UE33/UE43 - Opt1. Algorithmique
D.E.U.G. 2ème Année

Arnaud Legrand

15 octobre 2003

Année 2003-2004

Résumé

Ce document décrit le cours d'algorithmique dispensé aux étudiants de deuxième année de DEUG de L'UCBL au cours de l'année 2003-2004. Il est interdit aux étudiants de DEUG MIAS et ouvert à tous les autres DEUGs scientifiques. Il est indispensable à toute personne souhaitant suivre une licence d'informatique par la suite.

Table des matières

Introduction

Qu'est-ce qu'un ordinateur et qu'est-ce qu'on y trouve ? Un clavier, un écran, une carte mère, un processeur, de la mémoire, un disque dur, des périphériques (lecteur de disquettes, CD, DVD, imprimantes, etc.). Bref, plein de choses mais les seules vraiment indispensables, sont le processeur (qui fait des opérations arithmétiques élémentaires), la mémoire (dans laquelle le processeur peut stocker des résultats intermédiaires) et le bout de fil qui est entre les deux (également appelé «bus») qui se trouve sur la carte mère.

Comment se fait-il que l'on arrive à faire des choses aussi complexes avec un processeur qui, en gros, ne sait faire que des opérations arithmétiques élémentaires ? À un niveau un peu supérieur, nous nous intéresserons dans ce cours à la conception d'algorithmes (c'est-à-dire de méthodes) exprimés dans un langage élémentaire et «compréhensible» par un ordinateur. Nous utiliserons donc un pseudo-langage de haut niveau, en français et relativement souple, pendant les cours de façon à s'abstraire des difficultés posées par un vrai langage. Une fois ce pseudo-langage maîtrisé, nous traduirons nos algorithmes en langage C. Ce langage date des années 70/80 est relativement bas niveau. Cela signifie que le C manipule les mêmes sortes d'objets que que la plupart des ordinateurs, à savoir des caractères, des nombres et des adresses. Ce langage est disponible sur toute les plates-formes existantes à ce jour et quand il ne l'est pas (sur une plate-forme embarquée comme un robot ou un agenda électronique), on effectue une compilation croisée.

Qu'est-ce qu'une compilation et comment se fait-il qu'un même langage puisse être utilisé sur autant de processeurs différents (i386, PowerPc, Atari, Sparc, Mips, etc.) ? Comme nous l'avons déja dit, chaque processeur n'est capable de faire que des choses élémentaires. Pour compliquer les choses, chaque processeur est utilisable à l'aide d'un langage machine il existe différents types de langages machine. Un compilateur permet de transformer un programme écrit en un langage comme le C en langage machine. Il existe donc autant de compilateur que de familles de processeurs, de langages et de système d'exploitation (enfin presque car certains langages n'ont jamais été porté sur certains systèmes). Nous utiliserons le C sous linux.

Mais avant de programmer, il faut définir un algorithme. Un algorithme est une procédure de calcul bien définie, qui prend *en entrée* une valeur (ou un ensemble de valeurs), et qui produit, *en sortie*, une valeur ou un ensemble de valeurs. Un algorithme est donc une suite d'étapes de calcul permettant de passer de la valeur d'entrée à la valeur de sortie. Un algorithme permet donc de résoudre un problème combinatoire. L'énoncé du problème spécifie en termes généraux la relation désirée entre l'entrée et la sortie. L'algorithme décrit une procédure de calcul permettant d'établir cette relation.

1 Bases d'un langage algorithmique

Le langage algorithmique est un langage générique permettant de traiter des problèmes par concaténation d'instructions élémentaires. Il est à la base de tous les langages de programmation (enfin... tous les langages de programmations *impératifs*).

1.1 Structure de base

En matière de programmation, il n'y a pas grand chose d'obligatoire mais beaucoup de choses recommandées. En particulier, un programme a à peu près toujours la même organisation générale

```
1: Nom du programme
2: Déclaration de variables
3: Déclaration de fonctions
4: Début
5:    ...
6:    Liste des instructions
7:    ...
8: Fin
```

1.2 Déclaration de variables

Qu'est ce qu'une variable ? Une variable est un espace mémoire nommé, de taille fixée prenant au cours du déroulement de l'algorithme un nombre indéfini de valeurs différentes. Ce changement de valeur se fait par l'opération d'**affectation** (notée ← dans notre langage algorithmique). La variable diffère de la notion de **constante** qui, comme son nom l'indique, ne prend qu'une unique valeur au cours de l'exécution de l'algorithme.

À quoi sert la déclaration de variable ? La partie *déclaration de variable* permet de spécifier quelle seront les variables utilisées au cours de l'algorithme ainsi que le type de valeur qu'elles doivent respectivement prendre. Il est bien évident que l'on ne peut mémoriser une chaîne de caractères dans une variable de type "Entier". Le type des variables est utile à l'algorithmicien pour lui permettre de structurer ses idées. Il est très utile au programmeur car il permet de détecter des erreurs potentielles. Il est indispensable au compilateur car les différents types ne sont pas tous représentés de la même façon. La représentation d'un même type peut même varier d'un processeur à l'autre.

On utilisera différents types de variables (pour le moment) :

- **Entier** (1,2,3, ...)
- **Réel** (ou flottants) (1.0, 1.35, 1E+12, 3.1415926, ...)
- **Caractère** (a, b, c, !, ?, ...)
- **Booléen** (VRAI ou FAUX, par exemple $(2 < 3)$ **Ou** $(3 \neq 5)$ vaut VRAI)

Par exemple, "**Entier** A, B" définit 2 variables de type entières n'ayant aucune valeur (pour l'instant).

1.3 Instruction

Une instruction est une action élémentaire commandant à la machine un calcul, ou une communication avec un de ses périphériques (Entrant ou Sortant). Une instruction de base peut être :

une affectation et/ou opération arithmétique : l'affectation est l'action élémentaire principale puisque c'est par son intermédiaire que l'on peut modifier la valeur d'une variable. L'affectation a pour syntaxe *variable*← *valeur*.

```
1: { Exemple d'affectation }
2: Entier A
3: Début
4:    A← 1
5: Fin
6: {Cette séquence d'instructions donne la valeur entière "1" à la variable A.}
7: {Remarque : la valeur donnée est toujours du même type que la variable.}
```

un affichage : l'affichage est l'action élémentaire permettant a l'utilisateur de fournir un ou plusieurs résultats issus de son algorithme. Ainsi l'affichage peut être une simple phrase mais aussi peut permettre la visualisation du contenu (typé) d'une variable. L'affichage dans le langage algorithmique se fait par l'intermédiaire de la commande **Écrire**.

```
1: {Exemple d'utilisation de Écrire }
2: Entier A
3: Début
4:    A← 3
5:    Écrire ("La valeur de A est", A,".")
6:    Écrire ("La valeur de A+1 est", A+1,".")
7: Fin
8: {Cette séquence d'instructions donne la valeur entière "1" à la variable A.}
9: {Remarque : la valeur donnée est toujours du même type que la variable.}
```

Cette séquence d'instructions affiche la phrase suivante à l'écran :

```
La Valeur de A est 3.
La Valeur de A+1 est 4.
```

Dans le langage simple que nous utilisons pour écrire nos algorithmes, on indique directement ce qui doit être écrit en les séparant par des virgules. En C, le mécanisme est un peu différent. Le premier argument est une sorte de texte à trou que les autres arguments viennent remplir.

une lecture au clavier ou dans un fichier : la lecture au clavier est l'action élémentaire permettant de spécifier par une intervention humaine la valeur d'une variable. Cette action symbolise donc la communication avec un périphérique d'entrée tel que le clavier. Bien évidemment, la valeur saisie par l'utilisateur de l'algorithme se d'être du même type que la variable recevant la valeur. La saisie se fait par l'intermédiaire de la commande **Lire**.

```
1: {Exemple d'utilisation de Lire }
2: Entier A
3: Début
4:    Lire A
5:    Écrire ('La valeur de 2*A est ', 2 × A)
6: Fin
```

Si l'utilisateur saisit la valeur 10, nous aurons alors à l'écran :
```
La Valeur de 2*A est 20
```

1.3.1 Savoir déchiffrer une séquence d'instructions

▷ **Question 1** *Que fait la liste d'instructions suivantes ?*

1: $A \leftarrow 2$
2: $A \leftarrow A + 2$
3: $B \leftarrow A \times 2 + A$
4: $C \leftarrow 4$
5: $C \leftarrow B - C$
6: $C \leftarrow C + A - B$
7: $A \leftarrow B - C \times A$
8: $A \leftarrow (B - A) \times C$
9: $B \leftarrow (A + C) \times B$

Réponse □

▷ **Question 2** *Que fait la liste d'instructions suivantes ?*

1: $X \leftarrow -5$
2: $X \leftarrow X^2$
3: $Y \leftarrow -X - 3$
4: $Z \leftarrow (-X - Y)^2$
5: $X \leftarrow -(X + Y)^2 + Z$
6: $Y \leftarrow Z^X \times Y$
7: $Y \leftarrow -(Z + Y)$
8: $X \leftarrow X + Y - Z$
9: $Y \leftarrow X + Z$
10: $X \leftarrow (Y - Z)^2$
11: $Y \leftarrow X - Y$
12: $X \leftarrow (Y + Z)/(X/10)$
13: $Y \leftarrow ((X \times Z)/Y) \times 9$

Réponse □

1.3.2 Comprendre les principes de l'affectation

▷ **Question 3** *Comment inverser le contenu de deux variables ?*

```
1: {Inversion de deux variables}
2: Entier A, B, X
3: Début
4:     Lire A
5:     Lire B
6:     ...
7:     ...
8:     ...
9:     Écrire ('La valeur de A est ', A)
10:    Écrire ('La valeur de B est ', B)
11: Fin
```

Réponse □

▷ **Question 4** *Comment faire sans utiliser une variable supplémentaire ?*

```
1: {Inversion de deux variables}
2: Entier A, B
3: Début
4:     Lire A
5:     Lire B
6:     ...
7:     ...
8:     ...
9:     Écrire ('La valeur de A est ', A)
10:    Écrire ('La valeur de B est ', B)
11: Fin
```

Réponse □

▷ **Question 5** *Étant donnée X, comment calculer le plus rapidement possible X^{16} ?*

```
1: {Exponentiation}
2: Entier X
3: Début
4:     Lire X
5:     ...
6:     ...
7:     ...
8:     ...
9:     Écrire ('La valeur de X^16 est ', X)
10: Fin
```

Réponse □

1.3.3 Résolution de problèmes simples

▷ **Question 6** *Écrire un algorithme saisissant le prix "TTC" d'une marchandise et affichant le prix "Hors Taxe" sachant que cet article a une T.V.A. de 18,6%.* *Réponse* □

▷ **Question 7** *Écrire un algorithme saisissant 2 variables entières qui calcule et affiche leur moyenne.* *Réponse* □

▷ **Question 8** *Écrire un algorithme saisissant un temps en seconde que l'on transcrira en jours, heure, minutes, secondes.* *Réponse* □

▷ **Question 9** *En se basant sur l'exercice précédent, écrire un algorithme permettant de faire la différence entre deux horaires saisis en heure, minutes, secondes.* *Réponse* □

1.4 Les conditions

La condition en algorithmique est une instruction de branchement permettant de décider, dans un contexte donné, quelle sera la séquence d'instructions a appliquer. Elle permet ainsi à l'algorithme de prendre des décisions concernant son exécution. Sa syntaxe est :

```
1: ...
2: Si <condition> Alors
3:     ... Instructions A ...
4: Sinon
5:     ... Instructions B ...
6: ...
```

La section **Sinon** est facultative. La partie <condition> est essentielle puisque c'est elle qui décide de l'exécution des instructions conditionnées. Elle est de type **Booléen**. Cela signifie que :
- si <condition> vaut VRAI , seul le bloc Instructions A sera exécuté.
- si <condition> vaut FAUX , seul le bloc Instructions B (s'il existe) sera exécuté.

1.4.1 Savoir Interpréter une condition

On s'intéresse au bloc d'instructions suivant :

```
1: ...
2: Si (C − B) = B Alors
3:     A ← A + 1
4:     C ← C + B
5:     B ← A
6: Sinon
7:     B ← A
8:     A ← A − 1
9:     C ← C × B
10: ...
```

▷ **Question 10** *Donner les valeurs des variable A, B et C à la sortie de ce bloc d'instructions :*
- *pour* $A \leftarrow 2$, $B \leftarrow 3$, $C \leftarrow A \times B$ *Réponse*
- *pour* $A \leftarrow 1$, $B \leftarrow 5$, $C \leftarrow 3$ *Réponse*
- *pour* $A \leftarrow -3$, $B \leftarrow A \times A$, $C \leftarrow B - 5$ *Réponse*
- *pour* $A \leftarrow 8$, $B \leftarrow 3$, $C \leftarrow A - 2$ *Réponse*
- $A \leftarrow 10$, $B \leftarrow 1$, $C \leftarrow -B + A^2$ *Réponse*

□

1.4.2 Calculer les racines d'une équation du second degré

▷ **Question 11** *Saisir 3 entiers a, b, c et déterminer dans $\mathbb{R}$ (i.e. sans solution dans les complexes) les racines de l'équation $aX^2 + bX + c = 0$.*

Réponse □

1.5 Les boucles

Si le langage algorithmique se limitait aux structures précédentes, nous ne pourrions pas faire grand chose de plus qu'avec une calculatrice. On pourra nottemment remarquer que lorsque l'on

a un grand nombre d'opérations similaires à faires, le programme se déroulant de façon linéaire, il est nécessaire d'écrire ces opérations autant de fois que nécessaire. On introduit donc d'autres structures de contrôle : les boucles.

1.5.1 Boucle finie

La boucle **Pour** permet d'appliquer une opération (c'est-à-dire un ensemble d'instructions) à chaque élément d'une liste.

```
1: Entier i
2: Début
3:    Pour i dans {1..100} :
4:       Écrire (i^2)
5: Fin
```

Ces boucles peuvent se présenter sous différentes formes :

```
1: Pour i dans {1..100} :
2:    ...
3: Pour i= 1 à 100 :
4:    ...
5: Pour i= 1 à 100 de 5 en 5 :
6:    ...
7: Pour i=23 à 12 de -1 en -1 :
8:    ...
```

▷ **Question 12** *En vous inspirant de l'exemple précédent, écrivez un algorithme qui demande un nombre n, calcule et affiche la somme $\sum_{i=0}^{n} i^3$.* *Réponse* □

1.5.2 Boucle indéfinies

La boucle finie permet donc de réaliser des tâches répétitives à l'aide d'un ordinateur. Néanmoins, il n'est pas toujours possible de décrire simplement l'ensemble des indices que doit parcourir la boucle. Dans le cas, par exemple, où l'on cherche le premier entier $i > 1$ tel que la $i^4 + 4$ soit premier, on ne peut (à moins de réfléchir un peu) pas savoir la taille de l'ensemble d'indice à observer. Peut-être même qu'un tel nombre n'existe pas... Une chose est certaine, si ce nombre existe, une boucle **Tant que** permettra de le trouver.

```
1: Entier i
2: Début
3:    i← 2
4:    Tant que i^4 + 4 n'est pas premier :
5:       i← i+1
6:    Écrire ('Ce nombre existe et vaut ',i)
7: Fin
```

1.6 Les variables indicées (ou tableaux)

Une variable standard permet de stocker une valeur. Une variable indicée de 1 à n permet de stocker n valeurs. La déclaration d'un tableau dont les éléments ont un type de base, a une structure très proche d'une déclaration de variables ayant un type de base. La seule différence consiste à indiquer entre crochets le nombre d'éléments du tableau après le nom de la variable.

- **Entier** t[10]
- **Réel** a[100]

L'accès aux valeurs du tableau se fait donc à l'aides d'indices.

1.7 Les fonctions

Dans l'exemple de la section 1.5.2, le test **Tant que** ($i^4 + 4$ n'est pas premier) avait été utilisé. Ce n'est pas une instruction élémentaire et la détermination de la valeur de " $i^4 + 4$ n'est pas premier " nécessite un certain nombre de calculs. Il serait parfaitement possible d'insérer l'ensemble d'instruction correspondant avant et et au début du corps de la boucle **Tant que**. Cela nuirait cependant à la lisibilité de l'algorithme et ne serait pas très pratique. Lorsqu'un ensemble d'instructions réalise un certain algorithme et que cet ensemble est utilisé à différents endroits, il faut utiliser une fonction.

La structure d'une fonction ressemble à s'y méprendre à celle d'un programme si ce n'est qu'elle peut prendre un certain nombre de paramètres en entrée et qu'elle doit renvoyer une valeur.

```
type_retour NOM DE LA FONCTION(type_1 var_1, type_2 var_2, ...)
    {Déclaration des variables locales}
...
Début
    ...
    ...
        {Liste des instructions}
    ...
    ...
    Renvoyer     {une valeur de type type_retour}
Fin
```

Si a est de type **Réel** et b de type **Entier**, il est ensuite possible d'appeler une fonction POWER prenant deux arguments (le premier de type **Réel** et le second de type **Entier**) et renvoyant un **Réel** (par exemple) de la façon suivante : $res = \text{TOTO}(a,b)$

▷ **Question 13** *À titre d'exemple, comment écrire une fonction qui détermine si un nombre n'est pas premier ?* *Réponse* □

Notons quelque-chose d'extrèmement important concernant les variables (même si cette notion est plus sémantique qu'algorithmique). Une variable a une portée. Elle n'est visible que dans une certaine zone. Pour mieux comprendre cette notion, regardons l'exemple suivant :

```
{Porté 1}
Réel i

FOO()
Entier i
Début
    i← 1
    Écrire (i)
Fin

Début
    i← 3.14156926
    Écrire (i)
    FOO()
    Écrire (i)
Fin
```

La sortie de toute mise en œuvre raisonnable de l'algorithme précédent est :

```
3.14156926
1
3.14156926
```

La variable i qui est modifiée ligne 6 dans la fonction FOO() est celle qui est définie ligne 4 et pas celle qui est définie ligne 2. À la sortie de la fontion FOO(), i (définie ligne 2) n'est donc pas modifiée.

Il est donc possible de définir des variables en cours d'exécution du programme. Lorsque l'on est à l'extérieur de la fonction FOO(), il n'est pas possible d'accéder à la variable i définie ligne 4. L'algorithme suivant illustre cette impossibilité.

```
1:  {Porté 2}
FOO()
2:  Entier j
3:  Début
4:      j← 1
5:  Fin
6:
7:  Début
8:      FOO()
9:      Écrire (j)
10: Fin
```

Si l'on essayait de programmer cet algorithme tel quel, le compilateur nous dirait qu'en ligne 9, la variable j n'a pas été déclarée.

2 Équivalence langage algorithmique / langage C

```
1: {Mon joli programme}
2: {Déclaration des variables}
3: Entier a
4: Début
5:     a← 1
6:     {Une autre affectation...}
7:     a← 3
8: Fin
```

```
1  /******* Mon joli programme ********/
2  /* Déclaration des variables */
3  int a;
4  void main() {
5    a = 1;
6    /* Une autre affectation... */
7    a = 3;
8  }
```

```
1: Entier i=17
2: Réel x
3: Caractère c='@'
4: Booléen test=VRAI
5: Réel tab[10]
```

```
1  int   i = 17;
2  float x;
3  char  c = '@';
4  int   test = 1 ;
5  float tab[10];
```

```
1: Entier A
2: Début
3:     Lire (A)
4:     Écrire (A+1 vaut ', A+1)
5: Fin
```

```
1  {
2    int a;
3    scanf("%d",&a);
4    printf("A+1 vaut %d\n", a+1);
5  }
```

```
1: Si (((x = 0.5) Et (i mod 2 = 1)) Ou (x/i < 3)) Alors
2:     Écrire ('x vaut 1/2')
3: Si i ⩾ 0 Alors
4:     a← i + 1
5:     b← a^2
6: Sinon
7:     a← i − 1
8:     b← a^3
```

```
1  ○    if(((x == .5) && (i % 2 == 1)) || (x/i <○3)) {
2         printf("x vaut 1/2\n");
3  ○    }                                          ○
4       if(i >= 0) {
5  ○      a = i+1;                                 ○
6         b = a*a;
7  ○    } else {                                   ○
8         a = i-1;
9  ○      b = a*a*a;                               ○
10      }
```

```
1: Pour i=0 à N − 1 :
2:     Toto(i)
```

```
1  ○    for (i = 0; i < N; i++) {                  ○
2         Toto(i);
3  ○    }                                          ○
```

```
1: Pour i= 1 à 100 de 5 en 5 :
2:     Toto(i)
```

```
1  ○    for (i = 1; i <= 100; i=i+5) {             ○
2         Toto(i);
3  ○    }                                          ○
```

```
1: Pour i=23 à 12 de -1 en -1 :
2:     Toto(i)
```

```
1  ○    for (i = 23; i >=12; i--) {                ○
2         Toto(i);
3  ○    }                                          ○
```

```
1: Entier i, n, acc
2: Début
3:     acc← 0
4:     Lire (n)
5:     Pour i dans {0..n} :
6:         acc ← acc + i × i × i
7:     Écrire (Somme cubes de 1 à ',n,' = ',acc)
8: Fin
```

```
1  ○    {                                          ○
2         int i,n,acc;
3  ○      acc = 0;                                 ○
4         scanf("%d",&n);
5  ○      for(i = 0; i <= n; i++) {                ○
6           acc = acc + i*i*i ;
7  ○      }                                        ○
8         printf("Somme cubes de 1 à %d : %d\n",n,acc);
9  ○    }                                          ○
```

Booléen PAS_PREMIER(**Entier** n)

1: **Entier** j
2: **Début**
3: $j \leftarrow 2$
4: **Tant que** $((j < n)$ **Et** $(n \not\equiv 0[j]))$ **:**
5: $j \leftarrow j+1$
6: **Si** $(j=n)$ **Alors**
7: **Renvoyer** VRAI
8: **Sinon**
9: **Renvoyer** FAUX
10: **Fin**

```c
int pas_premier(int n) {
  int j = 2;
  while((j < n) && (n % j != 0)) {
    j++;
  }
  if(j == n) {
    return 1;
  } else {
    return 0;
  }
}
```

3 Quelques exercices

3.1 Amusettes

3.1.1 Sommation

▷ **Question 14** *Écrire une fonction qui, étant donné un entier n, renvoie $\sum_{i=1}^{n}\sum_{j=1}^{i} i+j$.* *Réponse*

Traduisez votre algorithme en langage C. *Réponse* □

3.1.2 Fibonacci

▷ **Question 15** *Écrire une fonction qui calcule itérativement le n-ème nombre de Fibonnacci défini par $F_0 = F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \leqslant 2$.* *Réponse*

Traduisez votre algorithme en langage C. *Réponse*

Quelle est la complexité en temps et en espace de votre algorithme ? □

▷ **Question 16** *Écrire une fonction qui calcule itérativement le n-ème nombre de Fibonnacci mais avec une complexité en espace plus faible que la version précédente.* *Réponse*

Traduisez votre algorithme en langage C. *Réponse* □

▷ **Question 17** *Écrire une fonction qui calcule récursivement le n-ème nombre de Fibonnacci.* *Réponse*

Traduisez votre algorithme en langage C. *Réponse* □

3.1.3 Catalan

▷ **Question 18** *Écrire une fonction qui calcule le n-ème nombre de Catalan, qui vaut 1, si $n = 1$ et $c_n = \sum_{k=1}^{n-1} c_k c_{n-k}$ sinon.* *Réponse* □

4 Calcul de complexité

Analyser un algorithme revient à prévoir les ressources (i.e. la quantité de mémoire) nécessaires à cet algorithme et à mesurer son temps d'exécution. En général, quand on analyse plusieurs

algorithmes candidats pour un problème donné, on arrive aisément à identifier le candidat le plus efficace. Ce type d'analyse peut révéler plusieurs candidats valables et permet d'éliminer les autres.

L'analyse d'un algorithme, même simple, peut s'avérer difficile. Il est donc nécessaire de se donner des outils mathématiques pour parvenir à nos fins.

4.1 Notation asymptotique

Les fonctions que l'on considère dans cette section sont des fonctions de $\mathbb{N}$ dans $\mathbb{N}$. Soient f et g deux fonctions.

Définition 1. On dira que $f = \mathcal{O}(g)$ si et seulement si il existe $c > 0$ et $n_0 \in \mathbb{N}$ tel que

$$\forall n \geqslant n_0 : f(n) \leqslant c \cdot g(n)$$

Définition 2. On dira que $f = \Omega(g)$ si et seulement si il existe $c > 0$ et $n_0 \in \mathbb{N}$ tel que

$$\forall n \geqslant n_0 : f(n) \geqslant c \cdot g(n)$$

Définition 3. On dira que $f = o(g)$ si et seulement si $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$.

Définition 4. On dira que $f = \Theta(g)$ si et seulement si $f = \mathcal{O}(g)$ et $g = \mathcal{O}(f)$. On remarquera que $f = \Theta(g)$ si et seulement si $g = \Theta(f)$.

▷ **Question 19** *Parmis les affirmation suivantes, lesquelles sont vraies ?*

$f(n)$	$g(n)$	$f = \mathcal{O}(g)$	$g = \mathcal{O}(f)$	$f = o(g)$	$g = o(f)$	$f = \Theta(g)$
$n+1$	$n^4 + 3n - 2$	...	...	...	...	...
$n+4$	$4n$	...	...	...	...	...
$\frac{1}{10}n^3 + 100n^2 + 10000$	n^3	...	...	...	...	...
2^{n+1}	2^n	...	...	...	...	...
2^{2n}	2^n	...	...	...	...	...
$(n+a)^b$	n^b	...	...	...	...	...
n^n	$n!$	...	...	...	...	...
2^n	$n!$	...	...	...	...	...
n *si* $n \equiv 0[2]$ *et 1 sinon*	n	...	...	...	...	...

Réponse □

Remarque. Comme on ne manipule que des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ et que l'on s'intéresse uniquement à leur comportement à l'infini, on pourra sommer les $\mathcal{O}$ et les Θ sans risque.

L'utilisation de la notation $\mathcal{O}$ permet de majorer le comportement d'une fonction et la notation Θ permet d'éviter d'avoir une majoration trop grossière. La notation Ω signifie qu'une fonction croit au moins aussi vite qu'un autre.

Ces notations permettent donc de comparer deux fonctions. On pourra faire l'analogie suivante avec la comparaison sur des nombres réels :

$$\begin{array}{rcl} f(n) = \mathcal{O}(g(n)) & \approx & a \leqslant b \\ f(n) = \Omega(g(n)) & \approx & a \geqslant b \\ f(n) = \Theta(g(n)) & \approx & a = b \\ f(n) = o(g(n)) & \approx & a \ll b \end{array}$$

4.2 Complexité d'un algorithme

Le temps d'exécution d'un algorithme sur une entrée particulière est le nombre d'opération élémentaires exécutées. Reprenons un à un les algorithmes de la section 3.1 et évaluons leur complexité

4.2.1 Sommation

Pour le moment, nous dirons que chaque ligne de notre pseudo-code demande une quantité de temps constante. L'execution de la ligne i prendra donc un temps c_i. Dans l'étude qui va suivre, le temps d'exécution va évoluer d'une formulation brouillonne utilisant tous les coûts des c_i, vers une formulation beaucoup plus simple, plus concise et plus facile à manipuler.

Entier CALCUL_SOMME(**Entier** n)	coût	fois
1: **Entier** $i, j, somme$		
2: **Début**		
3: $somme \leftarrow 0$	c_3	1
4: **Pour** i=1 **à** n **:**	c_4	n
5: **Pour** j=1 **à** i **:**	c_5	$\sum_{i=1}^{n} i$
6: $somme \leftarrow somme + i + j$	c_6	$\sum_{i=1}^{n} i$
7: **Renvoyer** $somme$	c_7	1
8: **Fin**		

Le coût de l'exécution de l'algorithme dépend donc de n. On a donc

$$\begin{aligned} C(n) &= c_4 + c_5 n + c_6 \sum_{j=1}^{n} j + c_7 \sum_{j=1}^{n} j + c_8 \\ &= (c_6 + c_7)\frac{n(n+1)}{2} + c_5 n + (c_4 + c_8) \\ &= \frac{c_6 + c_7}{2} n^2 + \left(\frac{c_6 + c_7}{2} + c_5 \right) n + (c_4 + c_8) \\ &= \Theta(n^2) \end{aligned}$$

4.2.2 Fibonacci

Itératif simple

Entier FIBO_SIMPLE(**Entier** n)	coût	fois
1: **Entier** $tab[n], i$		
2: **Début**		
3: $tab[0] \leftarrow 1$	c_3	1
4: $tab[1] \leftarrow 1$	c_4	1
5: **Pour** i=2 **à** n **:**	c_5	$n-1$
6: $tab[i] \leftarrow tab[i-1] + tab[i-2]$	c_6	$n-1$
7: **Renvoyer** $tab[n]$	c_7	1
8: **Fin**		

Le temps d'exécution de cet algorithme est donc $\Theta(n)$. On remarquera que l'on utilise un tableau de taille n et une variable de boucle. L'occupation mémoire est donc $\Theta(n)$.

Itératif rusé

```
Entier FIBO_RUSé(Entier n)                                   coût      fois
 1: Entier tab[3], i
 2: Début
 3:
 4:   Si n=0 Alors
 5:     Renvoyer 1                                           c5        1
 6:
 7:   Si n=1 Alors
 8:     Renvoyer 1                                           c8        1
 9:   tab[2]← 1                                             c9        1
10:   tab[1]← 1                                             c10       1
11:   tab[0]← tab[1]+tab[2]                                 c11       1
12:   Pour i=3 à n :                                        c12       n − 2
13:     tab[2]← tab[1]                                      c13       n − 2
14:     tab[1]← tab[0]                                      c14       n − 2
15:     tab[0]← tab[1] + tab[2]                             c15       n − 2
16:   Renvoyer tab[0]                                       c16       1
17: Fin
```

Le temps d'exécution de cet algorithme est donc encore $\Theta(n)$. Par contre, on n'utilise qu'un tableau de taille 3 et une variable de boucle. L'occupation mémoire est donc $\Theta(1)$, ce qui est bien meilleur.

Récursif

```
Entier FIBO_RéCURSIF(Entier n)                                coût      fois
 1: Entier res
 2: Début
 3:
 4:   Si n=0 Alors
 5:     Renvoyer 1                                           c5        1
 6:
 7:   Si n=1 Alors
 8:     Renvoyer 1                                           c8        1
 9:   res← FIBO_RéCURSIF(n − 1) + FIBO_RéCURSIF(n − 2)      c9 + C(n − 1) + C(n − 2)
10:   Renvoyer (res)                                        c10       1
11: Fin
```

On a donc

$$\begin{aligned} C(n) &= c_4 + c_5 + c_6 + c_7 + C(n-1) + C(n-2) \\ &= \alpha + C(n-1) + C(n-2) \end{aligned}$$

Il est parfaitement possible d'obtenir une forme close pour $C(n)$ mais nous nous contenterons pour l'instant d'un encadrement.

$$\begin{aligned} C(n) &= \alpha + C(n-1) + C(n-2) \\ &\geqslant \alpha + 2C(n-2) \qquad \text{car } C \text{ est croissante} \\ &\geqslant \alpha + 2(\alpha + 2C(n-4)) \\ &\geqslant \alpha + 2\alpha + 4(\alpha + C(n-6)) \\ &\geqslant \dots \\ &\geqslant \alpha + 2\alpha + 4\alpha + \ldots + 2^{n/2}\alpha \\ &\geqslant \alpha \left(\sum_{i=0}^{n/2} 2^i \right) = \alpha \left(2^{n/2+1} - 1 \right) \end{aligned}$$

Donc $C(n) = \Omega(2^{n/2})$. De même, on a

$$\begin{aligned} C(n) &= \alpha + C(n-1) + C(n-2) \\ &\leqslant \alpha + 2C(n-1) \qquad \text{car } C \text{ est croissante} \\ &\leqslant \alpha + 2(\alpha + 2C(n-2)) \\ &\leqslant \alpha + 2\alpha + 4(\alpha + C(n-3)) \\ &\leqslant \dots \\ &\leqslant \alpha + 2\alpha + 4\alpha + \dots + 2^n\alpha \\ &\leqslant \alpha \left(\sum_{i=0}^{n} 2^i \right) = \alpha \left(2^{n+1} - 1 \right) \end{aligned}$$

On a donc $C(n) = \mathcal{O}(2^n)$

L'évaluation de l'occupation mémoire est un peu différente. Un arbre d'appel permet de se convaincre aisément qu'il est en $\Theta(n)$. Le temps d'exécution de l'algorithme est de l'ordre du nombre de noeuds de l'arbre d'appel alors que l'occupation mémoire est de l'ordre de la hauteur de l'arbre.

5 Calcul du maximum et du minimum

▷ **Question 20** *Écrire un algorithme* ÉCRIRE_TABLEAU(*tableau,taille*) *qui affiche à l'écran les différentes valeurs d'un tableau tableau de taille taille.* *Réponse*
Traduisez votre algorithme en langage C. *Réponse* □

▷ **Question 21** *Écrire un algorithme* ÉCRIRE_TABLEAU(*tableau,taille*) *qui affiche à l'écran les différentes valeurs d'un tableau tableau de taille taille.* *Réponse*
Traduisez votre algorithme en langage C. *Réponse* □

▷ **Question 22** *Écrire un algorithme* LIRE_TABLEAU(*tableau,taille*) *qui lit taille élément au clavier et les affecte dans les différentes cellules du tableau tableau.* *Réponse*
Traduisez votre algorithme en langage C. *Réponse* □

Les algorithmes que nous allons écrire dans cette section n'utilisent que des affectations et des comparaisons. Il n'y pas besoin d'effectuer des opérations arithmétiques et c'est le nombre de comparaison que nous allons évaluer pour chacun des algorithmes que nous allons proposer.

5.1 Calcul du maximum

5.1.1 Algorithme

▷ **Question 23** *Écrire une fonction* MAXI_TABLEAU *qui prend en entrée un tableau d'entier et sa taille et renvoie un entier qui est la valeur du plus grand élément du tableau.* *Réponse* □

Une analyse rapide montre que, pour un tableau de taille n, on effectue exactement $n-1$ comparaisons et, au plus, n affectations. Dans quel cas effectue-t-on n affectations ? Dans quel cas, n'en effectue-t-on qu'une ? Cet algorithme est optimal en ce qui concerne le nombre de comparaisons.

▷ **Question 24** *Démontrer l'optimalité de l'algorithme précédent.* *Réponse* □

5.2 Calcul du maximum et du minimum

▷ **Question 25** *Proposez un algorithme naïf permettant de calculer à la fois le minimum et le maximum d'un tableau.* *Réponse* □

5.2.1 Algorithme rusé

L'idée pour améliorer l'algorithme est de regrouper les éléments à comparer par paires, i.e. de comparer à chaque tour de boucle $t[2k]$ et $t[2k+1]$, et ensuite comparer un seul des deux à *min_tmp* (lequel ?), et l'autre à *max_tmp*. Dans la suite, on distinguera bien les cas où la longueur du tableau est paire des cas où elle est impaire.

▷ **Question 26** *Écrire un algorithme un peu plus rusé et effectuant moins de comparaisons.*
Montrer que le nombre de comparaison est $\lceil \frac{n}{2} \rceil + n - 2$. *On peut montrer que cet algorithme est optimal mais celà sort du cadre de ce cours.* □

6 Tri

Qu'est-ce qu'un tri ? On suppose qu'on se donne un suite de n nombres entiers, et on veut les ranger en ordre croissant (ou décroissant) au sens large. Ainsi, pour $n = 7$, la suite $(5, 2, 3, 0, 6, 1, 1)$ devra devenir $(0, 1, 1, 2, 3, 5, 6)$.

Dans tout ce qui suit, on suppose que l'on trie des nombres entiers et que ceux-ci se trouvent dans un tableau *tab*.

6.1 Sélection

L'algorithme de tri le plus simple est le tri par sélection. Il consiste à trouver l'emplacement de l'élément le plus petit du tableau, c'est-à-dire l'entier m tel que $tab(i) \geqslant tab(m)$ pour tout i. Une fois cet emplacement trouvé, on échange les éléments $tab(1)$ et $tab(m)$. Puis on recommence ces opérations sur la suite $(tab(2), tab(3), \ldots, tab(n))$, ainsi on recherche le plus petit élément de cette nouvelle suite et on l'échange avec $tab(2)$. Et ainsi de suite ... jusqu'au moment où on n'a plus qu'une suite composée d'un seul élément $(tab(n))$.

Un exemple numérique pour le tri par sélection est donné ci-dessous dans la figure 1.

i=1	2	1	5	0	9	4
i=2	0	1	5	2	9	4
i=3	0	1	5	2	9	4
i=4	0	1	2	5	9	4
i=5	0	1	2	4	9	5
i=6	0	1	2	4	5	9

Tab. 1 – Exemple de par tri sélection.

▷ **Question 27** *Écrire un algorithme triant un tableau par sélection* *Réponse*

Évaluez la complexité de cet algorithme. *Réponse*

Traduisez cet algorithme en langage C. *Réponse* □

6.2 bulle

Une variante du tri par sélection est le **tri bulle**. Son principe est de parcourir la suite $(tab(1), tab(2), tab(3), \ldots, tab(n))$ en intervertissant toute paire d'éléments consécutifs $(tab(i), tab(i+1))$ non ordonnés. Ainsi aprés un parcours, l'élément maximum se retrouve en $tab(n)$. On recommence alors avec le préfixe $(tab(1), tab(2), tab(3), \ldots, tab(n-1))$, puis $(tab(1), tab(2), tab(3), \ldots, tab(n-2))$ ...

Le nom de tri bulle vient donc de ce que les plus grands nombres se déplacent vers la droite en poussant des bulles successives de la gauche vers la droite.La fonction correspondante utilise un

indice i qui marque la fin du préfixe à trier, et l'indice j qui permet de déplacer la bulle qui monte vers la borne i.

L'exemple numérique précédent (Figure 1) est donné avec le tri bulle ci-dessous dans la figure 2.

i=1	2	1	5	0	9	4
i=2	1	2	0	5	4	9
i=3	1	0	2	4	5	9
i=4	0	1	2	4	5	9
i=5	0	1	2	4	5	9

TAB. 2 – Exemple de tri bulle.

▷ **Question 28** *Écrire un algorithme triant un tableau par sélection* *Réponse*

Évaluez la complexité de cet algorithme. *Réponse*

Traduisez cet algorithme en langage C. *Réponse* □

6.3 fusion

Le tri fusion se base sur le fait qu'il est aisé de fusionner deux listes triées l_1 et l_2 en une liste triée l, et ce en temps $\mathcal{O}(|l_1|+|l_2|)$ (vu en TD et en section 7.2.2). Le tri se fait ensuite en divisant pour règner :

```
Liste TRI_FUSION(Liste L)
1: Début
2:    Séparer la liste L en deux listes L1 et L2 de longueur approximativement égales   O(|l|)
3:    L1 ← TRI_FUSION(L1)                                                               C(|l1|)
4:    L2 ← TRI_FUSION(L2)                                                               C(|l2|)
5:    L ← FUSION(L1,L2)                                                                 O(|l1| + |l2|)
6:    Renvoyer L
7: Fin
```

On obtient donc la récurrence $C(n) \leqslant \alpha.n + 2.C(n/2)$, qui se résout en utilisant le même type de technique qu'en section 4.2.2.

$$\begin{aligned} C(n) &\leqslant \alpha n + 2C(n/2) \\ &\leqslant \alpha n + 2(\alpha n/2 + 2C(n/4)) \\ &\leqslant \alpha n + \alpha n + 4C(n/4)) \\ &\leqslant \alpha n + \alpha n + 4(\alpha n/4 + 2C(n/8)) \\ &\leqslant \alpha n + \alpha n + \alpha n + 8C(n/8)) \\ &\leqslant \underbrace{\alpha n + \ldots + \alpha n}_{i \text{ fois}} + 2^i C(n/2^i) \end{aligned}$$

Or, comme $C(1) = 0$, dès que $n/2^i$ est inférieur à 1, on a $C(n) = \alpha.i.n$. Or $\frac{n}{2^i} \leqslant 1$ ssi $\log n \leqslant \log 2^i = i \log 2$. Donc on a $C(n) = \alpha.(\frac{\log n}{\log 2} + 1).n = \mathcal{O}(n \log n)$, ce qui est bien mieux que les autres tris précédemment étudiés.

7 Structures de données élémentaires

Dans cette section, nous nous intéressons à un certain nombre de structures de données.

7.1 Tableaux, pointeurs et structures

Un *tableau* de taille n est une structure très simple constituée de n emplacements consécutifs en mémoire. Il est donc possible d'accéder à un élément d'un tableau en temps constant pourvu que l'on connaisse sa position (ou indice). Un tableau est donc une structure très simple et très efficace. Il n'est cependant pas possible de modifier la taille d'un tableau, ce qui est gênant pour un certain nombre d'algorithmes. On dit cette structure est statique. Le nombre d'éléments qu'elle contient ne peut pas varier.

Comme nous l'avons vu en section 1.2, une variable est un espace mémoire nommé, de taille fixée. Il est donc possible de parler de l'adresse d'une variable (notée `&var` en C et que nous noterons $\#var$ dans ce cours). L'adresse d'une variable étant une valeur, elle a donc un type : c'est un *pointeur*. On distingue un pointeur sur un **Entier**, d'un pointeur sur un **Caractère**, d'un pointeur sur un **Réel**, même si ce sont tous des adresses. Le type du pointeur est nécessaire pour pouvoir le déréférencer, c'est à dire accéder à la valeur pointée par le pointeur (si `add` est un pointeur, la valeur pointée par `add` est notée `*add` en C et nous la noterons $\langle add \rangle$ dans ce cours). Un pointeur particulier est le pointeur *NIL* (`NULL` en C). Il représente une adresse inaccessible et c'est donc une valeur que l'on utilise pour dire que le pointeur n'a pas de valeur déterminée. Il est primordial de toujours initialiser ses pointeurs à la valeur *NIL*. Cela permet de détecter plus simplement les déréférencement de pointeurs non initialisés... On remarquera qu'un tableau peut être représenté par un pointeur (l'emplacement de son premier élément en mémoire) et un entier (sa taille).

Enfin, une *structure* (ou enregistrement) est un produit nommé de plusieurs types. Définissons un nouveau type **Individu** ainsi qu'une variable *ind* de ce type.

```
1: Structure Individu
2:     Caractère nom[128]
3:     Caractère login[9]
4:     Entier age
5:     Entier QI
6:     Booléen masculin
7: Individu ind
```

Il est ensuite aisé d'accéder aux différents *champs* de cette structure de la façon suivante :

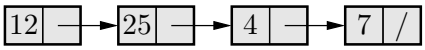
```
1: ind.name ← "Britney Spear"
2: ind.age ← 19
3: ind.QI ← 2
4: ind.masculin ← FAUX
5: ind.login ← "bspear"
```

En C, les *tableaux*, les *pointeurs* et les *structures* sont des structures de données de base (elles sont fournies par le langage). Nous allons maintenant voir un certain nombre d'autres structures de données qui n'existent pas au départ dans le langage C, mais que l'on peut construire assez aisément.

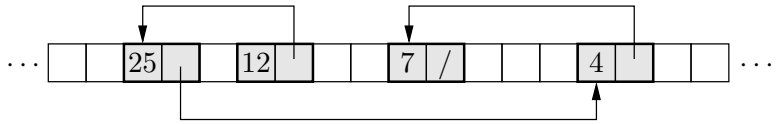
7.2 Liste chaînée

7.2.1 Définitions et exemples

Une *liste chaînée* est une structure de données dans laquelle les éléments sont rangés linéairement. Chaque élément est lié à son successeur et il n'est donc pas possible d'accéder directement à un élément quelconque de la liste.



Bien sûr, cette linéarité est purement virtuelle. À la différence du tableau, les éléments n'ont aucune raison d'êtres contigus ni ordonnés en mémoire.



La façon dont on met en œuvre ces structures dépend des langages même si la façon dont cela est présenté ici ressemble fortement à celle du langage C. Une façon simple de se représenter une liste, consiste à se dire qu'une liste l est soit vide [], soit constituée d'une tête t (qui est donc la valeur du premier élément de la liste) et d'une queue q (qui est le reste de la liste).

La manipulation d'une liste peut donc se faire grâce aux fonctions suivante :

- **Liste** LISTE_VIDE()
- **Liste** CONS(**Entier** t, **Liste** q)
- **Entier** TETE(**Liste** *list*)
- **Liste** QUEUE(**Liste** *list*)

La liste que nous avons donné en exemple peut donc se définir par :

$$l = \text{CONS}(12,\text{CONS}(25,\ \text{CONS}(4,\ \text{CONS}(7\ ,\ \text{LISTE\_VIDE}()))))$$

En terme de pointeurs et de structures, une liste d'entier peut se représenter grâce au type suivant :

```
1: Structure Maillon
2:     Entier valeur
3:     ↑Maillon  suivant
```

La liste que nous avons donné en exemple pourrait se définir par :

```
1: Maillon cell1, cell2, cell3, cell4
2: Début
3:     cell1.valeur ← 12
4:     cell1.suivant ← #cell2
5:     cell2.valeur ← 25
6:     cell2.suivant ← #cell3
7:     cell3.valeur ← 4
8:     cell3.suivant ← #cell4
9:     cell4.valeur ← 7
10:    cell4.suivant ← NIL
11: Fin
```

Pour conclure, retenons simplement qu'il est possible d'accéder en temps constant à la tête de la liste.

7.2.2 Exercices

▷ **Question 29** *Écrire une fonction qui calcule la longueur d'une liste :* *Réponse*

Écrire une fonction qui renvoie le $n^{\text{ème}}$ élément d'une liste : *Réponse*

Écrire une fonction qui indique si un élément appartient à une liste : *Réponse*

Écrire une fonction qui concatène deux listes : *Réponse*

Écrire une fonction qui renverse une liste : *Réponse*

Écrire une fonction qui insère un élément dans une liste triée : *Réponse*

Écrire une fonction qui fusionne deux listes triées : *Réponse* □

Pour vous entrainer, vous pourrez évaluer la complexité en mémoire et en temps de chacune de ces fonctions et les réécrire de façons itérative.

7.3 Pile

Une pile est une structure de donnée dynamique permettant de stocker un ensemble de données. On peut donc insérer un élément ou en supprimer un, à ceci près que l'on ne peut pas choisir l'élément que l'on supprime : c'est le dernier élément inséré. L'ordre dans lequel les éléments sont supprimés est donc inverse de celui dans lequel ils sont insérés (Last In First Out). Une bonne image de cette structure de données est la pile d'assiettes : on peut rajouter une assiette sur une pile d'assiettes mais on ne peut retirer que celle du dessus sans risquer de tout casser.

L'opération d'*insertion* dans une pile est généralement appelée EMPILER. L'opération de *suppression* est souvent appelée DÉPILER. Nous allons mettre en œuvre la structure de pile à l'aide d'un tableau.

```
1:  Structure Pile
2:      Entier sommet
3:      Réel tableau
4:
Booléen Pile-Vide(Pile P)
5:  Début
6:      Renvoyer (P.sommet=-1)
7:  Fin
8:
Empiler(Pile P, Réel x)
9:  Début
10:     P.sommet← P.sommet + 1
11:     P.tableau[P.sommet]← x
12: Fin
13:
Réel Dépiler(Pile P)
14: Début
15:     Si Pile-Vide(P) Alors
16:         Écrire ("Pile vide !")
17:     Sinon
18:         P.sommet← P.sommet − 1
19:         Renvoyer P.tableau[P.sommet + 1]
20: Fin
```

Cependant, dans cette mise en œuvre, comme on inclue notre pile dans un tableau, la taille de la pile est bornée. On peut s'affranchir de cette limitation en utilisant une simple liste chaînée. En effet, empiler un élément consiste simplement à rajouter un maillon, dépiler un élément consiste à en enlever un et à ce que la pile soit représentée par la queue de la liste précédente, tester si la pile est vide consiste à regarder s'il y a un maillon. Le type utilisé pourrait alors ressembler à ceci :

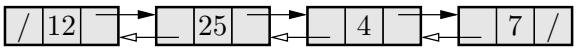
```
1:  Structure Maillon
2:      Réel valeur
3:      ↑Maillon suiv
4:
5:  Liste = ↑Maillon
6:
7:  Structure Pile
8:      Liste tete
```

On peut alors écrire les fonctions précédentes très simplement :

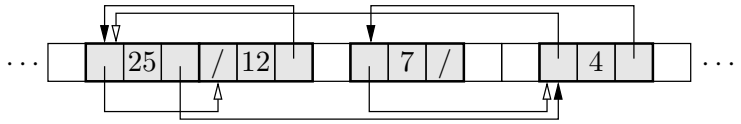
```
Booléen PILE-VIDE(Pile P)
 1: Début
 2:    Renvoyer (P.tete=NIL)
 3: Fin
 4:
 EMPILER(Pile P, Réel x)
 5: Début
 6:    P.tete← CONS(x,P.tete)
 7: Fin
 8:
Réel DÉPILER(Pile P)
 9: Début
10:    Si PILE-VIDE(P) Alors
11:       Écrire ("Pile vide !")
12:    Sinon
13:       haut_pile← TETE(P.tete)
14:       P.tete← QUEUE(P.tete)
15:       Renvoyer haut_pile
16: Fin
```

7.4 Liste doublement chaînée

Une liste doublement chaînée est similaire à une liste chaînée à ceci près qu'il est également possible d'accéder à son prédécesseur.



Bien sûr, cette linéarité est purement virtuelle. Tout comme pour la liste chaînée, les éléments n'ont aucune raison d'êtres contigus ni ordonnés en mémoire.



En terme de pointeurs et de structures, ce type peut donc se définir de la façon suivante.

```
 1: Structure Maillon
 2:    Entier valeur
 3:    ↑Maillon suivant
 4:    ↑Maillon precedant
```

La liste que nous avons donné en exemple pourrait se définir par

```
1:  Maillon cell1, cell2, cell3, cell4
2:  Début
3:      cell1.valeur ← 12
4:      cell1.suivant ← #cell2
5:      cell1.precedant ← NIL
6:      cell2.valeur ← 25
7:      cell2.suivant ← #cell3
8:      cell2.precedant ← #cell1
9:      cell3.valeur ← 4
10:     cell3.suivant ← #cell4
11:     cell3.precedant ← #cell2
12:     cell4.valeur ← 7
13:     cell4.suivant ← NIL
14:     cell4.precedant ← #cell3
15: Fin
```

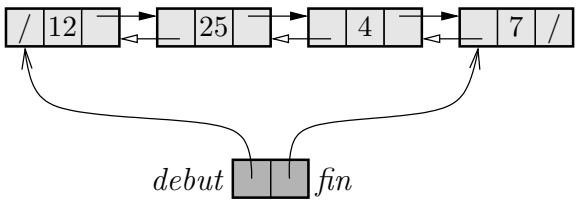
7.5 File

Une file est une structure de donnée dynamique différant d'une file par l'ordre d'extraction des éléments. Le premier élément inséré est le premier à être extrait d'une file. Les éléments sont donc supprimés dans le même ordre que celui dans lequel ils ont été insérés (First In First Out). Une bonne image de cette structure de données est la file d'attente au cinéma, au RU ou dans un supermarché : on fait la queue et, à moins de tricher, il n'est pas possible de sortir de la queue avant les personnes qui ont commencé à faire la queue avant vous.

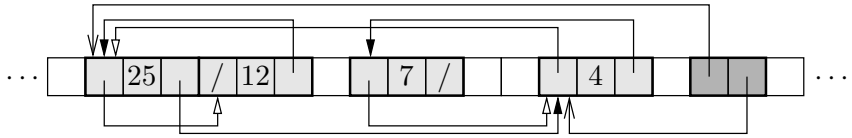
L'opération d'*insertion* dans une pile est généralement appelée ENFILER. L'opération de *suppression* est souvent appelée DÉFILER. Cette structure peut, comme pour une pile, se mettre en œuvre avec un tableau mais elle souffrira des même limitations que pour la pile. Le plus simple consiste à utiliser une liste doublement chaînée :

```
1:  Structure Maillon
2:      Réel valeur
3:      ↑Maillon suivant
4:      ↑Maillon precedant
5:
6:  DListe = ↑Maillon
7:
8:  Structure Pile
9:      ↑Maillon debut
10:     ↑Maillon fin
```

Une file où auraient été enfilées successivement les valeurs 12, 25, 4 et 7 se représenterait donc ainsi :

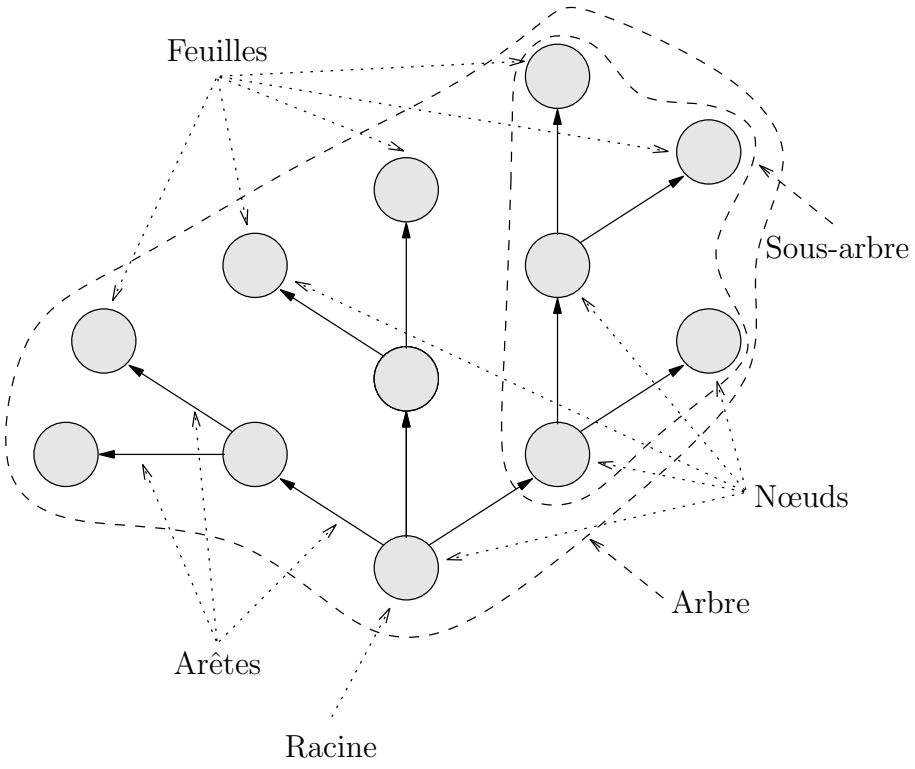


Bien sûr, une fois de plus, cette linéarité est purement virtuelle :



7.6 Arbres

Un arbre est un ensemble de nœuds (appelés aussi parfois sommets) reliés par des arêtes tel que chaque nœud (à part la racine qui en a 0) ait exactement une arête pointant vers lui. La racine est donc un nœud particulier puisqu'il n'a pas de prédécesseur. Les feuilles sont les noeuds sans sucesseurs.

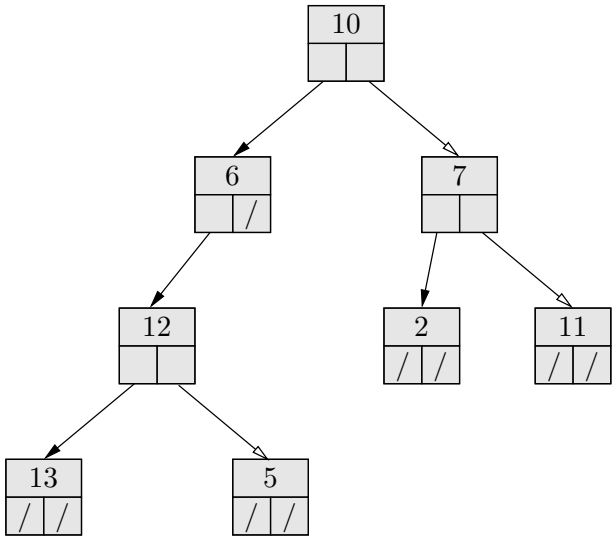


7.6.1 Arbres binaires

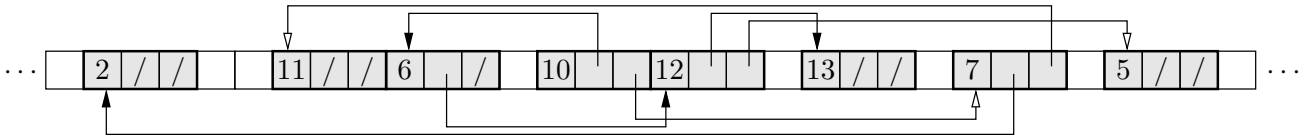
Un arbre binaire est un arbre tel que chaque noeud a au plus deux fils (ou successeurs). Il peut donc se représenter à l'aide de la structure suivante :

```
1: Structure Noeud
2:     Réel valeur
3:     ↑Noeud fg
4:     ↑Noeud fd
5: Un arbre est un ↑Noeud
```

Encore une fois, cette organisation est purement virtuelle. L'arbre suivant



est peut-être stocké ainsi en mémoire...



▷ **Question 30** *La récursivité se prête extrèmement bien à ce type de structure. Pour preuve, écrivons la recherche d'une valeur dans un arbre binaire.*

Réponse

Écrire une fonction qui calcule le nombre de nœuds d'un arbre donné. *Réponse*

Écrire une fonction qui calcule la somme des valeurs des nœuds d'un arbre donné. *Réponse*

Écrire une fonction qui calcule la plus grande des valeurs des nœuds d'un arbre donné.

Réponse □

On suppose disposer du contructeur **Arbre** NEWARBRE(**Entier** *val*, **Arbre** *gauche*, **Arbre** *droit*) qui, étant donné un entier *val* et deux arbres *gauche* et *droit*, construit un arbre dont la racine est étiquetée par *val* et dont le fils droit est *droit* et le fils gauche *gauche*. Ce constructeur, en association avec l' Arbre_Vide , permet de construire n'importe quel arbre binaire.

▷ **Question 31** *Il est alors aisé d'écrire une fonction qui convertit une liste en un arbre.*

Réponse

Évidemment, ce n'est pas la seule façon de transformer une liste en arbre. On aurait également pu mettre les valeur sur la gauche de l'arbre. L'idéal (pour l'équilibrage de l'arbre) étant de mettre à chaque étape la moitié de la liste à gauche et l'autre moitié à droite...

On peut également écrire simplement une fonction qui transforme un arbre en liste.

Réponse □

7.6.2 Arbres n-aires

Un arbre où chaque nœud a au plus n sommet est appelé arbre n-aire. Il peut bien évidemment se représenter par la structure suivante :

```
1: Structure Noeud
2:     Réel valeur
3:     ↑Noeud fils[n]
```

Quand il n'est pas possible de borner le degré de l'arbre, il suffit d'utiliser une des structures dynamiques que nous avons vu en début de section (liste, file, ...).

8 Réponses aux exercices

▷ Question 1, page 6

Il suffit de suivre l'évolution des différentes variables :

1:	$A \leftarrow 2$	*(A=2, B= ?, C= ?)*
2:	$A \leftarrow A + 2$	*(A=4, B= ?, C= ?)*
3:	$B \leftarrow A \times 2 + A$	*(A=4, B=12, C= ?)*
4:	$C \leftarrow 4$	*(A=4, B=12, C=4)*
5:	$C \leftarrow B - C$	*(A=4, B=12, C=8)*
6:	$C \leftarrow C + A - B$	*(A=4, B=12, C=0)*
7:	$A \leftarrow B - C \times A$	*(A=12, B=12, C=0)*
8:	$A \leftarrow (B - A) \times C$	*(A=0, B=12, C=0)*
9:	$B \leftarrow (A + C) \times B$	*(A=0, B=0, C=0)*

Elle affecte donc 0 aux variables A,B et C.

▷ Question 2, page 6

1:	$X \leftarrow -5$	*(X=-5, Y= ?, Z= ?)*
2:	$X \leftarrow X^2$	*(X=25, Y= ?, Z= ?)*
3:	$Y \leftarrow -X - 3$	*(X=25, Y=-28, Z= ?)*
4:	$Z \leftarrow (-X - Y)^2$	*(X=25, Y=-28, Z=9)*
5:	$X \leftarrow -(X + Y)^2 + Z$	*(X=0, Y=-28, Z=9)*
6:	$Y \leftarrow Z^X \times Y$	*(X=0, Y=-28, Z=9)*
7:	$Y \leftarrow -(Z + Y)$	*(X=0, Y=19, Z=9)*
8:	$X \leftarrow X + Y - Z$	*(X=10, Y=19, Z=9)*
9:	$Y \leftarrow X + Z$	*(X=10, Y=19, Z=9)*
10:	$X \leftarrow (Y - Z)^2$	*(X=100, Y=19, Z=9)*
11:	$Y \leftarrow X - Y$	*(X=100, Y=81, Z=9)*
12:	$X \leftarrow (Y + Z)/(X/10)$	*(X=9, Y=81, Z=9)*
13:	$Y \leftarrow ((X \times Z)/Y) \times 9$	*(X=9, Y=9, Z=9)*

Elle affecte 9 à X, Y et Z.

▷ Question 3, page 7

```
1: {Inversion de deux variables}
2: Entier A, B, X
3: Début
4:     Lire A
5:     Lire B
6:     X ← A
7:     A ← B
8:     B ← X
9:     Écrire ('La valeur de A est ', A)
10:    Écrire ('La valeur de B est ', B)
11: Fin
```

▷ Question 4, page 7

```
1: {Inversion de deux variables}
2: Entier A, B
3: Début
4:     Lire A
5:     Lire B
6:     A ← A + B
7:     B ← A − B
8:     A ← A − B
9:     Écrire ('La valeur de A est ', A)
10:    Écrire ('La valeur de B est ', B)
11: Fin
```

Néanmoins, cela ne marche que parce que les variables sont de type **Entier**. Ça ne marcherait pas avec des variables de type **Caractère**.

▷ Question 5, page 7

```
1: {Exponentiation}
2: Entier X
3: Début
4:     Lire X
5:     X ← X × X
6:     X ← X × X
7:     X ← X × X
8:     X ← X × X
9:     Écrire ('La valeur de X^16 est ', X)
10: Fin
```

▷ Question 6, page 7

```
1: {TTC}
2: Réel TTC, HT
3: Début
4:     Lire TTC
5:     HT ← TTC/1,186
6:     Écrire ('Prix hors-taxe : ', HT)
7: Fin
```

▷ Question 7, page 7

```
1: {Exponentiation}
2: Réel M
3: Entier A, B
4: Début
5:    Lire A
6:    Lire B
7:    M← (A + B)/2
8:    Écrire ('Moyenne de A et de B :', M)
9: Fin
```

▷ Question 8, page 7

```
1: {Horaire}
2: Réel M
3: Entier J, H, M, S, Temps
4: Début
5:    Écrire ('Entrez le temps à convertir')
6:    Lire Temps
7:    J← Temps/(24 × 60 × 60)
8:    Temps← Temps − J × (24 × 60 × 60)
9:    H← Temps/(60 × 60)
10:   Temps← Temps − H × (60 × 60)
11:   M← Temps/(60)
12:   Temps← Temps − M × 60
13:   S← Temps
14:   Écrire ('Soit ',J,' Jours, ', H,' Heures,', M,' Minutes et ', S ,' Secondes')
15: Fin
```

▷ Question 9, page 7

```
1: {Horaire2}
2: Entier H1, M1, S1, H2, M2, S2, Temps1, Temps2
3: Début
4:    Écrire ('Horaire 1')
5:    Écrire ('Heure')
6:    Lire H1
7:    Écrire ('Minute')
8:    Lire M1
9:    Écrire ('Seconde')
10:   Lire S1
11:   Écrire ('Horaire 2')
12:   Écrire ('Heure')
13:   Lire H2
14:   Écrire ('Minute')
15:   Lire M2
16:   Écrire ('Seconde')
17:   Lire S2
18:   Temps1← (H1 × 60 + M1) × 60 + S1
19:   Temps2← (H2 × 60 + M2) × 60 + S2
20:   Temps1← (Temps2 − Temps1)
21:   Si Temps1 < 0 Alors
22:      Temps1← −Temps1
23:   H1← Temps1/(60 × 60)
24:   Temps1← Temps1 − H1 × (60 × 60)
25:   M1← Temps1/(60)
26:   Temps1← Temps1 − M1 × 60
27:   S1← Temps1
28:   Écrire ('La différence est de ',H,' Heures,', M,' Minutes et ', S ,' Secondes')
29: Fin
```

▷ Question 10, page 8

$A = 3,\ B = 3,\ C = 9$

▷ Question 10 (suite)

$A = 0,\ B = 1,\ C = 3$

▷ Question 10 (suite)

$A = -4,\ B = -3,\ C = -12$

▷ Question 10 (suite)

$A = 9,\ B = 9,\ C = 9$

▷ Question 10 (suite)

$A = 11,\ B = 11,\ C = 100$

▷ Question 11, page 8

```
1:  Réel A, B, C, Delta, R1, R2
2:  Début
3:    Lire (A)
4:    Lire (B)
5:    Lire (C)
6:    (Delta) ← B^2 − 4 × A × C
7:    Si Delta > 0 Alors
8:      R1 ← (−B + √Delta)/(2 × A)
9:      R2 ← (−B − √Delta)/(2 × A)
10:     Écrire (' Les Racines sont ', R1, ' et ',R2)
11:   Sinon
12:     Si Delta = 0 Alors
13:       R1 ← −B/(2 × A)
14:       Écrire (' La Racine est ', R1)
15:     Sinon
16:       Écrire (' Il n'y a pas de racine réelle. ')
17: Fin
```

▷ Question 12, page 9

```
1:  Entier i, n, acc
2:  Début
3:    acc← 0
4:    Lire (n)
5:    Pour i dans {0..n} :
6:      acc ← acc + i × i × i
7:    Écrire ('La somme des cubes de 1 à ',n,' est ',acc)
8:  Fin
```

▷ Question 13, page 10

```
Booléen PAS_PREMIER(Entier n)
1:  Entier j
2:  Début
3:    j← 2
4:    Tant que (j < n) Et (n modulo j ≠ 0) :
5:      j← j+1
6:    Si j=n Alors
7:      Renvoyer VRAI
8:    Sinon
9:      Renvoyer FAUX
10: Fin
```

▷ Question 14, page 13

Entier CALCUL_SOMME(**Entier** n)
1: **Entier** $i, j, somme$
2: **Début**
3: $somme \leftarrow 0$
4: **Pour** i=1 **à** n **:**
5: **Pour** j=1 **à** i **:**
6: $somme \leftarrow somme + i + j$
7: **Renvoyer** $somme$
8: **Fin**

▷ Question 14 (suite)

```
1   int calcul_somme(int n) {
2     int i, j, somme = 0;
3     for (i = 1; i <= n; i++) {
4       for (j = 1; j <= i; j++) {
5         somme = somme + i + j;
6       }
7     }
8     return (somme);
9   }
```

▷ Question 15, page 13

Entier FIBO_SIMPLE(**Entier** n)
1: **Entier** $tab[n], i$
2: **Début**
3: $tab[0] \leftarrow 1$
4: $tab[1] \leftarrow 1$
5: **Pour** i=2 **à** n **:**
6: $tab[i] \leftarrow tab[i-1] + tab[i-1]$
7: **Renvoyer** $tab[n]$
8: **Fin**

▷ Question 15 (suite)

```
1   int calcul_Fibonnacci_iteratif_simple(int n)
2   {
3     int tab[100];
4     int i = 2;
5
6     if (n <= 0)
7       return -1;
8     tab[0] = 1;
9     tab[1] = 1;
10    for (i = 2; i <= n; i++) {
11      tab[i] = tab[i - 1] + tab[i - 2];
12    }
```

```
13 ○       return (tab[n]);                                                    ○
14         }
```

▷ Question 16, page 13

Entier FIBO_RUSé(**Entier** n)

1: **Entier** $tab[3], i$
2: **Début**
3: **Si** n=0 **Alors**
4: **Renvoyer** 1
5: **Si** n=1 **Alors**
6: **Renvoyer** 1
7: $tab[2] \leftarrow$ 1
8: $tab[1] \leftarrow$ 1
9: $tab[0] \leftarrow tab[1]+tab[2]$
10: **Pour** i=3 **à** n **:**
11: $tab[2] \leftarrow tab[1]$
12: $tab[1] \leftarrow tab[0]$
13: $tab[0] \leftarrow tab[1] + tab[2]$
14: **Renvoyer** $tab[0]$
15: **Fin**

▷ Question 16 (suite)

```
1  ○    int calcul_Fibonnacci_iteratif_ruse(int n)                            ○
2       {
3  ○      int tab[3];                                                         ○
4         int i = 2;
5  ○                                                                          ○
6         if (n <= 0)
7  ○        return -1;                                                        ○
8         if (n <= 1)
9  ○        return 1;                                                         ○
10        tab[2] = 1;
11 ○      tab[1] = 1;                                                         ○
12        tab[0] = tab[1] + tab[2];
13 ○      for (i = 3; i <= n; i++) {                                          ○
14          tab[2] = tab[1];
15 ○        tab[1] = tab[0];                                                  ○
16          tab[0] = tab[1] + tab[2];
17 ○      }                                                                   ○
18        return (tab[0]);
19 ○    }                                                                     ○
```

▷ Question 17, page 13

Entier FIBO_RéCURSIF(**Entier** n)
1: **Entier** res
2: **Début**
3: **Si** n=0 **Alors**
4: **Renvoyer** 1
5: **Si** n=1 **Alors**
6: **Renvoyer** 1
7: $res \leftarrow$ FIBO_RéCURSIF$(n-1)$ + FIBO_RéCURSIF$(n-2)$
8: **Renvoyer** (res)
9: **Fin**

▷ Question 17 (suite)

```
1   int calcul_Fibonnacci_recursif(int n)
2   {
3     int res = 0;
4
5     if (n == 0)
6       return 1;
7     if (n == 1)
8       return 1;
9     res = calcul_Fibonnacci_recursif(n-1) + calcul_Fibonnacci_recursif(n-2);
10    return (res);
11  }
```

▷ Question 18, page 13

Entier CATALAN_RéCURSIF(**Entier** n)
1: **Entier** res, k
2: **Début**
3: **Si** n=1 **Alors**
4: **Renvoyer** 1
5: $res \leftarrow 0$
6: **Pour** i=1 **à** $n-1$ **:**
7: $res \leftarrow res +$ CATALAN_RéCURSIF$(k) \times$ CATALAN_RéCURSIF$(n-k)$
8: **Renvoyer** (res)
9: **Fin**

▷ Question 19, page 14

$f(n)$	$g(n)$	$f = \mathcal{O}(g)$	$g = \mathcal{O}(f)$	$f = o(g)$	$g = o(f)$	$f = \Theta(g)$
$n+1$	$n^4 + 3n - 2$	*Vrai*	*Faux*	*Faux*	*Vrai*	*Faux*
$n+4$	$4n$	*Vrai*	*Vrai*	*Faux*	*Faux*	*Vrai*
$\frac{1}{10}n^3 + 100n^2 + 10000$	n^3	*Vrai*	*Vrai*	*Faux*	*Faux*	*Vrai*
2^{n+1}	2^n	*Vrai*	*Vrai*	*Faux*	*Faux*	*Vrai*
2^{2n}	2^n	*Faux*	*Vrai*	*Vrai*	*Faux*	*Faux*
$(n+a)^b$	n^b	*Vrai*	*Vrai*	*Faux*	*Faux*	*Vrai*
n^n	$n!$	*Faux*	*Vrai*	*Vrai*	*Faux*	*Faux*
2^n	$n!$	*Vrai*	*Faux*	*Faux*	*Vrai*	*Faux*
n si $n \equiv 0[2]$ et 1 sinon	n	*Vrai*	*Faux*	*Faux*	*Faux*	*Faux*

▷ Question 20, page 17

ÉCRIRE_TABLEAU(**Entier** *tableau*[], **Entier** *taille*)
1: **Entier** i
2: **Début**
3: **Pour** i=1 **à** *taille* **:**
4: **Écrire** ('Tableau[',i,'] = ',$tableau[i]$)
5: **Fin**

▷ Question 20 (suite)

```
1  ○    void affiche_tableau(int *tab, int n) {                    ○
2         int i;
3  ○      for (i = 0; i < n; i++) {                                 ○
4           printf("Tab[%d]=%d\n",i,tab[i])
5  ○      }                                                         ○
6       }
```

▷ Question 21, page 17

ÉCRIRE_TABLEAU(**Entier** *tableau*[], **Entier** *taille*)
1: **Entier** i
2: **Début**
3: **Pour** i=1 **à** *taille* **:**
4: **Écrire** ('Tableau[',i,'] = ',$tableau[i]$)
5: **Fin**

▷ Question 21 (suite)

```
1  ○    void affiche_tableau(int *tab, int n) {                    ○
2         int i;
3  ○      for (i = 0; i < n; i++) {                                 ○
4           printf("Tab[%d]=%d\n",i,tab[i])
5  ○      }                                                         ○
6       }
```

▷ Question 22, page 17

```
LIRE_TABLEAU(Entier tableau[], Entier taille)
1: Entier i
2: Début
3:    Pour i=1 à taille :
4:       Écrire ('Tableau[',i,'] = ?')
5:       Lire (tableau[i])
6: Fin
```

▷ Question 22 (suite)

```
1  ○     void lire_tableau(int *tab, int n) {                    ○
2          int i;
3  ○       for (i = 0; i < n; i++) {                             ○
4            printf("Tab[%d]= ?\n",i);
5  ○         scanf("%d",&(tab[i]));                              ○
6          }
7  ○     }                                                       ○
```

▷ Question 23, page 17

```
Entier MAXI_TABLEAU(Entier tableau[], Entier taille)
1: Entier i,max
2: Début
3:    max← tableau[1]
4:    Pour i=2 à taille :
5:       Si tableau[i]>max Alors
6:          max← tableau[i]
7:    Renvoyer max
8: Fin
```

▷ Question 24, page 17

Soit $\mathcal{A}$ UN algorithme qui résout le problème du MAX. Appliquons $\mathcal{A}$ à $[t[1]; t[2]; \ldots; t[n]]$, où $t[1]$ est le plus grand élément du tableau. Cet algo renvoie donc $t[1]$. On montre par l'absurde que tout élément qui n'est pas le maximum est comparé au moins une fois à un élément plus grand que lui : supposons que $t[2]$ ne soit pas comparé à plus grand que lui. Alors en appliquant $\mathcal{A}$ sur l entrée $[t[1]; t[1]+1; t[3]; \ldots; t[n]]$, on ne va pas changer le déroulement précédent de l'algorithme et la réponse retournée est incorrecte, ce qui contredit l'hypothèse de départ. Comme il y a n-1 éléments qui ne sont pas le maximum, il y a au moins $n-1$ comparaisons.

▷ Question 25, page 17

```
(Entier ×Entier) MAXI_TABLEAU(Entier tableau[], Entier taille)
 1: Entier i,max,min
 2: Début
 3:    max← tableau[1]
 4:    min← tableau[1]
 5:    Pour i=2 à taille :
 6:       Si tableau[i]>max Alors
 7:          max← tableau[i]
 8:       Si tableau[i]<min Alors
 9:          min← tableau[i]
10:    Renvoyer (max,min)
11: Fin
```

▷ Question 27, page 18

```
Entier TRI_SéLECTION(Entier tableau[], Entier n)
 1: Entier i, j, min_pos, echange
 2: Début
 3:    Pour i=1 à n − 1 :
 4:       min_pos← i
 5:       Pour j=i+1 à n :
 6:          Si tableau[j]<tableau[min_pos] Alors
 7:             min_pos← j
 8:       echange← tableau[i]
 9:       tableau[i]← tableau[min_pos]
10:       tableau[min_pos]← echange
11: Fin
```

▷ Question 27 (suite)

Il est facile de compter le nombre d'opérations nécessaires. À chaque itération, on démarre à l'élément $tab(i)$ et on le compare successivement à $tab(i+1), tab(i+2), \ldots, tab(n)$. On fait donc $n-i$ comparaisons. On commence avec $i=1$ et on finit avec $i=n-1$. Donc on fait $(n-1)+(n-2)+\cdots+2+1 = n(n-1)/2$ comparaisons, et $n-1$ échanges. Le tri par sélection fait donc de l'ordre de n^2 comparaisons.

▷ Question 27 (suite)

```
1  ○    void tri_selection_sans_copie(int* tab,int taille){                ○
2         int i,j,x,ind_min;
3  ○      for(i=0;i<taille-1;i++){                                         ○
4           ind_min=i;
5  ○        for(j=i+1;j<taille;j++){                                       ○
6             if(tab[j]<tab[ind_min]){
7  ○           ind_min=j;                                                  ○
8             }
9  ○        }                                                              ○
10          x=tab[i];
11 ○        tab[i]=tab[ind_min];                                           ○
12          tab[ind_min]=x;
```

```
13        }
14      }
```

▷ Question 28, page 19

Entier TRI_BULLE(**Entier** $tableau[]$, **Entier** n)

```
1:  Entier i, j, echange
2:
3:  Début
4:    Pour i=1 à n − 1 :
5:      Pour j=1 à n − i :
6:        Si tableau[j]>tableau[j + 1] Alors
7:          echange← tableau[j]
8:          tableau[j]← tableau[j + 1]
9:          tableau[j + 1]← echange
10: Fin
```

▷ Question 28 (suite)

On peut compter aussi trés facilement le nombre d'opérations et se rendre compte qu'il s'agit d'un tri en $\Theta(n^2)$ comparaisons et $\mathcal{O}(n^2)$ échanges (si par exemple le tableau est donné en ordre strictement décroissant).

▷ Question 28 (suite)

```
1   void tri_bulle(int* tab,int taille){
2     int i,j,x;
3     for(i=1;i<=taille-1;i++){
4       for(j=0;j<taille-i;j++){
5         if(tab[j]>tab[j+1]){
6           x=tab[j];
7           tab[j]=tab[j+1];
8           tab[j+1]=x;
9         }
10      }
11    }
12  }
```

▷ Question 29, page 21

Entier LONGUEUR(**Liste** L)

```
1: Début
2:   Si L = liste vide Alors
3:     Renvoyer 0
4:   Sinon
5:     Renvoyer (1+LONGUEUR(QUEUE(L))
6: Fin
```

▷ **Question 29 (suite)**

```
Entier INDICE(Entier n,Liste L)
 1: Début
 2:   Si (L = liste vide) Alors
 3:     Écrire ("Liste trop courte ! ")
 4:     Renvoyer -1
 5:   Sinon
 6:     Si (n = 1) Alors
 7:       Renvoyer TÊTE(L)
 8:     Sinon
 9:       Renvoyer INDICE(n − 1,QUEUE(L))
10: Fin
```

▷ **Question 29 (suite)**

```
Booléen APPARTIENT(Entier x, Liste L)
 1: Début
 2:   Si (L = liste vide) Alors
 3:     Renvoyer FAUX
 4:   Sinon
 5:     Si (TÊTE(L)=x) Alors
 6:       Renvoyer VRAI
 7:     Sinon
 8:       Renvoyer APPARTIENT(x,QUEUE(L))
 9: Fin
```

▷ **Question 29 (suite)**

```
Liste CONCAT(Liste L1,Liste L2)
1: Début
2:   Si (L1 = liste vide) Alors
3:     Renvoyer L2
4:   Sinon
5:     Renvoyer CONS(TÊTE(L1), CONCAT(QUEUE(L1),L2))
6: Fin
```

▷ **Question 29 (suite)**

Une première méthode peu efficace...

```
Liste RENVERSE(Liste L)
1: Début
2:   Si (L = liste vide) Alors
3:     Renvoyer L
4:   Sinon
5:     Renvoyer CONCAT(RENVERSE(QUEUE(L)),
       CONS(TÊTE(L),LISTE_VIDE()))
6: Fin
```

Et une seconde bien plus efficace.

```
Liste RenverseAux(Liste L,Liste aux)
 1: Début
 2:    Si (L = liste vide) Alors
 3:       Renvoyer aux
 4:    Sinon
 5:       Renvoyer RenverseAux(Queue(L), Cons(Tête(L),aux))
 6: Fin
 7:
Liste Renverse(Liste L)
 8: Début
 9:    Renvoyer RenverseAux(L,Liste_Vide())
10: Fin
```

▷ **Question 29 (suite)**

```
Liste Insertion_Triée(Entier x, Liste L)
 1: Début
 2:    Si (L = liste vide) Alors
 3:       Renvoyer Cons(x,Liste_Vide()))
 4:    Sinon
 5:       Si (Tête(L)>x) Alors
 6:          Renvoyer Cons(x,L)
 7:       Sinon
 8:          Renvoyer Cons(Tête(L), Insertion_Triée(x,Queue(L)))
 9: Fin
```

▷ **Question 29 (suite)**

```
Liste Fusion(Liste L1,Liste L2)
 1: Début
 2:    Si (L1 = liste vide) Alors
 3:       Renvoyer L2
 4:    Si (L2 = liste vide) Alors
 5:       Renvoyer L1
 6:    Si (Tête(L1)<Tête(L2)) Alors
 7:       Renvoyer Cons(Tête(L1), Fusion(Queue(L1),L2))
 8:    Sinon
 9:       Renvoyer Cons(Tête(L2), Fusion(L1,Queue(L2)))
10: Fin
```

▷ Question 30, page 26

```
Booléen APPARTIENT(Réel x, Arbre A)
 1: Début
 2:   Si (A = Arbre_Vide) Alors
 3:     Renvoyer FAUX
 4:   Sinon
 5:     Si (A.valeur)=x) Alors
 6:       Renvoyer VRAI
 7:     Sinon
 8:       Si (APPARTIENT(x,A.fg)) Alors
 9:         Renvoyer VRAI
10:       Si (APPARTIENT(x,A.fd)) Alors
11:         Renvoyer VRAI
12:       Renvoyer FAUX
13: Fin
```

▷ Question 30 (suite)

```
Entier NOMBRENOEUDS(Arbre A)
1: Début
2:   Si (A = Arbre_Vide) Alors
3:     Renvoyer 0
4:   Sinon
5:     Renvoyer (1+NOMBRENOEUDS(A.fg)+NOMBRENOEUDS(A.fd))
6: Fin
```

▷ Question 30 (suite)

```
Entier SOMMEVALEURNOEUDS(Arbre A)
1: Début
2:   Si (A = Arbre_Vide) Alors
3:     Renvoyer 0
4:   Sinon
5:     Renvoyer (A.valeur+SOMMEVALEURNOEUDS(A.fg)+ SOMMEVALEURNOEUDS(A.fd))
6: Fin
```

▷ Question 30 (suite)

On utilise bien sûr la fonction définie comme ci...

```
Entier MAX3(Entier a, Entier b, Entier c)
 1: Début
 2:   Si (a>b) Alors
 3:     Si (a>c) Alors
 4:       Renvoyer a
 5:     Sinon
 6:       Renvoyer c
 7:   Sinon
 8:     Si (b>c) Alors
 9:       Renvoyer b
10:     Sinon
11:       Renvoyer c
12: Fin
13:
Entier MAXVALEURNOEUDS(Arbre A)
14: Début
15:   Si (A = Arbre_Vide) Alors
16:     Renvoyer 0
17:   Sinon
18:     Renvoyer (MAX3(A.valeur,MAXVALEURNOEUDS(A.fg), MAXVALEURNOEUDS(A.fd)))
19: Fin
```

▷ Question 31, page 26

```
Arbre LIST2TREE(Liste L)
1: Début
2:   Si (L = Liste_Vide) Alors
3:     Renvoyer Arbre_Vide
4:   Sinon
5:     Renvoyer (NEWARBRE(TETE(L), Arbre_vide, LIST2TREE(QUEUE(L))))
6: Fin
```

▷ Question 31 (suite)

```
Liste TREE2LIST(Arbre A)
1: Début
2:   Si (A = Arbre_Vide) Alors
3:     Renvoyer Liste_Vide
4:   Sinon
5:     Renvoyer (CONS(A.valeur, CONCAT(TREE2LIST(A.fg), TREE2LIST(A.fd))))
6: Fin
```