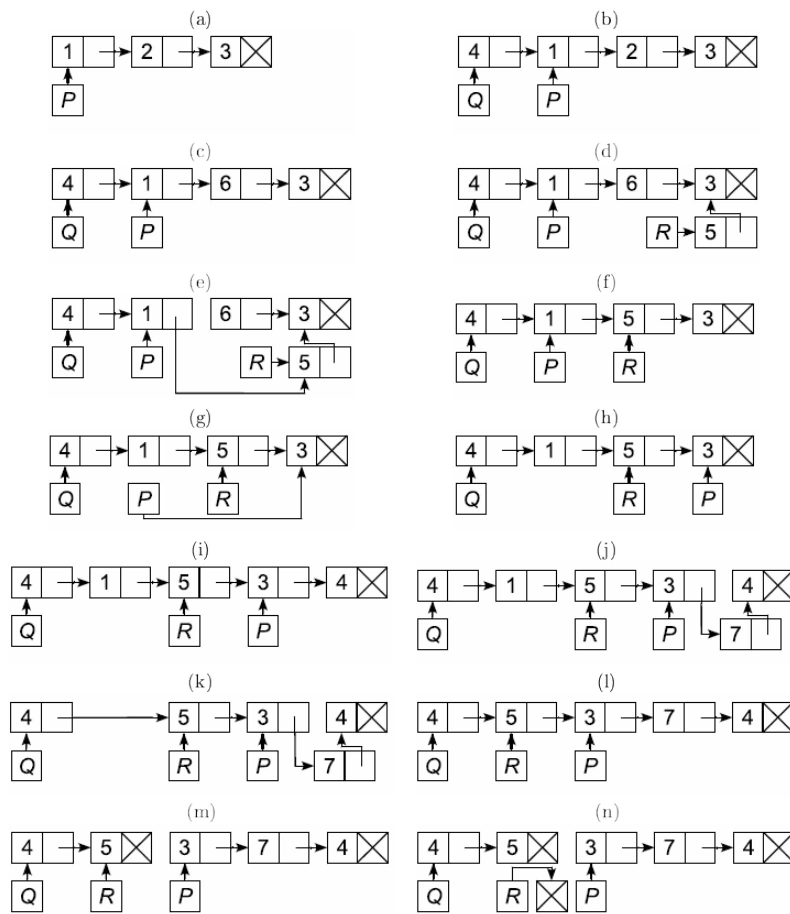

FIGURE 4.1 – Utilisation des listes chaînées : après la ligne 1 (a), 2 (b), 3 (c), 4 (d), 5 (e), réorganisation (f), après la ligne 6 (g), réorganisation (h), après la ligne 7 (i), 8 (j), 9 (k), réorganisation (l), après la ligne 10 (m), 11 (n).

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 Séance 13 (19/11/2008)

4.2.1 Premier élément mis à la fin d'une liste

L'algorithme 24 est aussi linéaire (on peut l'écrire en temps constant avec une liste doublement chaînée). Notons qu'on s'arrête juste 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, pour éviter de perdre l'adresse de la dernière cellule.

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

L'algorithme Supprimer vu en cours a une complexité en $O(n)$, donc SupprimeValeur (algorithme 25) a une complexité en $0(n^2)$.

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 SupprimeValeurD (algorithme 26), en utilisant une liste doublement chaînée, est linéaire.

En fait, on peut aussi arriver à cette complexité avec une liste simplement chaînée, en retenant l'adresse de la case précédente dans une mémoire, ou bien en procédant de manière décalée, et en testant si la *case suivante* contient e, et non la case courante, comme dans l'algorithme 27.

Données : L : liste simplement chaînée
Résultat : renvoie l'adresse de la dernière cellule d'une liste
Variables : Q, liste simplement chaînée.

```
début
  Q ← L;
  tant que Q ↑ succ ≠ NULL faire
    Q ← Q ↑ succ;
  fin
  renvoyer Q;
fin
```

Algorithme 23 : DerniereCellule(L : liste simplement chaînée) : liste simplement chaînée

Données : L : liste simplement chaînée
Résultat : modifie L pour déplacer son premier élément en dernière position
Variable : Q, liste simplement chaînée.

```
début
  Q ← L;
  si Q = NULL alors
    renvoyer NULL;
  fin
  tant que Q ↑ succ ≠ NULL faire
    Q ← Q ↑ succ;
  fin
  Q ↑ succ ← créerListe(L ↑ info, NULL);
fin
```

Algorithme 24 : PremierALaFin(L : liste simplement chaînée) : liste simplement chaînée

Données : L : liste simplement chaînée, e : entier
Résultat : supprime de L tous les éléments de valeur e
Variable : Q, liste simplement chaînée.

```
début
  Q ← L;
  si Q = NULL alors
    renvoyer NULL;
  fin
  tant que Q ≠ NULL faire
    si Q ↑ info = e alors
      Supprimer(L,Q);
    fin
    Q ← Q ↑ succ;
  fin
fin
```

Algorithme 25 : SupprimeValeur(L : liste simplement chaînée, e : entier)

```
Données : L : liste doublement chaînée, e : entier
Résultat : supprime de L tous les éléments de valeur e
Variable : Q, liste doublement chaînée.
début
    Q ← L ↑ succ;
    si Q = NULL alors
        renvoyer NULL;
    fin
    tant que Q ≠ L faire
        si Q ↑ info = e alors
            //Le prédécesseur de la case courante pointe vers son successeur :
            Q ↑ pred ↑ succ ← Q ↑ succ ;
            //Le successeur de la case courante pointe vers son prédécesseur :
            Q ↑ succ ↑ pred ← Q ↑ pred ;
        fin
        Q ← Q ↑ succ;
    fin
fin
```

Algorithme 26 : SupprimeValeurD(L : liste doublement chaînée, e : entier)

```
Données : L : liste simplement chaînée, e : entier
Résultat : supprime de L tous les éléments de valeur e
Variable : Q, liste simplement chaînée.
début
    Q ← L;
    si Q = NULL alors
        renvoyer NULL;
    fin
    tant que Q ↑ succ ≠ NULL faire
        si Q ↑ succ ↑ info = e alors
            Q ↑ succ ← Q ↑ succ ↑ succ;
        sinon
            Q ← Q ↑ succ;
        fin
    fin
fin
```

Algorithme 27 : SupprimeValeurB(L : liste simplement chaînée, e : entier)

4.3 Séance 14 (26/11/2008)

4.3.1 Suppression de doublons consécutifs d'une liste

Comme dans l'exercice précédent, il faut être un peu habile pour écrire l'algorithme 28 : si, en détectant un doublon dans la case suivante, on veut supprimer la case actuelle, alors il faut retenir l'adresse de la case précédente pour la relier à la suivante. Au lieu de ça, il vaut mieux supprimer la case suivante, en reliant l'actuelle à celle qui suivra.

```
Données : L : liste simplement chaînée non vide
Résultat : supprime de L tous les doublons consécutifs
Variable : Q, liste simplement chaînée.
début
  | Q ← L;
  | tant que Q ↑ succ ≠ NULL faire
  |   | si Q ↑ info = Q ↑ succ ↑ info alors
  |   |   | Q ↑ succ ← Q ↑ succ ↑ succ;
  |   | sinon
  |   |   | Q ← Q ↑ succ;
  |   | fin
  | fin
fin
```

Algorithme 28 : SupprimeDoublonsConsecutifs(L : liste simplement chaînée non vide)

4.3.2 Concaténation de deux listes simplement chaînées

L'algorithme 29 de concaténation de deux listes L_1 et L_2 en temps linéaire consiste à atteindre la dernière case de L_1 pour la lier à la première de L_2.

```
Données : L1, L2 : listes simplement chaînées
Résultat : modifie L1 pour lui concaténer L2
Variable : Q, liste simplement chaînée.
début
  | Q ← L1;
  | si Q = NULL alors
  |   | L1 ← L2;
  | fin
  | tant que Q ↑ succ ≠ NULL faire
  |   | Q ← Q ↑ succ;
  | fin
  | L1 ↑ succ ← L2
fin
```

Algorithme 29 : Concatene(L_1, L_2 : listes simplement chaînées)

4.3.3 Concaténation de deux listes doublement chaînées

Rappel sur la structure des listes doublement chaînée :

Une liste doublement chaînée est constituée de cases qui contiennent une valeur et qui pointent vers la case suivante ou vers la case précédente. Elle contient aussi une sorte de table des matières qui pointe vers la première et vers la dernière case, et vers qui pointent la première et la dernière case (voir dessin dans le cours). Ainsi, la première valeur de la liste doublement chaînée L est $L \uparrow$ succ $\uparrow$ info, sa dernière valeur est $L \uparrow$ pred $\uparrow$ info. L'intérêt est donc de pouvoir accéder à la case précédente, ou bien la dernière case, en temps constant.

En utilisant des listes doublement chaînées, on peut donc effectuer la concaténation en temps constant, avec l'algorithme 30. Il faut juste être soigneux pour assurer que les bonnes flèches pointent au bon endroit. De plus, contrairement à la version avec les listes simplement chaînées où L_2 restait une liste valide après concaténation, en version doublement chaînée, L_2 sera détruite dans l'opération puisque la dernière case de L_2 ne pointera plus vers la table des matières de L_2, mais vers celle de L_1.

Données : L_1, L_2 : listes doublement chaînées
Résultat : modifie L_1 pour lui concaténer L_2
début

// La dernière case de L_1 pointe vers la première de L_2 :
$L_1 \uparrow$ pred $\uparrow$ succ $\leftarrow L_2 \uparrow$ succ;
// La première case de L_2 pointe vers la dernière de L_1 :
$L_2 \uparrow$ succ $\uparrow$ pred $\leftarrow L_1 \uparrow$ pred;
// La dernière case de L_2 pointe vers la table des matières de L_1 :
$L_2 \uparrow$ pred $\uparrow$ succ $\leftarrow L_1$;
// La table des matières de L_1 pointe vers la dernière case de L_2 :
$L_1 \uparrow$ pred $\leftarrow L_2 \uparrow$ pred;

fin

Algorithme 30 : ConcateneDC(L_1, L_2 : listes doublement chaînées)

Chapitre 5

Arbres

5.1 Fin de la séance 14 (26/11/2008)

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 $A \uparrow$ info, et un pointeur vers la cellule-racine de son sous-arbre gauche $A \uparrow$ sag, ainsi qu'un pointeur vers la cellule-racine de son sous-arbre droit $A \uparrow$ 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 permet 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))

5.2 Séance 15 (03/12/2008)

5.2.1 Hauteur d'un arbre binaire

Idée de l'algorithme 31 : 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 = NULL alors
  | renvoyer 0;
  fin
  sinon si A ↑ sag = NULL et A ↑ sag = NULL alors
  | renvoyer 0;
  fin
  sinon
  | renvoyer 1 + max (Hauteur(A ↑ sag),Hauteur(A ↑ sad));
  fin
fin
```

Algorithme 31 : Hauteur(A : arbre binaire) : entier

5.2.2 Nombre de feuilles d'un arbre binaire

Idée de l'algorithme 32 : 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 ↑ sag = NULL et A ↑ sag = NULL alors
        renvoyer 1;
    fin
    sinon
        renvoyer NombreFeuilles(A ↑ sag)+NombreFeuilles(A ↑ sad);
    fin
fin
```

Algorithme 32 : NombreFeuilles(A : arbre binaire) : entier

5.2.3 Effeuillage d'un arbre binaire

Idée de l'algorithme 33 : l'effeuillage consiste à effeuiller récursivement le sous-arbre gauche et le sous-arbre droit de la racine de l'arbre.

```
Données : A : arbre binaire
Résultat : modifie A en l'effeuillant
début
    si A ↑ saq = NULL et A ↑ sad = NULL alors
        A ← NULL // cas où l'arbre de départ est une racine-feuille
    fin
    si (A ↑ sag ≠ NULL) alors
        si (A ↑ sag ↑ sag = NULL) et (A ↑ sag ↑ sad = NULL) alors
            A ↑ sag ← NULL // suppression du fils gauche de A, qui est une feuille;
        fin
        sinon
            Effeuille(A ↑ sag);
        fin
    fin
    si (A ↑ sad ≠ NULL) alors
        si (A ↑ sad ↑ sag = NULL) et (A ↑ sad ↑ sad = NULL) alors
            A ↑ sad ← NULL // suppression du fils droit de A, qui est une feuille;
        fin
        sinon
            Effeuille(A ↑ sad);
        fin
    fin
fin
```

Algorithme 33 : Effeuillage(A : arbre binaire) : entier

Chapitre 6

Tris

On notera dans la suite $|T|$ la taille d'un tableau T pour plus de simplicité.

6.1 Séance 16 (10/12/2008)

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

Dans tous les cas, pour calculer la taille de l'intersection de A et B, on va mettre dans une liste tout élément apparaissant dans A et B.

Sans doublons

Dans un premier temps on considérera qu'il n'y a pas de doublons, puis on ajoutera une procédure pour les prendre en compte qui ne changera rien à l'ordre de grandeur de la complexité des algorithmes.

Idée de l'algorithme 34 : pour chacun des $|A|$ éléments du premier tableau, on regarde en $O(|B|)$ s'il est présent parmi les $|B|$ 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 $O(|A||B|)$.

```
Données : A, B : tableaux d'entiers sans doublons, B est trié.
Résultat : Renvoie le nombre d'entiers présents dans les deux tableaux.
Variables : t, entier.
début
    t ← 0;
    pour i de 1 à |A| faire
        pour j de 1 à |B| faire
            si A[i] = B[j] alors
                t ← t + 1;
            fin
        fin
    fin
    renvoyer t;
fin
```

Algorithme 34 : TailleIntersectionNaif(A, B : tableaux) : entier

Idée de l'algorithme 35 avec un des deux tableaux trié : pour chacun des $|A|$ éléments du premier tableau, on regarde par dichotomie en $O(\log|B|)$ s'il est présent parmi les éléments du second tableau qui sont triés. On obtient donc une complexité en $O(|A|\log|B|)$.

Si le tableau B n'est pas trié, on peut commencer par le trier en $O(|B|\log|B|)$. La complexité totale est donc en $O(|A|\log|B| + |B|\log|B|)$, c'est à dire $O(\max(|A|, |B|)\log|B|)$, elle est meilleure qu'avec l'idée précédente.

Idée de l'algorithme 36 avec les deux tableaux triés : on parcourt simultanément les deux tableaux pour obtenir une complexité en $O(|A| + |B|)$.

Si on considère que les deux tableaux ne sont pas triés, on commence par trier le premier tableau en $O(|A|\log|A|)$, puis le second en $O(|B|\log|B|)$, ce qui donne une complexité totale en $O(|B|\log|B|+|A|\log|A|+|A| + |B|)$, soit $O(\max(|A|, |B|)\log\max(|A|, |B|))$, ce qui est encore meilleur que l'idée précédente.

Données : A, B : tableaux d'entiers sans doublons, B est trié.
Résultat : Renvoie le nombre d'entiers présents dans les deux tableaux.
Variables : t, entier.

```
début
    t ← 0;
    pour i de 1 à |A| faire
        si RechercheDicho(B,A[i]) alors
            t ← t + 1;
        fin
    fin
    renvoyer t;
fin
```

Algorithme 35 : TailleIntersection1Trié(A, B : tableaux) : entier

Données : A, B : tableaux d'entiers sans doublons, A et B sont triés.
Résultat : Renvoie le nombre d'entiers présents dans les deux tableaux.
Variables : t, entier.

```
début
    t ← 1;
    i ← 1; j ← 0;
    tant que i + j ≤ |A| + |B| faire
        si A[i] < B[j] alors
            i ← i + 1;
        sinon
            si B[j] < A[i] alors
                j ← j + 1;
            sinon
                i ← i + 1; j ← j + 1;
                t ← t + 1;
            fin
        fin
    fin
    renvoyer t;
fin
```

Algorithme 36 : TailleIntersection2Triés(A, B : tableaux) : entier

Avec doublons

Pour les deux premiers algorithmes, l'idée est de garder en mémoire dans une variable X l'intersection des deux tableaux déjà trouvée. A chaque fois qu'un élément est trouvé à la fois dans le tableau A et dans le tableau B, on vérifie s'il est déjà dans X. Si non, on l'y ajoute. Cette opération est donc faite $O(min(|A|,|B|))$ fois puisque $|A \cap B| \leq min(|A|,|B|)$. Quelle structure de données utiliser pour X ? Il faut en choisir une qui permet des opérations de recherche et d'insertion rapide :

- tableau trié : recherche en $O(\log n)$, insertion en $O(n)$
- tas : recherche en $O(n)$, insertion en $O(\log n)$
- arbre binaire de recherche : recherche en $O(\text{nauteur})$, insertion en $O(\text{nauteur})$ (où hauteur est logarithmique en moyenne, mais possiblement linéaire).
- pour avoir des opérations de recherche et d'insertion logarithmique, et donc garder le même ordre de grandeur de la complexité en $O(|A|\log|B|)$ pour l'algorithme 35, il faut utiliser une structure de données plus perfectionnée : arbre AVL [1] ou arbre rouge noir [2]...

Pour le troisième algorithme, c'est plus simple : il suffit de poursuivre l'incrémentation des compteurs i et j tant que des éléments identiques sont découverts, c'est à dire insérer juste après la ligne $t \leftarrow t+1;$: **tant que** $A[i] = A[i-1]$ **faire** $i \leftarrow i+1$, puis : **tant que** $B[j] = B[j-1]$ **faire** $j \leftarrow j+1$.

6.1.2 Test de l'existence de doublons

Un algorithme naïf pour vérifier si les n éléments d'un tableau sont différents est de tester pour chaque élément s'il est présent dans une des $n-1$ autres cases. Cet algorithme a une complexité quadratique. On peut faire mieux en commençant par trier le tableau en $O(n\log n)$ ce qui aura pour effet de rapprocher les doublons si le tableau en contient. Il suffit alors d'utiliser l'algorithme 37 qui parcourt en temps linéaire un tableau trié pour vérifier qu'il ne contient pas de doublon.

```
Données : T : tableau contenant n entiers triés.
Résultat : Renvoie VRAI si le tableau ne contient que des éléments différents, FAUX sinon
Variables : i, entier.
début
    i ← 1;
    tant que i < n et T[i] ≠ T[i+1] faire
        i ← i+1;
    fin
    renvoyer (i = n);
fin
```

Algorithme 37 : TousDifferents(T : tableau) : booléen

6.2 Séance 17 (17/12/2008)

6.2.1 Tri par fusion

Voir les algorithmes 38 et 39. Dans le pire comme dans le meilleur cas, l'algorithme a une complexité en $O(n\log n)$.

6.2.2 Tri de notes

L'algorithme 40 a une complexité linéaire en la taille du tableau. Si on le généralise pour trier un tableau d'entiers compris entre 0 et k, k étant fourni en paramètre, il a une complexité en $O(k+n)$.

6.2.3 Tri par base

Dans l'énoncé, quand il est dit que $T[j][l]$ désigne le $l^{\text{ème}}$ chiffre du $j^{\text{ème}}$ nombre de T, il s'agit du $l^{\text{ème}}$ chiffre *de poids faible*. Par exemple, si $T = [139, 1042, 225]$, $T[3][1] = 5$ et $T[3][4] = 0$.

L'algorithme 41 a une complexité en $O(kn)$.

1. http://fr.wikipedia.org/wiki/Arbre_AVL
2. http://fr.wikipedia.org/wiki/Arbre_rouge-noir

```
Données : T[1..n] : tableau.
Résultat : Renvoie un tableau trié contenant les éléments de T
début
    si n > 1 alors
        renvoyer Fusion(TriFusion(T[1..⌊n/2⌋], T[⌊n/2⌋ + 1..n]));
    sinon
        renvoyer T;
    fin
fin
```

Algorithme 38 : TriFusion(T : tableau) : tableau

```
Données : T1, T2 : tableaux triés.
Résultat : Renvoie un tableau trié contenant les éléments de T1 et T2
Variables : T, tableau d'entiers
début
    T ← T[1..|T1| + |T2|];
    i ← 1; j ← 1;
    tant que i + j ≤ |T1| + |T2| faire
        si i > |T1| alors
            T[i + j − 1] ← T2[j];
        sinon
            si j > |T2| alors
                T[i + j − 1] ← T1[i];
            sinon
                si T1[i] < T2[j] alors
                    T[i + j − 1] ← T1[i]; i ← i + 1;
                sinon
                    T[i + j − 1] ← T2[j]; j ← j + 1;
                fin
            fin
        fin
    fin
    renvoyer T;
fin
```

Algorithme 39 : Fusion($T1, T2$: tableaux triés) : tableau

```
Données : T : tableau d'entiers compris entre 0 et 20.
Résultat : Renvoie un tableau trié contenant les éléments de T
Variables : NbNotes[1..21], tableau
début
    //Création de NbNotes
    pour i de 1 à 21 faire
        NbNotes[i] ← 0;
    fin
    //Remplissage de NbNotes
    pour i de 1 à |T| faire
        NbNotes[T[i] + 1] ← NbNotes[T[i] + 1] + 1;
    fin
    //"Développement" de NbNotes
    j ← 1;
    pour i de 1 à |NbNotes| faire
        k ← 0;
        tant que k < NbNotes[i] faire
            T[j] ← i − 1;
            j ← j + 1; k ← k + 1;
        fin
    fin
    renvoyer T;
fin
```

Algorithme 40 : TriNotes(T : tableau d'entiers compris entre 0 et 20) : tableau

```
Données : T[1..n] : tableau d'entiers ayant au plus k chiffres.
Résultat : Renvoie un tableau trié contenant les éléments de T
Variables : F[0..9], tableau
début
    //Calcul de k :
    pour i de 1 à |T| faire
        k ← max(k, ⌊log10 T[i]⌋);
    fin
    //Initialisation du tableau de files :
    pour i de 0 à 9 faire
        F[i] ←créerFile();
    fin
    //Tri du tableau :
    pour i de 1 à k + 1 faire
        //Remplissage du tableau de files trié selon le i^ème chiffre :
        pour j de 1 à |T| faire
            ajouterFile(F[T[j][i]], T[j])
        fin
        //Remplissage de T, trié selon les i chiffres de poids faible :
        a ← 1  pour j de 0 à 9 faire
            tant que non(FileVide?(F[j])) faire
                T[a] ← têteFile(F[j]);
                retirerFile(F[j]);
                a ← a + 1;
            fin
        fin
    fin
    renvoyer T;
fin
```

Algorithme 41 : TriBase(T : tableau de n entiers) : tableau