# Introduction à la programmation

**Contrôle des connaissances**

- Interro sur 2 points (4ième semaine)
- DS sur 3 points
- Projet sur 3 points
- Examen sur 12 points

Pierre-André Wacrenier - LaBRI
wacrenier@labri.u-bordeaux.fr
http://dept-info.labri.u-bordeaux.fr/~wacren/mias302

## Les informatiques

- **Gestion** (comptabilité, stocks, production, documentation)
- **Transactionnelle** (banque, système de réservation)
- **Scientifique** : modélisation, simulation et calcul intensif (synthèse d'image, météorologie) ; analyse de données et aide à la décision (analyse d'image, analyse statistique)
- **Temps Réel** (calculateur embarqué, gestion de trafic urbain), Communication (télévision numérique, gestion de réseaux de télécommunication)
- **Bureautique** (traitement de texte, tableur, gestion de micro-entreprise)
- **Loisirs** et **Culture** (jeux, multimédia, Internet)
- …

## Les limites de l'informatique

**Limites théoriques** : tout n'est pas calculable.

Théorème de Gödel (1931) : Il existe une infinité de théorèmes n'ayant pas de démonstration dans tout système axiomatique suffisamment puissant pour exprimer + et *. G : "le théorème G n'est pas démontrable".

**Limites pratiques** : complexité du calcul.

Quelle est la fonction qui associe à la taille d'une donnée le nombre d'étapes de calcul nécessaire à son traitement ?

ex. calculer $10^x$ :

| taille de x | taille de $10^x$ |
|---|---|
| 1 (x < 10) | 10 chiffres |
| 2 (x < 100) | 100 |
| k | $10^k$ |

Ajouter 1 à la taille de la donnée => multiplier par 10 la taille du résultat.

**Limites budgétaires** (un bac+5 débutant gagne entre 170 et 230 kF et environ 270 kF après 10 ans)

## Introduction

**But**

Apprendre à traduire des algorithmes mathématiques simples dans un langage de programmation classique : le langage C.

Algorithme (d'Al-Kärezmi, Math. ~1250) :

*Suite d'opérations ou de raisonnements qui fournit la solution de certains problèmes.*

*Au moyen âge, procédé de calculs effectués à l'aide de chiffres arabes.*

*Ensemble de règles ou de procédés définis en vue d'obtenir un résultat déterminé au moyen d'un nombre fini d'opérations.*

## Exemples d'algorithme

Addition de deux entiers naturels :

| | | | |
|---|---|---|---|
| si retenue = 0 | | | ensemble de règles |
| 0+0=0 | 1+0=1 | 2+0=2 | ... |
| 0+1=1 | 1+1=2 | ... | |
| 0+2=2 | 1+2=3 | ... | |
| ... | ... | ... | |
| | | | 9+9=8 et retenue = 1 |
| si retenue = 1 ... | | | |
| 0+0=1 | 1+0=2 | 2+0=3 | ... |
| 0+1=2 | 1+1=3 | ... | |
| 0+2=3 | 1+2=4 | ... | |
| ... | ... | ... | |
| | | | 9+9=9 et retenue = 1 |

```
      1 1
    1 2 4 5
+     6 6 5
-----------
=   1 9 1 0
```

## Exemples d'algorithme

Divisibilité d'un nombre par

2 : le nombre doit se terminer par 0, 2 , 4, 6 ou 8

5 : le nombre doit se terminer par 0 ou 5

10 : le nombre doit se terminer par 0

3 :

- 0, 3, 6, 9 sont divisibles par
- la somme des chiffres du nombre doit être divisible par 3

Preuve ?

## Exemples d'algorithmes

Fonction puissance : $x^n$

$x^9 = x.x.x.x.x.x.x.x.x = x.(x.(x.(x.(x.(x.(x.(x)))))))$

Algorithme :

$x^{n+1} = x.x^n$

ICI, pour calculer $x^n$, il faut n - 1 multiplications.

Autre solution :

$x^9 = x.(x.x.x.x).(x.x.x.x) = x.(x.x.x.x)^2 = x.((x.x)^2)^2 = x.((x\ ^2)^2)^2$

Algorithme:

$x^{2n} = (x^n)^2$ si n est pair

$x^{2n+1} = x.(x^{2n})$

I Soient $2.\log_2(n)$ multiplications

## Ce que l'on fera

- calculer *n!* pour n petit,
- calculer le $n^{ième}$ terme d'une suite,
- décider si un nombre *n* est premier,
- réaliser un jeu simple,
- calculer l'exponentielle d'un complexe,
- calculer le produit d'un polynôme...

## Ce que l'on ne fera pas

- calculer des images de synthèse,
- piloter un robot,
- analyser des logiciels existants,
- pirater des logiciels,
- simuler une explosion thermo-nucléaire à l'aide de rayons lasers...

## Langage de programmation

Langage défini par une syntaxe (grammaire) très simple et très rigoureuse (non ambiguë) permettant de décrire l'enchaînement d'instructions.

EX. Langage machine (assembleur)

- li $1, 1
- la $4,0xFFFF0000
- li $2,8
- addi $3, $2, $1

Basic :

```
10 for i = 1 to 10
20    k = k + 1 ;
30    next i
40 goto 10
```

Impératifs : cobol, fortran, algol, pascal, C, modula, ada

Fonctionnels : lisp, scheme, ML

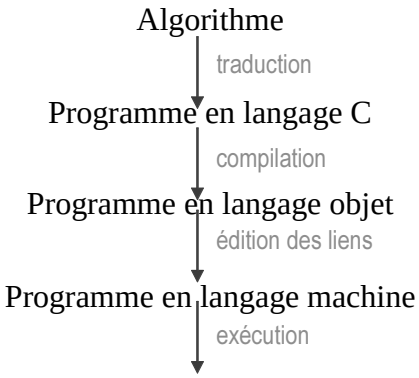
Logique : prolog

Objet : smalltalk, C++, eiffel, java

## Le langage C (1972)

Langage compilé de "haut niveau" :

- Puissant
- Concis
- Proche de la machine




Un programme écrit en langage C
peut être compilé et pour des machines différentes.

## Sommes et Produits

Expressions algébriques simples :

| Math | | C |
|---|---|---|
| $a + b$ | $\rightarrow$ | a + b |
| $a - b$ | $\rightarrow$ | a – b |
| $a \times b$ | $\rightarrow$ | a * b |
| $\frac{a}{b}$ | $\rightarrow$ | a / b |

Priorités des opérations :

- * et / sont prioritaires par rapport à + et -
- priorité à gauche pour les opérateurs de niveau équivalent

a * b + c = (a * b) + c

a*b/c + 3*a - 4*b = (((a*b)/c) + (3*a)) - (4*b)

a/b*c = (a/b) * c

## Parenthésage des expressions

a*b/c + 3*a - 4*b

(((a*b)/c) + (3*a)) - (4*b)

(a - b / c) * d * (e + f)

a / b / c est égal a /(b * c) mais pas à (a / (b / c))

Remarque

la fonction puissance n'existe pas en langage C :
il faut la programmer.

## Première fonction

```
int carre(int x){
        return x * x ;
}
```

Toute fonction possède :
- un nom (carre),
- un résultat (x*x) typé (int)
- des paramètres typés (int x)

Un **type** est une description d'un format de donnée,
c'est le mode d'emploi d'une case mémoire.

int : les entiers codés sur 32 bits $[-2^{31},2^{31}-1]$

## Affectation - Variable

Une variable est une case mémoire qui possède :
- un nom = adresse de la case
- un type = forme de la case
- une valeur = contenu de la case

Déclaration d'une variable : *type nom_de_variable ;*

```
int n ;        int a, A ;      int cumul ;
```

Affectation d'une variable : *nom_de_variable = expression ;*

```
n = 2 ;        a = n + 2 ;          n = n+ 1 ;
```

Une affectation n'est pas une équation, c'est une action :

1) Calculer la valeur de l'expression.

2) Ranger le résultat dans la case mémoire désignée par le nom de la variable.

## Fonction main()

```
int carre(int x){
        return x * x ;
}/*carre*/
void main(){ /*fonction principale */
        int k = 6 ;
        int r ;
        r = carre(3) + carre(2) ;
        r = carre(k) + 1 ;
        r = carre(carre(2)) ;
}/* main */
```

**Un programme C ne peut-être exécuté sans fonction main.**

## Fonction factorielle

Algorithme :

$$n! = \prod_{k=1}^{k=n} k = 1 \times 2 \times \cdots \times n$$

```
int factorielle(int n){
        int k ;
        int f = 1 ;
        for(k = 2 ; k <= n ; k++)
                f = f * k ;
        return f ;
} /* factorielle */
```

## Boucle « for »

```
int factorielle(int n){
        int k ;
        int f = 1 ;
        for(k = 2 ; k <= n ; k++ )
                f = f * k ;
        return f ;
} /* factorielle */
```

Structure de contrôle

action contrôlée

*for ( initialisation ; test ; incrémentation )*
*action ;*

1 exécuter la partie initialisation.
2 Si le test n'est pas validé alors aller en (6)
3 exécuter l'action (ou le bloc d'actions)
4 exécuter l'incrémentation (k = k + 1)
5 aller en (2)
6 ...

## Traduction en "langage machine" :

```
int factorielle(int n){
        int k ;
        int f = 1 ;
        for(k = 2 ; k <= n ; k++)
                f = f * k ;
        return f ;
} /* factorielle */
```

1 réserver une case pour k
2 réserver une case pour f et y mettre 1
3 mettre 2 dans la case de k
4 si k > n aller en 8
5 calculer f * k puis placer le résultat dans f
6 ajouter 1 à k
7 aller en 4
8 renvoyer la valeur de f

## Nombres à virgule flottante

**Calcul de la fonction exponentielle :** $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$

Impossibilité d'obtenir une valeur exacte.

•Calcul approché en fonction du rang ($n$)

•Calcul approché en fonction de la valeur du terme ($x^n/n!$)

a) Définir le profil de la fonction :

Résultat et paramètre sont réels : utilisation du type double.

[flottant codé sur 64 bits]

```
double exponentielle(double x) ;
double exponentielle(double x, int rang) ;
double exponentielle(double x, double epsilon) ;
```

## Fonction exponentielle

```
double puissance(double x, int n){
  int i ;
  double p = 1 ;
  for(i = 1 ; i <= n ; i++)
    p = p * x ;
  return p ;
}
```

```
double exponentielle(double x, int rang){
   int n ;
   double s = 1 ;
   for(n = 1 ; n <= rang ; n++)
      s = s + puissance(x,n) / factorielle(n) ;
   return s ;
}
```

## Fonction exponentielle

Intégrer le calcul de $x^n$ et celui de n! dans la boucle for :

```
double exponentielle(double x, int rang){
  int n ;
  double s = 1 , t = 1;
  for(n = 1 ; n <= rang ; n++)
  {
    t = t * x / n ;
    s = s + t ;
  }
  return s ;
}
```

| n | 1 | 2 | 3 |
|---|---|---|---|
| t | x/1 | (x/1).(x/2) | (x/1).(x/2).(x/3) |

## Fonction exponentielle

Calculer le terme jusqu'à ce qu'il soit *petit* :

```
double module(double x){
  if(x >= 0) return x ;
return -x ;
}
double exponentielle(double x, double epsilon){
  int n ;
  double s = 1 , t = 1;
  for(n = 1 ; module(t) > epsilon ; n++){
     t = t * x / n ;
     s = s + t ;
  }
  return s ;
}
```

module(t/s) > epsilon

s == s + t

## Instruction conditionnelle

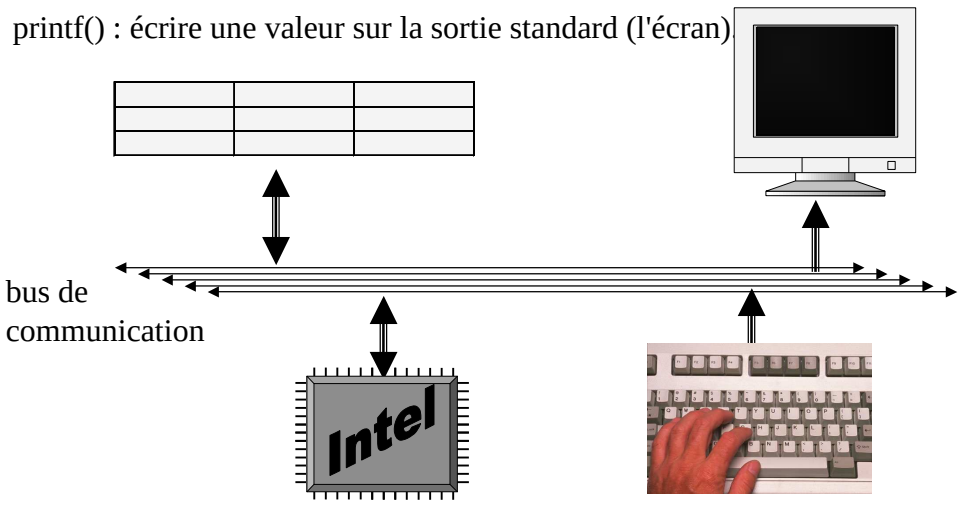
```
if (CONDITION)                  1 Si CONDITION est faux aller en 3
        Action A1 ;             2 Action A1 ;
Action A2;                      3 Action A2 ;

if (CONDITION)                  1 Si CONDITION est faux aller en 4
        Action A1 ;             2 Action A1 ;
else                            3 aller en 5
        Action A2 ;             4 Action A2
Action A3;                      5 Action A3
```

Expression conditionnelle

## Les Entrées/Sorties

scanf() : lire une valeur sur l'entrée standard (le clavier)

printf() : écrire une valeur sur la sortie standard (l'écran)



## printf()

```
printf("une chaîne de caractères") ;
```

Les caractères spéciaux :
un retour à la ligne \n
un entier %d

```
void main(){
   int i, n = 124 ;
   printf("un retour à la ligne \n") ;
   printf("un entier %d ",  n) ;
   /* afficher la table de multiplication par 7 */
   for (i=1 ; i <=10 ; i++)
        printf("%d x 7 = %d \n", i , 7*i) ;
}
```

## scanf()

Caractères spéciaux :
un entier %d
un double %lf

**&i désigne l'adresse de la case de i.**

```
void main(){
   int i, j ;
   printf("entre deux entiers :") ;
   scanf(" %d", &i) ;
   scanf("%d", &j) ;
   printf("%d + %d = %d\n", i, j , i + j) ;
}
```

## Entrées/Sorties de flottants

```
void main(){
   double pi = 3.14159 ;
   double r ;
   printf("Quel est le rayon ? \n") ;
   scanf( "%lf ", &r ) ;
   printf("Le périmètre est %lf \n", 2 * pi * r) ;
}
```

Les caractères spéciaux :
un retour à la ligne \n
un entier %d
un flottant double%lf

## Codage des informations : le BINAIRE

Informer = (étymologie) donner une forme, structurer

Information = signifiant + signifié

- représentation matérielle (forme, support, syntaxe,...) ;
- interprétation(s) (sémantique, essence, phénomène psychologique).

Pourquoi le binaire ?

Popularisé par Leibniz (vers 1700) en s'inspirant (très librement) de la philosophie chinoise du Yin / Yang.

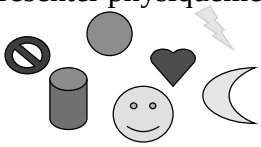
$\$\ \$\ \$\ \

## Le binaire

Pourquoi pas une base supérieure ?

• le nombre de chiffres utilisés est linéaire [ln(B)/ln(2)]
Un nombre en base 2 occupe 3,32 fois plus d'espace qu'en base 10.

• Il est facile d'analyser et de représenter physiquement un phénomène binaire.



•Les opérations sont très simples :

| | | | |
|---|---|---|---|
| 0 * 0 = 0 | 0 * 1 = 0 | 1 * 0 = 0 | 1 * 1 = 1 |
| 0 + 0 = 0 | 0 + 1 = 1 | 1 + 0 = 1 | 1 + 1 = 10 |

## Codage des nombres entiers

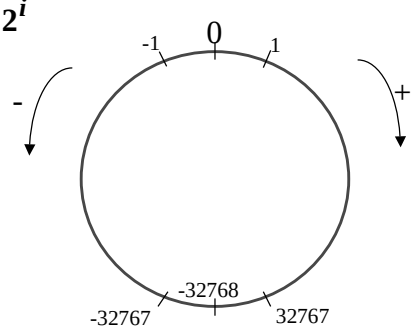
int : entier codé sur 32 bits [-2147483648, 2147483647] = [$-2^{31}$,$2^{31}$-1]

Codage *binaire en complément à deux* :

$$b_k b_{k-1} \ldots b_1 b_0 = -b_k 2^k + \sum_{i=0}^{i=k-1} b_i 2^i$$

(exemple sur 16 bits)

**32767 + 1 = -32768**



## Codage des nombres en virgule flottante

Virgule flottante = présence d'un exposant dans le codage

Norme IEEE754 : Signe Exposant Mantisse

S = $(-1)^{signe}$

Exposant codé par défaut $e_E e_{E-1} ... e_1 e_0 = 1 - 2^E + \sum_{i=0}^{i=E} e_i 2^i$

Mantisse $b_1 b_2 ... b_M = 1 + \sum_{i=1}^{i=M} b_i 2^{-i}$

float : nombre flottant codé sur 32 bits -- E=7 ~$10^{38}$ -- M=23 ~ 7 chiffres

double : flottant codé sur 64 bits -- E=10 ~$10^{308}$ -- M=52 ~15 chiffres

## Codage de 1,3

Signe $(-1)^s$ => S = **0**

Exposant $e_E e_{E-1} ... e_1 e_0 = 1 - 2^E + \sum_{i=0}^{i=E} e_i 2^i$

E = $2^0$ => $2^E$ - 1 = **011 …11**

Mantisse $b_1 b_2 ... b_M = 1 + \sum_{i=1}^{i=M} b_i 2^{-i}$

1,3 = $2^0$ (1 + 0 * ½ + 0,3)

= $2^0$ (1 + 0 * ½ + 1* ¼ + 0,05)

| 0,3 | 0,6 | 0,2 | 0,4 | 0,8 | 0,6 | 0,2 | 0,4 | 0,8 | 0,6 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |

## Chapitre II : suites récurrentes

**Récurrence à un terme :** $u_n = 2\,u_{n-1} + 1$.

Faire un programme qui calcule $u_n$ pour un $u_0$ donné.

a) définition du profil de la fonction

$u_n$ et n sont deux entiers `int suite1(int u0, int n) ;`

b) trouver l'algorithme :

| | | | |
|---|---|---|---|
| calculer | $u_1$ | à partir de | $u_0$ |
| calculer | $u_2$ | à partir de | $u_1$ |
| calculer | $u_n$ | à partir de | $u_{n-1}$ |

utiliser une boucle for pour appliquer n fois la règle u = 2u+1

## Récurrence à un terme

c) traduction

```
int suite1(int u0, int n){
  int i ;
  int u = u0 ;
  for(i=1 ; i <= n ; i++)
      u = 2 * u + 1 ;
  return u ;
}
```

Il n'y a pas d'indice : la variable **i** n'intervient que pour ***contrôler*** la boucle.

```
int suite1(int u0, int n){
        for( ; 1 <= n ; n--)
                u0 = 2 * u0 + 1 ;
        return u0 ;
}
```

## Suite récurrente à un terme

Cas général : on suppose disposer de la fonction int f(int x) ;

```
int suite_a_un_terme (int u0, int n){
  int i ;
  int u = u0 ;
  for(i=1 ; i <= n ; i++)
     u = f(u) ;
  return u ;
}
```

Pour bien faire, il faudrait passer la fonction en paramètre.

## Calcul de la série associée

$$\sum_{i=0}^{k} u_i \quad \text{où} \quad u_n = f(u_{n-1}) \quad \text{et} \quad u_0 = a$$

```
int serie(int u0, int k){
  int s = u0 ;
  int i ;
  for (i=1 ; i <= k ; i++)
     s = s + suite(u0,i) ;
  return s ;
}
```

```
int serie(int u0, int k){
   int u = u0, s = u0 ;
   int i ;
   for (i=1 ; i <= k ; i++) {
        u = f(u) ;
        s = s + u ;
   }
   return s ;
}
```

## Récurrences à 2 termes $u_{n+1} = f(u_n, u_{n-1})$

a) profil de la fonction

```
int suite2(int f(int, int), int u0, int u1, int n) ;
```

b) algorithme : faire n-1 fois une action calculant u en fonction des deux termes précédants : u = f (a,b)

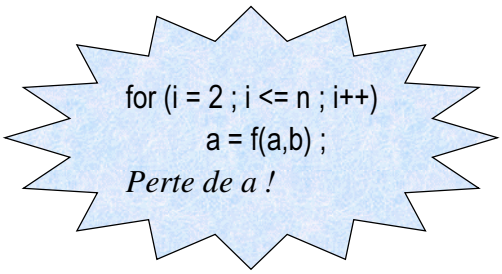
| | u0 | u1 | u2 | u3 | u4 |
|---|---|---|---|---|---|
| n=2 | b | a | u | | |
| n=3 | | b | a | u | |
| n=4 | | | b | a | u |

```
b = u0 ;      a = u1 ;      u = f(a,b) ;
              b = a ;       a = u ;       u = f(a,b) ;
                            b = a ;       a = u ;       u = f(a,b) ;
```

## Récurrences à 2 termes $u_{n+1} = f(u_n, u_{n-1})$

c) traduction

```
int suite_2(    int f(int, int), /* f est une fonction */
                int u0, int u1, int n)  /* n > 1 */
{
  int b = u0, a = u1, u ;
  int i ;
  for (i = 2 ; i <= n ; i++){
     u = f(a,b) ;
     b = a ;
     a = u ;
  }
  return u ;
}
```



## Cas particulier

$u_{2n} = u_{2n-1} * u_{2n-2}$

$u_{2n+1} = u_{2n} + 3$

$u_0 = 1$

i % 2
=
i modulo 2

```
int Exam98(int n)
{
  int a, b, tmp, i ;
  for (i = a = b = 1 ; i <= n ; i++)
  {
     if (i % 2 == 1)
         tmp = b + 3 ;
     else
         tmp = a * b ;
     b = a ;
     a = tmp ;
  }
  return a ;
}
```

## Cas particulier : autre solution

$u_{2n} = u_{2n-1} * u_{2n-2}$

$u_{2n+1} = u_{2n} + 3$

$u_0 = 1$

n / 2
division entière
3/2 == 1

```
int Exam98(int n)
{
   int pair = 1, impair = 4, i ;

   for(i = 1 ; i <= n / 2 ; i++)
   {
         pair = pair * impair ;
         impair = pair + 3 ;
   }
   if (n % 2 == 1)
         return impair ;
   return pair ;
}
```

## Récurrences à *k* termes

```
int suite_k_termes(int u0, int u1, int u2,...., int u20, int n)
{
  int u ;                              /*   n>20   * /
  int i ;
  for (i = 21 ; i <= n ; i++)
  {
     u = f(u20, u19, u18,..., u2, u1, u0) ;
     u0 = u1 ;
     u1 = u2 ;
      :   :
     u19 = u20 ;
     u20 = u ;
  }
  return u ;
}
```

Pour bien faire, il faudrait pouvoir appeler les variables par un nom paramétré

## Résolution d'équations numériques

Un joueur A doit deviner un nombre compris entre 0 et 1023 détenu par un joueur B.

A cette fin, le joueur A fait des propositions et reçoit, pour chacune d'elle, une réponse du joueur B qui peut être soit "trouvé", soit "trop grand" ou encore "trop petit". Simuler le joueur B :

Quelle est votre proposition ?
*124*
Trop grand
Quelle est votre proposition ?
*112*
Trop petit
Quelle est votre proposition ?
*116*
Gagné !

## Simulation du joueur B

```
#define MAX 1023
void main()
{
        int sol, prop ;
        randomize() ;
        sol = random(MAX+1) ; /*sol in [0,1023] */
        do{
                printf(”Quelle est votre proposition ?\n” ) :
                scanf("%d",&prop) ;
                if (prop>sol) printf("trop grand\n") ;
                else if (sol > prop) printf("trop petit\n") ;
        } while(sol != prop) ;
        printf("Gagné !") ;
}
```

## Instructions itératives

```
while (CONDITION)
        Action A1 ;
Action A2;
```

1 Si CONDITION est faux aller en 4
2 Action A1
3 aller en 1
4 Action A2 ;

**while** (cond) est équivalent à **for**(;cond;)

---

```
do
        Action A1 ;
while (CONDITION) ;
Action A2;
```

1 Action A1 ;
2 Si CONDITION est vrai aller en 1
3 Action A2

```
do                          Action A1 ;
        Action A1 ;         while (cond)
while (cond) ;                      Action A1 ;
Action A2;                  Action A2;
```

## Mémoriser l'intervalle et le nombre de coups

```
#define MAX 1023
void main(){
    int sol, prop, sup = MAX, inf = 0, coup = 0 ;
    randomize() ;
    sol = random(MAX+1) ; /*sol in [0,1023] */
    do{
            printf("entre un nombre entre %d et %d",inf,sup) ;
            scanf("%d",&prop) ;
            coup++ ;
            if (prop>sol) {
                    printf("trop grand\n") ;
                    sup = prop - 1 ;
            }else if (sol > prop){
                    printf("trop petit\n") ;
                    inf = prop + 1 ;
            }
    } while(sol != prop) ;
    printf("Gagné en %d coup(s)",coup) ;
}
```

## Simulation du joueur A

Ecrire un programme qui "trouve" le nombre détenu par le joueur B. Quelle stratégie (algorithme) doit adopter le joueur A pour gagner le plus rapidement possible ?

Proposer un nombre au hasard et recommencer.

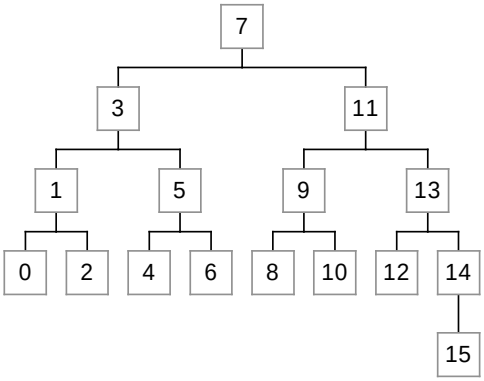
Proposer 0, puis 1, puis 2, ...et 1023 (ou un autre ordre)

Coût : 512 coups en moyenne.

Proposer un nombre au hasard dans l'intervalle de recherche.

# Méthode Dichotomique

*Diviser pour résoudre*

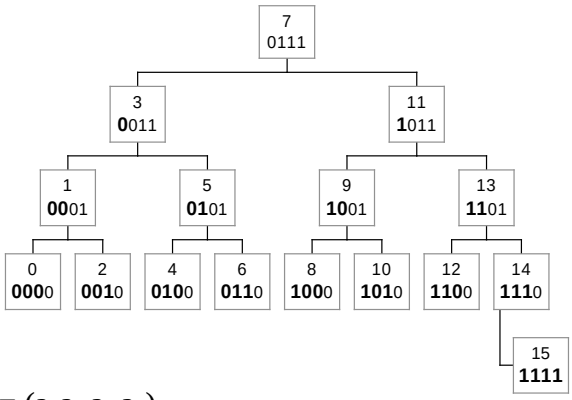



Arbre binaire de recherche équilibré.

# Résolution dichotomique

```
void main(){
int max = 1023, min = 0 , prop, rep ;
printf("choisissez un nombre entre %d et%d\n", min, max) ;
printf("-1 trop petit, 0 trouvé, +1 trop grand\n") ;
do{
    prop = (max+min) / 2 ;
    printf("ma proposition: %d\n", prop ) ;
    scanf("%d",&rep) ;
    if (rep < 0)
          min = prop + 1 ;
    else if (rep > 0)
           max = prop - 1 ;
    } while(rep != 0) ;
}
```

## Analyse de la méthode dichotomique

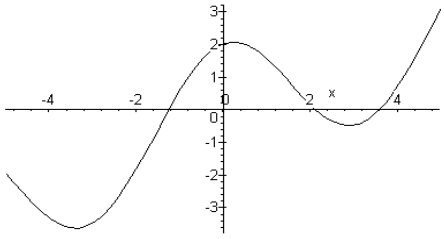



Posons sol = $(a_3 a_2 a_1 a_0)_2$.

Proposer $(0111)_2$ revient à demander la valeur de $a_3$ !

## Résolution d'équations numériques

Soit f une fonction réelle définie et continue sur [a,b] et possédant (au moins) une racine dans [a,b].



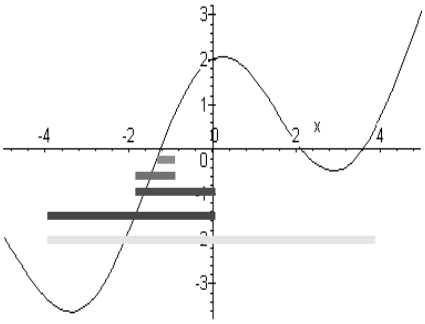

Trouver une solution approchée de f(x) = 0 en construisant une suite qui converge vers la solution.

- Méthode dichotomique
- Méthode de Lagrange
- Méthode de Newton

# Méthode dichotomique

```
double dichotomie(      double f(double), /* f est une fonction */
                        double a, /* f(a) < 0 */
                        double b, /* f(b) > 0 */
                        double epsilon)
{
   double x = a ;
   while (fabs(a-b) > epsilon){
      x = (a + b) / 2 ;
      if( f(x) > 0)
            b = x ;
      else
            a = x ;
   }
   return x ;
}
```

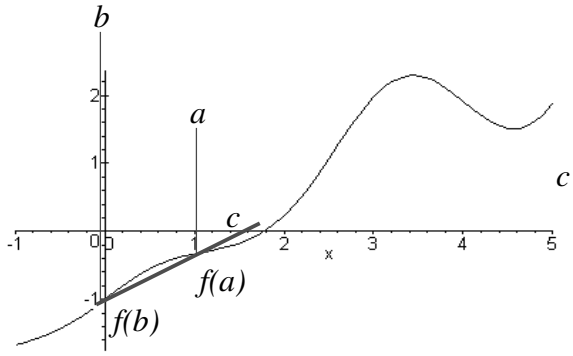
La fonction
double fabs(double d) ;
est définie dans math.h




# Exemple de résolution

```
double ma_fonction(double x)
{
        return 2 * x + cos(x) - 1 ;
}
void main()
{
        printf("la fonction 2x + cos(x) - 1 s'annule en %lf",
          dichotomie(ma_fonction, -5, 5, 1E-5) ) ;
}
```

# Méthode de Lagrange




$$c = \frac{a.f(b) - b.f(a)}{f(b) - f(a)}$$

définir la droite $u.x + v$ passant par $(a, f(a))$ et $(b, f(b))$

$$f(a) = u.a + v \qquad f(b) = u.b + v$$

$$u = \frac{f(a) - f(b)}{a - b} \qquad v = \frac{a.f(b) - b.f(a)}{a - b}$$

$$0 = \frac{f(a) - f(b)}{a - b} c + \frac{a.f(b) - b.f(a)}{a - b}$$

# Méthode de Lagrange

```
double lagrange( double f(double), /* f est une fonction */
                 double a,
                 double b,
                 double epsilon)
{
  double c ;
  while (fabs(a-b) > epsilon){
    c = (a*f(b) - b*f(a)) / (f(b) - f(a)) ;
    b = a ;
    a = c ;
    }
  return c ;
}
```

$$c = \frac{a.f(b) - b.f(a)}{f(b) - f(a)}$$

## Exemple de résolution

```
double ma_fonction(double x)
{
        return 2 * x + cos(x) - 1 ;
}
void main()
{
printf("la fonction 2x + cos(x) - 1 s'annule en %lf",
          lagrange( ma_fonction, -5, 5, 1E-5) ) ;
}
```

## Méthode de Lagrange + test de convergence

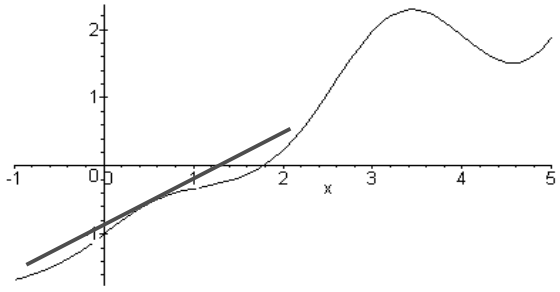
S'arrêter après au plus un certain nombre d'itérations.

```
#define MAXITER 100
double lagrange( double f(double), double a, double b, double epsilon)
{
  double c = a ; int cpt = 0 ;
  while (fabs(a-b) > epsilon && cpt < MAXITER){
    c = (a*f(b) - b*f(a)) / (f(b) - f(a)) ;
    b = a ;
    a = c ;
   cpt ++ ;
  }
  return c ;
}
```

## Méthode de Newton

Utiliser la tangente en un point à la place d'un couple de points :
$g(x) = f'(a).x - a.f'(a) + f(a)$ d'où $x = a - f(a)/f'(a)$
Suite réelle définie par

$$a_0 = a \quad \text{et} \quad a_{n+1} = a_n - f(a_n) / f'(a_n)$$




## Méthode de Newton

$$a_0 = a \quad \text{et} \quad a_{n+1} = a_n - f(a_n) / f'(a_n)$$

```
double newton(double f(double), double fprime(double),
              double a, double epsilon)
{
double b ;
do{
        a=b ;
        b = a -  f(a) / fprime(a) ;
} while(fabs(b-a) > epsilon) ;
return b ;
}
```

## Chapitre IV : données structurées

- une adresse : *numéro, type de voie, nom de voie, code postal, ville, pays.*
- une voiture : *numéro d'immatriculation, numéro de moteur couleur, année de production, modèle, marque.*
- une étoile : *nom, coordonnées polaires, distance, magnitude, couleur, type.*
- un micro-ordinateur : *µ-processeur, fréquence, carte-mère, mémoire, disque dur, carte graphique, carte son.*
- un nombre rationnel : *numérateur, dénominateur.*
- un point pondéré : *coordonnées, poids.*
- un nombre complexe : *partie réelle, partie imaginaire.*

## Structure "rationnel"

**Définition de la structure de donnée :**

```
struct rat{
   int num, deno ;              /* numérateur / dénominateur */
} ;
```

**Définition d'une variable de type struct rat :**

```
   struct rat r , x ;
```

**Affectation**

```
   r = {1,2} ; x = r ;
   r.num = 1 ; r.deno = 2 ;  /* r.num est une variable de type int */

x = x + r ; /* erreur : l'opération + n'est pas définie */
```

## Opérations sur les rat

*« Program = Algorithm + Data Structure »*

```
struct rat somme(struct rat a, struct rat b){
  struct rat c ;
  c.num = a.num * b.deno + b.num * a.deno ;
  c.deno = a.deno * b.deno ;
  return c ;
}

struct rat a, b, c ;
…
c = somme(a,b) ;
```

RMQ : utilisation du pgcd pour éviter les débordements inutiles

## Structure "point pondéré"

```
struct point_pondere {
  double x, y ;
  unsigned int poids ; /* poids >= 0 */
} ;
```

Déclaration de l'équivalence entre les types point_pondere et struct point_pondere :

```
typedef struct point_pondere point_pondere ;
```

Faire un programme qui simule le comportement suivant :

*Entre le premier point pondéré : 0 0 1*

*Entre le second point pondéré : 10 0 4*

*Le barycentre des 2 points est (8,0,5)*

## Calcul du barycentre

```
point_pondere barycentre(point_pondere a, point_pondere b)
 {
   point_pondere c ;
   c.poids = a.poids + b.poids ;
   c.x = (a.poids * a.x + b.poids * b.x) / c.poids ;
   c.y = (a.poids * a.y + b.poids * b.y) / c.poids ;
   return c ;
}

/*{
   return (point_pondere) {(a.poids * a.x + b.poids * b.x) / (a.poids + b.poids),
           (a.poids * a.y + b.poids * b.y) / (a.poids + b.poids),
            a.poids + b.poids
   } ; }*/
```

## Entrée/Sortie de PointPondéré

```
void ecrire(point_pondere a) {
   printf("(%lf , %lf , %d)",
          a.x, a.y, a.poids) ;
}
```

```
point_pondere  lire() {
   point_pondere a ;
   scanf("%lf", &a.x) ;
   scanf("%lf", &a.y) ;
   scanf("%d", &a.poids) ;
   return a ;
}
```

la fonction écrire n'a pas de résultat (void) ;

la fonction lire n'a pas de paramètre.

## Programme principal

```
Entre le premier point pondéré : 0 0 1
Entre le second point pondéré : 10 0 4
Le barycentre des 2 points est (8,0,5)
```

```
void main(){
  point_pondere a, b  ;
  printf("Entre le premier point pondéré :") ; a = lire() ;
  printf("Entre le second point pondéré :") ;  b = lire() ;
  printf(" Le barycentre des 2 points est ") ;
  ecrire(barycentre(a,b)) ;
}
```

## Les complexes

Rappels mathématiques : couple de $R^2$ noté $a + ib$

$a$ : partie réelle $b$ : partie imaginaire $i^2 = -1$

- Déclaration de la structure de donnée ;
- Définition des opérations usuelles :
  somme : $(a+ib) + (a'+ib') = (a+a') + i\,(b+b')$
  produit : $(a+ib)(a'+ib') = (a.a'-b.b') + i\,(a.b'+a'.b)$
  puissance,
  module : $\sqrt{a^2+b^2}$
- Entrées / sorties de complexe

## Structure complexe

```
struct complexe {
  double reelle, imaginaire ;
} ;
typedef struct complexe complexe ;

void main()
{
  complexe a, b = {1,0} ;
  a = b ;
  a.reelle = cos(3.14159 /  2) ;
  a.imaginaire = sin(3.14159 /  2) ;
  a = b + 1 ; /* ERREUR */
  a = 2 + 3*i ;  /* ERREUR */
}
```

## Somme de 2 complexes

Prototype (entête, profil) de la fonction :

Entrées : 2 complexes a et b

Résultat : le complexe a + b

```
complexe somme(complexe a, complexe b ) ;
```

algorithme : renvoyer le complexe ayant

- pour partie réelle, la somme des parties réelles ;
- pour partie imaginaire, la somme des parties imaginaires.

## Somme de 2 complexes

```
complexe somme(complexe a, complexe b )
 {
  complexe c ;
  c.reelle = a.reelle + b.reelle ;
  c.imaginaire = a.imaginaire + b.imaginaire ;
  return c ; /* return (complexe) {a.reelle + b.reelle , a.imaginaire + b.imaginaire } ;*/
}
```

## Somme de trois nombres complexes

```
complexe som3(complexe a, complexe b, complexe c)
{
  return somme(a,somme(b,c)) ;
}
```

## Produit de 2 complexes

```
complexe produit(complexe a, complexe b)
{
  complexe c ;
  c.reelle = a.reelle * b.reelle - a.imaginaire * b.imaginaire ;
  c.imaginaire = a.reelle * b.imaginaire + a.imaginaire * b.reelle ;
  return c ;
}
```

## Produit de quatre nombres complexes

```
complexe prod4(complexe a, complexe b, complexe c, complexe d)
{
  return produit(produit(a,b),produit(c,d)) ;
}
```

## Exponentiation d'un complexe

```
complexe puissance(complexe a, int n)
{
  int i ;
  complexe p = (complexe){1,0} ;
  for(i=1 ; i <= n ; i++)
     p = produit(p,a) ;
  return p ;
}
```

## Module d'un complexe

```
double module(complexe a)
{
  return sqrt(a.reelle * a.reelle + a.imaginaire * a.imaginaire) ;
}
```

## Entrées/Sorties de complexe

```
void ecrire(complexe a)
{
  printf("%lf + i %lf",a.reelle, a.imaginaire) ;
}
```

le complexe {1,1} s'affiche 1 + 1 i

```
complexe lire()
{
  complexe a ;
  printf("partie réelle :") ; scanf("%lf",&a.reelle) ;
  printf("partie imaginaire") ; scanf("%lf",&a.imaginaire) ;
  return a ;
}
```

## Limite des structures

```
struct V3 { /* vecteur de dimension 3 */
   double x, y, z ;
} ;
struct V3 addition (struct V3 a, struct V3 b){
   struct V3 c ;
   c.x = a.x + b.x ;
   c.y = a.y + b.y ;
   c.z = a.z + b.z ;
   return c ;
}
```

```
struct V30 { /* vecteur de dimension 30 */
   double x0, x1, x2, x3, x4, x5, x6,..., x29 ;
} ;
struct V30 addition (struct V30 a, struct V30 b){
   struct V3 c ;
   c.x0 = a.x0 + b.x0 ;
   c.x1 = a.x1 + b.x1 ;
   c.x2 = a.x2 + b.x2 ;
   c.x3 = a.x3 + b.x3 ;
   ::::::::::::::::::::::::::::::
   c.x28 = a.x28 + b.x28 ;
   c.x29 = a.x29 + b.x29 ;
   return c ;
}
```

## ChapitreV : tableaux

Un tableau est un ensemble de variables de même type.

Un tableau possède un nom et une taille (nombre d'éléments).

Déclaration : type *nom* [*taille*]

```
int tab[8] ; /* tableau de 8 entiers */
double vecteur[5] ; /* tableau de  5 doubles */
```

| tab | 12 | -3 | 23 | 0 | 12 | 2 | -5 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| vecteur | 12.4 | -3.04 | 42.3 | 90 | 12.9 |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |

**Les variables d'un tableau sont rangées de façon consécutive en mémoire.**

## Accès aux variables

nom d'une variable d'un tableau = nom du tableau + indice

*nom_du_tableau*[*indice*]

```
tab[0] = 2 ;
tab[3] = 34 ;
tab[6] = tab[2] * tab[4] + 1 ;
tab[8] = -1 ; /*  erreur */
```

tab

| 12 | -3 | 23 | 0 | 12 | 2 | -5 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

L'indice du premier élément d'un tableau est 0.

L'indice du dernier élément d'un tableau de k éléments est k-1

```
int tab[8] ; /* 8 éléments, les indices vont de 0 à 7 */
```

## Affectation

Mettre tous les éléments de tab à 0

=>  mettre à 0, une à une,  les variables de tab :

```
#define TAILLE 12
void main()
{
  int tab[TAILLE] ;
  int i ;
  for( i = 0 ; i < TAILLE ; i++)
    tab[i] = 0 ;
}
```

## Affectation (2)

```
int u[10], v[10] ;
```

Copier v dans u

```
u = v ; /* erreur : u est un tableau, pas une variable */
```

=> Une fonction ne peut pas renvoyer un tableau.

Solution :

```
int i ;
for (i=0 ; i < 10 ; i++)
        u[i] = v[i] ;
```

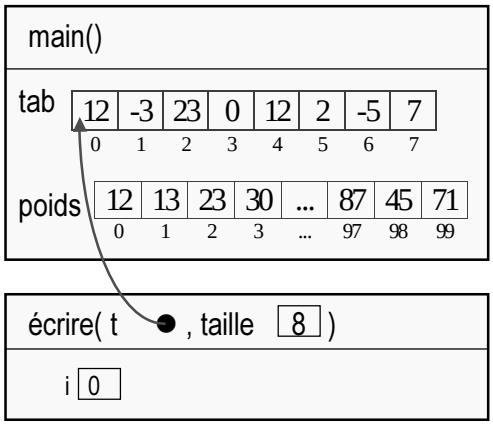
u est une constante, on ne manipule que ses éléments.

## Tableau passé en paramètre

```
void écrire(int t[ ], int taille)
{
  int i ;
  for(i = 0 ; i < taille ; i++)
    printf("  %d ", t[i]) ;
}
void main()
{
  int tab[8], poids[100]  ;
  ...
  écrire(tab, 8) ;
  écrire(poids,100) ;
}
```

## Passage d'un tableau en paramètre

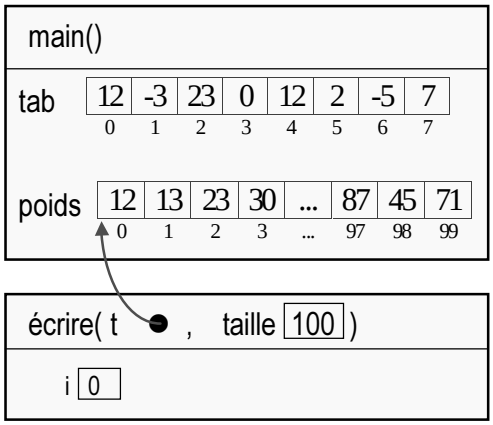
```
void écrire(int t[ ], int taille)
{
  int i ;
  for(i = 0 ; i < taille ; i++)
    printf("  %d ", t[i]) ;
}
void main()
{
  int tab[8], poids[100]  ;
...
  écrire( tab, 8) ;
  écrire(poids,100) ;
}
```




Le tableau n'est pas recopié lors de l'appel de la fonction : seule l'adresse du tableau est recopiée.

## Passage d'un tableau en paramètre (2)

```
void écrire(int t[ ], int taille)
{
  int i ;
  for(i = 0 ; i < taille ; i++)
    printf("  %d ", t[i]) ;
}
void main()
{
  int tab[8], poids[100]  ;
  ...
  écrire( tab, 8) ;
  écrire( poids,100) ;
}
```



## Quelques fonctions sur les tableaux

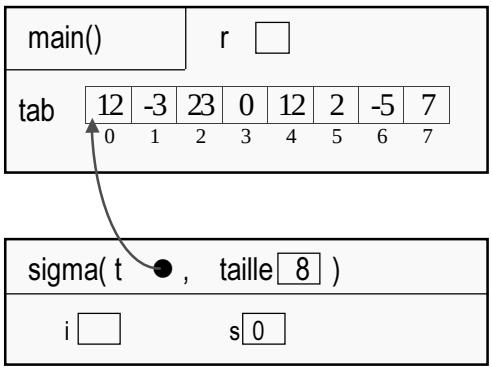
Ecrire des fonctions qui retournent :

- la somme des éléments d'un tableau d'entiers ;
- le produit des éléments d'un tableau d'entiers ;
- la valeur du plus grand élément d'un tableau ;
- l'indice de la première case contenant une valeur passée en paramètre ; on retournera -1 si la valeur recherchée n'apparaît pas dans le tableau.

Faire un programme qui lit (au clavier) deux vecteurs de dimension DIM puis qui affiche la somme et le produit scalaire de des vecteurs.

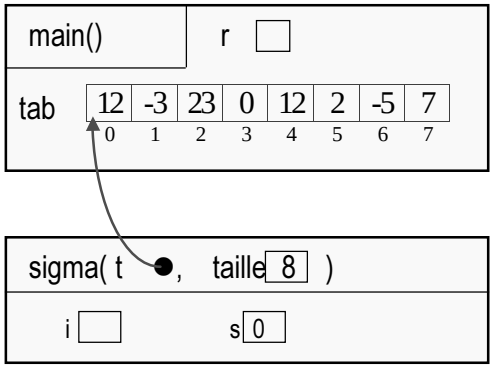
## Somme des éléments d'un tableau d'entiers

```
int sigma(int t[ ], int taille)
{
  int i, s = 0 ;
  for(i=0 ; i < taille ; i++)
    s = s + t[i] ;
  return s ;
}
void main()
{
  int r, tab[8] ;
  ...
  r = sigma(tab,8) ;
}
```



## Somme des éléments d'un tableau d'entiers

```
int sigma(int t[ ], int taille)
{
  int i, s = 0 ;
  for(i=0 ; i < taille ; i++)   s = s + t[i] ;
  return s ;
}
```




| |
|---|
| i = 0 s =0 |
| s= s + t[i] = 0 + t[0] = 0 + 12 |
| i = 1 s = 12 |
| s = s + t[i] = 12 + t[1] = 12 - 3 |
| i = 2 s =9 |
| ... |
| ... |
| i = 7 s = 41 |
| s= s + t[i] = 41 + t[7] = 41 + 7 |
| i = 8 s = 48 |
| retourne 48 |

## Produit des éléments d'un tableau d'entiers

```
int produit(int t[ ], int taille)
{
  int i, p = t[0] ;
  for(i=1 ; i < taille ; i++)
    p = p * t[i] ;
  return p ;
}
```

## Plus grand élément d'un tableau d'entiers

```
int max(int t[ ], int taille)
{
  int i, m = t[0] ;
  for(i=1 ; i < taille ; i++)
    if (t[i] > m)
        m = t[i] ;
  return m ;
}
```

## Recherche d'une valeur dans un tableau

Calculer l'indice de la première case contenant une valeur passée en paramètre ; on retournera -1 si la valeur recherchée n'apparait pas dans le tableau.

indice de 12 = 0
indice de 0 = 3
indice de 34 = -1

tab

| 12 | -3 | 23 | 0 | 12 | 2 | -5 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

prototype :

```
int indice(int t[ ], int taille, int val) ;
```

## Recherche d'une valeur dans un tableau

```
int indice(int t[ ], int taille, int val)
{
  int i = 0 ;
  while (i < taille && t[i] != val)
    i++ ;
  if (i < taille) return i ;
  return -1 ;
}
```

```
int indice(int t[ ], int taille, int val)
{
  int i ;
  for (i = 0 ; i < taille ; i++)
    if (t[i] == val) return i ;
    /* surtout pas de else !!! */
  return -1 ;
}
```

```
indice(tab, 8, 12) ;
indice(tab, 8, 3) ;
indice(tab, 8, 34) ;
```

tab

| 12 | -3 | 23 | 0 | 12 | 2 | -5 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

## Vecteurs

Simuler le comportement suivant :

Quelle est la dimension de l'espace vectoriel ? *2*

Entre le premier vecteur : *12.0 1.5*

Entre le second vecteur : *0.5 10*

Le produit scalaire de (12.0 , 1.5) et de (0.5 , 10) est 21.0

Leur somme est égale à (12.5 , 11.5)

## Vecteurs

```
/* Ecrire un vecteur -- exemple (1.23 , 123 , -34.44) */
void écrire_vecteur (double u[ ], int dim)
{
   int i ;
   printf ("( %lf",u[0]) ;
   for(i=1 ; i < dim ; i++) printf(" , %lf",u[i]) ;
    printf(" )") ;
}
double prodsca(double u[ ], double v[ ], int dim)  /* Produit scalaire*/
 {
   int i ;
   double p = 0;
   for(i=0 ; i < dim ; i++)
      p = p + u[i] * v[i] ;
   return p ;
}
```

## Lecture d'un vecteur

entrée : une dimension k

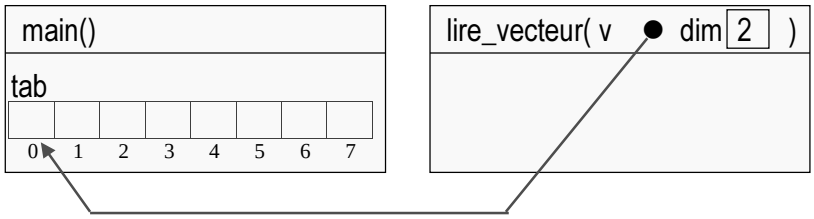
sortie : un tableau dont les k premières cases ont été affectées.

Prototype : on ne peut pas affecter directement un tableau

u = lire_vecteur(k)est interdit

Solution `void lire_vecteur(double v[ ], int dim) ;`

lire_vecteur(tab,2) ;




## lire_vecteur()

```
void lire_vecteur(double v[ ], int dim)
{
int i ;
for (i = 0 ; i < dim ; i++)
        scanf("%lf", &v[i]) ;
}
```

## addition()

```
void addition(double r[ ], double u[ ] , double v[ ], int dim)
{
int i ;                    /* r = u + v */
for (i = 0 ; i < dim ; i++)
  r[i] = u[i] + v[i] ;
}
```

## Vecteurs : programme principal

```
#define MAX 100
void main()
{
int d ; /* dimension */
double x[MAX], y[MAX] , res[MAX];
printf(" Quelle ... ? ") ;
scanf("%d",&d) ;
printf("Entre le premier vecteur") ;
lire_vecteur(x,d) ;
printf("Entre le second vecteur") ;
lire_vecteur(y,d) ;
printf("le produit scalaire de ") ;
ecrire_vecteur(x,d) ;
printf("et de") ;
```

```
Quelle est la dimension de l'espace vectoriel ? 2
Entre le premier vecteur : 12.0      1.5
Entre le second vecteur :0.5         10
Le produit scalaire de (12.0,1.5) et de (0.5,10) est 21.0
Leur somme est égale à (12.5 , 11.5)
```

```
ecrire_vecteur(y,d) ;
printf("est %lf\n", prodsca(x,y,d) ) ;
printf("Leur somme est égale à") ;
addition(res,x,y,d) ;
ecrire_vecteur(res,d) ;
}
```

## Algèbre de Boole (1847)

Éléments : { ***faux*** , ***vrai*** }

Opérations : **et** , **ou**

*Vrai* et *Vrai* = *Vrai*
*Vrai* et *Faux* = *Faux*
*Faux* et *Vrai* = *Faux*
*Faux* et *Faux* = *Faux*

*Vrai* ou *Vrai* = *Vrai*
*Vrai* ou *Faux* = *Vrai*
*Faux* ou *Vrai* = *Vrai*
*Faux* ou *Faux* = *Faux*

Eléments neutres

Commutative : *a* et *b* = *b* et *a*
Distributive : *(a* et *b)* ou *c* = *(a ou c) et (b ou c)*
Associative : *(a* et *b)* et *c* = *a* et *(b* et *c)*

Algèbre complémenté **non** : non(*Vrai*) = *Faux* , non(*Faux*) = *Vrai*

## Preuve par énumération

Lois de Morgan :

non(*a* ou *b*) = non(*a*) et non(*b*)

non(*a* et *b*) = non(*a*) ou non(*b*)

| *a* | b | a ou b | non (a ou b) | non(a) | non(b) | non(a) et non(b) |
|---|---|---|---|---|---|---|
| V | V | V | F | F | F | F |
| V | F | V | F | F | V | F |
| F | V | V | F | V | F | F |
| F | F | F | V | V | V | V |

Il existe $2^4$ opérations binaires :

| = | V | F |
|---|---|---|
| V | V | F |
| F | F | V |

| <> | V | F |
|---|---|---|
| V | F | V |
| F | V | F |

| => | V | F |
|---|---|---|
| V | V | F |
| F | V | V |

Prouver que :

- (a => b) $=_{def}$ non(a) ou non(b) $=_{def}$ (non(b) => non(a))
- (a <=> b) $=_{def}$ ( a = b)

## Opérations booléennes en langage C

On utilise les entiers pour coder les booléens :

**0** représente *faux*, tous les autres entiers représentant *vrai*.

et **&&** *(et alors : et non commutatif)*

ou **||** *(ou sinon : ou non commutatif)*

non **!**

Exemples :

```
(x && y)      ==    (x != 0 && y !=0)
! x || y      ==    x == 0 ||  y != 0
```

Les comparaisons sont des *expressions booléennes* (a > b).

On traduit *vrai* par différent de 0.

## Conjonction et disjonction

Soit t un tableau de n booléens, calculer t[0] et t[1] et … et t[n-1].

```
int conjonction(int t[], int n)
{
  int i ;
  for( i = 0 ; i < n ; i++)
    if (!t[i]) return 0 ;
  return 1 ;
}
```

```
for( i = 0 ; i < n - 1 && t[i] ; i++) ;
return t[i] ;
```

Soit t un tableau de n booléens, calculer t[0] ou t[1] ou … ou t[n-1].

```
int disjonction(int t[], int n)
{
  int i ;
  for( i = 0 ; i < n ; i++)
    if (t[i]) return 1 ;
  return 0 ;
}
```

```
for( i = 0 ; i < n - 1 && !t[i] ; i++) ;
return t[i] ;
```

## Démonstration automatique

Soit *f* et *g* deux fonctions booléennes de même arité (n).
Vérifier si *f* == *g*.
Tester pour *toutes les configurations t* si f(t) == g(t).

```
0  0 0 0 0 0 0
1  0 0 0 0 0 1
2  0 0 0 0 1 0
3  0 0 0 0 1 1
4  0 0 0 1 0 0
5  0 0 0 1 0 1
6  0 0 0 1 1 0
7  0 0 0 1 1 1
...
```

```
int successeur(int t[], int n)
{
  int i ;
  for(i = 0 ; i < n && t[i] ; i++) t[i] = 0 ;
  if (i == n) return 0 ; /* 1 1 1 1 -> 0 0 0 0 */
  t[i] = 1 ;
  return 1 ;
}
```

## Démonstration automatique

```
int egal( int f() , int g(),  int n)
{
  int i ;
  for(i = 0 ; i < n ; i++)
    t[i] = 0 ;
  while( f(t) == g(t) )
    if ( !successeur(t,n) )
      return 1 ;
  return 0 ;
}
```

## Codage des caractères

Code ASCII

**A**merican **S**tandard for **C**ommunication **I**nternational **I**nterchange

**A**merican **S**tandard **C**oding for **I**nformation **I**nterchange

**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange

Codage des caractères (anglo-saxons) sur 7 bits :

```
'0' -> 48    '1' -> 49    ...
'A' -> 65    'Z' -> 90
'a' -> 97    'z' -> 122
```

Code ISO-LATIN-1 : code ASCII étendu (8 bits) aux caractères accentués.

UNICODE : codage permettant de coder la plupart des écritures.

## Code ASCII

Programme affichant le code ASCII d'un caractère lu au clavier :

```
void main(){
  char c ;
  scanf("%c", &c) ;
  printf("%d", c) ;
}
```

Programme affichant le caractère correspondant à un code ASCII lu au clavier :

```
void main(){
  char c ;
  scanf("%d", &c) ;
  printf("%c", c) ;
}
```

La variable c contient un entier désignant un caractère.

## Fonctions standards

```
int est_minus(char c)
{
  return c >= 'a' && c <= 'z' ;
}

int est_majus(char c)
{
  return c >= 'A' && c <= 'Z' ;
}

int est_lettre(char c)
{
  return est_minus(c)  || est_majus(c) ;
}

int est_chiffre(char c)
{
  return c >= '0' && c <= '9' ;
}

int valeur_chiffre(char c)
{
  return c - '0' ;
}
```

## Chaînes de caractères

On appelle chaîne de caractères *(chaîne)* toute séquence de caractères terminée par le caractère nul (de code 0).

Les chaînes sont stockées dans des tableaux de caractères :

```
char une_chaine[] ={'U','n','e',' ','c','h','a','i','n','e',0} ;
char une_autre[] = "une autre chaine" ; /* le 0 est insérer automatiquement */
```

Pièges :

- une chaîne doit se terminer par le caractère nul :

```
char un_tableau[] ={'N','o','n',' ','c','h','a','i','n','e'} ;
```

- "**a**" est une chaîne, '**a**' est un caractère ;
- le caractère nul n'est pas le caractère '0'.

## Fonctions standards

```
int strlen(char chaine[])  ;
                                        /* longueur de la chaîne */
void strcpy(char destination[], char source[]) ;
                                        /* copie source dans destination */
void insert(char chaine[], int pos, char source[]) ;
                                        /* insère chaîne en position pos dans source */
void remove(int debut, int longueur, char source[]) ;
                                        /* supprime dans source longueur caractères à partir de début */
int match(char motif[], char source[]) ;
                                        /* donne la première position de la chaîne motif dans source (-1 sinon)*/
void replace(char motif[], char par[], char source[]) ;
                                        /* remplace toutes les occurrences de motif dans source par "par" */
```

## Longueur d'une chaîne

La chaîne "123456789" a pour longueur 9.

```
void strlen(char source[])
{
  int i ;
  for (i = 0 ; source[i] ; i++ )    /* source[i] != 0 */
    ;
  return i ;
}
```

## Copier

Soit une chaîne source, recopier la chaîne destination

```
void strcpy(char destination[], char source[])
{
  int i ;
  for (i = 0 ; source[i] !=0  ; i++)
    destination[i] = source[i] ;
  destination[i] = 0 ;
}
```

```
void strcpy(char destination[], char source[])
{
  int i = 0 ;
  while (destination[i] = source[i])
    i++ ;
}
```

Une affectation est aussi une expression !

## Mise en forme de texte

Recopier une chaîne dans une autre en remplaçant par un seul espace dans toute suite d'espaces consécutifs.

**"Ich bin ein berliner !"**

**"Ich bin ein berliner !"**

```
void nettoyer_espace(char destination[], char source[]){
  int s = 0 , d = 0 ;
  while (source[s] !=0 ) {
    while(source[s] == ' ' && source[s+1] == ' ' )
      s++ ;/* passer les blancs */
    destination[d] = source[s] ;
    s++ ;
    d++ ;
  }
  destination[i] = 0 ;
}
```

## Rechercher une chaîne dans une autre

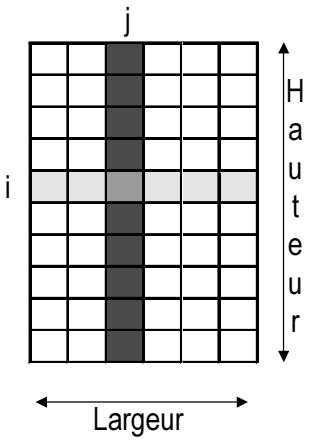
Retourne la position de la première occurrence d'un motif dans une chaîne.

match("456","012345689") retourne 4

```
int match(char motif[], char source[])
{
  int pos = 0, i = 0 ;
  do {
    if (motif[i] == 0)
      return pos ;
    else if (source[pos+i] == motif[i]))
      i++ ;
    else {
      pos++ ;
      i = 0 ;
      }
    } while(source[pos]) ;
  return -1 ;
}
```

# Matrices

Une matrice est représentée par un "tableau de tableaux" :




Déclaration du type *matrice*

```
#define Hauteur 425
#define Largeur 100
typedef double matrice [Hauteur][Largeur] ;
```

Déclaration d'une matrice : matrice m ;

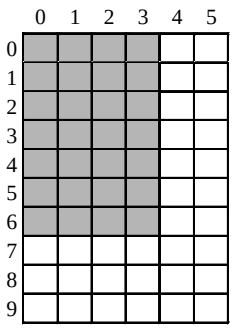
Accès à l'élément $M_{ij}$ : m[i][j]

# Afficher une matrice

```
void afficher(matrice m, int ligne, int colonne)
{
  int l, c ;
  for(l = 0 ; l < ligne ; l++)
    { /* pour chaque ligne */
      for(c = 0 ; c < colonne ; c++)
        prinft("%lf", m[l][c] ) ;
      printf("\n") ;
  } /* for */
}
```

colonne = 4

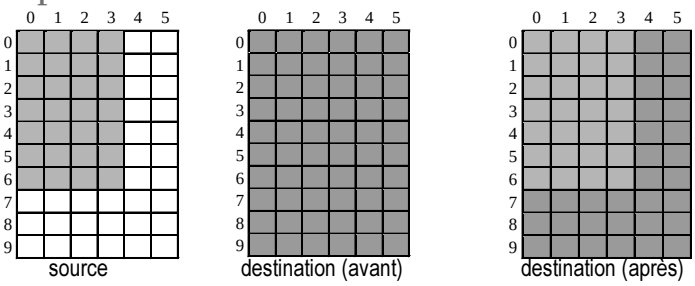
ligne =7



## Copier une matrice dans une autre

colonne = 4

ligne =7




```
void copier(matrice destination, matrice source, int ligne, int colonne)
{
  int l, c ;
  for(l = 0 ; l < ligne ; l++) /* pour chaque ligne */
    for(c = 0 ; c < colonne ; c++) /* pour chaque élément */
      destination[l][c] = source[l][c] ;
}
```

## Matrices diagonales

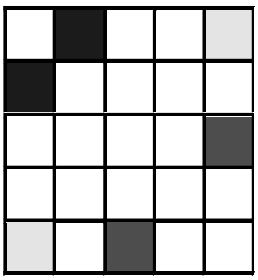
```
int est_diagonale(matrice a, int dim)
{
  int l, c ;
  for(l = 0 ; l < dim ; l++)
    for(c = 0 ; c < dim ; c++)
      if (a[l][c] && l  != c)
        return 0 ;
  return 1 ;
}
```

| | | | | |
|---|---|---|---|---|
| ■ | 0 | 0 | 0 | 0 |
| 0 | ■ | 0 | 0 | 0 |
| 0 | 0 | ■ | 0 | 0 |
| 0 | 0 | 0 | ■ | 0 |
| 0 | 0 | 0 | 0 | ■ |

```
for(l = 0 ; l < dim ; l++)
  for(c = 0 ; c < l ; c++)
    if (a[l][c] || a[c][l] )
      return 0 ;
```

## Matrices symétriques

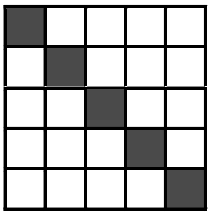
```
int est_symetrique(matrice a, int dim)
{
  int l, c ;
  for(l = 0 ; l < dim ; l++)
    for(c = 0 ; c < dim ; c++)
      if (a[l][c] != a[c][l] )
    return 0 ;
  return 1 ;
}
```



```
for(l = 0 ; l < dim ; l++)
    for(c = 0 ; c < l; c++)
```

## Matrice triangulaire

```
int est_triangulaire_inf(matrice a, int dim)
{
  int l, c ;
  for(l = 0 ; l < dim ; l++)
    for(c = 0 ; c < l; c++)
      if (a[l][c]) return 0 ;
  return 1 ;
}
```

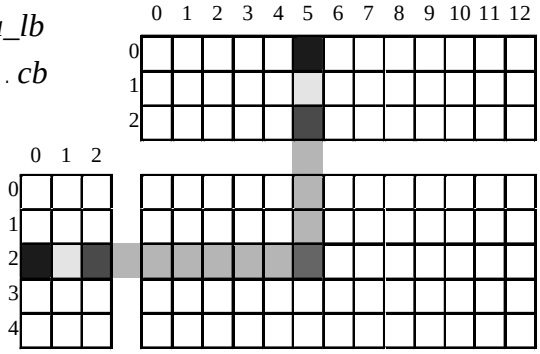


## Produit de matrices

**A** de dimensions *la* . *ca_lb*

**B** de dimensions *ca_lb* . *cb*

$$\mathbf{P}_{lc} = \sum_{k=0}^{k<ca\_lb} \mathbf{A}_{lk}.\mathbf{B}_{kc}$$



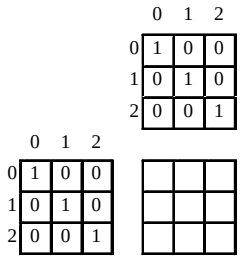

```
p[l][c] = 0 ;
for(k=0 ; k < ca_lb ; k++)
   p[l][c] = p[l][c] + a[l][k] * b[k][c] ;
```

## Produit de matrices

```
void produit(matrice p, matrice a, matrice b,
                    int la, int ca_lb,  int cb)
{
int c, l, k ;
for(l=0 ; l < la ; l++)
  for(c = 0 ; c < cb ; c++)
  {
     p[l][c] = 0 ;
    for(k=0 ; k < ca_lb ; k++)
       p[l][c] = p[l][c] + a[l][k] * b[k][c] ;
  }
} /* produit */
```

## Carré d'une matrice

```
void carre(matrice a, int dim)
{
  produit(a,a,a,dim,dim,dim) ;
}
```




```
void carre(matrice a, int dim)
{
  matrice aux ;
  copier(aux, a, dim, dim) ;
  produit(a, aux, aux, dim, dim, dim) ;
}
```

```
void main(){
  matrice m ;
/* placer la matrice identité dans m */
  carre(m,3) ;
  afficher(m,3,3) ;
}
```

## Résolution d'un système d'équations

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *x* | + | *2y* | + | *3z* | = | *14* | *(a)* |
| *2x* | + | *3y* | + | *z* | = | *11* | *(b)* |
| *3x* | - | *y* | - | *2z* | = | *-5* | *(c)* |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *x* | + | *2y* | + | *3z* | = | *14* | *(a)* |
| | - | *y* | - | *5z* | = | *-17* | *(b') = (b - 2a)* |
| | - | *7y* | - | *11z* | = | *-47* | *(c') = (c - 3a)* |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *x* | + | *2y* | + | *3z* | = | *14* | *(a)* |
| | - | *y* | - | *5z* | = | *-17* | *(b')* |
| | | | | *24z* | = | *72* | *(c' - 7b')* |

## Algorithme de Gauss

Entrée : une matrice M de dimensions l . c (où c > l)

Pour i = 0,1,2,…,l-2 faire

trouver pivot tel que pivot >= i m[i][pivot] !=0

échanger les lignes m[i] et m[pivot]

pour k = i+1 … l-1 faire

remplacer la ligne m[k] par m[k] - (m[k][i]/m[i][i]) * m[i]

## Fonction *gauss*

```
void gauss(matrice m, int l, int c){
  int i, j, k, pivot ;
  double x ;
  for(i =  0 ; i < l -1 ; i++){
    /* déterminer le pivot le plus grand possible */
    pivot = i ;
    for(k = i + 1 ; k <  l ; k++)
      if (fabs(m[k][i]) > fabs(m[pivot][i])
        pivot = k ;
```

```
        if (pivot > i) { /* échanger les lignes m[i] et m[pivot] */
        for(k = i  ; k <  c ; k++){
           x = m[pivot][k] ;
           m[pivot][k] = m[i][k] ;
           m[i][k] = x ;
        }
        for(k = i + 1 ; k < l ; k++){    /* calculer la ligne m[k] */
           x = m[k][i] / m[i][i] ;
           for( j = i  ; j < c ; j++)
              m[k][j] = m[k][j] -  x * m[i][j] ;
        } /* calculer la ligne */
     }/* for i */
  } /* gauss */
```

*Gauss*

# Tableaux & Structures

```
typedef struct {
   double x, y ;
   int poids ;
} point_pondere ;
```

- Calculer le barycentre d'un ensemble de points pondérés ;
- Calculer le point le plus proche d'un autre point ;
- Calculer les deux points les plus proches.

```
void main()
{
   int i = 0 , j = 1 ;
   point_pondere tab[10] ; /* tableau de 10 points pondérés */
   tab[i].x = 1.23 ;
   tab[i].y = 12.4 ;
   tab[i].poids = 101 ;
   tab[j] = tab[i] ;
}
```

## Barycentre d'un ensemble de points pondérés

Entrée : un tableau de points pondérés et sa taille.

Sortie : le barycentre de cet ensemble.

```
point_pondere barycentre(point_pondere pt[], int taille)
{
  int i ;
  point_pondere res = pt[0] ;
  for (i = 1 ; i < taille ; i++)
    res = bary2(res, pt[i]) ;
  return res ;
}
```

```
for (i = 1 ; i < taille ; i++) {
   res.x = res.x + pt[i].poids * pt[i].x ;
   res.y = res.y + pt[i].poids * pt[i].y ;
   res.poids = res.poids + pt[i].poids  ;
}
res.x = res.x /res.poids ;
res.y = res.y /res.poids ;
```

## Point le plus proche d'un autre point

```
double distance(point_pondere a, point_pondere b){
  return sqrt( (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y) ) ;
}

point_pondere plus_proche(point_pondere x,
                          point_pondere pt[], int taille){
  int indice_min = 0 ;
  for (i = 1 ; i < taille ; i++)
    if (distance(x, pt[indice_min]) > distance(x, pt[i]) )
        indice_min = i ;
  return pt[indice_min] ;
}                          /* Optimisation des appels à distance() */
```

## Les deux points les plus proches

```
void les_plus_proches(point_pondere pt[], int taille, point_pondere res[])
{
  int i ;
  point_pondere r ;
  res[0] = pt[0] ; res[1] = pt[1] ;
  for(i = 0 ; i < taille ; i ++){
    r = plus_proche(pt[i], pt, taille) ;
    if (distance(res[0], res[1] ) > distance (pt[i],r)) {
      res[0] = pt[i] ;
      res[1] = r ;
    }
  }
}
```

## Les deux points les plus proches

```
void les_plus_proches(point_pondere pt[], int taille, point_pondere res[])
{
        int i ;
        point_pondere r ;
        res[0] = pt[0] ; res[1] = pt[1] ;
        for(i = taille-1 ; i > 0 ; i--){
                r = plus_proche(pt[i], pt, i ) ;
                if (distance(res[0], res[1] ) > distance (pt[i],r)) {
                        res[0] = pt[i] ;
                        res[1] = r ;
                }
        }
}                               /* complexité : Taille²/2 comparaisons  */
```

## Tris

Entrée : un tableau tab contenant ensemble de n points pondérés

Sortie : un tableau res contenant une permutation des éléments de tab ordonnés par leur poids : res[i] <= res[i+1] .

| Algorithme : | tab (poids) | res (poids) |
|---|---|---|
| | **3** 5 2 6 8 2 | 3 |
| | **3 5** 2 6 8 2 | 3 5 |
| | **3 5 2** 6 8 2 | 2 3 5 |
| | **3 5 2 6** 8 2 | 2 3 5 6 |
| | **3 5 2 6 8** 2 | 2 3 5 6 8 |
| | **3 5 2 6 8 2** | 2 2 3 5 6 8 |

*Ajouter un à un les éléments de* tab *à* res *tout en maintenant* res *ordonné.*

## Ordonner un tableau de points

```
void ordonner_suivant_poids(point_pondere tab[],
                            point_pondere res[], int n){
  int i ;
  for(i=0; i < n ; i++)
    inserer_suivant_poids(tab[i], res, i) ; /* i est la taille de res */
}
```

avec

```
void inserer_suivant_poids(   point_pondere p,
                              point_pondere res[], int n) ;
```

## Insérer un point suivant le poids

Entrées : un point p et un tableau ordonné res de n éléments

Exemple p = 5, n = 6

| res.poids | 1 | 1 | 2 | 3 | 8 | 9 | - |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

• Trouver l'emplacement de p

| | 1 | 1 | 2 | 3 | 8 | 9 | - |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | **4** | 5 | 6 |

• Faire de la place pour p

| | 1 | 1 | 2 | 3 | | 8 | 9 |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | **4** | 5 | 6 |

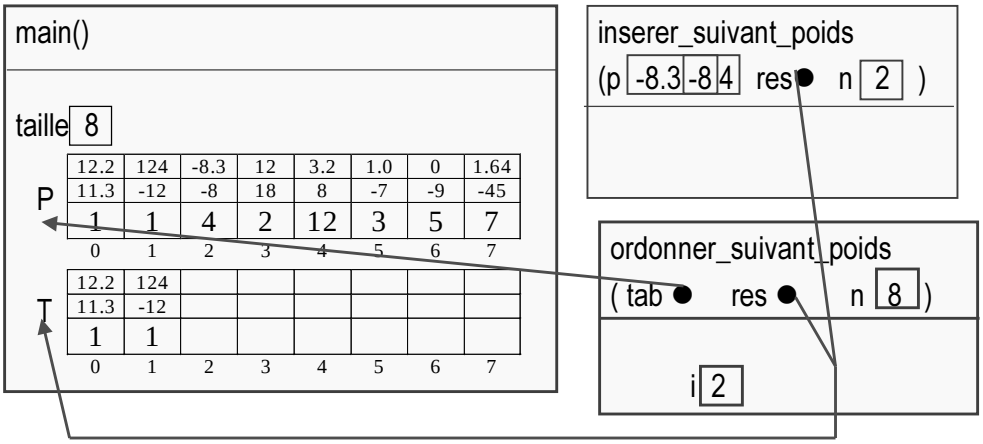
• Recopier p

| | 1 | 1 | 2 | 3 | **5** | 8 | 9 |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | **4** | 5 | 6 |

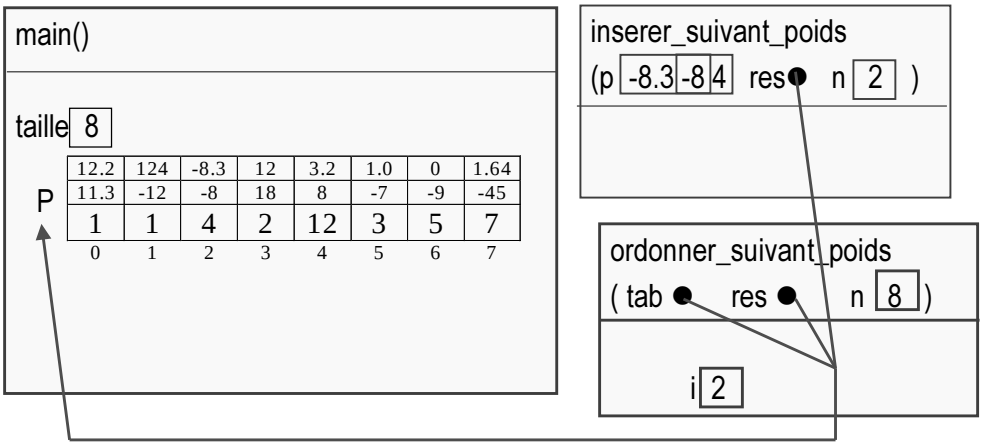
## Insérer un point suivant le poids

```
void inserer_suivant_poids( point_pondere p,
                            point_pondere res[], int n){
  int i = 0, j ;
  while(i < n && p.poids > res[i].poids)     /* trouver la place de p */
    i++ ;
  for(j = n ; j > i  ; j--)                  /* faire de la place pour p */
    t[ j ] = t[ j-1 ] ;
  t[i] = p ;                                 /* recopier p */
}
```

## ordonner _suivant_poids(P,T,8)




## ordonner _suivant_poids(P,P,8)




Il faut vérifier soigneusement le déroulement du programme.

## Améliorer le tri

Offrir la possibilité de trier suivant différentes caractéristiques.

• Passage d'une fonction de comparaison en paramètre.

Diminuer la complexité de la recherche de la position.

• Utiliser une recherche dichotomique

(Diminuer le nombre d'affectation => changer l'algorithme)

```
void ordonner(int superieur(point_pondere , point_pondere),
              point_pondere tab[ ],
              point_pondere res[ ], int n){
  int i ;
  for(i=0; i < n ; i++)
    inserer (superieur, tab[i], res, i) ; /* i est la taille de res */
}
```

## Recherche dichotomique dans un tableau

```
void inserer(  int superieur(point_pondere , point_pondere),
        point_pondere p, point_pondere res[ ], int n){
  int deb = 0 , fin = n-1, i ;

  while(fin > deb) {                          /* trouver la place de p */
    i = (deb + fin) / 2 ;
    if (superieur(p, res[i]) )  deb = i + 1 ;
    else fin = i - 1;
  }

  for(j = n ; j > deb  ; j--)                 /* faire de la place pour p */
    t[ j ] = t[ j-1 ] ;
  t[ deb ] = p ;                              /* recopier p */
}
```

## Comparaisons de points pondérés

```
int est_plus_lourd(point_pondere x, point_pondere y){
  return x.poids > y.poids ;
}

int est_plus_leger(point_pondere x, point_pondere y){
  return x.poids < y.poids ;
}

int est_plus_eloigne(point_pondere x, point_pondere y){
   return distance({0,0,1},x)  > distance({0,0,1}, y)  ;
}
```

## Utilisation des fonctions de comparaison

```
#define MAX 50
void main(){
  point_pondere mes_points[MAX] ,  par_poids[MAX],
                        par_eloignement [MAX] ;
  int taille ;

  /* …. initialisation de mes_points et de taille … */
  ordonner(est_plus_lourd, mes_points, par_poids, taille)  ;

  ordonner(est_plus_eloigne, mes_points, par_eloignement, taille)  ;
  /* … */
}
```

## Représentation des polynômes

$P(x) = 3x^4 + 2.3x^3 + x - 1$

| 0 | 0 | 0 | 3 | 2.3 | 0 | 1 | -1 |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

$$P(x) = \sum_{i=0}^{DEGRE\_MAX} P[i].x^i$$

```
#define DEGRE_MAX 100
typedef double polynome [DEGRE_MAX + 1] ;
polynome P ;
```

Autres représentations

```
typedef struct{
  int degre ;
  double coef[DEGRE_MAX + 1] ;
} poly_deg ;
```

```
typedef struct{
  int degre ;
  double coef ;
} poly_coef [NB_COEF] ;
```

## Calcul du degré

Indice du plus grand coefficient non nul

```
int degre(polynome p)
{
  int d ;
  for(d = DEGRE_MAX ; d > 0 && ! p[d] ; d--) ;
  return d ;
}
```

### Polynôme nul

```
int est_nul(polynome p)
{
  return degre(p) == 0 && p[0] == 0 ;
}
```

## Copier

```
void copier(polynome destination, polynome source)
{
  int i ;
  for(i = 0 ; i <= DEGRE_MAX ; i++)
    destination[i] = source[i] ;
}
```

## Initialiser

```
void RAZ(polynome p)
{
  int i ;
  for(i = 0 ; i <= DEGRE_MAX ; i++)
        p[i] = 0  ;
}
```

## Évaluer un polynôme : méthode de Hörner

$$a_k x^k + a_{k-1} x^{k-1} \ldots + a_2 x^2 + a_1 x + a_0$$
$$=$$
$$\Big(\big(\ldots\big((0 \,.x + a_k)x + a_{k-1}\big)x + a_{k-2}\big)x + \ldots\Big)x + a_1\Big)x + a_0$$

```
double valeur(double x,  polynome p)
{
  double s = 0 ;
  int i ;
  for(i = degre(p) ; i >= 0 ; i--)
    s = s * x + p[i] ;
  return s ;
}
```

## Somme

```
void somme(polynome res, polynome p, polynome q)
{
  int i ;
  for(i = 0 ; i <= DEGRE_MAX ; i++){
    res[i] = p[i] + q[i] ;
}
```

## Produit

$p(x)*q(x)$ où $p(x) = 3x^2 - 2x + 1$ $q(x) = -4x^2 + 5x + 6$

$(3*-4)\ x^4 + (3*5 + -2*-4)\ x^3 + (3*6 + -2*5 + 1*-4)\ x^2 + (-2 * 6 + 1 * 5)\ x + (1*6)$

$$P(x) = \sum p_i.x^i \qquad Q(x) = \sum q_i.x^i$$

$$R(x) = P(x) * Q(x) = \sum r_i.x^i \qquad r_i = \sum_{i=j+k} p_j.q_k$$

## Produit de polynômes

```
void produit(polynome res, polynome p, polynome q)
{
  int i, j ;
  for(i = 0 ; i <= DEGRE_MAX ; i++)
  {
    res[i] = 0 ;
    for(j = 0 ; j <= i ; j++)
      res[i] = res[i] + p[i] * q[i-j] ;
  }
}
```

Complexité : pour calculer un r[i] on fait i + 1 sommes et i + 1 multiplications

=> **DEGRE_MAX² opérations**

*Le nombre d'opérations ne dépend pas du degré des polynômes.*

## Produit de polynômes

```
void produit(polynome res, polynome p, polynome q)
{
  int i, j , deg_p, deg_q ;
  deg_p =  degre(p) ;
  deg_q = degre(q) ;
  RAZ(res) ;
  if (deg_p + deg_q <= DEGRE_MAX)
    for(i = 0 ; i <= deg_p ; i++)
      if(p[i] != 0)
        for(j = i ; j <= deg_q ; j++)
          res[j] = res[j] + p[i] * q[j-i] ;
}
```

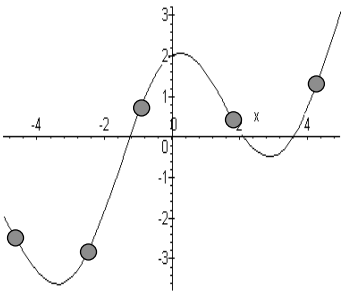
## Polynôme d'Euler

$$E_n(x) = \prod_{i=1}^{i=n} (1 - x^i)$$

```
void Euler(polynome p, int n){
  int i ;
  polynome a, b ;
  RAZ(a) ;
  RAZ(b) ;
  a[0] = b[0] = 1 ;

  for(i=1 ; i <= n ; i++){
    a[i] = -1 ;
    produit(p,a,b) ;
    copier(b,p) ;
    a[i] = 0 ;
    }
  }
```

# Interpolation de Lagrange

Calculer les coefficients
d'un polynôme de degré n
défini par n+1 points $(a_i,b_i)$




Base de Lagrange $L_i(a_j) = \begin{cases} 1 \text{ si } j = i \\ 0 \text{ sinon} \end{cases}$

$$P(x) = \sum_{i=0}^{n} b_i L_i(x)$$

d'où $$L_i(x) = \prod_{i \neq j} \frac{x - a_j}{a_i - a_j}$$

# Base de Lagrange

```
typedef struct {double x,y;} point ;
void base(polynome p, point pt[], int nb_pts, int i)
{
  polynome q, r ;
  int j ;
  double k = 1 ;
  RAZ(p) ;
  RAZ(q) ;
  p[0] = 1 ;
  q[1] = 0 ;

  for(j = 0 ; j < nb_pts ; j++)
    if(i != j){
      q[0] = -pt[j].x ;
      copier(r, p) ;
      produit(p,r,q) ;
      k = k * (pt[i] - pt[j]) ;
    }
  for(j=0 ; j <= nb_pts ; j++)
    p[j] = p[j] / k ;
}
```

## Fonction Lagrange

```
void Lagrange(polynome p, point pt[], int nb_points)
{
  int i, j ;
  polynome L ;
  RAZ(p) ;
  for(i = 0 ; i <= nb_points ; i++){
    base(L, pt, nb_points, i) ;
    for(j = 0 ; j <= nb_points ; j++)
      p[j] = p[j] + pt[i].b * L[j] ;
  }
}
```

## Arithmétique

Calcul du "Plus Grand Commun Diviseur"

$$\text{pgcd}(a,b) = \max(\{k \mid a = 0\ [k] \text{ et } b = 0\ [k]\ \})$$

$$\text{pgcd}(a,a) = a \qquad \text{et} \qquad \text{pgcd}(a+b,a) = \text{pgcd}(b,a)$$

preuve: posons k = pgcd(a+b,a) et k' = pgcd(a,b)

•k divise aussi b :

si a = x.k et a+b = y.k alors b = (y-x).k

•k' divise aussi a+b :

si a = x'.k' et b = y'.k' alors a + b = (x'+y').k'

•par définition on a k >= k' et k' >= k et donc k = k'

## Algorithme d'Euclide

pgcd($a$,$a$) = $a$ et pgcd($a$+$b$,$a$) = pgcd($b$,$a$)

```
int pgcd(int a, int b){ /* a > 0 et b > 0 */
  if (a == b) return a ;
  if (a > b) return pgcd(a-b,b) ;
  return pgcd(b-a,a) ;
}
```

C'est une fonction ***récursive*** (qui s'appelle elle-même).

```
int pgcd(int a, int b){
  while (a != b)
    if (a > b) a = a - b ;
    else b = b - a ;
  return a ;
}
```

## Algorithme d'Euclide

Exemple : pgcd(20,6)

| 20 | 14 | 8 | 2 | 2 | 2 |
|---|---|---|---|---|---|
| 6 | 6 | 6 | 6 | 4 | 2 |

On a en fait pgcd($a$,$b$) = pgcd($a\%b$, $b$) posons $a = k.b + a\%b$

on a pgcd($a$,$b$) = pgcd($x.b + a\%b$, $b$) pour $0 <= x <= k$

```
int pgcd(int a, int b){
  while (a && b)
    if (a > b) a = a % b ;
    else b = b % a ;
  return a + b ;
}
```

```
int pgcd(int a, int b){
  int r ;
  while (b){
    r = a % b ;
    a = b ;
    b = r ;
  }
  return a ;
}
```

## Complexité de l'algorithme d'Euclide

La complexité d'un algorithme est une fonction :

C : taille des données --> nombre d'opérations

3 complexités → minimale (meilleur des cas)
→ en moyenne
→ maximale (pire des cas)

Nombre de divisions dans le pire des cas (pour une taille donnée) :

Fibonacci $F_{k+2} = F_k + F_{k+1}$

pgcd($F_i$ , $F_{i+1}$) demande i étapes de calculs

taille($F_i$) ~ i bits algorithme linéaire

pgcd(a,b) demande moins de log(a+b) étapes

```
while (b){
  r = a % b ;
  a = b ;
  b = r ;
}
```

## Tests de primalité

Un nombre premier possède exactement deux diviseurs 1 et lui-même (*par définition* 1 n'est pas premier).

Il existe un nombre infini de nombre premiers [*Euclide*] :

Supposons que k soit le plus grand nombre premier: k! + 1 = 1 [n] pour tout n premier <= k

Quelque soit $x > 1$, il existe un nombre premier entre $x$ et $2x$. [*Tchebychev*]

Il y a environ n/ln(n) nombres premiers inférieurs à n.

Deux problèmes principaux :

- Décider si un un nombre est premier,
- Décomposer un nombre en ses facteurs premiers.

## Primalité

Écrire une fonction booléenne premier(n) qui retourne vrai ssi n est un nombre premier.

```
int premier(int n){
  int d ;
  if (n < 2) return 0 ;
  for(d = 2 ; d < n ; d++)
    if (n % d == 0) return 0 ;
  return 1 ;
}
```

*n divisions*

```
int premier(int n){
  int d , rac = (long) sqrt((double)n) ;
  if (n < 3) return n == 2 ;
  if (n % 2 == 0) return 0 ;
  for(d = 3 ; d <= rac ; d = d + 2)
    if (n % d == 0) return 0 ;
  return 1 ;
}
```

*racine(n)/2 divisions*

## Primalité (2)

Ne tester ni les multiples de 2 ni ceux de 3.

```
int premier(int n){
  int d , rac = (long) sqrt((double)n) ;
  if (n < 4) return n == 2 || n == 3;
  if (n % 2 == 0 || n % 3 == 0) return 0 ;
  for(d = 6 ; d <= rac ; d = d + 6)
    if (n % (d - 1) == 0 ||  n % (d + 1) == 0 ) return 0 ;
  return 1 ;
}
```

*racine(n)/3 divisions*

éliminer les multiples de 2,3 et 5 : partir de d =30

tester n % (d +/- x) == 0

pour x = 1, 7, 11, 13, 17, 19, 23, 29 [on testera 8 nombres sur 30]

## Test de Fermat

Petit théorème de Fermat :

*si a est premier alors* $m^{a-1} = 1\ [a]$ *pour tout entier m de* ]1,*a*[.

1 234 567 est composite car $2^{1\,234\,566} = 354\,219\ [1\,234\,567]$

```
int puissance(int m, int exp, int mod){
        if (!exp) return 1 ;
        if (exp % 2) /* si exp est impair */
                return (m *  puissance(m,exp - 1, mod)) % mod ;
        return puissance( (m * m) % mod , exp / 2,  mod)) ;
}
```

Applications : tests *probabilistes* de primalité.

On peut dire si un entier est composite sans savoir le factoriser.

## Nombres de Carmichaël

nombre pseudo-premier absolu :

nombres composites *m* tels que $m^{a-1} = 1\ [a]$ *pour tout entier* $m < a$

Exemples : 561  - 1105 - 1729 - 2465 - 2821

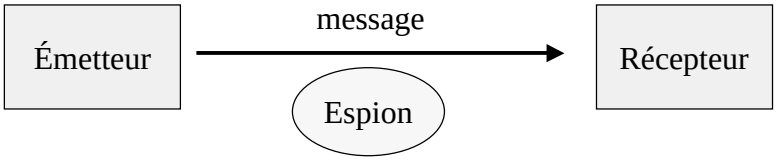
(les nombres de Carmichaël comportent au moins 3 facteurs)

```
int carmichaël (int a){
  int m ;
  for(m = 2;  m < a ; m++)
    if (puissance(m, a-1, a)  != 1) return 0 ;
  return ! premier(m) ;
}
```

## Cryptographie

On utilise un codage pour transmettre des informations confidentielles. Le coût de l'espionnage doit être prohibitif par rapport à la valeur de l'information. => il faut pouvoir changer de code très facilement.

Problème : *le récepteur doit pouvoir décoder.*




L'émetteur et le récepteur doivent convenir du code utilisé :
*Utilisation de clefs secrètes ou d'algorithmes secrets.*

## Cryptographie

Codage RSA [*Rivest - Shamir - Adelman 1979*] à clef publique
*le récepteur donne l'algorithme pour encoder les informations.*

"Réciproque" du théorème de Fermat :

*Soient n =p.q (avec p et q premiers) alors*
*si d.e = 1 [(p-1).(q-1)] alors* $a^{de} = a\ [n]$ *pour tout a*

Codage : remplacer chaque paquet *a* de 8 caractères, par $\boldsymbol{a^e[n]}$
Décodage : remplacer chaque paquet *a* de 8 caractères, par $\boldsymbol{a^d[n]}$

- Trouver deux grands nombres premiers p et q ;
- Choisir un nombre e premier avec (p-1).(q-1) ;
- Calculer d par Bézout. (d.e + v.(p-1).(q-1) = 1)

**Pour calculer d il faut connaître p et q.**

## Développement décimal d’un rationnel

```
76        |   7
 60       |-------
  40      | 10,857142 857142 ...
   50     |
    10    |
     30   |
      20  |
       6  |
```

- Afficher le développement décimal d’un rationnel.
- Calculer sa période.

## Afficher le développement décimal

```
void main(){
  int quotient, reste, num, deno ;
  int i, decim ;
  /* saisir le numérateur, le denominateur et le nombre de decimales */
  …
  /* fin de saisie */
  quotient = num / deno ;
  reste = (num % deno) * 10 ;
  printf("%ld,", quotient) ;
  for(i=0 ; i < decim ; i ++){
    quotient = reste / deno ;
    reste = (reste % deno) * 10;
    printf("%ld", quotient) ;
  }
} /* main */
```

```
R   Q
76, 10,
604 8 5
050 7 1
103 4 2
020 8
60
    .
..
```

## Calcul de la période

```
#define MAX 30000
void main() {
  int reste[MAX], quotient[MAX], num, deno ;
  int i ;
  /* saisir le numérateur, le denominateur et le nombre de decimales */
  …
  /* fin de saisie */
  quotient[0] = num / deno ;
  reste[0] = (num % deno) * 10 ;
  for(i=0 ; indice(reste[i],reste,i) == -1 ; i ++) {
    quotient[i+1] = reste[i] / deno ;
    reste[i+1] = (reste[i] % deno) * 10 ;
  } /* la période commence en indice(reste[i],reste,i) + 1 et se termine en i*/
…
```

| | reste | quotient |
|---|---|---|
| 0 | 60 | 10 |
| 1 | 40 | 8 |
| 2 | 50 | 5 |
| 3 | 10 | 7 |
| 4 | 30 | 1 |
| 5 | 20 | 4 |
| 6 | 60 | 2 |

## Changement de base

Information = sémantique + support

= valeur + représentation

Changer de base = changer de support

Soit $a_k\, a_{k-1}\ldots a_2 a_1 a_0$ la représentation d'un entier $x$ en base A.
Écrire l'entier $x$ en base B.

- Calculer la valeur de $x$ ;
- Calculer la représentation de $x$ en base B.

valeur($a_k\, a_{k-1}\ldots a_2 a_1 a_0$) = $a_k A^k + a_{k-1} A^{k-1} \ldots a_2 A^2 + a_1 A + a_0$

$\Rightarrow$ *évaluation d'un polynôme*

## Représentation -> Valeur (1)

Entrées : un entier base, un tableau support et sa taille.

Sortie: la valeur de support.

```
int valeur(int base, int support[], int taille)
{
  int i, s = 0 ;
  for(i = 0 ; i < taille ; i++)
    s = s + support[i] * puissance(base,i) ;
  return s ;
}
```

De l'ordre de taille.log(taille) multiplications...

## Représentation -> Valeur (2)

On intègre le calcul de la puissance :

```
int valeur(int base, int support[], int taille)
{
  int i, p = 1, s = 0 ;
  for(i = 0 ; i < taille ; i++)
  {
    s = s + support[i] * p ;
    p = p * base ;
  }
  return s ;
}
```

De l'ordre de 2.taille multiplications...

## Représentation -> valeur : méthode de Hörner

$$a_k X^k + a_{k-1} X^{k-1} \ldots + a_2 X^2 + a_1 X + a_0$$

$$=$$

$$\left(\left(\ldots\left(\left(\left(0 + a_k\right)X + a_{k-1}\right)X + a_{k-2}\right)X + \ldots\right)X + a_1\right)X + a_0$$

```
int valeur(int base, int support[], int taille)
{
  int i, s = 0 ;
  for(i = taille -1 ; i >= 0 ; i--)
    s = s * base + support[i] ;
  return s ;
}
```

De l'ordre de taille multiplications.

## Valeur -> Représentation

Calculer la représentation de $1635_{10}$ en base 10…

1635 = (1635 / 10) * 10 + (1635 % 10)
= 163 * 10 + 5
= (16 * 10 + 3)*10 + 5
= (1 * 10 + 6) * 10 + 3)*10 + 5

Calculer la représentation de $135_{10}$ en base 8…

$135_{10}$ = (135 / 8)*8 + (135 % 8)
= 16 * 8 + 7
= (2 * 8 + 0) * 8 + 7
= $207_8$

## Valeur -> Représentation

Entrées : un valeur et une base.

Sortie : un tableau support contenant la représentation de l'entier dans la base convenue.

```
#define MAX 1000            /* on dispose de MAX chiffres au plus */

void representation(int valeur, int base, int support[])
{
  int i ;
  for(i = 0 ; i <= MAX ; i++){
    support[i] = valeur % base ;
    valeur = valeur / base ;
  }
}
```

## Insérer

Insérer la chaîne `" ein"` en position 7 dans `"ich bin berliner"`

1) faire de la place (`"ich bin berlberliner"`)

2) recopier la chaîne (`"ich bin ein berliner"`)

```
void insert(char chaîne[], int pos, char source[])
{
  int lgc, i ;
  lgc = strlen(chaîne) ;
  for(i = strlen(source) ;  i <= pos ; i--) /* faire de la place */
    source[lgc+ i] = source[i] ;
  for(i = 0 ; i < lgc ; i++)       /* recopier la chaîne */
    source[pos+i] = chaîne[i] ;
}
```

## Supprimer une section de chaîne

```
void remove(int debut, int longueur, char source[])
{
  int i ;
  i = debut ;
  while(source[i] = source[i+longueur]) i++ ;
}
  ou encore
  while (source[i+ longueur] != 0)
  {
    source[i] = source[i+ longueur] :
    i++ ;
  }
  source[i] = 0 ;
```

## Remplacer un motif par un autre

```
void replace(char motif[], char par[], char source[]){
  int pos = 0, nb = 0 , lg = strlen(motif) ;
  pos = match(motif,source) ;
  while(pos >= 0) {
    remove(pos,lg,source) ;
    insert(par,pos,source) ;
    pos = match(motif,source) ;
  }
}
```

Problème l'appel : replace("a","aa",chaîne) boucle *infiniment*.
Solution  pos = match(motif,source+pos+strlen(par) ) ;
Démarrer la recherche juste après l'insertion.