INFO-F-101 – Programmation
Recueil d'Exercices

Nadjet Benseba Emmanuel Dall'Olio Martin De Wulf
Gilles Geeraerts Joël Goossens Christophe Macq
Olivier Markowitch Thierry Massart Patrick Meyer

Septembre 2006

Table des matières

Table de correspondance

Cette table indique dans quels chapitres du syllabus de théorie se trouvent les exercices.

Section	Chapitre du syllabus
Instructions élémentaires	3
Structures itératives	3
Fonctions	4
Tableaux	6
Manipulation de caractères	6
Complexité	9
Tris de tableaux	10
Invariants	11
Classes et pointeurs	13
Listes	14

Chapitre 1

Instructions élémentaires

EX. 1 Si les variables `a`, `b`, `c` contiennent respectivement les nombres 2, 6 et 1, quelles sont leurs valeurs après l'exécution de chacune des suites d'assignations ci-dessous ?

a.
```
a = b;
c = a;
```

b.
```
a = a + 1;
b = c + 1;
c = 2 * c;
```

c.
```
a = b;
b = a;
```

d.
```
c = a;
a = b;
b = c;
```

e.
```
b = - a;
b = 2 * b;
a = b;
```

f.
```
a = a * a;
a = a * a;
```

EX. 2 Écrire un algorithme qui affiche a^{17} (a lu sur *input*) en employant le moins de multiplications possible.

EX. 3 Écrire une suite d'assignations permettant d'échanger les valeurs de 2 variables `a` et `b`.

EX. 4 Qu'écrit sur *output* le programme suivant quand on lui fournit en *input* les valeurs 2 et 6 ?

```
#include <iostream>

using namespace std ;

int main()
{
  int a,b;

  cin >> a;
  a *= 2;
  cin >> b;
  b += a;
  cout << a << endl << b << endl;
}
```

EX. 5 Qu'écrit sur *output* le programme suivant quand on lui fournit en *input* les valeurs 2, 6 et 4 ?

```
#include <iostream>
```

```
using namespace std ;

int main()
{
  int a,b;

  cin >> a >> b;
  a = b;
  cin >> b;
  b += a;
  cout << a << " " << b << endl;
}
```

EX. 6 Qu'écrit sur *output* le programme suivant quand on lui fournit en *input* les valeurs 2 et 6 ?

```
#include <iostream>

using namespace std ;

int main()
{
  int a,b;

  cin >> b >> a;
  a = b+1;
  cout << a << endl;
  a = b+1;
  cout << a << endl;
  a += 1;
  cout << a << " " << (a+1) << endl;
}
```

EX. 7 Qu'écrit sur *output* le programme suivant quand on lui fournit en *input* :

1. les valeurs 2 et 6 ?
2. les valeurs 8 et 3 ?
3. les valeurs 3 et 3 ?

```
#include <iostream>

using namespace std ;

int main()
{
  int a,b;

  cin >> a >> b;
  if (a > b)
  {
    cout << a << endl;
    a = b;
  }
  cout << a << endl;
}
```

EX. 8 Qu'écrit sur *output* le programme suivant quand on lui fournit en *input* :

1. les valeurs 2 et 6 ?
2. les valeurs 8 et 3 ?
3. les valeurs 3 et 3 ?

```
#include <iostream>

using namespace std ;

int main()
{
  int a,b;

  cin >> a >> b;
  if (a > b)
  {
    cout << a << endl;
    a = b;
    cout << a << endl;
  }
}
```

EX. 9 Écrire un programme qui lit 3 nombres, et qui, si deux d'entre eux ont la même valeur, imprime cette valeur (le programme n'imprime rien dans le cas contraire).

EX. 10 Expliquer ce que fait l'algorithme ci-dessous en insérant, après chaque instruction d'affichage en *output*, des commentaires donnant la condition à satisfaire pour que l'instruction d'affichage soit exécutée.

```
int a;

if (a > 0)
{
  if (a > 1)
  {
     if (a > 2)
     {
        cout << (a-2) << endl;
     }
     else
     {
        cout << (a-1) << endl;
     }
  }
  else
  {
    cout << a << endl;
  }
}
else
{
  cout << "Erreur" << endl;
}
```

EX. 11 Écrire le morceau de code qui si, `a` est supérieure à 0, teste si `a` vaut 1, auquel cas il imprime « `a vaut 1` » et qui, si `a` n'est pas supérieure à 0, imprime « `a est inférieure ou égale à 0` ».

EX. 12 Indentez correctement le morceau de code suivant :

```
if (a > 2)
{
if (a > 3)
{
if (a == 4)
cout << "message 1" << endl;
else
cout << "message 2" << endl;
cout << "message 3" << endl;
}
}
else
cout << "message 4" << endl;
```

Justifiez l'utilisation des accolades.

EX. 13 Écrire le programme qui lit en *input* trois entiers a, b et c. Si l'entier c est égal à 1 alors le programme affiche sur l'*output* la valeur de $a + b$, si c vaut 2 alors le programme affiche la valeur de $a - b$, si c égal 3 alors l'*output* sera la valeur de $a \cdot b$. Enfin si la valeur 4 est assignée à c alors le programme affiche la valeur de $a^2 + b \cdot a$. Si c contient une autre valeur le programme affiche un message d'erreur.

EX. 14 Écrire un programme qui imprime la moyenne arithmétique de deux nombres lus sur *input*.

EX. 15 Écrire un programme qui lit deux nombres sur *input* et qui soustrait le plus petit au plus grand et qui affiche le résultat.

EX. 16 Écrire un programme qui lit deux nombres a et b sur *input* et qui calcule et affiche le nombre c tel que b soit la moyenne de a et c.

EX. 17 Écrire un programme qui lit trois nombres sur *input* et qui imprime les deux plus grands.

EX. 18 Si `a`,`b`,`c` sont des variables entières et `arret`, `test1`, `test2`, `test3` des variables booléennes, on demande la valeur de ces variables après chacune des instructions ci-dessous :

```
a=2;
b=3;
c=4;
test1=true;
test2=(b>=a) and (c>= b);
test3=test1 or test2
arret=test3 and (not test2);
a+=1;
b-=1;
c-=2;
test1=true;
test2=(b>=a) and (c>=b);
test3=test1 or test2;
arret=arret or test2;
```

EX. 19 Parmi les douze propositions logiques suivantes, lesquelles sont équivalentes ?

1. $(a > b) \wedge (N \neq 0)$
2. $(a \leq b) \vee (N = 0)$
3. $(a \leq b) \wedge (N = 0)$
4. $\neg(a < b) \wedge \neg(N = 0)$
5. $(a \vee a) \wedge (a \vee \neg a) \wedge (\neg b \vee a) \wedge (\neg b \vee \neg a) \wedge (N = 0)$
6. $\neg(a > b) \wedge (N \neq 0)$
7. $\neg((a > b) \wedge (N \neq 0))$
8. $\neg((a > b) \vee (N \neq 0))$
9. $\neg(a > b) \vee \neg(N \neq 0)$
10. $\neg(a > b) \wedge \neg(N \neq 0)$
11. $(a > b) \vee (N \neq 0)$
12. $\neg(a \Rightarrow b) \wedge (N = 0)$

EX. 20 Écrire le code calculant la valeur de chacune des propositions précédentes.

EX. 21 Écrire de manière optimisée et en n'utilisant que les opérateurs logiques `et`, `ou` et `not`, le code correspondant à l'expression logique suivante (où $\oplus$ représente un `ou` exclusif) :

$$b \wedge \neg(a \oplus b) \wedge (a \vee \neg b) \wedge (\neg a \Rightarrow b)$$

Chapitre 2

Structures itératives

2.1 Boucles simples

EX. 22 Que fait cette suite d'instructions :

```
a=1;
while (a <= 5)
{
  cout << a << " ";
  a+=1;
}
cout << a << endl;
```

EX. 23 Que fait cette suite d'instructions :

```
a=1;
do
{
  cout << a << " ";
  a+=1;
}
while (a <= 5);
cout << a << endl;
```

EX. 24 Que fait cette suite d'instructions :

```
a=0;
while (a < 5)
{
  a+=1;
  cout << a << " ";
}
cout << a << endl;
```

EX. 25 Que fait cette suite d'instructions :

```
a=0;
do
{
  a+=1;
  cout << a;
```

```
}
while(a != 5);
cout << a << endl;
```

EX. 26 Écrire un programme qui lit un nombre `n` sur *input* et qui affiche tous les nombres naturels entre 0 et `n` compris, de manière croissante.

Variantes :

1. Tous les nombres entre 0 et `n` compris, de manière décroissante ;
2. Tous les nombres pairs entre 0 et `n` compris, de amnière croissante ou décroissante ;
3. Tous les nombres entre 0 et `n` bornes non-comprises, de manière coirssante ;
4. *etc.* Inventez-en d'autres !

EX. 27 Écrire un programme qui calcule la moyenne des résultats d'une interrogation, ces notes étant données en *input*, la fin des données étant signalée par la valeur sentinelle -1 (on suppose aussi que la suite des notes contient toujours au moins un élément).

EX. 28 Écrire l'équivalent de l'instruction suivante avec une boucle `while` et puis avec une boucle `do` :

```
int i;

for(i=a;i<=b;++i)
{
  <instruction;>
}
```

EX. 29 Modifier le code ci-dessous en utilisant une boucle `for` :

```
int i,sum=0;

cin >> i ;
while(i>0)
{
  i--;
  sum+=2*i;
}
cout << sum << endl;
```

EX. 30 Modifier le code ci-dessous en utilisant une boucle `while` :

```
double prod;
int i,cpt;

prod=1;
cpt=0;
for(cin >> i;i<100;++i)
{
  cpt++;
  prod*=i;
}
cout << (prod/cpt) << endl;
```

EX. 31 Écrire un programme qui affiche sur output la valeur de $n!$ (où $n! = n \times (n-1) \times (n-2) \cdots 2 \times 1$) avec n lu sur *input*.

EX. 32 Écrire un programme qui lit sur *input* une liste de valeurs entières se trouvant dans l'intervalle $[-max, max]$ et terminée par une valeur sentinelle $> max$ (on peut prendre $max = 100$ pour fixer les idées). Écrire un programme qui affiche le nombre de valeurs < 0 de cette liste de valeurs.

EX. 33 Écrire un programme qui affiche sur *output* les nombres entiers strictement positifs dont le carré est inférieur à un nombre entier n lu sur *input*.

EX. 34 Écrire un programme qui affiche sur output les puissances de 2 à exposants entiers positifs qui sont inférieures à n lu sur *input*.

EX. 35 Écrire un programme qui affiche sur output les 10 premières puissances (à exposants entiers > 0) d'un nombre a lu sur *input*.

EX. 36 Écrire le programme qui lit en *input* un nombre $Max > 0$ et une suite de nombres entiers strictement positifs. On demande d'afficher sur l'*output* les premiers éléments de cette suite, déterminés par la condition que leur somme ne dépasse pas Max. De façon précise, si $a_1, a_2, a_3, \ldots, a_n$ est la suite de nombres, on demande d'afficher les nombres $a_1, a_2, a_3, \ldots, a_k$ tels que $a_1 + a_2 + \cdots + a_k \leq Max$ et si $k < n$, $a_1 + a_2 + \cdots + a_{k+1} > Max$.

Aucune hypothèse n'est faite quant à l'existence d'une solution. La suite de nombres est suivie d'une valeur sentinelle -1.

EX. 37 Écrire le programme qui lit en *input* deux nombres b_i, b_s et puis une liste non vide de valeurs entières triées par ordre croissant. On demande d'afficher sur l'*output* tous les nombres de cette liste appartenant à l'intervalle $[b_i, b_s]$. On suppose que la dernière valeur de la liste est toujours strictement supérieure à b_s.

Avec :

3	25	-7	-5	1	3	12	18	42	51
b_i	b_s	liste							

On imprime :

3 12 18

EX. 38 On suppose que sur l'*input* se trouve une suite d'entiers positifs, terminée par la valeur sentinelle -1. Comparer le premier (noté A) et le dernier élément (noté B) de la suite et afficher un message suivants :

A `est inférieur à` B
A `est strictement supérieur à` B
`La suite est vide`

Si la suite ne compte qu'un élément, on considère que $A = B$.

EX. 39 Ecrire un programme qui calcule F_n le n^{e} nombre de FIBONACCI dont la définition est la suivante : $\forall n \geq 2 : F_n = F_{n-1} + F_{n-2}$ avec $F_0 = 0$ et $F_1 = 1$.

EX. 40 Le *nombre d'or* est la limite de la suite :

$$F_2/F_1\ ,\ F_3/F_2\ ,\ F_4/F_3\ ,\ \ldots\ ,\ F_{n+1}/F_n\ ,\ \ldots$$

où F_n, le nombre de FIBONACCI de rang n, est défini par :

$$F_n = F_{n-1} + F_{n-2} \qquad F_0 = 0 \qquad F_1 = 1$$

Écrire un algorithme qui calcule ce nombre d'or avec une précision ε (ε est lu sur *input*, $1.0E-5$ par exemple).

EX. 41 Que réalise l'algorithme suivant ?

```
#include <iostream>

using namespace std ;

int main()
{
 int d, r;

 cin >> r >> d;
 while ( r >= d )
 {
   r -= d;
 }
 cout << r << endl;
}
```

EX. 42 Que réalise l'algorithme suivant ? Utilisez des exemples pour vous donner l'intuition.

```
#include <iostream>

using namespace std ;

int main()
{
 int a, n, d, r;

 cin >> a >> n;
 r=a;
 d=n;
 while ( r >= d )
 {
   d *= 3;

 }
 while (d != n)
 {
   d /= 3;

   while ( r >= d )
   {
   r -= d;
   }
 }
 cout << r << endl;
}
```

EX. 43 Écrire un programme qui lit en *input* une suite de nombres strictement positifs (sentinelle : 0, la suite comporte au moins deux nombres) et indique si elle est croissante, décroissante, strictement croissante, strictement décroissante ou non triée.

EX. 44 Écrire le code qui calcule la valeur approchée de π sur base de la série suivante (due à Leibniz). Les calculs s'arrêtent lorsque la valeur du dernier terme ajouté est inférieure à un ε

donné, ou lorsque le nombre de termes considérés atteint une borne également donnée.

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$$

EX. 45 Écrire le code qui calcule la valeur approchée de π sur base de la série suivante. Les calculs s'arrêtent lorsque la valeur du dernier terme ajouté est inférieure à un ε donné, ou lorsque le nombre de termes considérés atteint une borne également donnée.

$$\frac{\pi}{8} = \frac{1}{1 \cdot 3} + \frac{1}{5 \cdot 7} + \frac{1}{9 \cdot 11} + \cdots$$

EX. 46 Écrire le code qui calcule la valeur approchée de π sur base de la série suivante (due à Euler). Les calculs s'arrêtent lorsque la valeur du dernier terme ajouté est inférieure à un ε donné, ou lorsque le nombre de termes considérés atteint une borne également donnée.

$$\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots$$

EX. 47 Le nombre e peut être défini comme la limite d'une série :

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!} + \cdots = \sum_{i \geq 0} \frac{1}{i!}$$

Il est connu que l'approximation :

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}$$

diffère de e d'au plus deux fois le terme suivant dans la série, c'est-à-dire d'au plus $2 \cdot \frac{1}{(n+1)!}$.

Écrire le code calculant e avec une approximation dont l'erreur est inférieure à une constante donnée (en ignorant les imprécisions dues aux arrondis sur machine).

✍: 25 min.

EX. 48 On peut calculer approximativement le sinus de x en effectuant la sommation des n premiers termes de la série infinie

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

où x est exprimé en radians.
Réécrire cette somme sous la forme $\sin(x) = \sum_{i \geq 0} f(i, x)$. (On vous demande donc de trouver $f(i, x)$). Écrire ensuite le code calculant de cette manière la valeur de $\sin(x)$ où x est lu sur *input*.
Continuer l'addition des termes successifs dans la série jusqu'à ce que la valeur d'un terme devienne inférieure (en amplitude) à une constante ε (exemple : $\varepsilon = 10^{-5}$)

2.2 Boucles imbriquées

EX. 49 Écrire une programme qui lit sur *input* une valeur naturelle `n` et qui affiche à l'écran un carré de `n` caractères `X` de côté, comme suit (pour `n` $= 6$) :

```
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
```

Variantes : afficher uniquement le bord du carré, afficher un rectangle, afficher le triangle supérieur,...

EX. 50 Écrire un programme qui lit sur *input* une valeur naturelle n et qui affiche ensuite à l'écran toutes les paires (i, j) telles que $0 \leq i \leq \mathtt{n}$ et $0 \leq j \leq \mathtt{n}$.

Ces paires seront affichées sur n lignes de la façon suivante :

$$\begin{array}{cccc} (0,0) & (0,1) & \cdots & (0,\mathtt{n}) \\ (1,0) & (1,1) & \cdots & (1,\mathtt{n}) \\ \vdots & \vdots & \vdots & \vdots \\ (\mathtt{n},0) & (\mathtt{n},1) & \cdots & (\mathtt{n},\mathtt{n}) \end{array}$$

EX. 51 Écrire un progamme qui lit sur *input* une valeur naturelle n et qui affiche ensuite à l'écran toutes les paires (i, j) telles que $0 \leq i \leq \mathtt{n}$ et $0 \leq j \leq i$. Ces paires seront affichées sur n lignes de la façon suivante :

$$\begin{array}{cccccc} (0,0) & & & & & \\ (1,0) & (1,1) & & & & \\ \vdots & \vdots & \vdots & \vdots & & \\ (\mathtt{n}-1,0) & (\mathtt{n}-1,1) & \cdots & (\mathtt{n}-1,\mathtt{n}-1) & & \\ (\mathtt{n},0) & (\mathtt{n},1) & \cdots & \cdots & & (\mathtt{n},\mathtt{n}) \end{array}$$

EX. 52 Refaire l'exercice 49 en supposant que n est impaire et en dessinant des 0 sur les deux diagonales principales à la places des X.

Exemple, pour $\mathtt{n} = 7$:

```
0XXXXX0
X0XXX0X
XX0X0XX
XXX0XXX
XX0X0XX
X0XXX0X
0XXXXX0
```

Chapitre 3

Les Fonctions

EX. 53 Écrire une fonction à valeur entière qui calcule x exposant i.

EX. 54 Écrire une fonction à valeur booléenne qui teste la parité d'un nombre.

EX. 55 La distance euclidienne entre 2 points de coordonnées (x_1, y_1) et (x_2, y_2) est donnée par la formule :

$$s = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Écrire une fonction qui reçoit les coordonnées de 2 points et calcule la distance entre ces 2 points (vous pouvez utiliser la fonction `sqrt`).

EX. 56 Écrire une fonction à valeur booléenne qui teste la primalité d'un nombre.

EX. 57 Écrire une fonction à valeur entière qui retourne le nombre de nombres premiers strictement inférieurs à une valeur fournie.

EX. 58 Écrire la fonction `swap` qui effectue l'échange de ses deux paramètres entires `a` et `b`.

EX. 59 Écrire la fonction `Fibonacci` qui reçoit en paramètre 2 nombres de Fibonacci consécutifs, disons F_n et F_{n+1} pour fixer les idées. Après l'appel, on souhaite disposer des 2 nombres Fibonacci suivants, c'est-à-dire F_{n+2} et F_{n+3}.

EX. 60 Écrire une fonction qui trie trois variables de manière croissante. Par exemple, considérons les trois variables sont `a`, `b` et `c` contenant respectivement 5, 3 et 7. Après l'appel à la fonction (`trie(a,b,c)`, par exemple), nous aurons `a` $= 3$, `b` $= 5$ et `c` $= 7$.

EX. 61 Chercher les erreurs dans les algorithmes suivants :

(a)
```
// Cette fonction retourne la factorielle de n
int factorielle(int n);
{
  int i:

  for(i=1;i<=N;++i)
  {
    n*=i;
  }
  return(n);
}
```

(b)
```
// Cette fonction multiplie par 2 la valeur contenue dans a.
void fois_2(int a)
{
  a*=2;
}
```

EX. 62 Un algorithme pour trouver le plus grand commun diviseur de deux nombres entiers a été découvert par Silver et Terzian en 1962. Dans beaucoup de cas, il est plus rapide que l'algorithme d'Euclide.

Étant donné deux nombres entiers a et b, l'algorithme procède en trois étapes :

1. Déterminer la plus grande puissance k de 2 qui divise à la fois a et b (où k est un nombre naturel) ; remplacer a par $a/2^k$ et b par $b/2^k$.

2. A présent, a ou b est impair. Si $a \neq b$, faire ce qui suit :

   – $t \leftarrow \mid a - b \mid$
   – Si t est pair, remplacer t par $t/2$. Répéter ceci tant que t est pair.
   – $a \leftarrow t$ si $a > b$, $b \leftarrow t$ sinon.
   – Si $a \neq b$, répéter l'étape 2.

3. à présent, $a = b$. Le plus grand commun diviseur des deux nombres donnés vaut $2^k \cdot a$.

En appliquant cet algorithme à 504 et 420, nous obtenons

a	b
504	420
252	210
126	105
21	105
21	21

Réponse : $2^2 \times 21 = 84$

Écrire une fonction `Silver_Terzian` qui réalise cet algorithme.

 ✍: 25 min. 

EX. 63 Écrire une fonction qui calcule la valeur approchée de π sur base de la série suivante. Les calculs s'arrêtent lorsque la valeur du dernier terme ajouté est inférieure à un ε donné, ou lorsque le nombre de termes considérés atteint une borne également donnée.

$$\frac{\pi}{4} = 4 \cdot arctan\left(\frac{1}{5}\right) - arctan\left(\frac{1}{239}\right)$$

avec la série :

$$arctan(x) = \frac{x}{1+x^2}\left(1 + \frac{2}{3}\cdot\frac{x^2}{1+x^2} + \frac{2}{3}\cdot\frac{4}{5}\cdot(\frac{x^2}{1+x^2})^2 + \cdots\right) = \frac{x}{1+x^2}\sum_{i=0}^{\infty} c_i(\frac{x^2}{1+x^2})^i$$

où $c_0 = 1$
et pour $i > 0, c_i = \frac{2 \cdot i}{2 \cdot i+1} c_{i-1}$

EX. 64 Deux suites (x_i) et (y_i) sont définies par

$$\begin{aligned} y_0 &= 2 \\ x_0 &= 1 \\ x_{i+1} &= \frac{2}{y_{i+1}} \\ y_{i+1} &= \frac{y_i + x_i}{2} \end{aligned}$$

Nous admettons que les suites convergent toutes les deux vers $\sqrt{2}$, avec de plus $x_i < \sqrt{2} < y_i$. Ecrire une fonction `rac2` qui calcule deux approximations supérieure et inférieure de $\sqrt{2}$, de différence inférieure à ε (prévoyez aussi d'arrêter les calculs si le nombre d'étapes dépasse une borne fixée).

EX. 65 Soit r un nombre réel positif. Deux suites de nombres réels x_n et y_n sont spécifiées par :

$x_1 = 1$,
$y_1 = 1$,

et pour $n \geq 1$:

$$\begin{cases} x_{n+1} = x_n + r.y_n \\ y_{n+1} = x_n + y_n \end{cases}$$

Il n'est pas difficile de prouver $y_n \neq 0$ pour $n \geq 1$, et aussi que le quotient $\frac{x_n}{y_n}$ converge vers $\sqrt{r}$.

Écrire une fonction qui calcule la valeur approchée $\frac{x_n}{y_n}$ de $\sqrt{r}$, où n est la plus petite valeur pour laquelle la condition suivante est vraie :
$(x_n)^2 < (r+\varepsilon)(y_n)^2$ **et** $(x_n)^2 > (r-\varepsilon)(y_n)^2$.
(ε étant comme d'habitude une valeur donnée). L'exécution sera interrompue si le nombre d'étapes devient trop grand.

EX. 66 On souhaite disposer d'un module tools contenant les fonctions `maximum`, `minimum`, `abs` pour des nombres entiers. On vous demande d'écrire ce module, le fichier d'en-tête correspondant et un exemple de module principal qui l'utilise.

EX. 67 Dans certaines conditions, on peut calculer une abscisse où une fonction f s'annule par la méthode de Newton.

Partant d'une approximation initiale x_0 (valeur quelconque qu'il vaut mieux prendre près de l'abscisse cherchée) on calcule les approximations suivantes $x_1, x_2, ..., x_i, x_{i+1}$ par la formule suivante :

$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$

On s'arrête lorsque l'approximation x_i est suffisamment précise.

De cette façon on peut calculer la racine n^{me} d'un nombre a. Il faut calculer la valeur de x pour laquelle la fonction $f(x) = x^n - a$ s'annule.

Ayant $f'(x) = n \cdot x^{n-1}$ on obtient l'évaluation suivante de x :

$x_{i+1} = x_i - \frac{x_i^n - a}{n \cdot x_i^{n-1}}$

On calcule ainsi les approximations successives pour $\sqrt[n]{a}$ jusqu'à ce que :

$|{x_i}^n - a| \leq \varepsilon$ ($\varepsilon = 10^{-5}$ par exemple).

Écrire une fonction racine à valeur réelle qui calcule la racine n^{me} d'un nombre a (a réel et n entier donnés en paramètre). Cette fonction prend $x_0 = 1$.

✍: 40 min.

EX. 68 Écrire une fonction `pyramide` qui crée une pyramide de hauteur n (n passé comme paramètre) avec les chiffres suivants (à l'aide de boucles imbriquées) :

```
       1
      232
     34543
    4567654
   567898765
  67890109876
 7890123210987
890123454321098
 90123456765432109
 0123456789876543210
123456789010987654321
```

(Ne pas écrire 11 chaînes de caractères à plusieurs chiffres).

Chapitre 4

Les tableaux

4.1 Tableaux Simples

EX. 69 Écrire un programme qui lit une série de valeurs lues au clavier et qui ensuite les affiche dans l'ordre inverse. La fin de la série est marquée par la lecture de la valeur -1 et on supposera qu'il n'y aura jamais plus de 100 valeurs lues.

EX. 70 Écrire la fonction `int find_max(int t[],int n)` qui renvoie l'indice du plus grand des éléments du tableau `t`, qui est de taille `n`. On supposera que le tableau n'est pas vide.

EX. 71 Écrire une fonction `symétrie_vecteur` qui opère une symétrie sur les éléments d'un vecteur V de dimension n, c'est-à-dire échange la première et la dernière composante du vecteur, la deuxième et l'avant-dernière,... $(v_1, v_2, \ldots, v_n)$ devient $(v_n, v_{n-1}, \ldots, v_1)$.

EX. 72 Écrire une fonction `produit_scalaire` qui renvoie le produit scalaire de deux vecteurs de longueur n donnés en paramètre (u, v) :

$$\sum_{i=1}^{n} u_i \cdot v_i$$

EX. 73 Écrire une fonction qui tasse un vecteur d'entiers : tous les zéros se retrouvent à la fin, l'ordre des autres nombres ne doit pas être conservé.

Exemple : (0,1,0,2,0,0,3,4) devient (4,1,3,2,0,0,0,0).

EX. 74 Écrire une fonction qui tasse un vecteur d'entiers : tous les zéros se retrouvent à la fin et l'ordre des autres nombres n'est pas modifié.
Exemple : (1,9,0,0,7,8,0,4) devient (1,9,7,8,4,0,0,0).

EX. 75 Soient deux vecteurs POS et DON de longueur n. Écrire une fonction `imprime_ind` qui affiche les éléments de DON dans l'ordre indiqué par POS (qui contient les entiers de 1 à n).

Exemple ($n = 4$)

DON	9	6	8	2

POS	2	4	3	1

donne 6 2 8 9.

EX. 76 (QUESTION DE L'EXAMEN D'AOÛT 2006) Un nombre premier est un entier naturel strictement supérieur à 1, n'admettant que deux entiers naturels diviseurs distincts : 1 et lui-même.

On vous demande d'écrire un crible, c'est-à-dire une fonction qui trouve tous les nombres premiers inférieurs à un entier n donné.

La fonction devra avoir la signature `void crible(int t[],int n)` et remplira le tableau `t` de tous les nombres premiers inférieurs à `n`. La fin de la liste de nombres premiers sera marquée par la valeur sentinelle -1. Le tableau `t` sera supposé suffisamment grand pour stocker `n+1` entiers.

Par exemple, après l'appel `crible(t,20)`, le tableau `t` aura le contenu suivant :

2	3	5	7	11	13	17	19	-1	$\cdots$

L'évaluation de votre solution sera principalement basée sur son efficacité.

EX. 77 Le coefficient binomial C_n^p est la valeur $\frac{n!}{p!(n-p)!}$. Les coefficients binomiaux peuvent être calculé itérativement, sans effectuer ni multiplication, ni division, au moyen des formules suivantes :
$C_0^0 = 1$
$C_1^0 = 1$
$C_n^p = 0$ si $p > n$ ou si $p < 0$
$C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$ sinon.
La représentation classique se fait alors sous la forme d'un triangle, appelé Triangle de Pascal.

```
ligne 0      1
ligne 1      1   1
ligne 2      1   2   1
ligne 3      1   3   3   1
ligne 4      1   4   6   4   1
ligne 5      1   5   10  10  5    1
ligne 6      1   6   15  20  15   6   1
ligne 7      1   7   21  35  35   21  7   1
```

Les coefficients C_n^k, $0 \leq k \leq n$ figurent à la n^{e} ligne. Le triangle est construit en plaçant des 1 aux extrémités de chaque ligne et en complétant la ligne en reportant la somme de deux nombres adjacents de la ligne supérieure : celui juste au dessus, et celui à sa gauche.

Écrire une fonction `void pascal(vecteur v, int n)` qui renvoie dans le vecteur `v` la n^{e} ligne du triangle de Pascal. Le tableau sera supposé suffisamment grand.

EX. 78 Écrire une fonction `gliss_1_g` effectuant un glissement circulaire gauche d'une position sur un vecteur.

✍: 40 min.

EX. 79 Écrire une fonction qui effectue un glissement circulaire gauche de K positions sur un vecteur. Chaque élément ne doit être déplacé qu'une seule fois et votre fonction ne doit pas utiliser de vecteur de travail.

4.2 Tableaux à Deux Dimensions : les Matrices

EX. 80 Écrire :

1. une fonction `init_mat_m_n` qui initialise une matrice $m \times n$ à composantes entières (m et n sont des constantes données).
2. une fonction `imprime_mat_m_n` qui imprime une telle matrice.

EX. 81 Écrire une fonction `trace` à valeur réelle qui calcule la trace d'une matrice A de dimension $n \times n$ donnée en paramètre :

$$\texttt{trace}(A) = \sum_{i=1}^{n} A_{ii}$$

EX. 82 Écrire une fonction qui effectue le produit de la matrice $A(l \times m)$ par la matrice $B(m \times n)$.

EX. 83 Rappel : une matrice A_{ij} (i de 1 à n, j de 1 à n) est dite symétrique si elle est égale à sa transposée, c'est-à-dire si :

$\forall i, j : 1 \leq i, j \leq n : A_{ij} = A_{ji}$

et antisymétrique si :

$\forall i, j : 1 \leq i, j \leq n : A_{ij} = -A_{ji}$.

Écrire les fonctions `symetrie` et `antisymetrie` qui testent respectivement la symétrie et l'antisymétrie d'une matrice carrée.

✍: 40 min.

EX. 84 Écrire une fonction rotation qui effectue une rotation, en place, de +90° (dans le sens trigonométrique) dans une matrice carrée $n \times n$.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \text{ devient : } \begin{pmatrix} 3 & 6 & 9 \\ 2 & 5 & 8 \\ 1 & 4 & 7 \end{pmatrix}$$

Chapitre 5

Manipulations de caractères

EX. 85 Écrire une fonction `length` qui renvoie la longueur d'une chaîne.

EX. 86 Écrire une fonction qui remplace par le contenu de la variable c, de type caractère, le k-ième caractère de s.

EX. 87 Écrire une fonction qui renvoie une chaîne formée des n premiers caractères de s (ou s si n est trop grand).

Variantes :
- Une fonction qui renvoie une chaîne formée des n derniers caractères.
- Une fonction qui renvoie une chaîne formée des n caractères du string s à partir de la position p (comptée à partir de 0).

EX. 88 Écrire une fonction qui renvoie la valeur vrai *si et seulement si* tous les caractères de A se trouvent dans le même ordre dans B, mais pas nécessairement de façon contiguë.

Par exemple :

inclus('mai', 'machin') = vrai ;
inclus('mai', 'misérable') = faux.

EX. 89 Écrire une fonction `MAT_premiers` qui met en majuscule la première lettre de chaque mot d'une chaîne. On suppose que le string est composé de caractères alphabétiques et de blancs, ces derniers séparant les mots.

EX. 90 Écrire un fonction `Supprime_double` ayant comme paramètre un vecteur V de n composantes de type caractères, qui supprime, en place, les caractères identiques qui se suivent.

Exemple :

```
hassaaard
1␣␣␣␣␣␣␣n
```

devient

```
hasard
1␣␣␣␣␣␣n
```

 ✍: 40 min. 

EX. 91 Soit une chaîne V ($|V| \leq MAXV$) et deux chaînes C et R.
On demande d'écrire la fonction $FANDR$ avec 4 paramètres : V, C, R et $DROP$, qui cherche dans V la chaîne C et en remplace chaque occurrence par la chaîne R (en décalant si nécessaire les autres caractères de V). Si des caractères doivent être décalés après $MAXV$, ils sont perdus (il faut néanmoins poursuivre le traitement). L'entier $DROP$ contiendra le nombre de ces caractères perdus.
Variantes :
- Après un remplacement la recherche se poursuit par le caractère suivant le dernier caractère de remplacement ; seuls les caractères initialement dans V font l'objet de la recherche.
- Après un remplacement la recherche se poursuit par le premier caractère de remplacement.

EX. 92 Écrire une fonction qui donne la valeur numérique d'une chaîne de chiffre en base 10.
Variantes : signe - éventuel, point décimal, notation scientifique (e.g. 3.9E+5).

EX. 93 Écrire une fonction qui transforme un nombre entier en un string.

 ✍: 35 min. 

EX. 94 Dans un string, on cherche le maximum des longueurs des suites de caractères tous différents.
Exemple 1 :

$$A\,B\,C\,C\,\underbrace{D\,C\,A\,B\,E}\,E\,D\,F$$

ABC, DCA, ABE, EDF, ... sont des suites de caractères distincts de longueur 3.
$DCABE$ est une suite de caractères distincts de longueur 5, il n'y a pas de suite de longueur supérieure à 5 dont les caractères soient tous distincts. La réponse est donc 5.
Exemple 2 :

$$\overbrace{A\,B\,\underbrace{C\,D\,E}\,C\,A\,B\,F}\,A$$

$DECABF$ est une suite admissible maximale de longueur 6.

Chapitre 6

Complexité

EX. 95 Trouvez la complexité au pire cas du code suivant :

```
for (int i = 0; i < n; i=i+3)
  t[i]=0;
```

EX. 96 Trouvez la complexité au pire cas de la fonction suivante :

```
typedef int matrice[dim][dim];
void somme(matrice c, matrice a, matrice b, int n)
{
    for (int i = 0; i < n; ++i) {
       for (int j = 0; j < n; ++j) {
         c[i][j] = a[i][j] + b[i][j];
       }
    }
}
```

EX. 97 Trouvez la complexité au pire cas du code suivant :

```
for (int j=0;j<4;j++)
   for (int i = 0; i < n; i++)
      t[i]*=t[i];
```

EX. 98 Trouvez la complexité au pire cas de la fonction suivante :

```
typedef int matrice[dim][dim];
void produit(matrice c, matrice a, matrice b, int n)
{
    for (int i = 0; i < n; ++i) {
       for (int j = 0; j < n; ++j) {
          c[i][j] = 0;
          for (int k = 0; k < n; ++k)
              c[i][j] += a[i][k] * b[k][j];
       }
    }
}
```

EX. 99 Trouvez la complexité au pire cas de la fonction suivante :

```
void tri_insertion(int V[])
{
    int tmp, j;

    for (int i = 1; i < n; i++) {
      tmp = V[i];
      j = i - 1;
      while (j >= 0 and V[j] > tmp) {
         V[j + 1] = V[j];
         j--;
      }
      V[j + 1] = tmp;
    }
}
```

EX. 100 Trouvez la complexité au pire cas et minimale de la fonction suivante :

```
typedef int matrice[dim][dim];
bool symetrie(matrice a,int n)
{
    bool sym = true;

    for (int i = 0; i < n and sym; i++)
        for (int j = 0; j < i and sym; j++)
           if (a[i][j] != a[j][i])
              sym = false;
     return (sym);
}
```

EX. 101 Trouvez la complexité au pire cas du code suivant :

```
int i=0,j=0;

while(j<n)
{
  t[j*n+i]=i;
  if (i==n-1)
   {
    j=j+1;
    i=0;
   }
   else
    i++;
}
```

EX. 102 Trouvez la complexité au pire cas du code suivant :

```
i = 1;
while (i <= n)
{
  j = 1;
  while (j <= m)
  {
    cout << j;
    ++j;
    ++i;
```

```
  }
  ++i;
}
```

EX. 103 Trouvez la complexité au pire cas du code suivant :

```
for (int i=1; i <= n; ++i)
{
  j = 1;
  while (j <= m)
  {
    r = 2*(i+j)+r;
    ++j;
  }
  k = 1;
  while (k <= l)
  {
    r = r*k;
    ++k;
  }
  cout << r;
}
```

EX. 104 Trouvez la complexité au pire cas du code suivant :

```
for (int i=1; i<n; ++i)
{
  for (int j=1;j<= n; j = j*2)
  {
    cout <<j;
  }
}
```

EX. 105 En faisant l'hypothèse que `n` est positif, donner, en termes de grand O et simplifiée, la complexité au pire cas du code suivant :

```
int c = 0;

for (int i = n * n; i > 0; i = i / 2)
  for (int j = n; j > 0; j = j / 4)
    c++;

cout << c << endl;
```

EX. 106 Trouvez les complexités au pire cas et minimale du code suivant, en supposant que la fonction `boolrand` renvoie `true` ou `false` avec la même probabilité et qu'elle est de complexité constante.

```
for (int i=0; i<n; ++i)
  v[i] = true;
for(int i=0; i<n-1; ++i)
  w[i] = boolrand();
w[n-1] = false;
flag = true;
j = 1;
while(flag)
```

```
{
  if (v[j]==w[j])
    for(int i=j; i<n-1; ++i)
      w[i] = boolrand();
  else
    flag = false;
  ++j;
}
```

✍: 10 min.

EX. 107 On se donne la fonction suivante :

```
int incseq (int V[], int N) {
  int i = 0, k = 0, maxseq = 0;

  while (i < N - 1) {
    int seq = 1;

    for (int j = i; j < N - 1 && V[j] <= V[j + 1]; ++j) ++seq;

    if (seq > maxseq) {
      maxseq = seq;
      k = i;
    }

    i += seq;
  }
  return k;
}
```

Donner la complexité au pire cas de l'algorithme en fonction de `N`.

EX. 108 (QUESTION DE L'EXAMEN D'AOÛT 2006) On se donne la fonction C++ suivante :

```
void binaire (int r[], int n)
{
  for (int i = 1; i < n; ++i) {
      int tempr = r[i];
      int bi = -1;
      int bs = i;
      while (bs - bi > 1) {
        int j = (bs + bi) / 2;
        if (tempr < r[j]) bs = j; else bi = j;
      }
      for (int j = i; j > bs; --j) r[j] = r[j - 1];
      r[bs] = tempr;
    }
}
```

1. Que fait cette fonction ? Détailler les différentes étapes de l'algorithme.
2. Donner sa complexité au pire cas, sous forme d'un grand $\mathcal{O}$ et en fonction de la taille `n` du vecteur traité.

Chapitre 7

Les tris de tableaux

7.1 Tris simples

EX. 109 Écrire une fonction triant un vecteur par sélection.

EX. 110 Écrire une fonction triant un vecteur par insertion.

EX. 111 Écrire une fonction triant un vecteur par insertion avec recherche préalable du minimum.

7.2 Tri par énumération

EX. 112 Écrire une fonction triant un vecteur par énumération (les valeurs possibles pour les composantes : $0 \dots 10$).

Pour rappel, le tri par énumération fonctionne comme suit, en supposant qu'on désire trier le tableau `V` :

1. On commence par *compter* le nombre d'occurrences de chaque valeur du tableau `V`, dans un tableau `count`. À la fin de cette étape, `count`$[i]$ contiendra le nombre d'occurrences de la valeur i dans `V`.
2. On somme ensuite les valeurs contenues dans `count` de manière à ce que `count`$[i]$ contienne le nombre de valeurs $\leq i$ présentes dans `V`.
3. `count` permet alors de recopier les valeurs de `V` dans un nouveau vecteur `W`, et ce, de manière triée. Pour ce faire, on parcourt `V` élément par élément. Chaque valeur v présente dans `V`, doit être placée dans la `count`$[v]^{\text{e}}$ case de `W`, et `count`$[v]$ doit être décrémentée pour éviter les collisions.

Commencez par exécuter *à la main* le tri par énumération sur le vecteur `V` suivant :

$$V = \boxed{4}\boxed{8}\boxed{4}\boxed{2}\boxed{8}\boxed{1}\boxed{4}$$

7.3 Tri Shell

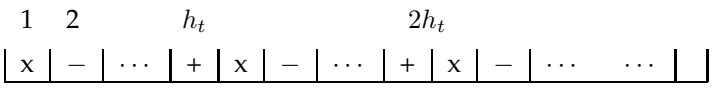
EX. 113 **Méthode Shell (***diminishing increment sort***)**

Les performances de l'algorithme de l'insertion linéaire sont essentiellement fonction du nombre de fois que l'on devra « tourner » dans le `while`. Si au départ le vecteur A est presque trié, le nombre d'itérations du `while` est petit et l'algorithme peut alors être très performant. Par contre, si le vecteur de départ est dans un ordre tout à fait quelconque, on peut montrer que le nombre de déplacements est proportionnel à n^2 pour l'ensemble des n éléments. Cela est dû au fait que l'on

ne déplace les éléments que d'une position à la fois dans le vecteur. Donald Shell s'est dit que l'on devrait obtenir de meilleures performances si au lieu de faire des « sauts » d'une seule position dans le vecteur, on faisait des sauts plus longs d'un seul coup, la longueur de ses sauts diminuant à chaque grande étape du tri. A la dernière de ces grandes étapes on sera quand même obligé de comparer des éléments successifs (« saut » de longueur 1), ce qui revient à la méthode précédente, mais comme à ce moment le vecteur sera déjà presque trié, cette dernière étape sera très rapide. Le fait que l'on compare des éléments de plus en plus proches justifie le nom de « méthode de l'incrément décroissant » que l'on donne parfois à cette méthode.

Description de la méthode :

Soit un vecteur A de dimension n. La première étape consiste à le considérer comme étant formé d'un nombre h_t de sous-vecteurs imbriqués les uns dans les autres qu'on trie indépendamment, c'est-à-dire qu'on est amené à trier le sous-vecteur $(A_1, A_{1+h_t}, A_{1+2h_t}, ...)$ puis le sous-vecteur $(A_2, A_{2+h_t}, A_{2+2h_t}, ...)$ et ainsi de suite jusqu'au sous-vecteur $(A_{h_t}, A_{2h_t}, A_{3h_t}, ...)$. On considère donc bien h_t sous-vecteurs dont les éléments sont distants de h_t positions :



La manière de trier ces sous-vecteurs peut être tout à fait quelconque, nous utiliserons ici l'insertion linéaire déjà étudiée.
A l'étape suivante on subdivise (en pensée) A en un nombre h_{t-1} de sous-vecteurs (avec $h_{t-1} < h_t$) de la même manière et on trie ces h_{t-1} sous-vecteurs à leur tour. Après quoi on recommence avec $h_{t-2} < h_{t-1}$ sous-vecteurs et ainsi de suite jusqu'à un nombre h_1 de sous-vecteurs qui vaudra obligatoirement 1, c'est-à-dire qu'à cette dernière étape on trie d'un seul coup tout le vecteur A par insertion linéaire (étape rapide, voir ci-dessus).
Le problème qui reste à résoudre est le choix des nombres $h_t, h_{t-1}, ..., h_1(= 1)$. En principe n'importe quelle suite décroissante se terminant par 1 peut convenir mais le choix de la suite peut influencer les performances de l'algorithme. Il semble, empiriquement, qu'une suite offrant de bonnes performances soit :

$$\begin{array}{lcll} h_1 & = & 1 & \\ h_{s+1} & = & 3\,h_s + 1 & \forall\, s \geq 1 \end{array}$$

On prendra comme premier élément h_t de la suite (le plus grand) : h_t tel que $h_{t+2} \geq n$.

En résumé, pour appliquer cette méthode il faut d'abord calculer la séquence des incréments $h_s(s = 1, 2, ..., t)$ (ou du moins le h_t qui est le premier incrément utilisé). Ensuite il faut faire une boucle sur les différents h_s à l'intérieur de laquelle il faut faire une boucle sur les différents sous-vecteurs à trier selon l'insertion linéaire.

Remarques :

– La complexité de cette méthode ne se justifie que pour n très grand.
– Ce tri n'est pas stable.

7.4 Tri Cocktail Shaker

EX. 114 Le tri bulle fait avancer les plus grands éléments vers la fin du vecteur au fur et à mesure de l'exécution. On obtient donc ainsi une tendance au regroupement des plus petits éléments en début de vecteur. D'autre part, beaucoup de comparaisons portant sur ces éléments sont redondantes d'une itération à l'autre. Il serait donc intéressant de placer ces petits éléments en position définitive et de ne plus les considérer lors d'étapes ultérieures. Pour cela il faut connaître une borne inférieure en deçà de laquelle le vecteur est définitivement trié, tout comme dans le tri bulle on avait une borne supérieure au-delà de laquelle le vecteur était définitivement trié. Pour réaliser cela, on effectue une étape du tri bulle une fois dans un sens (selon les indices croissants des composantes), ce qui fournit une borne supérieure, puis une fois dans l'autre sens (selon les indices décroissants des composantes), ce qui fournit une borne inférieure.

On continue de la sorte en alternant le sens de parcours du vecteur jusqu'à ce qu'il soit complètement trié, c'est-à-dire jusqu'à ce que les bornes se rejoignent. C'est à cette alternance de sens de parcours du vecteur que cette méthode de tri doit son nom de « cocktail shaker ». Nous vous demandons de programmer cette méthode.

Remarque :

Les tris « bulle » et « *cocktail shaker* » sont tous deux stables et ont tous deux un temps d'exécution proportionnel à n^2.

7.5 Tri par ventilation

EX. 115 **Tri par ventilation**

Soit une table A composée de n éléments du type

clé	information satellite

On veut trier A, en ordre croissant, sur clé.

La méthode du tri par ventilation s'applique dans le cas où chaque clé s'exprime comme une suite de composantes $a_1, a_2, \cdots, a_k$. (Les a_i peuvent être des sous-clés). Ainsi, par exemple, un nombre s'écrit comme une suite de chiffres compris entre 0 et 9, un mot est formé de lettres, etc.

La méthode consiste à effectuer des tris successifs sur chaque composante de la clé en allant de gauche à droite. Par exemple, en supposant que les clés sont des nombres à quatre chiffres, on effectue d'abord un tri sur le chiffre des milliers, puis pour chaque intervalle ayant le même chiffre pour les milliers on trie sur le chiffre des centaines, ensuite sur celui des dizaines, et enfin sur celui des unités.

Chapitre 8

Les invariants

Ex. 115bis Exprimez en logique du premier ordre les afirmations suivantes (vous supposerez que tous le tableaux sont de taille `N`) :

1. La variable `v` est positive ou nulle.
2. La variable `v` contient une valeur comprise entre 5 et 8 (bornes incluses).
3. Si la variable `v` est positive, alors la variable `w` est négative ou nulle.
4. Le tableau `V` ne contient que des valeurs positives ou nulles.
5. Au moins une case du tableau `V` contient une valeur supérieure ou égale à 5.
6. La variable `v` contient la somme des éléments du tableau `W`.
7. La variable `v` contient la somme des `i` premiers éléments du tableau `W`.
8. Les `i` premiers éléments du tableau `V` sont inférieurs à 6.
9. Tous les éléments du tableau `V` sont inférieurs ou égaux à `max`.
10. `max` est la valeur maximale contenue dans le tableau.
11. Le tableau `V` contient toutes des valeurs différentes.
12. Le tableau `V` est trié (de façon croissante).
13. Les `i` premières cases du tableau `V` sont triées (de façon croissante).
14. Les cases comprises entre les indices k_1 et k_2 du tableau `V` sont triées.
15. Si la variable `i` est positive, les `i` premières cases du tableau `V` sont triées de façon croissantes ; sinon, le tableau `V` est initialisé à 0.
16. La partie du tableau `V` comprise entre les indices k_1 et k_2 est triée, et il n'existe pas de plus grande partie (en terme de nombre de cases) du tableau qui soit triée.

EX. 116 Donnez formellement l'invariant de la boucle suivante (on suppose que `m` et `n` sont positifs) :

```
i = 0;
j = 0;

while (i != n)
{
  j += m;
  ++i;
}
```

EX. 117 Soit le morceau de code suivant (on suppose que `n` est > 0) :

```
int v[n];
int w[n];
int i=0, s=0;

while(i<n)
{
    s += v[i]*w[i];
    ++i;
}
```

– Expliquez en français ce que fait cette boucle.
– Exprimez à l'aide d'une formule logique la précondition et la postcondition de la boucle.
– Donnez-en l'invariant.
– Servez-vous de ces informations pour démontrer que la boucle fait bien ce que vous aviez prévu (en supposant qu'elle se termine).

EX. 118 Même question pour la boucle suivante (On suppose cette fois que `max` > 0 et `v` $[0] = 0$) :

```
sum = v[0];
i = 0;

while (sum < max and i < n)
{
  sum += v[i];
  ++i;

}
```

EX. 119 Soit le morceau de code suivant : Donnez formellement l'invariant de la boucle suivante et montrez que cette boucle se termine.

```
// n > 0
i = 1;
a = 0;
b = 1;

while (i < n)
{
  ++i;
  b += a;
  a = b-a;
}
```

– Que calcule ce morceau de code ?
– Quels sont la post-condition et l'invariant de la boucle ?
– Supposons que cette boucle se termine. Démontrez qu'elle calcule bien ce que vous aviez prévu.
– Enfin, prouvez que cette boucle se termine.

EX. 120 À quoi sert la boucle suivante ?

```
q = 0;
r = j1;
while (r >= j2)
{
  ++q;
  r -= j2;
}
```

Démontrez que cette boucle est correcte (qu'elle fait bien ce que vous venez de prédire). Pour ce faire, calculez d'abord l'invariant et la postcondition du `while`.

EX. 121 Voici la fonction $plus$:

```
int plus (unsigned int x, unsigned int y)
{
  while(y>0)
  {
    ++x;
    --y;
  }
  return x;
}
```

Démontrez que cette boucle calcule bien la somme de `x` et `y` dans `x`. Pour ce faire, aidez-vous de l'invariant du `while`, et de la fonction de terminaison.

EX. 122 La boucle suivante est censée imprimer la valeur maximale de `v`. Démontrez que c'est effectivement correct en utilisant l'invariant de la boucle.

```
i = 0;

while (i < n-1)
{
  if (v[i] > v[i+1])
  {
    save = v[i];
    v[i] = v[i+1];
    v[i+1] = save;
  }
  ++i;
}

cout << v[n-1];
```

 ✍: 40 min. 

EX. 123 (**Question de janvier 2003**) On se donne la fonction suivante, qui fait l'hypothèse que le vecteur V passé en paramètre est trié.

```
int RD (int V[], int n, int x) {
  int bi = 0, bs = n - 1, m = (n - 1) / 2;

  while (bs >= bi && V[m] != x) {
    if (V[m] < x) bi = m + 1;
    else bs = m - 1;
    m = (bi + bs) / 2;
  }

  if (x != V[m]) m = -1;

  return m;
}
```

– Quelle est l'utilité de cette fonction ?
– Donner l'invariant de la boucle.
– Démontrer la terminaison de la boucle.

EX. 124 (**Question d'août 2000**) Soit l'algorithme suivant :

```
const N = 200;
const Max = 1000;
int max_val, max_nb;
int T[N];
/* le vecteur est initialisé avec des valeurs
 *  aléatoires entre 0 et MAX
 */
for (int i=0; i<N; i++) T[i] = my_random(0, Max) ;
max_val = -1 ;
max_nb = 0 ;
for (int i=0; i<N; i++) {
    if (T[i] > max_val) {
        max_val = T[i];
        max_nb = 1;
    }
    else if (T[i] == max_val) max_nb++;
}
```

- Exprimez par une formule logique du premier ordre le résultat de l'algorithme (c'est-à-dire l'effet qu'il a sur `max_val` et `max_nb`).
- Donnez l'invariant de cet algorithme. Justifiez votre réponse en montrant que l'invariant est préservé au fil des itérations, qu'il est vérifié en début d'itération si la pré-condition est vérifiée ($Pre \Rightarrow I$) et qu'il implique la post-condition si la condition de sortie de boucle est vérifiée ($\texttt{i} = \texttt{N} \wedge I \Rightarrow Post$)

EX. 125 Voici la fonction `incseq` :

```
int incseq (int V[], int N) {
  int i = 0, k = 0, maxseq = 0 ;

  while (i < N - 1) {
    int seq = 1 ;

    for (int j = i; j < N - 1 && V[j] <= V[j+1]; ++j) ++seq ;

    if (seq > maxseq) {
      maxseq = seq ;
      k = i ;
    }

    i += seq ;
  }

  return k ;
}
```

- Décrivez l'effet de cette fonction. Exprimez sa valeur de retour en fonction des valeurs de ses paramètres.
- Donnez formellement la postcondition de cette fonction.
- Donnez l'invariant de la boucle principale (**Question de l'examen de juin 2003**).
- Concluez-en que la fonction a bien l'effet que vous aviez prévu (en supposant qu'elle se termine)
- Prouvez-en la terminaison.

Chapitre 9

Les pointeurs et les classes

9.1 Les pointeurs

EX. 126 Que contiennent `i` et `j` après les instructions suivantes :

```
int i = 10;
int *ip = &i;
int j = *ip;
*ip += 1;
ip += 1 ;
if (i == j)
  i = 5 ;
else if (j+1 == i)
  i = 6 ;
else if (j+2 == i)
  i = 7 ;
```

EX. 127 Décrivez l'effet des instructions suivantes. Indiquez celles qui provoquent une erreur de compilation et/ou une erreur à l'exécution.

1. ```
   int i = 10;
   int *ip= &i;
   int *ip = i;
   double *ipp = &i;
   ```
2. ```
   int j = 5;
   int *ip= &j;
   int *ip2 = ip;
   int *ip3 = &ip;
   int *ip4 = *ip;
   int **ip5 = &ip;
   int k = ip;
   int l = *ip;
   *ip = 2;
   ip + = 1;
   *ip = new int;
   ip = new int;
   ```
3. ```
   int j = 5;
   int *ip= &j;
   int *ip2 = new int;
   delete j;
   ```
   ```

```
delete ip2;
delete *ip2;
delete ip;
```

4.
```
int V[5]={1,2,3,4,5} ;
int i ;
int * ip ;
int ** ipp ;
i = V[1] ;
ip = V ;
i = *ip ;
++ip ;
i = *ip ;
i = ip[1] ;
V[2] = 4 ;
ipp = &ip ;
i = **ipp ;
```

5.
```
int * V ;
int * p ;
int i ;
V = new int[5] ;
V[0] = 1 ;
V[1] = 2 ;
p = &(V[0]) ;
delete[] V ;
i = *p ;
```

9.2 Les classes

EX. 128 Écrire une classe `Etudiant` qui possède comme champs : un entier `matricule` et un tableau d'entiers de taille 3 nommé `cotes` (chaque case de `cotes` correspond à un cours)
- Écrire une fonction qui reçoit un objet de type `Etudiant` et retourne la moyenne de ses cotes.
- Écrire un constructeur qui à partir d'un matricule et de trois cotes, crée un `Etudiant`.
- Écrire un programme dans lequel on crée un tableau de 5 éléments de type `Etudiant` (représentant un groupe de 5 étudiants) et qui calcule les moyennes des notes obtenues par les étudiants du groupe pour chacun des cours.

EX. 129 Écrire une classe `Point` qui permet de représenter un point dans un système à deux coordonnées x et y. Écrire une fonction qui reçoit deux objets de type `Point` et retourne la distance euclidienne entre ces deux points.

EX. 130 Quelle est l'effet de ces instructions ?

```
class MyClass {
 public:
  int i ;
  double d ;
  int V[3] ;
  int * W ;
} ;

MyClass x ;
x.i = 9 ;
x.W = new int[4] ;
x.W[0] = x.i ;
```

```
MyClass * p = new MyClass ;
p->i = x.i ;
p->i = x.W[0] ;
double * pp ;
pp = &(x.d) ;
*pp = 9 ;
p->V[0] = x.V[2] = 4 ;
delete p ;
x.d = *pp ;
```

EX. 131 Écrire une classe `Trait` qui représente un trait formé de maximum 10 segments de droite dans un espace à deux dimensions. Un trait sera donc représenté par au plus 11 objets de type `Point`. Comme on veut éviter de créer des `Point` inutiles, votre classe `Trait` contiendra un tableau de 11 pointeurs vers des objets de type `Point`, qui seront créés au moment voulu.

Vous écrirez :

1. Un constructeur qui initialise la classe `Trait` à l'aide de deux points. Les cases inutilisées du vecteur de points seront mises à `NULL`;
2. Une fonction qui reçoit un objet de type `Trait` et un pointeur vers un objet de type `Point` et qui ajoute le `Point` dans la première case libre de l'objet de type `Trait` (de manière à alonger le trait).
3. Une fonction recevant un *pointeur* vers un objet de type `Trait` et calculant sa longueur.

EX. 132 On peut encoder un polynôme dans un vecteur dont les composantes ont 2 champs : un champ donnant le degré du monôme et un champ donnant le coefficient correspondant. On suppose les monômes triés par ordre décroissant sur leur degré.

Ayant :

```
const int n=20;

class monome
{
public:
  int degre;
  double coeff;
};
```

Le polynôme $27x^{11} + 5x^4 - 3$

est codé par :

p

11	4	0	-1	...	...	degre
27	5	-3	?	...	...	coefficient

On utilise la valeur -1 dans le champ `degre` comme sentinelle.

Écrivez le prototype d'une classe `polynome` qui stockera des polynômes sous la forme d'un tableau de `monome`, de taille `n`. Vous prévoirez :
– un constructeur qui initialisera le polynôme à 0.
– un constructeur qui recevra un tableau de `monome` en paramètres et qui s'en servira pour initialiser le polynôme. On supposera que les monômes sont classés dans le tableau par ordre décroissant.
– une fonction d'initialisation `init`, qui lira sur l'*input* une série de couples d'entiers $\langle d, v\rangle$ qui représentent chacun un monôme $v{\cdot}x^d$ du polynôme (on supposera que les monômes sont entrés en ordre décroissant). La liste de couples d'entiers sera terminée par une valeur sentinelle -1. Le polynôme sera initialisé avec ces valeurs.

– une fonction `affiche` qui affiche le polynôme.

EX. 133 On vous demande d'écrire les fonctions suivantes, qui manipulent des polynômes.

1. Une fonction `int eval(double x)` qui évalue le polynôme en `x`.
   Amélioration : utiliser la méthode de Horner. Cette méthode repose sur la constatation suivante :
   Ayant $p(x) = \sum_{i=0}^{n} c_i \cdot x_i$ on a
   $$p(x) = (\ldots((c_n \cdot x + c_{n-1}) \cdot x + c_{n-2})\ldots) \cdot x + c_0$$
2. Écrire une fonction `polynome somme(polynome p, polynome q)` qui retourne la somme des polynômes `p` et `q`.
   Exemple :
   $$\text{Si } p = 3x^4 + 7, q = x^7 - x^4 + 2x \text{ alors } r = x^7 + 2x^4 + 2x + 7.$$
3. Écrire une fonction `polynome produit(polynome p, polynome q)` qui retourne le polynôme produit de `p` et `q`.
   Exemple :
   Si $p = 3x^4 + 7, q = x^7 - x^4 + 2x$ alors $r = 3x^{11} - 3x^8 + 7x^7 + 6x^5 - 7x^4 = 14x$.
4. Écrire une fonction `polynome derivee(polynome p, int n)` qui retourne la dérivée `n`^{ième} du polynôme.
   Exemple :
   Si $p = 3x^4 + x^2 + 7$, et
   $n = 1$ alors $r = 12x^3 + 2x$
   $n = 2$ alors $r = 36x^2 + 2$.
5. Écrire `void division(polynome p, polynome d, polynome &q, polynome &r)`, une fonction qui calcule dans `q` le polynôme quotient et dans `r` le polynôme reste de la division de `p` par `d`.

Chapitre 10

Les listes

10.1 Les listes linéaires simples : fonctions élémentaires

Ayant les déclarations de la classe `elem` et du type `liste` donné à la figure 10.1,

EX. 134 Écrire une fonction `int longueur(liste L)` qui retourne la longueur de la liste `L` (c'est-à-dire le nombre d'éléments chaînés entre-eux pour former cette liste).

EX. 135 Écrire une fonction `bool existe(liste L, int k)` qui retourne un booléen indiquant si la valeur `k` apparaît dans la liste `L`.

EX. 136 Écrire une fonction `void insered (liste & L, int k)` qui insère un élément portant l'information `k` en début de la liste `L`.

EX. 137 Écrire une fonction `void inseref (liste & L, int k)`. Même chose que `insered`, mais en fin de la liste.

EX. 138 Écrire une fonction `void inserem (liste & L, int k, int pos)` qui insère un élément portant l'information `k` dans la liste `L`, en position `pos`, c'est-à-dire juste après le $(\texttt{pos}-1)^{\text{e}}$ élément. Si la liste contient moins de $\texttt{pos} - 1$ éléments, un message d'erreur sera imprimé et la liste ne sera pas modifiée. On suppose que `pos` est un entier > 0.

EX. 139 Écrire une fonction `void inserap (liste L, int k, int l)` qui insère un élément portant l'information `k` derrière l'information `l` dans la liste `L`. Si l'information `l` n'existe pas dans la liste, un message d'erreur sera imprimé et l'insertion ne sera pas effectuée.

EX. 140 Écrire une fonction `void inserav (liste & L, int k, int l)`. Même chose que `inserap`, mais insérer `k` *devant* l'information `l`.

EX. 141 Écrire une fonction `void retired (liste & L)` qui retire le premier élément de la liste `L`. Si la liste est vide, un message d'erreur sera imprimé.

EX. 142 Écrire une fonction `void retiref (liste & L)`. Même chose que `retired`, mais avec le dernier élément.

EX. 143 Écrire une fonction `void retirem (liste & L, int pos)`. Même chose que `retired` mais c'est le $\texttt{pos}^{\text{e}}$ élément qui doit être retiré. Si la liste contient moins de `pos` éléments, imprimer un message d'erreur. On suppose que `pos` est un entier strictement positif.

```
class elem;

typedef elem *liste;

class elem
{
   public:
     int info;              // Information
     liste next;            // Pointeur vers le suivant
     elem(){}
     elem(int i,liste p);
};

elem::elem(int i,liste p)
{
  info=i;
  next=p;
}
```

FIG. 10.1 – déclarations de la classe elem et du type liste

✍: 25 min.

EX. 144 Écrire une fonction `void inverse (liste & L)`. Cette fonction doit « inverser » la liste `L`, c'est-à-dire la modifier de telle sorte que ce qui était le premier élément devienne le dernier, le deuxième devienne l'avant dernier, ..., jusqu'au dernier qui devient le premier.

EX. 145 Écrire une fonction `void attache (liste & L1, liste & L2)`. Cette méthode doit attacher la liste `L2` derrière la liste `L1`. La liste `L2` devra être vide après l'appel.

10.2 Listes circulaires avec élément bidon au début

Ayant les déclarations de la classe `elem` et du type `liste` donné à la figure 10.1,

EX. 146 Écrire un fonction retournant une liste vide.

EX. 147 Idem que l'exercice 135 pour une liste circulaire avec élément bidon au début.

EX. 148 Idem que l'exercice 137 pour une liste circulaire avec élément bidon au début.

EX. 149 Idem que l'exercice 140 pour une liste circulaire avec élément bidon au début.

10.3 Listes linéaires simples triées

EX. 150 Ayant les déclarations de la classe `elem` et du type `liste` donné à la figure 10.1, réaliser une fonction `void inserTrie(liste & L, elem *e)` qui insère l'élément pointé par `e` dans la liste dont la tête est donnée par `L`. Cette liste est triée en ordre décroissant et doit le rester après l'insertion.

10.4 Listes implémentées dans des vecteurs

EX. 151 Dans toutes les méthodes de tri étudiées précédemment, on déplaçait physiquement en mémoire les éléments qui ne se trouvaient pas à leur place. Si ces éléments comprennent des informations satellites, celles-ci doivent bien entendu « suivre » leur clé lors des déplacements. Si les informations satellites sont longues, leur déplacement peut prendre beaucoup de temps ; dans ce cas il peut être intéressant d'ajouter à chacun des éléments un pointeur de façon à chaîner les éléments sous forme de liste. Le déplacement physique est alors remplacé par une manipulation de pointeurs. En fin de tri l'ordre des éléments ne sera plus donné par la position physique de ces éléments mais en suivant les pointeurs de chaînage. La plupart des méthodes de tri que nous avons vues peuvent être exploitées en se servant d'une liste, il y a cependant des cas où cela ne serait pas possible ou du moins peu performant. Par exemple, pour la méthode Shell, il serait très difficile de scinder le vecteur donné en sous-vecteurs imbriqués si les éléments n'étaient pas consécutifs en mémoire.

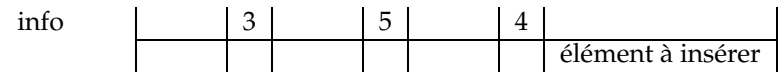
Tri par insertion avec liste

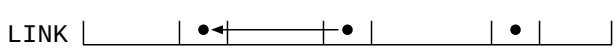
Au vecteur $info$ contenant les éléments à trier, on ajoute un vecteur $next$ de même longueur et destiné à contenir les pointeurs de chaînage. Appelons $tete$ la tête de liste, c'est-à-dire le pointeur vers le premier élément de la liste. Supposons que l'on doive effectuer un tri de façon à obtenir les éléments en ordre décroissant (il est évident que l'on procéderait de manière analogue pour un ordre croissant).

Initialisation : tete = 0
$next[0] = -1$;

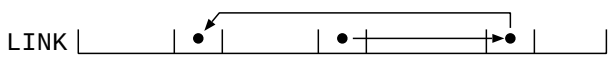
ce qui revient à dire qu'au départ la liste contient le seul élément $info[0]$; à tout moment la liste contiendra les éléments déjà triés et rien qu'eux, l'ordre étant bien entendu donné par les pointeurs. On ajoute un à un les autres éléments dans cette liste ; pour cela, on parcourt les composantes de $info$ de la 2e à la ne (n = nombre d'éléments à trier) et on insère l'élément considéré en position correcte dans la liste de la façon suivante : en partant de $tete$ on parcourt la liste jusqu'à rencontrer un élément inférieur à celui à insérer. A ce moment on place l'élément à insérer devant l'élément trouvé (rappelons-nous que l'on veut un ordre décroissant). Quand on dit « placer devant » il s'agit évidemment d'un placement logique c'est-à-dire par manipulation de pointeurs.

Exemple :





devient



Si en suivant la liste on arrive au bout de celle-ci cela signifie que l'élément à insérer est le plus petit rencontré jusqu'à présent, il vient donc en queue de liste.


✍: 30 min.


Exercice :

Écrire la fonction qui réalise le tri décrit ci-dessus.

10.5 Listes doublement liées

EX. 152 Chaque élément d'une telle liste comprend deux champs de type pointeur : l'un contenant l'adresse de l'élément qui le suit, l'autre l'adresse de l'élément qui le précède. Il est donc possible de parcourir une telle liste dans les deux sens.

On définit :

```
#include <iostream>
#include <climits>

using namespace std ;

/* Minimum and maximum values : a 'signed int' can hold INT_MIN and INT_MAX */

class elem;

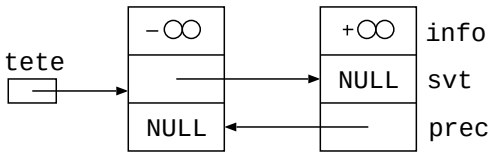
typedef elem *liste;

class elem
{
public:
  int info;               // Information
  liste next, previous;  // Pointeur vers le suivant et le précédent
  elem(int i, liste p, liste q);
};


elem::elem(int i, liste p, liste q)  // Initialisation
{
   info = i;
   next = p;
   previous = q;
}

void initliste(liste &L)               // Initialisation
{
   L = new elem (INT_MIN, NULL, NULL);
   L->next = new elem (INT_MAX, NULL, L);
}
```

Pour simplifier les tests, on placera en début et en fin de liste des sentinelles. La liste est donc initialisée à :



On demande d'écrire une fonction `void insere(liste L, int k)` qui fait une insertion de l'entier `k` dans la liste `L` en en conservant l'orde croissant.

10.6 Intersection et différence symétrique

 ✍: 25 min. 

EX. 153 On considère 2 listes triées par ordre croissant sans répétition sur le champ *info* des éléments. On désire qu'après traitement la 1re liste contienne les éléments qui étaient communs aux 2 listes et que la 2e contienne les éléments qui n'apparaissaient que dans l'une de ces listes.
Autrement dit, si les listes de têtes a et b représentent respectivement les ensembles A et B, après traitement, a représentera l'intersection :

$$A \cap B = \{x \in A \text{ et } x \in B\}$$

et b représentera la différence symétrique :

$$A \triangle B = \{x \in A \backslash B \text{ ou } x \in B \backslash A\}$$

Ces listes seront évidemment encore triées.
Exemples :
A = {2,4,5,7,8,12,17} B = {3,4,9,12,15,17,20}.

deviendra, après traitement

A = $\{4, 12, 17\}$ B = $\{2, 3, 5, 7, 8, 9, 15, 20\}$

Nous donnons deux solutions. La première consiste à traiter les éléments de chaque liste un par un. La seconde consiste à traiter les éléments par blocs. Dans l'exemple ci-dessus, les éléments 5,7,8 peuvent être « déplacés » en une fois. Cette dernière solution permet quelques économies d'assignations (comparer les traitements de chacun des 3 cas).
Afin d'éviter les tests d'insertion en début de liste, on place en tête de chaque liste un élément bidon que l'on supprime à la fin du traitement.

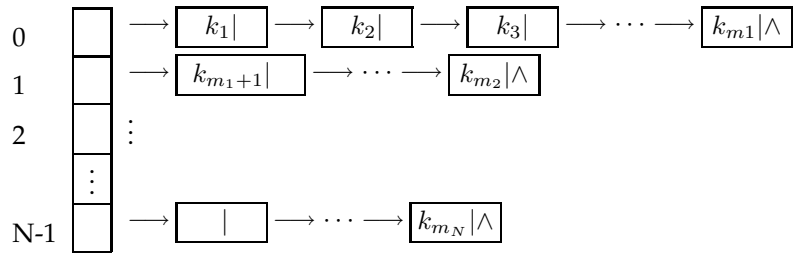
10.7 Liste triée scindée en plusieurs sous-listes

Soit une liste triée scindée en n sous-listes, la tête de chaque sous-liste se trouvant dans une composante du vecteur V. On dispose donc d'un vecteur V de pointeurs dont chacune des composantes (numérotées de 0 à $N-1$) représente une tête de liste triée (listes simples, sans en-tête) telles que

$$k_i \leq k_{i+1} \qquad \forall i | 1 \leq i < m_N \qquad \text{(voir dessins)}$$

En particulier, on a $k_{m_i} \leq k_{m_i+1} \forall\, i | 0 \leq i < N-1$ c'est-à-dire que l'élément de fin d'une liste est

$\leq$ au premier élément de la liste suivante.



✍: 20 min.

EX. 154 Écrire une fonction `liste recherche (liste L, int k)` qui renvoie un pointeur vers la 1^{e} occurrence de l'élément d'information k.
Hypothèse : on suppose qu'aucune sous-liste n'est vide.

EX. 155 Écrire une fonction `suppression(liste & L, int i, int j)` avec $(1 \leq \texttt{i} \leq \texttt{j})$ qui supprimera dans cette structure tous les éléments compris entre le $\texttt{i}^{e}$ et le $\texttt{j}^{e}$ (inclus).

L'algorithme sera décomposé en deux phases :

1. Passer les $\texttt{i} - 1$ premiers éléments.
2. Supprimer les éléments `i` à `j`.

Pour la phase 1, on s'arrêtera dès que les $\texttt{i} - 1$ premiers éléments auront été passés. Dans ces conditions, deux cas peuvent se présenter : on s'arrête au milieu d'une liste et on devra détruire un morceau en « milieu de liste » ou on s'arrête à la fin d'une liste et on détruira à partir du début de la liste suivante.

Si on ne doit pas exécuter la phase 1 ($\texttt{i} = 1$) on s'arrangera pour initialiser les variables de manière à simuler l'arrêt en bout d'une pseudo liste no. -1.

La phase 2 est découpée en deux parties :
(a) La destruction à partir du milieu d'une liste.
(b) La destruction à partir du début d'une liste.
L'étape (b) peut être répétée si nécessaire.

10.8 Listes cycliques

EX. 156 Considérons les définitions suivantes :

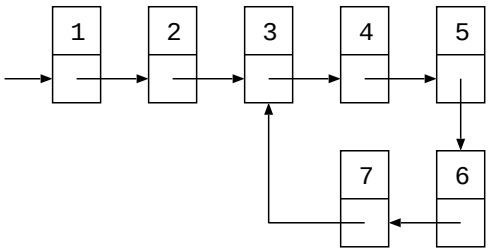
```
class element;

typedef element *liste;

class element
{
public:
  int info;
  bool marque;
  liste svt;
};
```

où le champ `marque` est supposé *faux* par défaut.

Cette structure est utilisée pour représenter des listes « cycliques » formées d'une partie transitoire et d'une partie périodique selon le schéma suivant :



On demande d'écrire une fonction qui, recevant l'adresse du 1er élément, imprime les informations des éléments de la liste en séparant la partie transitoire de la partie périodique par un astérisque. Dans l'exemple, on obtiendra :

1 2 * 3 4 5 6 7

10.9 Parcours de 3 listes triées

EX. 157 Soit :

```
class element;

typedef element *liste

class element
{
public:
  int info;
  liste svt;
};
```

On dispose de trois listes quelconques (elles peuvent être vides) et triées en ordre croissant. Leurs têtes correspondantes sont `Liste1`, `Liste2`, `Liste3` de type `liste`. Dans une même liste toutes les infos sont différentes.

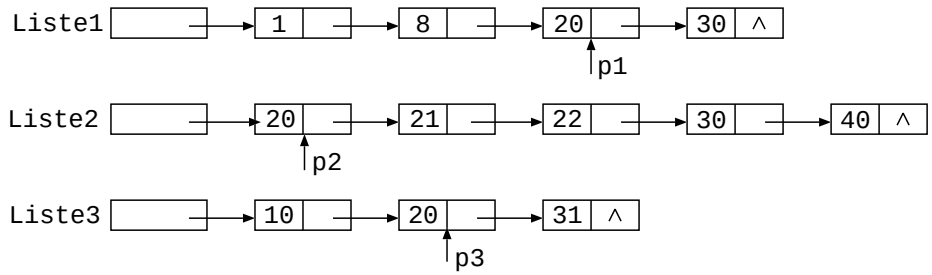
On voudrait savoir s'il existe un triplet (`p1`,`p2`,`p3`) tel que `p1` pointe vers un élément de `Liste1`, `p2` vers un élément de `Liste2` et `p3` vers un élément de `Liste3`, et tel que :

info de `p1` = info de `p2` = info de `p3`

Remarquez qu'il peut en exister plusieurs.

On demande d'écrire une fonction `void triplet (liste Liste1,liste Liste2, liste Liste3, liste &p1, liste &p2, liste &p3)` qui recherche le premier tel triplet et qui le retourne via `p1`, `p2` et `p3`. S'il n'y en a pas, `p1` prendra la valeur $\wedge$.

Exemple :



Remarque : vous ne pouvez parcourir les listes qu'une seule fois.

10.10 Listes à différents niveaux

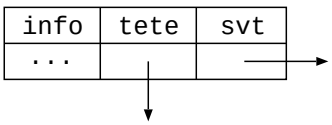
EX. 158 On donne les déclarations :

```
class element ;

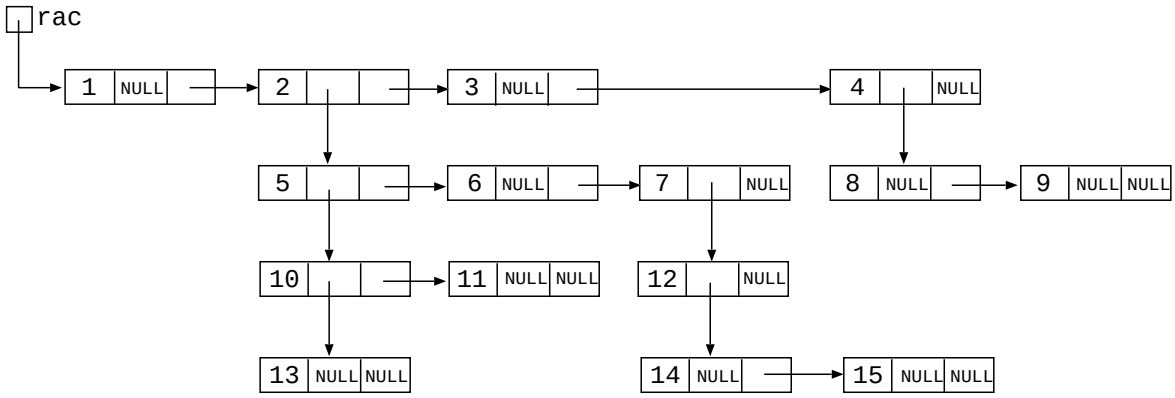
typedef element * liste ;

class element
{
  public:
    int info ;
    liste svt ;
    liste tete ;
} ;
```

On considère des structures du type de celle donnée en exemple ci-dessous où chaque élément peut, à la fois, faire partie d'une liste et être la tête d'une autre.



Exemple :



Ceci est une structure à 4 niveaux

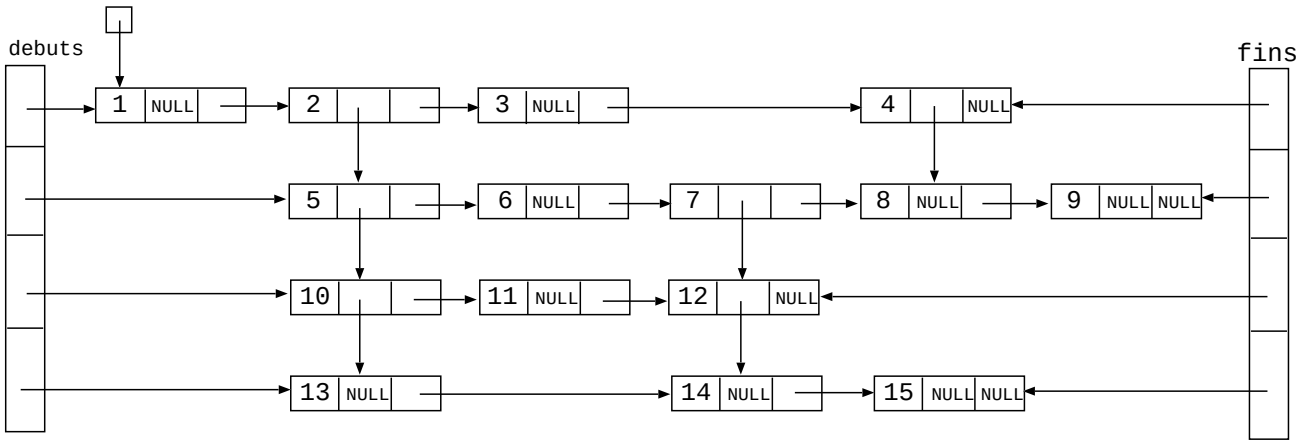
- La sous-structure contenant les infos 5, 6 et 7 est une liste de niveau 2 de trois éléments. Sa tête est contenue dans l'élément qui supporte l'info 2.
- La sous-structure contenant les infos 8 et 9 est aussi une liste de niveau 2. Sa tête est contenue dans l'élément qui supporte l'info 4.

– La sous-structure contenant l'info 13 est une liste de niveau 4 à un seul élément. Sa tête est contenue dans l'élément qui supporte l'info 10.

✍: 40 min.

Écrivez la procédure `lineariser(liste rac, liste fins[n] liste debuts[n])` qui doit :
– Relier ensemble les listes de même niveau.
– Installer dans `debuts[i]` et `fins[i]` respectivement, un pointeur vers le premier et un pointeur vers le dernier élément de la liste de niveau `i` ainsi obtenue pour tout i de 1 à n.

Par exemple, si `rac` donne accès à la structure représentée ci-dessus, après exécution de la procédure, on aura :



Attention : on ne peut parcourir la structure qu'une seule fois.
