

URCA – UFR SCIENCES

# RAPPORT INFO00401

Concepts d'Algorithmie avancée

Luca ALVARO, Antoine THEBAULT  
01/01/2020

## Sommaire

Sommaire .....	1
Algorithmes de base .....	4
Normalisation de données d'un tableau (Dimension 1).....	4
Calcul du maximum de 2 images binaires.....	6
Multiplication de 2 matrices carrées .....	9
Calcul d'un nombre parfait .....	12
Concaténer 2 tableaux.....	13
Suite de Fibonacci avec un Vecteur .....	16
Fibonacci avec des Variables.....	17
Majorité d'un tableau .....	18
Conversion d'un nombre base 10 en nombre binaire .....	20
Algorithmes de Tri.....	22
Tri via un Arbre.....	22
Tri de l'arbre.....	22
Tri à bulle.....	25
Tri par Sélection .....	26
Algorithmes de probabilité .....	28
Simuler un jeu de tennis .....	28
Algorithmes de formalisme de pointeur.....	30
Copier une chaîne de caractère .....	30
Supprimer l'occurrence.....	31
Inverser les éléments d'un tableau.....	34
Palindrome.....	36
Copier un tableau.....	37
Algorithmes récursifs .....	39
Ackerman récursif .....	39
Tri d'un tableau par pivot (tri rapide) récursif .....	40
Algorithmes de Pile .....	42
Définition des structures.....	42
Créer une Pile.....	42
Empiler une Pile .....	43
Dépiler une Pile.....	44
Sommet d'une pile .....	44

Déterminer si une pile est vide .....	45
Vider une Pile .....	45
Afficher une Pile .....	46
Egalité de deux Piles .....	47
Trier une Pile .....	48
Ackerman avec Pile (Iteratif) .....	50
Algorithmes de Liste .....	53
Définition des structures .....	53
Créer une liste .....	53
La liste est-elle vide ? .....	54
Afficher la liste .....	54
Chercher un élément dans la liste .....	55
Ajouter en dernière position .....	56
Ajouter la valeur à la nième position .....	58
Retirer la valeur à la dernière position .....	60
Retirer la valeur à la position n .....	62
Retirer la valeur (première occurrence) .....	63
Afficher Liste .....	64
La liste est vide ? .....	65
Vider la liste .....	66
Egalité de deux listes .....	66
Construire une liste binaire à partir d'un tableau .....	67
Incrémentation de 1 une liste binaire .....	68
Trier une liste .....	70
Fusionner deux listes triées .....	71
Extraire une chaine de la liste .....	72
Algorithmes d'Arbre .....	74
Définition des structures .....	74
Créer un Arbre .....	74
L'arbre est-il vide ? .....	75
Choix du parcours pour l'affichage .....	75
Parcours Prefixé .....	76
Parcours Infiné .....	76
Parcours Suffixe .....	77
Calcul de la hauteur .....	77
Parcours pour le calcul de la hauteur .....	78

L'arbre est-il équilibré ? .....	78
Deux Arbres sont-ils égaux ? .....	79
Parcours pour l'égalité .....	79
Parcours en largeur .....	80
Vida(n)ge de l'Arbre .....	83
Vidage (récursif) .....	83
Ajout Logique .....	84
Ajout en Largeur .....	85
Equilibrage de l'arbre .....	88
Algorithmes de Huffman .....	92
Définition des structures .....	92
Recherche du maximum .....	92
Transformation de la table de priorité .....	93
Décodage de Huffman .....	95
Encodage de Huffman .....	97
Algorithmes d'Arbre R&N .....	100
Vérifier si un arbre R&N respecte les règles .....	100

**Ci-joint :**

- L'ensemble des codes des Algorithmes en C
- Un lanceur avec exemples
- Deux versions de ce dossier (PDF/DOCX)

Tous les algorithmes sont accessibles via le lien suivant (GitHub du projet)

***<https://github.com/Angom8/INFO0401>***

## Algorithmes de base

### Normalisation de données d'un tableau (Dimension 1)

Soit un tableau d'entier  $T_1$ , on désire exploiter ses données pour une autre réalisation, pour cela nous aurons besoin de normaliser les données de ce tableau sur l'intervalle  $[a, b]$ .

$T_1$	6	1	19	4	8	2	17
-------	---	---	----	---	---	---	----

Comme la plus grande valeur c'est 19 dans  $T_1$  alors on doit diviser l'ensemble des valeurs par 19.

**Algorithme :** Normalisation

**Données :**      \*  $t_1$  : tableau d'entiers;  
                  \*  $t_2$  : tableau de réels;  
                   $i, n, a, b$  : entiers;  
                   $max$  : réel;

**Début**

// Lecture de la taille

lire(n);

// Lecture de l'intervalle  $[a, b]$

lire(a);

lire(b);

// Permute a et b si besoins

**Si**  $a > b$  **Alors**

$a \leftarrow a + b$ ;

$b \leftarrow a - b$ ;

$a \leftarrow a - b$ ;

**FinSi**

// Lecture de  $t_1$

$t_1 \leftarrow \text{allouer}(\text{taille}(\text{entier}) * n)$ ;

**Pour**  $i$  allant de 1 à  $n$  **Faire**

    lire(  $t_1(i)$  );

**FinPour**

// Affichage de  $t_1$

**Pour**  $i$  allant de 1 à  $n$  **Faire**

    afficher( $t_1(i)$ );

**FinPour**

// Recherche de la valeur maximale

$max \leftarrow t_1(1)$ ;

**Pour**  $i$  allant de 1 à  $n$  **Faire**

**Si**  $t_1[i] > max$  **Alors**

$max \leftarrow t_1(i)$ ;

**FinSi**

**FinPour**

afficher(max);

$t_2 \leftarrow \text{allouer}(\text{taille}(\text{réel}) * n);$

// Normalisation

**Pour** i allant de 1 à n **Faire**

**Si** a < 0 **Alors**

$t_2[i] \leftarrow (t_1[i] \text{ div } \text{max}) * (b + \text{absolue}(a)) + a;$

**Sinon**

$t_2[i] \leftarrow (t_1[i] \text{ div } \text{max}) * (b - a) + a;$

**FinSi**

    afficher( $t_2[i]$ );

**FinPour**

// Nettoyage de la mémoire

liberer( $t_1$ );

liberer( $t_2$ );

**Fin**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      //Declarations
6      int * t1;
7      float * t2;
8      int i, n, a, b;
9      float max;
10
11     //Lecture de la taille
12     do{
13         printf("Saisir une taille : ");
14         scanf("%d",&n);
15     }while(n<1);
16
17     //Lecture de l'intervalle [a, b]
18     printf("Saisir l'intervalle a : ");
19     scanf("%d",&a);
20     printf("Saisir l'intervalle b : ");
21     scanf("%d",&b);
22
23     //Permute a et b si besoins
24     if(a>b){
25         a += b;
26         b = a-b;
27         a = a-b;
28     }
29
30     //Lecture de t1
31     t1 = (int *)calloc(n, sizeof(int));
32
33     for(i=0;i<n;i++){
34         printf("Saisir la case t1[%d] : ", i);
35         scanf("%d",t1+i);
36     }
37
38     //Affichage de t1
39     for(i=0;i<n;i++){
40         printf("%d ", t1[i]);
41     }
42     printf("\n");
43
44     //Recherche de la valeur maximale du tableau
45     max = t1[0];
46     for(i=0;i<n;i++){
47         if(t1[i]>max){
48             max = t1[i];
49         }
50     }
51     printf("Max = %.0f \n", max);
52
53     t2 = (float *)calloc(n, sizeof(float));
54
55     //Normalisation
56     printf("Resultat : \n");
57     for(i=0;i<n;i++){
58         if(a < 0){
59             t2[i] = (t1[i]/max)*(b+abs(a)) + a;
60         }else{
61             t2[i] = (t1[i]/max)*(b-a) + a;
62         }
63         printf("%.2f ", t2[i]);
64     }
65     printf("\n");
66
67     //Nettoyage de la mémoire
68     free(t1);
69     free(t2);
70
71     exit(0);
72 }
```

Exemple normalisation entre [0, 1] :

```
Saisir une taille : 7
Saisir l'intervalle a : 0
Saisir l'intervalle b : 1
Saisir la case t1[0] : 6
Saisir la case t1[1] : 1
Saisir la case t1[2] : 19
Saisir la case t1[3] : 4
Saisir la case t1[4] : 8
Saisir la case t1[5] : 2
Saisir la case t1[6] : 17
6 1 19 4 8 2 17
Max = 19
Resultat :
0.32 0.05 1.00 0.21 0.42 0.11 0.89
```

### Complexité :

Affichage =  $2n+1$

Affectation = Si  $a > b$  :  $n+6$  Sinon :  $n+3$

Addition = Si  $a > b$  :  $n+1$  Sinon :  $n$

## Calcul du maximum de 2 images binaires

0	0	1
1	0	0
1	0	1

A

ou

0	0	1
1	0	0
1	0	1

B

= ?

**Algorithme :** CalculMaxMatricesBinaires

**Données :** \*\*  $M_A$ , \*\*  $M_B$ , \*\*  $M_C$  : matrice d'entiers;  
j, i, n : entiers;

**Début**

// Lecture de la taille  
lire(n);

// Allocation dynamique  
 $M_A \leftarrow \text{allouer}(\text{taille}(*\text{entier}) * n);$   
 $M_B \leftarrow \text{allouer}(\text{taille}(*\text{entier}) * n);$   
 $M_C \leftarrow \text{allouer}(\text{taille}(*\text{entier}) * n);$

**Pour** i allant de 1 à n **Faire**

$M_A(i) \leftarrow \text{allouer}(\text{taille}(\text{entier}) * n);$   
 $M_B(i) \leftarrow \text{allouer}(\text{taille}(\text{entier}) * n);$   
 $M_C(i) \leftarrow \text{allouer}(\text{taille}(\text{entier}) * n);$

**FinPour**

// Lecture de la matrice A

**Pour** i allant de 1 à n **Faire**

**Pour** j allant de 1 à n **Faire**  
lire( $M_A(i, j)$ );

```

    FinPour
FinPour

// Lecture de la matrice B
Pour i allant de 1 à n Faire
    Pour j allant de 1 à n Faire
        lire(MB(i, j));
    FinPour
FinPour

// Affichage de la matrice A
Pour i allant de 1 à n Faire
    Pour j allant de 1 à n Faire
        afficher(MA(i, j));
    FinPour
FinPour

// Affichage de la matrice B
Pour i allant de 1 à n Faire
    Pour j allant de 1 à n Faire
        afficher(MB(i, j));
    FinPour
FinPour

// Calcul du max de 2 images binaires
Pour i allant de 1 à n Faire
    Pour j allant de 1 à n Faire
        Si MA(i, j) > MB(i, j) Alors
            MC(i, j) ← MA(i, j);
        Sinon
            MC(i, j) ← MB(i, j);
        FinSi
    FinPour
FinPour

// Affichage de la matrice C, le résultat
Pour i allant de 1 à n Faire
    Pour j allant de 1 à n Faire
        afficher(MC(i, j));
    FinPour
FinPour

// Nettoyage de la mémoire
Pour j allant de 1 à n Faire
    liberer(MA(i));
    liberer(MB(i));
    liberer(MC(i));
FinPour
liberer(MA);
```



liberer(M<sub>B</sub>);  
liberer(M<sub>C</sub>);

Fin

```
4  int main(){
5      //Declarations
6      int ** Ma, ** Mb, ** Mc;
7      int i, j, n;
8
9      //Lecture de la taille
10     do{
11         printf("Saisir une taille : ");
12         scanf("%d",&n);
13     }while(n<1);
14
15     //Allocation dynamique
16     Ma = (int **)calloc(n, sizeof(int *));
17     Mb = (int **)calloc(n, sizeof(int *));
18     Mc = (int **)calloc(n, sizeof(int *));
19
20     for(i=0;i<n;i++){
21         Ma[i] = (int *)calloc(n, sizeof(int));
22         Mb[i] = (int *)calloc(n, sizeof(int));
23         Mc[i] = (int *)calloc(n, sizeof(int));
24     }
25
26     //Lecture de Ma
27     for(i=0;i<n;i++){
28         for(j=0;j<n;j++){
29             do{
30                 printf("Saisir la case Ma[%d, %d] : ", i, j);
31                 scanf("%d",&Ma[i][j]);
32             }while(Ma[i][j] != 1 && Ma[i][j] != 0);
33         }
34     }
35
36     //Lecture de Mb
37     for(i=0;i<n;i++){
38         for(j=0;j<n;j++){
39             do{
40                 printf("Saisir la case Mb[%d, %d] : ", i, j);
41                 scanf("%d",&Mb[i][j]);
42             }while(Mb[i][j] != 1 && Mb[i][j] != 0);
43         }
44     }
45
46     //Affichage de Ma
47     for(i=0;i<n;i++){
48         for(j=0;j<n;j++){
49             printf("%d ", Ma[i][j]);
50
51             printf("\n");
52         }
53         printf("\n");
54
55         //Affichage de Mb
56         for(i=0;i<n;i++){
57             for(j=0;j<n;j++){
58                 printf("%d ", Mb[i][j]);
59             }
60             printf("\n");
61         }
62         printf("\n");
63
64         //Calcul du max de 2 img binaires
65         for(i=0;i<n;i++){
66             for(j=0;j<n;j++){
67                 if(Ma[i][j] > Mb[i][j]){
68                     Mc[i][j] = Ma[i][j];
69                 }else{
70                     Mc[i][j] = Mb[i][j];
71                 }
72             }
73         }
74
75         //Affichage de Mc
76         for(i=0;i<n;i++){
77             for(j=0;j<n;j++){
78                 printf("%d ", Mc[i][j]);
79             }
80             printf("\n");
81         }
82         printf("\n");
83
84         //Nettoyage de la mémoire
85         for(i=0;i<n;i++){
86             free(Ma[i]);
87             free(Mb[i]);
88             free(Mc[i]);
89         }
90         free(Ma);
91         free(Mb);
92         free(Mc);
93
94         exit(0);
95     }
```

Exemple de calcul du max de 2 matrices binaires :

```
Saisir une taille : 3
Saisir la case Ma[0, 0] : 0
Saisir la case Ma[0, 1] : 0
Saisir la case Ma[0, 2] : 1
Saisir la case Ma[1, 0] : 1
Saisir la case Ma[1, 1] : 0
Saisir la case Ma[1, 2] : 0
Saisir la case Ma[2, 0] : 1
Saisir la case Ma[2, 1] : 0
Saisir la case Ma[2, 2] : 1
Saisir la case Mb[0, 0] : 1
Saisir la case Mb[0, 1] : 0
Saisir la case Mb[0, 2] : 0
Saisir la case Mb[1, 0] : 0
Saisir la case Mb[1, 1] : 1
Saisir la case Mb[1, 2] : 1
Saisir la case Mb[2, 0] : 1
Saisir la case Mb[2, 1] : 0
Saisir la case Mb[2, 2] : 0

Ma :
0 0 1
1 0 0
1 0 1

Mb :
1 0 0
0 1 1
1 0 0

Resultat Mc :
1 0 1
1 1 1
1 0 1
```

**Complexité :**  
Affichage =  $3n^2$   
Affectation =  $n^2+3n+3$   
Lecture =  $2n^2+1$

## Multiplication de 2 matrices carrées

**Algorithme :** ProduitMatriciel

**Données :**     \*\* Ma, \*\* Mb, \*\* Mc : matrice d'entiers;  
                  i, j, k, n : entiers;

**Début**

// Lecture de la taille

lire(n) ;

// Allocation dynamique

$M_A \leftarrow \text{allouer}(\text{taille}(* \text{entier}) * n);$

$M_B \leftarrow \text{allouer}(\text{taille}(* \text{entier}) * n);$

$M_C \leftarrow \text{allouer}(\text{taille}(* \text{entier}) * n);$

**Pour** i allant de 1 à n **Faire**

$M_A(i) \leftarrow \text{allouer}(\text{taille}(\text{entier}) * n);$

$M_B(i) \leftarrow \text{allouer}(\text{taille}(\text{entier}) * n);$

$M_C(i) \leftarrow \text{allouer}(\text{taille}(\text{entier}) * n);$

**FinPour**

// Lecture de la matrice A. On admet que lire() gère la lecture de matrices comme fait précédemment

lire( $M_A$ ) ;

// Lecture de la matrice B

lire( $M_B$ ) ;

// Affichage de la matrice A. On admet que afficher() gère l'affichage de matrices comme fait précédemment

afficher( $M_A$ ) ;

// Affichage de la matrice B

afficher( $M_B$ ) ;

// Multiplication de la matrice A et B. On fait la somme des  $a(i, k) * b(k, j)$

**Pour** i allant de 1 à n **Faire**

**Pour** j allant de 1 à n **Faire**

$M_C(i, j) \leftarrow 0 ;$

**Pour** k allant de 1 à n **Faire**

$M_C(i, j) \leftarrow M_C(i, j) + (M_A(i, k) * M_B(k, j)) ;$

**FinPour**

**FinPour**

**FinPour**

```
// Affichage de la matrice C, le résultat  
afficher(MC) ;
```

```
// Nettoyage de la mémoire
```

```
Pour j allant de 1 à n Faire
```

```
    liberer(MA(i));
```

```
    liberer(MB(i));
```

```
    liberer(MC(i));
```

```
FinPour
```

```
liberer(MA);
```

```
liberer(MB);
```

```
liberer(MC);
```

**Fin**

```
4  int main(){
5      //Declarations
6      int ** Ma, ** Mb, ** Mc;
7      int i, j, k, n;
8
9      //Lecture de la taille
10     do{
11         printf("Saisir une taille : ");
12         scanf("%d",&n);
13     }while(n<1);
14
15     //Allocation dynamique
16     Ma = (int **)calloc(n, sizeof(int *));
17     Mb = (int **)calloc(n, sizeof(int *));
18     Mc = (int **)calloc(n, sizeof(int *));
19
20     for(i=0;i<n;i++){
21         Ma[i] = (int *)calloc(n, sizeof(int));
22         Mb[i] = (int *)calloc(n, sizeof(int));
23         Mc[i] = (int *)calloc(n, sizeof(int));
24     }
25
26     //Lecture de Ma
27     for(i=0;i<n;i++){
28         for(j=0;j<n;j++){
29             printf("Saisir la case Ma[%d, %d] : ", i, j);
30             scanf("%d",&Ma[i][j]);
31         }
32     }
33
34     //Lecture de Mb
35     for(i=0;i<n;i++){
36         for(j=0;j<n;j++){
37             printf("Saisir la case Mb[%d, %d] : ", i, j);
38             scanf("%d",&Mb[i][j]);
39         }
40     }
41
42     //Affichage de Ma
43     for(i=0;i<n;i++){
44         for(j=0;j<n;j++){
45             printf("%d ", Ma[i][j]);
46         }
47         printf("\n");
48     }
49     printf("\n");
51     printf("\n");
52
53     //Affichage de Mb
54     printf("Mb : \n");
55     for(i=0;i<n;i++){
56         for(j=0;j<n;j++){
57             printf("%d ", Mb[i][j]);
58         }
59         printf("\n");
60     }
61     printf("\n");
62
63     //Initialisation de la matrice C en 0
64     for(i=0;i<n;i++){
65         for(j=0;j<n;j++){
66             Mc[i][j] = 0;
67         }
68     }
69
70     //Multiplication de la matrice A et B
71     for(i=0;i<n;i++){
72         for(j=0;j<n;j++){
73             for(k=0;k<n;k++){
74                 Mc[i][j] += Ma[i][k]*Mb[k][j];
75             }
76         }
77     }
78
79     //Affichage de Mc
80     printf("Resultat Mc : \n");
81     for(i=0;i<n;i++){
82         for(j=0;j<n;j++){
83             printf("%d ", Mc[i][j]);
84         }
85         printf("\n");
86     }
87     printf("\n");
88
89     //Nettoyage de la mémoire
90     for(i=0;i<n;i++){
91         free(Ma[i]);
92         free(Mb[i]);
93         free(Mc[i]);
94     }
95     free(Ma);
96     free(Mb);
97     free(Mc);
98
99     exit(0);
100 }
```

**Exemple de multiplication de 2 matrices :**

```
Saisir une taille : 3
Saisir la case Ma[0, 0] : 4
Saisir la case Ma[0, 1] : 5
Saisir la case Ma[0, 2] : 2
Saisir la case Ma[1, 0] : 3
Saisir la case Ma[1, 1] : 6
Saisir la case Ma[1, 2] : 1
Saisir la case Ma[2, 0] : 4
Saisir la case Ma[2, 1] : 2
Saisir la case Ma[2, 2] : 3
Saisir la case Mb[0, 0] : 5
Saisir la case Mb[0, 1] : 2
Saisir la case Mb[0, 2] : 1
Saisir la case Mb[1, 0] : 2
Saisir la case Mb[1, 1] : 3
Saisir la case Mb[1, 2] : 5
Saisir la case Mb[2, 0] : 4
Saisir la case Mb[2, 1] : 2
Saisir la case Mb[2, 2] : 3

Ma :
4 5 2
3 6 1
4 2 3

Mb :
5 2 1
2 3 5
4 2 3

Resultat Mc :
38 27 35
31 26 36
36 20 23
```

**Complexité :**

Affichage =  $3n^2$

Affectation =  $n^3 + n^2 + 3n + 3$

Lecture =  $2n^2 + 1$

## Calcul d'un nombre parfait

**Algorithme :** estParfait

**Données :** i, nb, tmp : entiers

**Début**

// Lecture du nombre

lire(nb) ;

// Initialisation

tmp ← 0 ;

// Calcul d'un nombre parfait

**Pour** i allant de 1 à nb div 2 **Faire**

**Si** nb%i = 0 **Alors**

        tmp ← tmp + i ;

**FinSi**

**FinPour**

// Vérifier si le nombre est parfait

**Si** nb = tmp **Alors**

    afficher("Le nombre n est parfait") ;

**Sinon**

    afficher("Le nombre n n'est pas parfait") ;

**FinSi**

**Fin**

```
4  int main(){
5      //Declarations
6      int i, nb, tmp;
7
8      //Lecture de la taille
9      printf("Saisir un nombre : ");
10     scanf("%d",&nb);
11
12     //Initialisation
13     tmp = 0;
14
15     //Calcul d'un nombre parfait
16     for(i=1;i<=(nb/2);i++){
17         if((nb%i) == 0){
18             tmp += i;
19         }
20     }
21     printf("\n");
22
23     //Vérification si le nombre est parfait
24     if(nb == tmp){
25         printf("Le nombre %d est parfait \n", nb);
26     }else{
27         printf("Le nombre %d n'est pas parfait \n", nb);
28     }
29 }
```

Exemple de calcul de nombre parfait :

Saisir un nombre : 28	Saisir un nombre : 30
Le nombre 28 est parfait	Le nombre 30 n'est pas parfait

**Complexité :**

Affichage = 1

Affectation =  $k+1$  (k le nb de fois ou  $nb\%i == 0$ )

Lecture = 1

## Concaténer 2 tableaux

A    

1	4	6	8
---	---	---	---

    B    

2	3	9	10
---	---	---	----

=    C    

1	2	3	4	6	8	9	10
---	---	---	---	---	---	---	----

Les Tableaux sont déjà triés et la concaténation doit respecter l'ordre des tableaux d'origine.

**Algorithme :** ConcatenationTableau

**Données :**    i, j,  $n_1$ ,  $n_2$  : entiers  
                  \* a, \* b, \* c : tableaux d'entiers

**Début**

// Lecture de la taille

lire( $n_1$ ) ;

lire( $n_2$ ) ;

// Allocation dynamique

a ← allouer(taille(\* entier)\* $n_1$ );

b ← allouer(taille(\* entier)\* $n_2$ );

c ← allouer(taille(\* entier)\*( $n_1+n_2$ ));

```
// Lecture du tableau A
Pour i allant de 1 à  $n_1$  Faire
    Lire(a(i));
FinPour

// Lecture du tableau B
Pour i allant de 1 à  $n_2$  Faire
    Lire(b(i));
FinPour

// Affichage du tableau A
afficher(a) ;
// Affichage du tableau B
afficher(b) ;

// Concatener A et B
i  $\leftarrow$  0;
j  $\leftarrow$  0;

TantQue i <  $n_1$  OU j <  $n_2$  Faire
    Si i  $\geq$   $n_1$  Alors
        c(i+j)  $\leftarrow$  b(j);
        j  $\leftarrow$  j+1;
    Sinon Si j  $\geq$   $n_2$  Alors
        c(i+j)  $\leftarrow$  a(i);
        i  $\leftarrow$  i+1;
    Sinon Si a(i) < b(j) Alors
        c(i+j)  $\leftarrow$  a(i);
        i  $\leftarrow$  i+1;
    Sinon
        c(i+j)  $\leftarrow$  b(j);
        j  $\leftarrow$  j+1;
    Finsi
    FinSi
    FinSi
FinTantQue

// Affichage du tableau C
afficher(c) ;

// Nettoyage de la mémoire
liberer(a);
liberer(b);
liberer(c);

Fin
```

```

4  int main(){
5      //Declarations
6      int i, j, n1, n2;
7      int * a, * b, * c;
8
9      //Lecture de la taille
10     do{
11         printf("Saisir la taille du premier tableau : ");
12         scanf("%d",&n1);
13     }while(n1 < 1);
14
15     do{
16         printf("Saisir la taille du deuxieme tableau : ");
17         scanf("%d",&n2);
18     }while(n2 < 1);
19
20     //Initialisation
21     a = (int *)calloc(n1, sizeof(int));
22     b = (int *)calloc(n2, sizeof(int));
23     c = (int *)calloc(n1+n2, sizeof(int));
24
25     //Lecture du tableau A
26     printf("Saisir a[0] : ");
27     scanf("%d",&a[0]);
28     for(i=1;i<n1;i++){
29         do{
30             printf("Saisir a[%d] : ",i);
31             scanf("%d",&a[i]);
32             }while(a[i-1] > a[i]);
33     }
34     printf("\n");
35
36     //Lecture du tableau B
37     printf("Saisir b[0] : ");
38     scanf("%d",&b[0]);
39     for(i=1;i<n2;i++){
40         do{
41             printf("Saisir b[%d] : ", i);
42             scanf("%d",&b[i]);
43             }while(b[i-1] > b[i]);
44     }
45     printf("\n");
46
47     //Affichage du tableau A
48     for(i=0;i<n1;i++){
49         printf("%d ",a[i]);
50     }
51     printf("\n");
52
53     //Affichage du tableau B
54     for(i=0;i<n2;i++){
55         printf("%d ",b[i]);
56     }
57     printf("\n");
58
59     //Concatener
60     i=0;
61     j=0;
62     while(i<n1 || j<n2){
63         if(i>=n1){
64             c[i+j] = b[j];
65             j++;
66         }else if(j>=n2){
67             c[i+j] = a[i];
68             i++;
69         }else if(a[i] < b[j]){
70             c[i+j] = a[i];
71             i++;
72         }else{
73             c[i+j] = b[j];
74             j++;
75         }
76     }
77
78     //Affichage du tableau C
79     for(i=0;i<(n1+n2);i++){
80         printf("%d ",c[i]);
81     }
82     printf("\n");
83
84     //Nettoyage de la mémoire
85     free(a);
86     free(b);
87     free(c);
88
89     exit(0);
90 }

```

### Exemple concatenation de 2 tableaux :

<pre> Saisir la taille du premier tableau : 3 Saisir la taille du deuxieme tableau : 2 Saisir a[0] : 1 Saisir a[1] : 2 Saisir a[2] : 5  Saisir b[0] : 4 Saisir b[1] : 7 </pre>	<pre> A : 1 2 5 B : 4 7 Resultat C : 1 2 4 5 7 </pre>
--	---

### Complexité :

Affichage =  $2(n_1+n_2)$

Affectation =  $2(n_1+n_2) + 2$

Lecture =  $n_1+n_2+2$



## Suite de Fibonacci avec un Vecteur

**Algorithme :** VecteurFibonacci

**Données :** i, n : entiers;  
\* tab : tableau d'entiers;

**Début**

// Lecture du nombre d'opération  
lire(n);

// Initialisation  
tab  $\leftarrow$  allouer(taille(entier)\*n);  
tab(0)  $\leftarrow$  0;  
tab(1)  $\leftarrow$  1;

// Calcul de la suite de Fibonacci  
**Pour** i allant de 2 à n **Faire**  
    tab(i)  $\leftarrow$  tab(i-1) + tab(i-2);  
**FinPour**

// Afficher la suite de Fibonacci  
**Pour** i allant de 1 à n **Faire**  
    afficher(tab(i));  
**FinPour**

**Fin**

```
5  int main(){
6      //Declarations
7      int n, i;
8      long int * tab;
9
10     //Lecture du nombre d'opération
11     do{
12         printf("Saisir le nombre d'opération supp à 2: ");
13         scanf("%d",&n);
14     }while(n < 2);
15
16     //Initialisation
17     tab = (long int *)calloc(n, sizeof(long int));
18     tab[0] = 0;
19     tab[1] = 1;
20
21     //Calcul de la suite de Fibonacci
22
23     for(i=2;i<n;i++){
24         tab[i] = tab[i-1]+tab[i-2];
25     }
26
27     //Affiche la suite de Fibonacci
28     for(i=0;i<n;i++){
29         printf("%d ", tab[i]);
30     }
31     printf("\n");
32
33     free(tab);
34
35     return 0;
36 }
```

**Exemple calcul de la suite de Fibonacci avec un vecteur :**

```
Saisir le nombre d'opération (supp à 2): 20  
Suite de Fibonacci :  
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

**Complexité :**

Affichage = n

Affectation = n+2

Addition = n-1

## Fibonacci avec des Variables

**Algorithme :**

**Données :** i, n, a, b, c : entiers;

**Début**

// Lecture du nombre d'opération

lire(n) ;

// Initialisation

a  $\leftarrow$  0 ;

afficher(a);

b  $\leftarrow$  1;

afficher(b);

// Calcul de la suite de Fibonacci et affichage

**Pour** i allant de 2 à n **Faire**

c  $\leftarrow$  a+b;

afficher(c);

a  $\leftarrow$  b;

b  $\leftarrow$  c;

**FinPour**

**Fin**

```
5  int main(){
6      //Declarations
7      int n, i;
8      long int a, b, c;
9
10     //Lecture du nombre d'opération
11     do{
12         printf("Saisir le nombre d'opération supp à 2: ");
13         scanf("%d",&n);
14     }while(n < 2);
15
16     //Initialisation
17     a = 0;
18     printf("%d ", a);
19     b = 1;
20     printf("%d ", b);
21
22     //Calcul de la suite de Fibonacci et affichage
23     for(i=2;i<n;i++){
24         c = a+b;
25         printf("%d ", c);
26         a = b;
27         b = c;
28     }
29     printf("\n");
30
31     exit(0);
32 }
```

Exemple calcul de la suite de Fibonacci avec des variables :

Saisir le nombre d'opération supp à 2: 23

Suite de Fibonacci :

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711

**Complexité :**

Affichage =  $n + 1$

Affectation =  $3n-1$

Addition =  $n-1$

## Majorité d'un tableau

**Algorithme :** majoritéTableau

**Données :** i, n, nb, compteur, tmp1, tmp2 : entiers;  
\* tab : tableau d'entiers;

**Début**

// Lecture de la taille du tableau  
lire(n);

// Initialisation  
tab ← allouer(taille(entier)\*n);

// Lecture du tableau déjà trié  
**Pour** i allant de 1 à n **Faire**  
    lire(tab(i));

**FinPour**

// Affichage du tableau

```
Pour i allant de 1 à n Faire  
    afficher(tab(i));  
FinPour  
  
// Calcul du majoré  
tmp2 ← tab(0);  
compteur ← 0;  
tmp1 ← 1;  
Pour i allant de 2 à n Faire  
    Si tmp2 != tab(i) Alors  
        Si tmp1 > compteur Alors  
            compteur ← tmp1;  
            nb ← tmp2;  
        FinSi  
        tmp1 ← 1;  
        tmp2 ← tab(i) ;  
    Sinon  
        tmp1 ← tmp1+1;  
    FinSi  
FinPour  
  
// Affichage du résultat  
afficher(nb, compteur);  
  
// Nettoyage de la mémoire  
liberer(tab);
```

**Fin**

```
5  int main(){  
6      //Declarations  
7      int n, i, nb, compteur, tmp1, tmp2;  
8      int * tab;  
9  
10     //Lecture de la taille du tableau  
11     do{  
12         printf("Saisir la taille du tableau : ");  
13         scanf("%d",&n);  
14     }while(n < 1);  
15  
16     //Initialisation  
17     tab = (int *)calloc(n, sizeof(int));  
18  
19     //Lecture du tableau  
20     printf("Saisir tab[0] : ");  
21     scanf("%d",&tab[0]);  
22     for(i=1;i<n;i++){  
23         do{  
24             printf("Saisir tab[%d] : ",i);  
25             scanf("%d",&tab[i]);  
26             }while(tab[i-1] > tab[i]);  
27     }  
28     printf("\n");  
29  
30     //Affiche le tableau  
31     for(i=0;i<n;i++){  
32         printf("%d ", tab[i]);  
33     }  
34  
35     printf("\n");  
36  
37     //Calcul du majoré  
38     tmp2 = tab[0];  
39     compteur = 0;  
40     tmp1 = 1;  
41     for(i=1;i<n;i++){  
42         if(tmp2 != tab[i]){  
43             if(tmp1 > compteur){  
44                 compteur = tmp1;  
45                 nb = tmp2;  
46             }  
47             tmp1 = 1;  
48             tmp2 = tab[i];  
49         }else{  
50             tmp1++;  
51         }  
52     }  
53  
54     //Affichage du résultat  
55     printf("Le majoté est %d qui est compté %d fois \n", nb, compteur);  
56  
57     //Nettoyage de la mémoire  
58     free(tab);  
59  
60     exit(0);  
61 }
```

**Exemple calcul du majoré :**

```
Saisir la taille du tableau : 10
Saisir tab[0] (ordre croissant) : 1
Saisir tab[1] (ordre croissant) : 4
Saisir tab[2] (ordre croissant) : 4
Saisir tab[3] (ordre croissant) : 6
Saisir tab[4] (ordre croissant) : 6
Saisir tab[5] (ordre croissant) : 6
Saisir tab[6] (ordre croissant) : 6
Saisir tab[7] (ordre croissant) : 7
Saisir tab[8] (ordre croissant) : 7
Saisir tab[9] (ordre croissant) : 8

tab :
1 4 4 6 6 6 6 7 7 8
Le majoré est 6 qui est compté 4 fois
```

**Complexité :**

Affichage =  $n+1$

Affectation =  $4n$  ou  $2n+2$  ou  $n+3$

Lecture =  $n+1$

## Conversion d'un nombre base 10 en nombre binaire

**Algorithme :** conversionBinaire

**Données :** i, n, nb, tmp : entiers;  
\* v : tableau d'entiers;

**Début**

// Lecture du nombre à convertir  
lire(nb);

// Calcul de la taille du tableau

tmp  $\leftarrow$  nb;

i  $\leftarrow$  0 ;

**TantQue** tmp > 0 **Faire**

    tmp  $\leftarrow$  tmp – puissance(2, i);

    i  $\leftarrow$  i+1;

**FinTantQue**

n  $\leftarrow$  i;

// Initialisation du tableau v

v  $\leftarrow$  allouer(taille(entier)\*n);

// Calcul du nombre binaire

**Pour** i descendant de n à 1 **Faire**

**Si** (nb % 2) = 0 **Alors**

        v(i)  $\leftarrow$  0;

**Sinon**

```
        v(i) ← 1;
    FinSi
    nb ← nb div 2 ;
FinPour

// Affichage du tableau v
Pour i allant de 1 à n Faire
    afficher(v(i));
FinPour

retourner(v);

Fin
```

```
129  /*Convertie un nombre base 10 en base 2*/ 148
130  int * conversion(int nb){                  149
131      //Declarations                          150
132      int i, tmp;                             151
133      int * v;                                152
134
135      //Calcul de la taille du tableau        153
136      tmp = nb;                               154
137      for(i=0;tmp > 0;i++){                   155
138          tmp -= pow(2,i);                    156
139      }                                       157
140      n = i;                                158
141
142      //Initialisation du tableau v          159
143      v = (int *)malloc(sizeof(int)*n);       160
144
145      //Calcul du nombre binaire              161
146      for(i=n-1;i>=0;i--){                   162
147          if(nb%2 == 0){                      163
148
149              v[i] = 0;
150          }else{
151              v[i] = 1;
152          }
153
154          nb /= 2;
155      }
156
157      //Affichage du tableau v
158      printf("v : ");
159      for(i=0;i<n;i++){
160          printf("%d ", v[i]);
161      }
162      printf("\n");
163
164      return v;
165  }
```

Exemple de conversion d'un nombre base 10 en binaire :

```
Saisir un nombre en base 10 :50
nombre binaire :
1 1 0 0 1 0
```

**Complexité :**

Affichage = n

Affectation =  $2n+2k+4$  (k nb de tour de la boucle tant que)

Lecture = 1

## Algorithmes de Tri

### Tri via un Arbre

On utilise un arbre et le parcours Infini pour trier un tableau. Chaque valeur est ajoutée dans l'arbre selon ses rapports avec les autres nœuds, de façon récursive.

L'arbre est ensuite lu, permettant d'afficher les valeurs triées du tableau.

**Algorithme :** triViaArbre

**Données :**

*a :	Structure Arbre;
i :	Entier;
*b :	Structure Noeud;
*T :	Tableau d'entiers ;

**Début**

```
creerArbre(a);  
Pour i allant de 1 à taille(T) Faire  
    Si a->racine == NULL Alors  
        allouer(b)  
        a->racine ← b;  
    Sinon  
        ajouterTriElement(T[i], a->root);  
    FinSi  
FinPour  
afficher(a, 'f');
```

**Fin**

```
int triTableauViaArbre(int *T, int n){  
    Arbre *a = malloc(sizeof(Arbre));  
    for(int i=0;i<n;i++){  
        if(a->root == NULL){  
            Node* b = malloc(sizeof(Node));  
            a->root = b;  
        }  
        else{  
            ajouterTriElement(T[i], a->root);  
        }  
    }  
    afficher(a, 'f');  
}
```

### Tri de l'arbre

**Algorithme :** ajouterTriElement

**Données :**

*a :	Structure Arbre;
i, ecart1, ecart2, tmp :	Entier;
*b,*c :	Structure Noeud;
*T :	Tableau d'entiers ;

**Début**

```
allouer(b);
b->valeur ← n;
Si c->fg == NULL Alors
    c->fg ← b;
    b->top ← c;
Sinon Si c->rgt == NULL Alors
    c->fd ← b;
    b->top ← c;
    //puis inversion si valeur plus petit qu'à gauche
    Si c->fd->valeur < c->fg->valeur Alors
        tmp ← c->fd->valeur;
        c->fd->valeur ← c->fg->valeur;
        c->fg->valeur ← tmp;
    FinSi
Sinon
    Si c->fg->valeur < n Alors
        ajouterTriElement(n, c->fg); //ajout récursif à gauche
    Sinon Si c->fd->valeur < n Alors
        ajouterTriElement(n, c->fd); //ajout récursif à droite
    Sinon
        ecart1 = n-c->fg->valeur;
        ecart2 = n-c->fd->valeur;

        Si ecart1>ecart2 Alors //ajout récursif à droite avec inversion de la valeur à insérer
            tmp ← c->fd->valeur;
            c->fd->valeur ← n;
            ajouterTriElement(tmp, c->fd);

        Sinon
            //ajout récursif à gauche avec inversion de la valeur à insérer
            tmp ← c->fg->valeur;
            c->fg->valeur ← n;
            ajouterTriElement(tmp, c->fg);
        FinSi
    FinSi
FinSi
Fin
```



```
void ajouterTriElement(int n, Node* c){

    int ecart1, ecart2, tmp;
    Node* b = malloc(sizeof(Node));
    b->value = n;
    if(c->lft == NULL){
        c->lft = b;
        b->top = c;
    }
    else if(c->rgt == NULL){
        c->rgt = b;
        b->top = c;//puis inversion si valeur plus petit qu'à gauche

        if(c->rgt->value < c->lft->value){
            tmp = c->rgt->value;
            c->rgt->value = c->lft->value;
            c->lft->value = tmp;
        }
    }
    else {
        if(c->lft->value < n){
            ajouterTriElement(n, c->lft);//ajout récursif à gauche
        }

        else if(c->rgt->value < n){
            ajouterTriElement(n, c->rgt);//ajout récursif à droite,
        }

        else{
            ecart1 = n-c->lft->value;
            ecart2 = n-c->rgt->value;

            if(ecart1>ecart2){//ajout récursif à droite avec inversion de la valeur à insérer
                tmp = c->rgt->value;
                c->rgt->value = n;
                ajouterTriElement(tmp, c->rgt);
            }
            else{
                //ajout récursif à gauche avec inversion de la valeur à insérer
                tmp = c->lft->value;
                c->lft->value = n;
                ajouterTriElement(tmp, c->lft);
            }
        }
    }
}
```

## Tri à bulle

Algorithme : trierBulle

Données :

*T :	Tableau d'entiers
stop :	Booléen
i, valtmp :	Entier

**Début**

stop  $\leftarrow$  Faux;

**TantQue** stop == Faux **Faire**

    i = 0;

    stop  $\leftarrow$  Vrai;

**TantQue** i < taille(T)-1 **Faire**

**Si** T[i] > T[i+1] **Alors**

            valtmp  $\leftarrow$  T[i+1];

            T[i+1]  $\leftarrow$  T[i];

            T[i]  $\leftarrow$  valtmp;

            stop  $\leftarrow$  Faux;

**FinSi**

        i  $\leftarrow$  i+1;

**FinTantQue**

**FinTantQue**


**Fin**

```
int main(){
    //Declaration
    int n,i, j, tmp;
    n = 10;
    int t[10] = {1, 5, 7, 2, 3, 9, 8, 4, 10, 6};
    int stop = 0;

    for(i=0;i<n;i++){
        printf("%d ", t[i]);
    }
    printf("\n");
    while(stop == 0){
        i = 0;
        stop = 1;
        while(i < n-1){
            if(t[i] > t[i+1]){
                tmp = t[i+1];
                t[i+1] = t[i];
                t[i] = tmp;
                stop = 0;
            }
            i++;
        }
    }

    for(i=0;i<n;i++){
        printf("%d ", t[i]);
    }
    printf("\n");

    exit(0);
}
```



```
1 5 7 2 3 9 8 4 10 6
1 2 3 4 5 6 7 8 9 10
```

**Complexité** :  $n^2$  dans le pire,  $n$  au mieux

## Tri par Sélection

**Algorithme** : trierBulle

**Données** :

*T :	Tableau d'entiers
stop :	Booléen
i,j, tmp,min :	Entier

**Début**

**Pour**  $i$  allant de 0 à  $\text{taille}(T) - 1$  **Faire**

$\text{min} \leftarrow i$ ;

```

    Pour j allant de i + 1 à taille(T) Faire
        Si T[j] < T[min] Alors
            min ← j ;
        FinSi
    FinPour
    Si min != i Alors
        tmp ← T[min] ;
        T[min] ← T[i] ;
        T[i] = tmp ;
    FinSi
FinPour
Fin
```

```
int main(){
    //Declaration
    int n,i, j, tmp, min;
    n = 10;
    int t[10] = {1, 5, 7, 2, 3, 9, 8, 4, 10, 6};
    int stop = 0;

    for(i=0;i<n;i++){
        printf("%d ", t[i]);
    }

    printf("\n");

    for(i=0;i<n-1;i++){
        min = i;
        for(j=i+1;j<n;j++){
            if(t[j] < t[min]){
                min = j;
            }
        }
        if(min != i){
            tmp = t[min];
            t[min] = t[i];
            t[i] = tmp;
        }
    }

    for(i=0;i<n;i++){
        printf("%d ", t[i]);
    }

    printf("\n");

    exit(0);
}
```

```
1 5 7 2 3 9 8 4 10 6
1 2 3 4 5 6 7 8 9 10
```

Complexité :  $n^2$

## Algorithmes de probabilité

### Simuler un jeu de tennis

**Algorithme :** simuTennis

**Données :** score1, score2 : entiers;  
p : réel;  
avantage : booléen;

**Début**

// Lecture de la probabilité de gagner  
lire(p) ;

// Initialisation  
score1  $\leftarrow$  0 ;  
score2  $\leftarrow$  0 ;  
avantage  $\leftarrow$  faux ;

// Simulation du jeu

**TantQue** (score1 != 4 **ET** score2 != 4 **ET** !avantage) **OU** (score1 != 5 **ET** score2 != 5 **ET** avantage) **Faire**

**Si** score1 < 3 | score2 < 3 **Alors**

**Si** aléatoire() < p **Alors**

            score1  $\leftarrow$  score1+1;

**Sinon**

            score2  $\leftarrow$  score2+1;

**FinSi**

**Sinon**

        avantage  $\leftarrow$  vrai ;

**Si** aléatoire() < p **Alors**

**Si** score2 = 4 **Alors**

                score2  $\leftarrow$  3;

**Sinon**

                score1  $\leftarrow$  score1+1 ;

**FinSi**

**Sinon**

**Si** score1 = 4 **Alors**

            score1  $\leftarrow$  3;

**Sinon**

            score2  $\leftarrow$  score2+1 ;

**FinSi**

**FinSi**

**FinSi**

**FinTantQue**

// Affiche si on a gagné ou perdu

**Si** score1 = 4 | score1 = 5 **Alors**

    afficher("vous avez gagné le jeu !") ;

**Sinon**

afficher("vous avez perdu le jeu !");

**FinSi**

**Fin**

```
5  int main(){
6      //Declaration
7      int score1, score2, avantage;
8      float p;
9
10     //Lecture de la probabilité de gagnée
11     do{
12         printf("Saisir votre probabilité de gagner entre 0 et 100 : ");
13         scanf("%f",&p);
14     }while(p > 100 && p < 0);
15
16     //Initialisation
17     score1 = 0;
18     score2 = 0;
19     avantage = 0;
20     srand(time(NULL));
21
22     //Simulation du jeu
23     while((score1 != 4 && score2 != 4 && avantage == 0) || (score1 != 5 && score2 != 5 && avantage == 1)){
24         if(score1 < 3 || score2 < 3){
25             if(rand()%100 < p){
26                 score1++;
27             }else{
28                 score2++;
29             }
30         }else{
31             avantage = 1;
32             if(rand()%100 < p){
33                 if(score2 == 4){
34                     score2 = 3;
35                 }else{
36                     score1++;
37                 }
38             }else{
39                 if(score1 == 4){
40                     score1 = 3;
41                 }else{
42                     score2++;
43                 }
44             }
45         }
46     }
47
48     //Affiche si on a gagné ou perdu
49     if((score1 == 4 && avantage == 0) || score1 == 5){
50         printf("Vous avez gagné le jeu ! \n");
51     }else{
52         printf("Vous avez perdu le jeu ! \n");
53     }
54 }
```

### Exemple de jeu de tennis :

Saisir votre probabilité de gagner entre 0 et 100 : 30  
Vous avez perdu le jeu !

Saisir votre probabilité de gagner entre 0 et 100 : 80  
Vous avez gagné le jeu !

Saisir votre probabilité de gagner entre 0 et 100 : 50  
Vous avez perdu le jeu !

Saisir votre probabilité de gagner entre 0 et 100 : 50  
Vous avez gagné le jeu !

**Complexité :**

Affichage = 1

Lecture = 1

## Algorithmes de formalisme de pointeur

### Copier une chaîne de caractère

**Algorithme :** copieChaine

**Données :**      n : entier;  
                  \* p1, \* p2 : caractères;  
                  tab(100) : tableau statique de caractères;  
                  \* tab\_copy : tableau de caractères;

**Début**

// Lecture de la chaine de caractère (max 100)  
lire(tab);

// Calcul de la taille de la chaine

p1 ← tab;  
**TantQue** \*p1 != '\0' **Faire**  
    p1 ← p1+1;

**FinTantQue**

n ← p1-tab;

// Initialisation

tab\_copy ← allouer(taille(caractère));

// Copie du tableau

p1 ← tab;  
p2 ← tab\_copy;  
**TantQue** \*p1 != '\0' **Faire**  
    \*p2 ← \*p1;  
    p1 ← p1+1;  
    p2 ← p2+1;

**FinTantQue**

// Affichage du résultat

afficher(tab\_copy);

// Nettoyage de la mémoire

liberer(tab\_copy);

**Fin**

```
4  int main(){
5      //Declarations
6      int n;
7      char * p1, * p2;
8      char tab[100], * tab_copy;
9
10     //Lecture de la chaine de caractère
11     printf("Saisir une chaine de caractere max(100) : ");
12     scanf("%s",tab);
13
14     //Calcul de la taille de la chaine
15     for(p1=tab;*p1 != '\0';p1++){
16     }
17     n = p1-tab;
18
19     //Initialisation
20     tab_copy = (char *)calloc(n, sizeof(char));
21
22     //Copie
23     for(p1=tab, p2 = tab_copy;*p1 != '\0';p1++, p2++){
24         *p2 = *p1;
25     }
26
27     //Affichage du résultat
28     printf("résultat : %s de taille %d \n", tab_copy, n);
29
30     //Nettoyage de la mémoire
31     free(tab_copy);
32
33     exit(0);
34 }
```

Exemple de copie de chaine de caractère :

```
Saisir une chaine de caractere max(100) : bonjour
résultat : bonjour de taille 7
```

**Complexité :**

Affichage = n

Affectation = 4n+5

Lecture = 1

## Supprimer l'occurrence

**Algorithme :** supprimerOccurrence

**Données :** n, nb : entiers;  
\* p1, \* p2 : entiers;  
\* tab : tableau d'entiers;

**Début**

```
// Lecture de la taille du tableau
lire(n);
```

```
// Lecture de l'occurrence
```



```
lire(nb);

// Initialisation
tab ← allouer(taille(entier));

// Lecture du tableau
Pour p1 allant de tab à (tab+n) Faire
    lire(p1);
FinPour

// Affichage du tableau
Pour p1 allant de tab à (tab+n) Faire
    afficher(*p1);
FinPour

// Supprimer toutes les occurrences
p2 ← tab;
Pour p1 allant de tab à (tab+n) Faire
    *p2 ← *p1;
    Si *p2 != nb Alors
        p2 ← p2+1;
    FinSi
FinPour
n ← p2-tab;

// Affichage du résultat
Pour p1 allant de tab à (tab+n) Faire
    afficher(*p1);
FinPour

// Nettoyage de la mémoire
liberer(tab);

Fin
```

```
4  int main(){
5      //Declarations
6      int n, nb;
7      int *p1, *p2;
8      int * tab;
9
10     //Lecture de la taille du tableau
11     do{
12         printf("Saisir la taille du tableau : ");
13         scanf("%d",&n);
14     }while(n < 1);
15
16     //Lecture de l'occurrence
17     printf("Saisir l'occurrence à supprimer : ");
18     scanf("%d",&nb);
19
20     //Initialisation
21     tab = (int *)calloc(n, sizeof(int));
22
23     //Lecture du tableau
24     for(p1=tab;p1<tab+n;p1++){
25         printf("Saisir tab[%ld] = : ", p1-tab);
26         scanf("%d",p1);
27     }
28
29     //Affichage du tableau
30     for(p1=tab;p1<tab+n;p1++){
31         printf("%d ", *p1);
32     }
33     printf("\n");
34
35     //Supprimer toutes les occurrences
36     for(p1=p2=tab;p1<tab+n;p1++){
37         *p2 = *p1;
38         if(*p2 != nb){
39             p2++;
40         }
41     }
42     n = p2-tab;
43
44     //Affichage du résultat
45     printf("Résultat : \n");
46     for(p1=tab;p1<tab+n;p1++){
47         printf("%d ", *p1);
48     }
49     printf("\n");
50
51     //Nettoyage de la mémoire
52     free(tab);
53
54     exit(0);
55 }
```

#### Exemple supprimer l'occurrence :

```
Saisir la taille du tableau : 10
Saisir l'occurrence à supprimer : 5
Saisir tab[0] = : 1
Saisir tab[1] = : 5
Saisir tab[2] = : 5
Saisir tab[3] = : 2
Saisir tab[4] = : 4
Saisir tab[5] = : 8
Saisir tab[6] = : 5
Saisir tab[7] = : 6
Saisir tab[8] = : 3
Saisir tab[9] = : 5

tab :
1 5 5 2 4 8 5 6 3 5
Résultat :
1 2 4 8 6 3
```

#### Complexité :

Affichage =  $2n$

Affectation =  $n+k+3$  (k le nb d'occurrences supprimées)

Lecture =  $n+2$

## Inverser les éléments d'un tableau

**Algorithme :** inversionTableau

**Données :**        n, tmp : entiers;  
                  \* p1, \* p2 : entiers;  
                  \* tab : tableau d'entiers;

**Début**

```
// Lecture de la taille du tableau  
lire(n);
```

```
// Initialisation  
tab ← allouer(taille(entier));
```

```
// Lecture du tableau  
Pour p1 allant de tab à (tab+n) Faire  
    lire(p1);  
FinPour
```

```
// Affichage du tableau  
Pour p1 allant de tab à (tab+n) Faire  
    afficher(*p1);  
FinPour
```

```
// Inverser les éléments du tableau  
p2 ← tab+n-1;  
Pour p1 allant de tab à (tab+n/2) Faire  
    tmp ← *p1;  
    *p2 ← *p1;  
    *p2 ← tmp;  
    p2 ← p2-1;  
FinPour
```

```
// Affichage du résultat  
Pour p1 allant de tab à (tab+n) Faire  
    afficher(*p1);  
FinPour
```

```
// Nettoyage de la mémoire  
liberer(tab);
```

**Fin**

```
4  int main(){
5      //Declarations
6      int n, tmp;
7      int *p1, *p2;
8      int * tab;
9
10     //Lecture de la taille du tableau
11     do{
12         printf("Saisir la taille du tableau : ");
13         scanf("%d",&n);
14     }while(n < 1);
15
16     //Initialisation
17     tab = (int *)calloc(n, sizeof(int));
18
19     //Lecture du tableau
20     for(p1=tab;p1<tab+n;p1++){
21         printf("Saisir tab[%ld] = : ", p1-tab);
22         scanf("%d",p1);
23     }
24
25     //Affichage du tableau
26     for(p1=tab;p1<tab+n;p1++){
27         printf("%d ", *p1);
28     }
29     printf("\n");
30
31     //Inverse les éléments du tableau
32     for(p1=tab, p2=tab+n-1;p1<tab+(n/2);p1++, p2--){
33         tmp = *p1;
34         *p1 = *p2;
35         *p2 = tmp;
36     }
37
38     //Affichage du résultat
39     printf("Résultat : \n");
40     for(p1=tab;p1<tab+n;p1++){
41         printf("%d ", *p1);
42     }
43     printf("\n");
44
45     //Nettoyage de la mémoire
46     free(tab);
47
48     exit(0);
49 }
```

#### Exemple inversion des éléments d'un tableau :

```
Saisir la taille du tableau : 10
Saisir tab[0] = : 9
Saisir tab[1] = : 8
Saisir tab[2] = : 7
Saisir tab[3] = : 6
Saisir tab[4] = : 5
Saisir tab[5] = : 4
Saisir tab[6] = : 3
Saisir tab[7] = : 2
Saisir tab[8] = : 1
Saisir tab[9] = : 0

tab :
9 8 7 6 5 4 3 2 1 0
Résultat :
0 1 2 3 4 5 6 7 8 9
```

#### Complexité :

Affichage =  $2n$   
Affectation =  $2n+2$   
Lecture =  $n+1$

## Palindrome

Un Palindrome est un mot consistant des mêmes caractères même s'il est inversé. Par exemple, Bob à l'envers donne Bob, tout comme Elle donne Elle.

**Algorithme :** chaineEstPalindrome

**Données :**        n : entier  
                  \* p1, \* p2 : char  
                  tab(30) : tableau de caractère  
                  test : booléen

**Début**

```
// Initialisation
test ← 1;

// Lecture de la chaine de caractère
lire(tab);

// Affichage du tableau
p1 ← tab;
TantQue *p1 != '\0' Faire
    afficher(*p1);
    p1 ← p1+1;
FinTantQue

// Calculer la taille de la chaine de caractère
n ← p1-tab;

// Test si c'est un Palindrome
p2 ← tab+n-1;
Pour p1 allant de tab à (tab+n/2) Faire
    Si *p1 != *p2 Faire
        test ← 0;
    FinSi
    p2 ← p2-1;
FinPour

// Affichage du résultat
Pour p1 allant de tab à (tab+n) Faire
    afficher(*p1);
FinPour

Si test = 0 Faire
    afficher("n'est pas un palindrome");
Sinon
    afficher("est un palindrome");
FinSi
```

**Fin**

```
4  int main(){
5      //Declarations
6      short int test = 1;
7      int n;
8      char *p1, *p2;
9      char tab[30];
10
11     //Lecture de la chaine de caractère
12     printf("Saisir une chaine de caractere (max 30): ");
13     scanf("%s", tab);
14
15     //Affichage du tableau
16     for(p1=tab; *p1 != '\0'; p1++){
17         printf("%c", *p1);
18     }
19     printf("\n");
20     //Calculer la taille de la chaine de caractère
21     n = p1-tab;
22
23     //Test si c'est un Palindrome
24
25     for(p1=tab, p2=tab+n-1; p1<tab+(n/2); p1++, p2--){
26         if(*p1 != *p2){
27             test = 0;
28         }
29
30     //Affichage du résultat
31     printf("Résultat : \n");
32     for(p1=tab; p1<tab+n; p1++){
33         printf("%c ", *p1);
34     }
35     if(test == 0){
36         printf(" n'est pas un palindrome \n");
37     }else{
38         printf(" est un palindrome \n");
39     }
40
41     exit(0);
42 }
```

#### Exemple de test Palindrome :

```
Saisir une chaine de caractere (max 30): bob
bob
Résultat :
bob est un palindrome
```

#### Complexité :

Affichage =  $2n$

Affectation =  $1.5n+k+4$  ( $k$  = le nombre de char != dans le « test »)

Lecture = 1

## Copier un tableau

Algorithme : copieTableau

Données :  
n : entier  
\* p1, \* p2 : entiers  
\* tab1, \* tab2 : tableaux d'entiers

Début

```
// Lecture de la taille du tableau
lire(n);
```

```
// Initialisation
tab1 ← allouer(taille(entier));
tab2 ← allouer(taille(entier));
```

```
// Lecture du tableau
Pour p1 allant de tab1 à (tab1+n) Faire
    lire(p1);
FinPour

// Affichage du tableau
Pour p1 allant de tab1 à (tab1+n) Faire
    afficher(*p1);
FinPour

// Copier tab1 dans tab2
p2 ← tab2;
Pour p1 allant de tab1 à (tab1+n) Faire
    *p2 = *p1 ;
    p2 ← p2+1 ;
FinPour

// Affichage du résultat
Pour p1 allant de tab2 à (tab2+n) Faire
    afficher(*p1);
FinPour
```

**Fin**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      //Declaration
6      int n;
7      int * p1, * p2;
8      int * tab1, * tab2;
9
10     //Lecture de la chaine de caractère
11     printf("Saisir une chaine de caractere max(100) : ");
12     scanf("%s",tab);
13
14     //Calcul de la taille de la chaine
15     for(p1=tab;*p1 != '\0';p1++){
16     }
17     n = p1-tab;
18
19     //Initialisation
20     tab_copy = (char *)calloc(n, sizeof(char));
21
22
23     //Copie
24     for(p1=tab, p2 = tab_copy;*p1 != '\0';p1++, p2++){
25         *p2 = *p1;
26     }
27
28     //Affichage du résultat
29     printf("résultat : %s de taille %d \n", tab_copy, n);
30     return 0;
31 }
```

Exemple de copie de tableau :

```
Saisir la taille n : 5
Saisir tab[0] :1
Saisir tab[1] :4
Saisir tab[2] :5
Saisir tab[3] :4
Saisir tab[4] :5

tab : 1 4 5 4 5
résultat tab_copy : 1 4 5 4 5
```

Complexité :

Affichage =  $2n$

Affectation =  $n+3$

Lecture =  $n+1$

## Algorithmes récursifs

### Ackerman récursif

Algorithme : AckermanRecuratif

Données :  $n, m$  : entiers;

Début

// Fonction calcul d'Ackerman

**Si**  $m = 0$  **Alors**

retourner( $n+1$ );

**Sinon Si**  $n = 0$  **Alors**

retourner(AckermanRecuratif( $m-1, 1$ ));

**Sinon**

retourner(AckermanRecuratif( $m-1, \text{AckermanRecuratif}(m, n-1)$ ));

**FinSi**

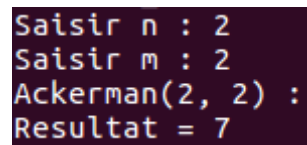
**FinSi**

Fin



```
4 //Fonction Ackerman récursif
5 int Ackerman(int m, int n){
6     if(m == 0){
7         return (n+1);
8     }else if(n==0){
9         return (Ackerman(m-1,1));
10    }else{
11        return (Ackerman(m-1, Ackerman(m, n-1)));
12    }
13 }
14
15 int main(){
16     //Déclarations
17     int m,n;
18
19     //Initialisation
20
21     do{
22         printf("saisir n: ");
23         scanf("%d",&n);
24     }while(n<0);
25
26     do{
27         printf("saisir m : ");
28         scanf("%d",&m);
29     }while(n<0);
30
31     //Affichage du résultat
32     printf("%d %d \n",n,m);
33     printf("%d \n", Ackerman(m,n));
34
35     exit(0);
36 }
```

Exemple Ackerman Récursif :



```
Saisir n : 2
Saisir m : 2
Ackerman(2, 2) :
Resultat = 7
```

## Tri d'un tableau par pivot (tri rapide) récursif

**Algorithme :** triRapide

**Données :** inf, sup, G, D, i, j, P, K : entiers;  
\* T : tableau d'entiers;

**Début**

// Initialisation

G ← inf;

D ← sup;

i ← G;

j ← sup-1;

P ← T(sup);

// Calcul tri par pivot

**TantQue** i != j **Faire**

**TantQue** T(i) < P **ET** i != j **Faire**

        i ← i+1;

**FinTantQue**

**TantQue** T(j) > P **ET** i != j **Faire**

        j ← j-1;

**FinTantQue**

```
K ← T(i);  
T(i) ← T(j);  
T(j) ← K;
```

**FinTantQue**

**Si** T(i) > P **Alors**

```
K ← T(i);  
T(i) ← T(sup);  
T(sup) ← K;
```

**FinSi**

**Si** G < i **Alors**

```
triRapide(T, G, i);
```

**FinSi**

**Si** D > i+1 **Alors**

```
triRapide(T, i+1, D);
```

**FinSi**

**Fin**

```
1  #include <stdio.h>  
2  #include <stdlib.h>  
3  
4  void TriRapide(int * T, int inf, int sup){  
5      int G, D, i, j, P, K;  
6  
7      G = inf;  
8      D = sup;  
9      i = G;  
10     j = sup-1;  
11     P = T[sup];  
12     printf(" p = %d \n", P);  
13  
14     while(i != j){  
15         while((T[i] < P) && (i != j)){  
16             i++;  
17         }  
18  
19         while((T[j] > P) && (i != j)){  
20             j--;  
21         }  
22  
23         K = T[i];  
24         T[i] = T[j];  
25         T[j] = K;  
26     }  
27  
28     if(T[i] > P){  
29         K = T[i];  
30         T[i] = T[sup];  
31         T[sup] = K;  
32     }  
33  
34     if(G < i){  
35         TriRapide(T, G, i);  
36     }  
37  
38     if(D > i+1){  
39         TriRapide(T, i+1, D);  
40     }  
41 }
```

```
Saisir taille de T :10  
Saisir T[0] :5  
Saisir T[1] :2  
Saisir T[2] :4  
Saisir T[3] :6  
Saisir T[4] :8  
Saisir T[5] :7  
Saisir T[6] :9  
Saisir T[7] :3  
Saisir T[8] :2  
Saisir T[9] :1  
Tab :  
5 2 4 6 8 7 9 3 2 1  
Resultat du tableau trié :  
1 2 2 3 4 5 6 7 8 9
```

# Algorithmes de Pile

## Définition des structures

**Définir :** Structure cellule

**Début**

    valeur : entier;  
    \* suivant : structure cellule;

**Fin**

**Nommer** Cellule

**Définir :** Structure pile

**Début**

    \* premier : structure cellule;

**Fin**

**Nommer** Pile

```
4  /*Element de la pile de type int*/
5  typedef struct Cell Cell;      12  /*Pile de premier element first*/
6  struct Cell                    13  typedef struct Pile Pile;
7  {                               14  struct Pile
8      int value;                 15  {
9      Cell *nxt;                 16      Cell *first;
10 };                             17  };
```

## Créer une Pile

**Algorithme :** creerPile

**Données :** \* p : structure pile;

**Début**

    // Allocation et initialisation de la pile  
    p ← allouer(taille(structure pile));  
    p → premier ← NULL;  
    retourner(p);

**Fin**

```
19  /*Initialisation d'une pile */
20  Pile* creerPile(){
21      Pile *p = malloc(sizeof(Pile));
22      p->first = NULL;
23      return p;
24  }
```

## Empiler une Pile

**Algorithme :** empilerPile

**Données :**     \* p : structure pile;  
                  val : entier;  
                  \* c : structure cellule;

**Début**

```
// Allocation et initialisation de la cellule
c ← allouer(taille(structure cellule));
c → valeur ← val;
c → suivant ← NULL;
```

```
// Empiler la pile
```

```
Si p != NULL OU c != NULL Faire
    Si p → premier != NULL Faire
        c → suivant ← p → premier;
        p → premier ← c;
    Sinon
        c → suivant ← NULL;
        p → premier ← c;
FinSi
```

**FinSi**

**Fin**

```
26  /*On ajoute la valeur val dans notre pile*/
27  void empilerPile(Pile *p, int val){
28
29      Cell *new_c = malloc(sizeof(*new_c));
30      if (p == NULL || new_c == NULL){
31          }
32      else{
33          if(p->first != NULL){/*Ajout d'un élément de la pile*/
34              new_c->value = val;
35              new_c->nxt = p->first;
36              p->first = new_c;
37          }
38          else{/*Ajout du premier élément de la pile*/
39              new_c->value = val;
40              new_c->nxt = NULL;
41              p->first = new_c;
42          }
43      }
44
45  }
```

## Dépiler une Pile

**Algorithme :** depilerPile

**Données :**      \* p : structure pile;  
                  val : entier;  
                  \* c : structure cellule;

**Début**

// Initialisation

\* c  $\leftarrow$  p  $\rightarrow$  premier;

// Depiler la pile

**Si** p  $\neq$  NULL **ET** p  $\rightarrow$  premier  $\neq$  NULL **Faire**  
    p  $\rightarrow$  premier  $\leftarrow$  c  $\rightarrow$  suivant;  
    liberer(c);

**FinSi**

**Fin**

```
47  /*On retire le sommet*/
48  void depilerPile(Pile *p){
49
50      Cell *elem = p->first;
51
52      if (p != NULL && p->first != NULL){
53          printf("--%d\n", elem->value);
54          p->first = elem->nxt;
55          free(elem);
56      }
57
58  }
```

## Sommet d'une pile

**Algorithme :** sommet

**Données :**      \* p : structure pile;

**Début**

// Retourne le sommet de la pile

retourner(p  $\rightarrow$  premier  $\rightarrow$  valeur);

**Fin**

```
60  /*Retourne la valeur du sommet de la pile*/  
61  int sommet(Pile *p){  
62  
63      Cell *elem = p->first;  
64  
65      return elem->value;  
66  
67  }
```

## Déterminer si une pile est vide

**Algorithme :** pileEstVide

**Données :**     \* p : structure pile;  
                  test : booléen;

**Début**

// Initialisation

test ← faux;

// Test si la pile est vide

**Si** p != NULL **ET** p → premier != NULL **Faire**

    p → premier ← c → suivant;

    liberer(c);

**FinSi**

**Fin**

```
69  /*Retour 0 ou 1 selon si la pile est vide ou non*/  
70  int pileEstVide(Pile *p){  
71      int i = 1;  
72      if(p != NULL && p->first != NULL){  
73          i = 0;  
74      }  
75      return i;  
76  }
```

## Vider une Pile

**Algorithme :** viderPile

**Données :**     \* p : structure pile;

**Début**

// Vide la pile

**TantQue** p → premier != NULL **Faire**

```
        depiler(p);  
FinTantQue
```

**Fin**

```
78  /*Vide complètement la pile*/  
79  void viderPile(Pile *p){  
80  
81      while(p->first != NULL){  
82          depilerPile(p);  
83      }  
84  
85  }
```

## Afficher une Pile

**Algorithme :** afficherPile

**Données :** \* p, \* tmp : structure pile;

**Début**

// Affichage de la pile

**TantQue** p → premier != NULL **Faire**  
 empiler(tmp, sommet(p));  
 afficher(sommet(p));  
 depiler(p);

**FinTantQue**

**TantQue** tmp → premier != NULL **Faire**  
 empiler(p, sommet(tmp));  
 depiler(tmp);

**FinTantQue**

// Nettoyage de la mémoire  
liberer(tmp);

**Fin**

```
131  /*Affiche le contenu de la pile
132  **On suit le principe d'une pile (on ne peut accéder qu'à la première valeur au sommet),
133  **mais avec le formalisme pointeur, on pourrait très bien remonter la pile telle une chaîne
134  */
135  void afficherPile(Pile *p){
136
137      Pile *ptmp = creerPile();
138
139      printf("La pile : ");
140
141      while (p != NULL && p->first != NULL){
142          printf("%d ", sommet(p));
143          empilerPile(ptmp, sommet(p));
144          depilerPile(p);
145      }
146
147      /*On repile la pile*/
148      while (ptmp != NULL && ptmp->first != NULL){
149          empilerPile(p, sommet(ptmp));
150          depilerPile(ptmp);
151      }
152      printf(" - FIN\n");
153
154      /*Nettoyage de la mémoire*/
155      free(ptmp);
156  }
```

## Egalité de deux Piles

**Algorithme :** estEgalePile

**Données :**     \* p1, \* p2, \* tmp1, \* tmp2 : structure pile;  
                  test : booléen;

**Début**

// Initialisation

tmp1 ← creerPile();

tmp2 ← creerPile();

test ← vraie;

// Test si les deux piles sont égales

**TantQue** p1 → premier != NULL **ET** p2 → premier != NULL **Faire**  
    empiler(tmp1, sommet(p1));  
    empiler(tmp2, sommet(p2));

**Si** sommet(p1) != sommet(p2) **Faire**  
        test ← faux;



**FinSi**

depiler(p1);  
depiler(p2);

**FinTantQue**

// On vérifie si la taille est différente

**Si** p1 → premier != NULL **OU** p2 → premier != NULL **Faire**  
test ← faux;

**FinSi**

// On reempile les piles

**TantQue** tmp1 → premier != NULL **ET** tmp2 → premier != NULL **Faire**  
empiler(p1, sommet(tmp1));  
empiler(p2, sommet(tmp2));  
depiler(tmp1);  
depiler(tmp2);

**FinTantQue**

// Nettoyage de la mémoire

liberer(tmp1);  
liberer(tmp2);

**Fin**

```
87  /*On compare deux piles p1 et p2
88  **Retour 1 en cas d'égalité, 0 sinon
89  **On suit le principe d'une pile (on ne peut accéder qu'à la première valeur au sommet),
90  **mais avec le formalisme pointeur, on pourrait très bien remonter la pile telle une chaîne
91  */
92  int estegalePile(Pile *p1, Pile *p2){
93
94      Pile *ptmp1 = creerPile();
95      Pile *ptmp2 = creerPile();
96      short int stop = 1;
97
98      while (p1 != NULL && p1->first != NULL && p2 != NULL && p2->first != NULL && stop == 1){
99          if(sommet(p1) != sommet(p2)){
100              stop = 0;
101          }
102          else{
103              empilerPile(ptmp1, sommet(p1));
104              empilerPile(ptmp2, sommet(p2));
105              depilerPile(p1);
106              depilerPile(p2);
107          }
108      }
109
110      /*On verifie si la taille est différente*/
111      if( (p2->first != NULL)|(p1->first != NULL)){
112          stop = 0;
113      }
114
115      /*On reempile les piles*/
116      while (ptmp1 != NULL && ptmp1->first != NULL && ptmp2 != NULL && ptmp2->first != NULL){
117          empilerPile(p1, sommet(ptmp1));
118          empilerPile(p2, sommet(ptmp2));
119          depilerPile(ptmp1);
120          depilerPile(ptmp2);
121      }
122
123      /*Nettoyage de la mémoire*/
124      free(ptmp1);
125      free(ptmp2);
126
127      return stop;
128
129  }
```

## Trier une Pile

**Algorithme :** trierPile

**Données :** \* p, \* tmp1, \* tmp2 : structure pile;  
min : entier;

**Début**

// Initialisation

```
tmp1 ← creerPile();
tmp2 ← creerPile();

// Tri la pile
TantQue p → premier != NULL Faire
    min ← sommet(p);
    TantQue p → premier != NULL Faire
        Si sommet(p) < min Faire
            min ← sommet(p);
        FinSi
        empiler(tmp1, sommet(p));
        depiler(p) ;
    FinTantQue

    TantQue tmp1 → premier != NULL Faire
        Si sommet(tmp1) != min Faire
            empiler(p, sommet(tmp1));
        FinSi
        depiler(tmp1) ;
    FinTantQue

    empiler(tmp2, min);
FinTantQue

//On replace les valeurs dans p
TantQue tmp2 → premier != NULL Faire
    empiler(p, sommet(tmp2));
    depiler(tmp2);
FinTantQue

// Nettoyage de la mémoire
liberer(tmp1);
liberer(tmp2);
```

**Fin**

```

158  /*Tri dans l'ordre croissant les valeurs d'une pile*/
159  void trierPile(Pile * p){
160      /*Declarations*/
161      int min;
162      Pile * tmp1 = creerPile();
163      Pile * tmp2 = creerPile();
164
165      while(p->first != NULL){
166          min = sommet(p);
167          while(p->first != NULL){
168              if(sommet(p) < min){
169                  min = sommet(p);
170              }
171              empiler(tmp1, sommet(p));
172              depiler(p);
173          }
174
175          while(tmp1->first != NULL){
176              if(sommet(tmp1) != min){
177                  empiler(p, sommet(tmp1));
178              }
179              depiler(tmp1);
180          }
181
182          empiler(tmp2, min);
183      }
184
185      while(tmp2->first != NULL){
186          empiler(p, sommet(tmp2));
187          depiler(tmp2);
188      }
189
190      /*Nettoyage de la mémoire*/
191      free(tmp1);
192      free(tmp2);
193  }

```

#### Exemple de pile :

```

La pile : 20 15 10 5 - FIN
Sommet : 20

Depiler la pile ...
Sommet : 15

Est vide : 0

La pile : 15 10 5 - FIN

tri de la pile ...
La pile : 5 10 15 - FIN

Est vide : 1

```

## Ackerman avec Pile (Iteratif)

**Algorithme :** AckermanPile

**Données :** n, m : entiers;  
\* p : Pile;

**Début**

```

// Initialisation
p ← creerPile();
empiler(p, m);
empiler(p, n);

```

```

// Fonction calcul d'Ackerman
TantQue NON pileEstVide(p) Faire

```

```

n ← sommet(p);
depiler(p);

Si NON pileEstVide(p) Alors
    m ← sommet(p);
    depiler(p);
Sinon
    Sortir;
FinSi

Si m = 0 Alors
    empiler(p, n+1);
Sinon Si n = 0 Alors
    empiler(p, m-1);
    empiler(p, 1);
Sinon
    empiler(p, m-1);
    empiler(p, m);
    empiler(p, n-1);
FinSi
FinSi
FinTantQue

```

**Fin**

63	int main(){	84	depiler(p);
64	int m,n;	85	}else{
65	do{	86	break;
66	printf("saisir n: ");	87	}
67	scanf("%d",&n);	88	
68	}while(n<0);	89	if(m == 0){
69	do{	90	empiler(p, n+1);
70	printf("saisir m : ");	91	}else if(n == 0){
71	scanf("%d",&m);	92	empiler(p, m-1);
72	}while(n<0);	93	empiler(p, 1);
73		94	}else{
74	Pile * p = creerPile();	95	empiler(p, m-1);
75	empiler(p,m);	96	empiler(p, m);
76	empiler(p,n);	97	empiler(p, n-1);
77		98	}
78	while(!pileEstVide(p)){	99	
79	n = sommet(p);	100	}
80	depiler(p);	101	printf("résultat : %d \n", n);
81		102	
82	if(!pileEstVide(p)){	103	exit(0);
83	m = sommet(p);	104	}
84	depiler(p);		

Exemple de calcul d'Ackerman récursif :

```
Saisir n : 3  
Saisir m : 3  
Ackerman(3, 3)  
Resultat : 61
```

## Algorithmes de Liste

Afin de simplifier nos calculs et ajouter un meilleur contrôle sur le nombre d'éléments, nous avons ajouté un attribut « taille » en plus, s'incrémentant ou se décrémentant selon au fil des utilisations de la liste.

### Définition des structures

Définir : Structure élément

**Début**

valeur : entier;  
\* prochain : structure élément;

**Fin**

Nommer élément\_liste

Définir Structure Liste

**Début**

\*premier, dernier : structure élément;  
taille : entier;

**Fin**

Nommer : Liste

```
4  /*Element de la pile de type int*/
5  typedef struct Cell Cell;
6  struct Cell
7  {
8      int value;
9      Cell *nxt;
10 };
11
12 /*Liste de taille n, de premier element first et de dernier element last*/
13 typedef struct Liste Liste;
14 struct Liste
15 {
16     Cell *first;
17     Cell *last;
18     int size;
19 }
20 ;;
```

### Créer une liste

Algorithme : CreerListe

Données : \*l : structure liste;

**Début**

// Allocation de la liste  
Allouer(l) ;

// Initialisation des pointeurs et de la taille  
l->premier ← NULL ;

l->dernier ← NULL ;  
taille ← 0;

**Fin**

```
22  /*Initialisation d'une list
23  **La liste ne contient aucune case
24  **La taille est de 0
25  */
26  Liste* creerListe(){
27      Liste * l= malloc(sizeof(Liste));
28      l->first = NULL;
29      l->last = NULL;
30      l->size = 0;
31      return l;
32  }
```

## La liste est-elle vide ?

**Algorithme** : estVideListe

**Données** : \*l : liste;

**Début**

retourner(l->premier == NULL);

**Fin**

```
272  /*Retour 0 ou 1 selon si la liste est vide ou non*/
273  int listeEstVide(Liste *l){
274      int i = 1;
275      if(l != NULL && l->first != NULL){
276          i = 0;
277      }
278      return i;
279  }
```

## Afficher la liste

**Algorithme** : afficherListe

**Données** : \*l : structure liste

\*e : structure élément

**Début**

e ← l->premier ;

**TantQue** e != NULL **Faire**

Ecrire(e->valeur) ;

e ← e->prochain ;

**FinTantQue**

**Fin**

```
256  /*Affiche le contenu de la liste*/
257  void afficherListe(Liste *l){
258
259      printf("La liste : ");
260
261      Cell* elem = l->first;
262
263      while(elem != NULL){
264          printf("%d ", elem->value);
265          elem = elem->nxt;
266      }
267
268      printf("Taille : %d - FIN\n", l->size);
269
270 }
```

## Chercher un élément dans la liste

Grace à la taille, il est bien plus facile de se déplacer dans la liste et de gérer les cas où l'utilisateur saisit des valeurs invalides. Ceci nous évite notamment de faire un parcours de toute la boucle jusqu'à trouver la valeur.

Retourne la n-ième valeur d'une liste. Retourne la dernière valeur si l'utilisateur rentre un n supérieur à la taille de la liste. S'il saisit une valeur inférieure à 0, parcours la liste dans le sens inverse !

**Algorithme :** chercherElementListe

**Données :**        \* l : structure liste  
                  n, i, j : entiers  
                  \* e : structure élément

**Début**

i ← 0  
j ← n  
e ← l->premier ;

**Si e != NULL Alors**

    //Si n est plus grand que la taille, on retourne la dernière valeur de la liste

**Si j > l->taille Alors**

        retourner l->dernier;

**Sinon**

**Si j < 0 Alors**

            j ← (l->taille-n)%l->taille; //On augmente j pour coller à la taille et parcourir dans le

sens inverse

**FinSi**

**Tant que i < j Faire**

            e ← e->prochain;

            i ← i + 1;

**FinTantQue**

        retourner e;



**FinSi**  
**Sinon**  
retourner NULL; //l est vide  
**FinSi**

```
35  /*Retourne la n-ième valeur d'une liste. Retourne la dernière valeur si OutOfBounds à la taille.
36  Cell* chercherElementListe(Liste *l, int n){
37
38      int i = 0;
39      int j = n;
40      Cell* c = l->first;
41
42      if(c != NULL){
43          if(j>l->size){
44              return l->last;
45          }
46          else {
47              if(j<0){
48                  j = (l->size-n)%l->size;
49              }
50              while(i < j){
51                  c = c->nxt;
52                  i++;
53              }
54              return c;
55          }
56      }
57      else{
58          return NULL;
59      }
60
61  }
```

## Ajouter en dernière position

**Algorithme :** ajouterDernierElementListe

**Données :**

*l :	Liste
valeur :	Entier
*nouvel_element,* tmp :	Structure Élément

**Début**

\*nouvel\_element ← malloc(sizeof(\*nouvel\_element));

**Si l != NULL ET nouvel\_element != NULL Alors**

**Si l->premier != NULL Alors** //Ajout d'un élément lambda

```
//On définit la nouvelle case  
nouvel_element->valeur ← valeur;  
nouvel_element->prochain ← NULL;  
//On redéfinit la dernière case  
tmp ← l->dernier;  
tmp->prochain ← nouvel_element ;  
//On définit la nouvelle case comme dernière case  
l->dernier ← nouvel_element;  
//On augmente la taille  
l->taille ← l->taille + 1;
```

**Sinon**

```
//Ajout du premier élément de la pile  
nouvel_element->valeur ← valeur ;  
nouvel_element->prochain ← NULL;  
l->premier ← nouvel_element ;  
l->dernier ← nouvel_element;  
l->taille ← 1;
```

**FinSi**

**FinSi**

**Fin**

```
63  /*On ajoute la valeur val à la fin de notre liste*/
64  void ajouterDernierElementListe(Liste *l, int val){
65
66      Cell *new_c = malloc(sizeof(*new_c));
67      Cell *tmp;
68
69      if (l == NULL || new_c == NULL){
70      }
71      else{
72          if(l->first != NULL){/*Ajout d'un élément de la pile standard*/
73
74              //On définit la nouvelle case
75              new_c->value = val;
76              new_c->nxt = NULL;
77
78              //On redéfinit la dernière case
79              tmp = l->last;
80              tmp->nxt = new_c;
81
82              //On définit la nouvelle case comme dernière case
83              l->last = new_c;
84
85              //On augmente la taille
86              l->size++;
87
88          }
89          else{/*Ajout du premier élément de la pile*/
90              new_c->value = val;
91              new_c->nxt = NULL;
92              l->first = new_c;
93              l->last = new_c;
94              l->size = 1;
95
96          }
97      }
98
99  }
```

## Ajouter la valeur à la nième position

**Algorithme :** ajouterElementListe

**Données :**

*l :	Liste
valeur, n,j:	entier
*nouvel_element,* tmp :	Structure Élément

**Début**

**Si** l != NULL ET nouvel\_element != NULL **Alors**

**Si** l->premier != NULL **Alors** //Ajout d'un élément de la liste standard

**Si** n<l->taille et n>0 **Alors**

```
//On définit la nouvelle case  
nouvel_element->valeur ← valeur;  
nouvel_element->prochain = chercherElementListe(l, n+1);  
//On redéfinit la case n-1  
tmp ← chercherElementListe(l, n-1);  
tmp->prochain ← nouvel_element;  
//On augmente la taille  
l->taille ← l->taille + 1 ;
```

**Sinon**

**Si**  $n \geq l \rightarrow \text{taille}$  **Alors**

ajouterDernierElementListe(l, valeur);

**Sinon**

$j \leftarrow (l \rightarrow \text{taille} - n) \% l \rightarrow \text{taille};$

ajouterElementListe(l, valeur, j);

**FinSi**

**FinSi**

**Sinon** //Ajout du premier élément de la pile

nouvel\_element->valeur = valeur;

nouvel\_element->prochain = NULL;

l->premier ← new\_c;

l->dernier ← new\_c;

l->taille ← l->taille + 1;

**FinSi**

**FinSi**

**Fin**

```
101  /*On ajoute la valeur val à la place n de notre liste*/
102  void ajouterElementListe(Liste *l, int val, int n){
103
104      Cell *new_c = malloc(sizeof(*new_c));
105      Cell *tmp;
106      int j;
107
108      if (l == NULL || new_c == NULL){
109      }
110      else{
111          if(l->first != NULL){/*Ajout d'un élément de la liste standard*/
112
113              if(n<l->size && n>0){
114                  //On définit la nouvelle case
115                  new_c->value = val;
116                  new_c->nxt = chercherElementListe(l, n+1);
117
118                  //On redéfinit la case n-1
119                  tmp = chercherElementListe(l, n-1);
120                  tmp->nxt = new_c;
121
122                  //On augmente la taille
123                  l->size++;
124              }
125              else{
126                  if(n>=l->size){
127                      ajouterDernierElementListe(l, val);
128                  }
129                  else{
130                      j = (l->size-n)%l->size;
131                      ajouterElementListe(l,val, j);
132                  }
133              }
134          }
135      }
136      else{/*Ajout du premier élément de la pile*/
137          new_c->value = val;
138          new_c->nxt = NULL;
139          l->first = new_c;
140          l->last = new_c;
141          l->size = 1;
142      }
143  }
144  }
145
146  }
```

## Retirer la valeur à la dernière position

**Algorithme :** supprimerDernierElementListe

**Données :**

*l :	Liste
*elem :	Structure Élément

**Début**

**Si** l->premier != NULL **Alors**

**Si** l->taille > 1 **Alors**

elem ← chercherElementListe(l, l->taille-2); //Retourne l'avant-dernière case

elem->prochain ← NULL; //Suppression du dernier élément

libérer(l->dernier);

l->taille ← l->taille - 1; //Réduction de la taille

l->dernier ← elem; //Redéfinition de la dernière case

**Sinon**

//La liste ne contient qu'une seule case

elem ← l->premier; //Retourne la seule et unique case

l->premier ← NULL;

l->dernier ← NULL;

l->taille ← 0;

libérer(elem);

**FinSi**

**FinSi**

**Fin**

```
148 /*On retire la dernière cell de notre liste*/
149 void supprimerDernierElementListe(Liste *l){
150
151     Cell *elem;
152
153     if (l != NULL){
154         if(l->size > 1){
155             elem = chercherElementListe(l, l->size-2); //Retourne l'avant-dernière case
156             printf("--%d\n", l->last->value);
157
158             elem->nxt = NULL; //Suppression du dernier élément
159             free(l->last);
160
161             l->size--; //Réduction de la taille
162
163             l->last = elem; //Redéfinition de la dernière case
164         }
165         else{//La liste ne contient qu'une seule case
166             elem = l->first; //Retourne la seule et unique case
167             printf("--%d\n", elem->value);
168             l->first = NULL;
169             l->last = NULL;
170             l->size = 0;
171             free(elem);
172         }
173     }
174
175 }
```

## Retirer la valeur à la position n

**Algorithme :** supprimerElementListe

**Données :**

\*l : Liste  
\*elem1, \*elem2, \*elem3 : Structure Élément  
i: Entier

**Début**

**Si** l->premier != NULL **Alors**

**Si** l->taille > 1 **Alors**

**Si** n < l->taille & n > 0 **Alors** //Suppression lambda

elem1 = chercherElementListe(l, n); //Retourne l'élément n

elem2 = chercherElementListe(l, n-1); //Retourne l'élément n-1

elem3 = elem1->prochain ; //Retourne l'élément n+1

elem2->prochain ← elem3; //On saute l'élément n à supprimer

l->taille ← l->taille; //Réduction de la taille

libérer(elem1);

**Sinon**

**Si** n >= l->taille **Alors** //n est plus grand que la taille, on supprime le dernier élément  
supprimerDernierElementListe(l);

**Sinon**

**Si** n == 0 **Alors** //premier élément de la liste à supprimer

elem1 ← chercherElementListe(l, 0); //Retourne l'élément n

elem2 ← elem1->prochain ; //Retourne l'élément n+1

l->premier ← elem2;

l->taille ← l->taille; //Réduction de la taille

libérer(elem1);

**Sinon** //n négatif, on parcourt dans le sens inverse

j ← (l->taille-n)%l->taille;

supprimerElementListe(l, j); //Appel récursif

**FinSi**

**FinSi**

**FinSi**

**Sinon** //La liste ne contient qu'une seule case : on la supprime donc

elem1 ← l->premier; //Retourne la seule et unique case

l->premier ← NULL;

l->dernier ← NULL;

l->taille ← 0;

libérer(elem1);

**FinSi**

## FinSi

### Fin

```
148  /*On retire la dernière cell de notre liste*/
149  void supprimerDernierElementListe(Liste *l){
150
151      Cell *elem;
152
153      if (l != NULL){
154          if(l->size > 1){
155              elem = chercherElementListe(l, l->size-2); //Retourne l'avant-dernière case
156              printf("--%d\n", l->last->value);
157
158              elem->nxt = NULL; //Suppression du dernier élément
159              free(l->last);
160
161              l->size--; //Réduction de la taille
162
163              l->last = elem; //Redéfinition de la dernière case
164          }
165          else{//La liste ne contient qu'une seule case
166              elem = l->first; //Retourne la seule et unique case
167              printf("--%d\n", elem->value);
168              l->first = NULL;
169              l->last = NULL;
170              l->size = 0;
171              free(elem);
172          }
173      }
174
175  }
```

## Retirer la valeur (première occurrence)

**Algorithme :** supprimerValeurListe

### Données :

*l :	Liste
*elem :	Structure Élément
i, retour, valeur :	Entiers

### Début

```
elem ← l->premier;
i = 0;
retour = 0;
//On se déplace vers l'occurrence
TantQue elem != NULL ET elem->valeur != valeur Faire
    elem = elem->prochain;
    l ← l + 1;
FinTantQue
//Si fin de la liste
Si i < l->taille Alors
    supprimerElementListe(l, i);
```



retour  $\leftarrow$  1;

**FinSi**

retourner retour; // On a bien supprimé la valeur

**Fin**

```
/*On retire la cell n de notre liste*/
void supprimerElementListe(Liste *l, int n){

    Cell *elem1;
    Cell *elem2;
    Cell *elem3;
    int j;

    if (l != NULL){
        if(l->size > 1){
            if(n<l->size & n > 0){
                elem1 = chercherElementListe(l, n); //Retourne l'élément n
                elem2 = chercherElementListe(l, n-1); //Retourne l'élément n-1
                elem3 = elem1->nxt; //Retourne l'élément n+1
                printf("--%d\n", elem1->value);

                elem2->nxt = elem3; //On saute l'élément n à supprimer

                l->size--; //Réduction de la taille

                free(elem1);
            }
            else{
                if(n>=l->size){
                    supprimerDernierElementListe(l);
                }
                else{
                    if(n == 0){
                        elem1 = chercherElementListe(l, 0); //Retourne l'élément n
                        elem2 = elem1->nxt; //Retourne l'élément n+1
                        printf("--%d\n", elem1->value);
                        l->first= elem2;
                        l->size--; //Réduction de la taille

                        free(elem1);
                    }
                    else{
                        j = (l->size-n)%l->size;
                        supprimerElementListe(l, j);
                    }
                }
            }
        }
        else{ //La liste ne contient qu'une seule case : on la supprime donc
            elem1 = l->first; //Retourne la seule et unique case
            printf("--%d\n", elem1->value);
            l->first = NULL;
            l->last = NULL;
            l->size = 0;
            free(elem1);
        }
    }
}
```

## Afficher Liste

Algorithme : afficherListe

**Données :**

\*l : Liste  
\*elem : Structure Élément

**Début**

elem ← l->premier;

Ecrire("La liste :") ;

**Tant que** elem != NULL **Faire**

Ecrire(" " + elem->valeur + " ");

elem ← elem->prochain;

**FinTantQue**

Ecrire("Taille : " + l->taille + " - FIN");

**Fin**

```
/*Affiche le contenu de la liste*/
void afficherListe(Liste *l){

    printf("La liste : ");

    Cell* elem = l->first;

    while(elem != NULL){
        printf("%d ", elem->value);
        elem = elem->nxt;
    }

    printf("Taille : %d - FIN\n", l->size);
}
```

## La liste est vide ?

**Algorithme :** listeEstVide

**Données :**

\*l : Liste  
i : Booléen

**Début**

i ← False;

**Si** l != NULL et l->first != NULL **Alors**

i ← True;

}

retourner i;

**Fin**

```
/*Retour 0 ou 1 selon si la liste est vide ou non*/  
int listeEstVide(Liste *l){  
    int i = 1;  
    if(l != NULL && l->first != NULL){  
        i = 0;  
    }  
    return i;  
}
```

## Vider la liste

**Algorithme :** viderListe

**Données :**

*l :	Liste
i :	Booléen

**Début**

**TantQue** l->premier != NULL **Alors**  
    supprimerDernierElementListe(l);  
**FinTantQue**

**Fin**

```
/*Vide complètement la liste*/  
void viderListe(Liste *l){  
    while(l->first != NULL){  
        supprimerDernierElementListe(l);  
    }  
}
```

## Egalité de deux listes

**Algorithme :** estEgaleListe

**Données :**

*l1, *l2 :	Liste
stop :	Booléen
*elem1, *elem2 :	Structure Élément

**Début**

stop ← True ;  
elem1 ← l1->premier;  
elem2 ← l2->premier;  
//On verifie si la taille est différente  
**Si** l1->taille != l2 ->taille **Alors**  
    stop ← False

**FinSi**

**TantQue** elem1 != NULL ET elem2 != NULL ET stop == True **Faire**

**Si** elem1->value != elem2->value **Alors**

        stop ← False

**Sinon**

        elem1 ← elem1->prochain;

        elem2 ← elem2->prochain;

**FinSi**

**FinTantQue**

retourner stop;

**Fin**

```
/*On compare deux listes l1 et l2
**Retour 1 en cas d'égalité, 0 sinon
*/
int estEgaleListe(Liste* l1, Liste* l2){

    Cell* elem1 = l1->first;
    Cell* elem2 = l2->first;
    short int stop = 1;

    /*On verifie si la taille est différente*/
    if(l1->size != l2->size){
        stop = 0;
    }

    while (elem1 != NULL && elem2 != NULL && stop == 1){
        if(elem1->value != elem2->value){
            stop = 0;
        }
        else{
            elem1 = elem1->nxt;
            elem2 = elem2->nxt;
        }
    }

    return stop;
}
```

## Construire une liste binaire à partir d'un tableau

**Algorithme :** constructionListeBinaire

**Données :**     n, i : entiers;  
                  \* v : tableau d'entiers;  
                  \* l : Liste;

**Début**

    //Initialisation

```
l ← creerListe();

//On verifie si la taille est différente
Pour i allant de 1 à n Faire
    ajouterDernierElementListe(l, v(i));
FinPour

//Affichage de la Liste
afficherListe(l);

retourner(l);
```

**Fin**

```
166  /*Construit une liste à partir d'un tableau d'entiers*/
167  Liste * constructionListB(int * v, int n){
168      int i;
169      Liste * l = creerListe();
170
171      for(i=0;i<n;i++){
172          ajouterDernierElementListe(l, v[i]);
173      }
174
175      afficherListe(l);
176
177      return l;
178  }
```

## Incrémentation de 1 une liste binaire

**Algorithme :** incrementationListeBinaire

**Données :** counter : entier;  
\* v : tableau d'entiers;  
\* l : Liste;  
\* c : Cellule;

**Début**

```
//Incrémentation
Si l != NULL ET l → premier != NULL Alors
    counter ← l → taille-2;
    c ← l → dernier;
    TantQue c != NULL ET counter >= 0 ET c → valeur != 0 Faire
        c → valeur ← 0;
```

```
c ← chercherElementListe(l, counter);  
counter ← counter-1;
```

**FinTantQue**

**Si** c != NULL **ET** counter >= 0 **Alors**

```
c → valeur ← 1;
```

**Sinon**

```
ajouterDernierElementListe(l, 0);
```

**FinSi**

**FinSI**

```
//Affichage de la Liste  
afficherListe(l);
```

**Fin**

```
180  /*Incremente de 1 une liste binaire*/  
181  void incrementationListeB(Liste * l){  
182      int counter;  
183      Cell * c;  
184      if(l != NULL && l->first != NULL){  
185          counter = l->size-2;  
186          c = l->last;  
187          while(c != NULL && counter >= 0 && c->value != 0){  
188              c->value = 0;  
189              c = chercherElementListe(l,counter);  
190              counter--;  
191          }  
192  
193          if(c != NULL && counter >= 0){  
194              c->value = 1;  
195          }else{  
196              ajouterDernierElementListe(l, 0);  
197          }  
198      }  
199      afficherListe(l);  
200  }
```

**Exemple liste binaire :**

```
Saisir un nombre en base 10 :50  
v : 1 1 0 0 1 0  
  
Construction de la liste :  
La liste : 1 1 0 0 1 0 Taille : 6 - FIN  
Incrementation de 1 :  
La liste : 1 1 0 0 1 1 Taille : 6 - FIN  
Incrementation de 1 :  
La liste : 1 1 0 1 0 0 Taille : 6 - FIN
```

## Trier une liste

**Algorithme :** trierListe

**Données :**

*l :	Liste
stop:	Booléen
valtmp	Entier
*elem :	Structure Elément Liste

**Début**

stop ← Faux;

**TantQue** stop == Faux **Faire**

elem ← l->premier;

stop ← Vrai;

**TantQue** elem->prochain != NULL) **Faire**

**Si** elem->prochain->valeur < elem->valeur **Alors**

valtmp ← elem->prochain->valeur;

elem->prochain->valeur ← elem->valeur;

elem->valeur ← valtmp;

stop ← Faux;

**FinSi**

elem ← elem->prochain;

**FinTantQue**

**FinTantQue**

**Fin**

```
/*Tri dans l'ordre croissant les valeurs d'une liste*/
void trierListe(Liste *l){

    int valtmp;
    int stop = 0;
    Cell * elem;

    while(stop == 0){
        elem = l->first;
        stop = 1;
        while(elem->nxt != NULL){
            if(elem->nxt->value < elem->value){
                valtmp = elem->nxt->value;
                elem->nxt->value = elem->value;
                elem->value = valtmp;
                stop = 0;
            }
            elem = elem->nxt;
        }
    }
}
```

## Fusionner deux listes triées

**Algorithme :** fusionnerListes

**Données :**

*l1, *l2, *l3 :	Liste
i :	Booléen
*elem1, *elem2:	Structure Element Liste

**Début**

```
l3 ← creerListe();  
elem1 ← l1->premier;  
elem2 ← l2->premier;
```

**TantQue** elem1 != NULL || elem2 != NULL **Faire**

**Si** elem1 != NULL **Alors**

**Si** elem2 != NULL **Alors**

**Si** elem1->valeur < elem2->valeur **Alors**  
                ajouterDernierElementListe(l3, elem1->valeur);  
                elem1 ← elem1->prochain ;

**Sinon**

            ajouterDernierElementListe(l3, elem2->valeur);  
            elem2 ← elem2->prochain;

**FinSi**

**Sinon**

        ajouterDernierElementListe(l3, elem1->valeur);  
        elem1 ← elem1->prochain;

**FinSi**

**Sinon**

        ajouterDernierElementListe(l3, elem2->valeur);  
        elem2 ← elem2->prochain;

**FinSi**

**FinTantQue**

```
return l3;
```

**Fin**



```
Liste* fusionnerListes(Liste *l1, Liste *l2){

    Liste* l3 = creerListe();
    Cell *elem1 = l1->first;
    Cell *elem2 = l2->first;

    while(elem1 != NULL || elem2 != NULL){

        if(elem1 != NULL){
            if(elem2!=NULL){
                if(elem1->value < elem2->value){
                    ajouterDernierElementListe(l3, elem1->value);
                    elem1 = elem1->nxt;
                }
                else{
                    ajouterDernierElementListe(l3, elem2->value);
                    elem2 = elem2->nxt;
                }
            }
            else{
                ajouterDernierElementListe(l3, elem1->value);
                elem1 = elem1->nxt;
            }
        }
        else{
            ajouterDernierElementListe(l3, elem2->value);
            elem2 = elem2->nxt;
        }
    }

    return l3;
}
```

## Extraire une chaine de la liste

**Algorithme :** extraireChaine

**Données :**

*l :	Liste
a,b,c :	Booléen
elem :	Structure Element Liste

**Début**

c ← 0;

**TantQue** elem != NULL **Faire**

```
Si c >= a && c <= b Alors  
    Ecrire(elem->valeur);
```

```
FinSi  
elem ← elem->prochain;  
c ++;
```

FinTantQue

Fin

```
void extraireChaine(Liste *l, int a, int b){  
  
    Cell* elem = l->first;  
    int c = 0;  
  
    while(elem != NULL){  
        if(c >= a && c <= b){  
            printf("%d", elem->value);  
        }  
        elem = elem->nxt;  
        c ++;  
    }  
  
}
```

Exemple d'une liste :

```
Ajoute fin liste v=5 :  
La liste : 5 Taille : 1 - FIN  
Ajoute fin liste v=10 :  
La liste : 5 10 Taille : 2 - FIN  
Ajoute milieu de la liste v=20 :  
La liste : 5 20 10 Taille : 3 - FIN  
Chercher l'element v=10 :  
Trouvé !  
Chercher l'element v=15 :  
Non Trouvé !  
Liste est vide ? 0  
Supprimer element v=5 :  
--5  
La liste : 20 10 Taille : 2 - FIN  
Trier la liste :  
La liste : 10 20 Taille : 2 - FIN  
Supprimer le dernier element :  
--20  
La liste : 10 Taille : 1 - FIN  
Vider la liste :  
--10  
La liste : Taille : 0 - FIN
```

## Algorithmes d'Arbre

**Note :** Les piles utilisées dans les algorithmes ci-dessous ne stockent pas des entiers mais des Nœuds

### Définition des structures

Définir : Structure noeud

Début

```
    valeur :      entier;  
    * pere, fd, fg :  structure noeud;
```

Fin

Nommer nœud\_arbre

Définir Structure Arbre

Début

```
    *racine :      structure noeud;  
    taille :      entier;
```

Fin

Nommer : Arbre

```
/*Liste de taille n, de premier element root*/  
typedef struct Arbre Arbre;  
struct Arbre  
{  
    Node *root;  
    int size;  
};  
  
/*Element de l'arbre de type int*/  
typedef struct Node Node;  
struct Node  
{  
    int value;  
    Node *lft, *rgt, *top;  
};
```

### Créer un Arbre

Algorithme : CreerArbre

Données : \*a : structure Arbre;

Début

```
// Allocation de l'arbre  
Allouer(a) ;  
  
// Initialisation de la racine et de la taille  
a->racine ← NULL ;  
taille ← 0;
```

Fin

```
/*Initialisation d'un arbre
**L'arbre ne contient aucune case
**La taille est de 0
*/
Arbre* creerArbre(){
    Arbre * a = malloc(sizeof(Arbre));
    a->root = NULL;
    a->size = 0;
    return a;
}
```

## L'arbre est-il vide ?

**Algorithme** : estVideArbre

**Données** : \*a : arbre;

**Début**

retourner(a->racine == NULL);

**Fin**

```
/*Initialisation d'un arbre
**L'arbre ne contient aucune case
**La taille est de 0
*/
Arbre* creerArbre(){
    Arbre * a = malloc(sizeof(Arbre));
    a->root = NULL;
    a->size = 0;
    return a;
}
```

## Choix du parcours pour l'affichage

**Algorithme** : Afficher

**Données**

\*a : Structure Arbre  
c : caractère ;

**Début**

**Selon** (c) :

**cas** 'p' :

afficherPrefixe(a->racine) ;

**fincas** ;

**cas** 'i' :

afficherInfine(a->racine) ;

**fincas** ;

**Autrement :**

afficherSuffixe(a->racine) ;  
fincas ;

**FinSelon**

**Fin**

```
void afficher(Arbre *a, char c){  
  
    switch(c){  
        case('p'):  
            afficherPostfixe(a->root);  
            break;  
        case('i'):  
            afficherInfine(a->root);  
            break;  
        default:  
            afficherPrefixe(a->root);  
            break;  
    }  
}
```

## Parcours Prefixé

Algorithme : afficherPrefixe

Données :

\*n : structure nœud ;

**Début**

**Si** n != NULL **Alors**

Ecrire(e->valeur) ;  
afficherPrefixe(e->fg) ;  
afficherPrefixe(e->fd) ;

**FinSi**

**Fin**

## Parcours Infiné

Algorithme : afficherInfine

Données :

\*n : structure nœud ;

**Début**

**Si** n != NULL **Alors**

afficherInfine (e->fg) ;  
Ecrire(e->valeur) ;  
afficherInfine (e->fd) ;

**FinSi**

**Fin**

## Parcours Suffixe

**Algorithme :** afficherSuffixe

**Données :**

\*n : structure nœud ;

**Début**

**Si** n != NULL **Alors**

    afficherSuffixe (e->fg) ;

    afficherSuffixe (e->fd) ;

    Ecrire(e->valeur) ;

**FinSi**

**Fin**

```
void afficherPrefixe(Node *e){
    if(e != NULL){
        printf("%d", e->value);
        afficherPrefixe(e->lft);
        afficherPrefixe(e->rgt);
    }
}

void afficherInfixe(Node *e){
    if(e != NULL){
        afficherInfixe(e->lft);
        printf("%d", e->value);
        afficherInfixe(e->rgt);
    }
}

void afficherPostfixe(Node *e){
    if(e != NULL){
        afficherPostfixe(e->lft);
        afficherPostfixe(e->rgt);
        printf("%d", e->value);
    }
}
```

## Calcul de la hauteur

**Algorithme :** calculHauteur

**Données :**

\*a : structure Arbre

h : entier

**Début**

**Si** NON estVideArbre(a) **Alors**

    h ← parcoursHauteur(a->racine, 0) ;

**FinSi**

retourner h ;

**Fin**

```
int calculHauteur(Node * e){  
    int h = 0;  
    if(e != NULL){  
        h = parcoursHauteur(e, 0);  
    }  
    return h;  
}
```

## Parcours pour le calcul de la hauteur

**Algorithme :** parcoursHauteur

**Données :**

\*n :                structure Noeud  
h, htmp1, htmp :       entiers

**Début**

**Si** n != NULL **Alors**

    htmp1 ← parcoursHauteur(n->fg, h+1) ;

    htmp2 ← parcoursHauteur(n->fd, h+1) ;

**Si** htmp2 > htmp1 **Alors**

        htmp1 ← htmp2 ;

**FinSi**

**FinSi**

**retourner** htmp1 ;

**Fin**

```
int parcoursHauteur(Node *e, int h){  
    int htmp1 = h;  
    int htmp2 = h;  
    if(e != NULL){  
        htmp1 = parcoursHauteur(e->lft, h+1);  
        htmp2 = parcoursHauteur(e->rgt, h+1);  
        if(htmp1 < htmp2){  
            htmp1 = htmp2;  
        }  
    }  
    return htmp1;  
}
```

## L'arbre est-il équilibré ?

**Algorithme :** estEquilibre

**Données :**

\*a :                Structure Arbre  
hg, hd :           entiers

**Début**

```
hg ← 0 ;  
hd ← 0 ;  
Si a->racine->fg != NULL Alors  
    hg ← calculHauteur(a->root->fg, 0) ;  
Finsi  
Si a->racine->fd != NULL Alors  
    hd ← calculHauteur(a->root->fd, 0) ;  
FinSi  
retourner (hd+1==hg | hg+1==hd | hg == hd) ;
```

**Fin**

```
int estEquilibre(Arbre *a){  
    int h1 = 0;  
    int hr = 0;  
  
    if(a->root->lft != NULL){  
        h1 = calculHauteur(a->root->lft);  
    }  
    if(a->root->rgt != NULL){  
        hr = calculHauteur(a->root->rgt);  
    }  
    return (hr+1==h1 | h1+1==hr | hr == hr);  
}
```

## Deux Arbres sont-ils égaux ?

Algorithme : estEgal

Données :

\*a1, \*a2 :                      Structure Arbre

**Début**

**retourner** parcoursEgal(a1->racine, a2->racine, 1);

**Fin**

```
int estEgal(Arbre *a1, Arbre *a2){  
    return parcoursEgal(a1->root, a2->root, 1);  
}
```

## Parcours pour l'égalité

Algorithme : parcoursEgal

Données :

\*n1, \*n2 :                      structures Noeud

i, tmp :                        entiers

**Début**

```
tmp = i ;  
Si n1 != NULL ET n2 != NULL Alors  
    Si n1->valeur != n2->valeur Alors  
        tmp = 0 ;
```



**Sinon**

tmp ← parcoursEgal(n1->fg, n2->fg, tmp);

tmp ← parcoursEgal(n1->fd, n2->fd, tmp);

**FinSi**

**Sinon**

**Si** n1 != n2 **Alors**

tmp ← 0;

**FinSi**

**FinSi**

**retourner** (tmp);

**Fin**

```
int parcoursEgal(Node *e1, Node*e2, int n){
    int n_tmp = n;
    if(e1 != NULL & e1 != NULL){
        if(e1->value != e2->value){
            n_tmp = 0;
        }
        else{
            n_tmp = parcoursEgal(e1->lft, e2->lft, n_tmp);
            n_tmp = parcoursEgal(e1->rgt, e2->rgt, n_tmp);
        }
    }
    else{
        if(e1 != e2){
            n_tmp = 0;
        }
    }
    return n_tmp;
}
```

## Parcours en largeur

Algorithme : parcoursLargeur

Données :

*p1, *p2 :	Pile de Nœuds
done :	Entier
tmp :	Nœud
a :	Arbre

**Début**

done ← 0;

p1 = creerPile();

p2 = creerPile();

empilerPile(p1, a->racine);

**TantQue** done != 1 **Faire**

done ← 1

//depiler et stocker nouveaux fils dans une pile

```
TantQue sommet(p1) != NULL Faire
    empilerPile(p2, sommet(p1)->valeur->fg) ;
    Si (sommet(p2) != NULL) Alors
        done ← 0 ;
        Ecrire(sommet(p2)->valeur->valeur) ;
    Sinon
        Ecrire('[]') ;
    FinSi
    Si
        empilerPile(p2, sommet(p1)->valeur->fd) ;
    Si (sommet(p2) != NULL) Alors
        done ← 0 ;
        Ecrire(sommet(p2)->valeur->valeur) ;
    Sinon
        Ecrire('[]') ;
    FinSi
    depilerPile(p1) ;
FinTantQue
    //répéter tant que pile pas vide au départ (= pas de fils). Stockage des null, donc arrêt quand un
parcours de pile = que des null
    p1 = p2 ;
    viderPile(p2) ;
FinTantQue
    viderPile(p1);
    viderPile(p2);
    libérer(p1);
    libérer(p2);
Fin
```

```
int parcouresLargeur(Arbre *a){

    int done = 0;
    Pile *p1 = creerPile();
    Pile *p2 = creerPile();
    Node * tmp;

    //prendre racine
    empilerPile(p1, a->root);

    while(done != 1){
        done = 1;
        //depiler et stocker nouveaux fils dans une pile
        while(p1->first != NULL){

            empilerPile(p2, sommet(p1)->value->lft);
            tmp = sommet(p1)->value->lft;

            if(tmp != NULL){
                done = 0;
                printf("%d", sommet(p2)->value->value);
            }
            else{
                printf("[ ]");
            }

            empilerPile(p2, sommet(p1)->value->rgt);
            tmp = sommet(p1)->value->rgt;

            if(tmp != NULL){
                done = 0;
                printf("%d", sommet(p2)->value->value);
            }
            else{
                printf("[ ]");
            }
            depilerPile(p1);
        }
        //répéter tant que pile pas vide au départ (= pas de fils)
        p1 = p2;
        viderPile(p2);
    }
}
```

## Vida(n)ge de l'Arbre

Algorithme : estVide

Données :

\*a:                    Structure Arbre

**Début**

**Si** a != NULL ET a->racine != NULL **Alors**  
    vidage(a->racine);

**FinSi**

a->racine ← NULL;

a->size ← 0;

**Fin**

## Vidage (récuratif)

Algorithme : vidange

Données :

\*e:                    Structure Noeud

**Début**

**Si** e != NULL **Alors**

    vidange(e->fg);

    vidange(e->fd);

    e->fg ← NULL;

    e->fd ← NULL;

    e->top ← NULL;

    libérer(e);

**FinSi**

**Fin**

```
void vidage(Node *e){
    if(e != NULL){
        vidage(e->lft);
        vidage(e->rgt);
        e->lft = NULL;
        e->rgt = NULL;
        e->top = NULL;
        free(e);
    }
}

void viderArbre(Arbre *a){
    if(a != NULL){
        vidage(a->root);
    }
    a->root = NULL;
    a->size = 0;
}
```

## Ajout Logique

**Algorithme :** ajoutLogiqueArbre

**Données :**

\*e, \*ne :                    Structure Noeud  
v :                            entier

**Début**

//On peut ajouter depuis la racine ou de n'importe quel nœud. On suppose l'Arbre dont il fait partie non-vide.

**Si** e->fg == NULL **Alors** //Notre nœud est vide à gauche

    allouer(ne); //Allocation du nouvel élément

    ne->valeur ← v;

    ne->top ← e;

    e->fg ← ne;

**Sinon**

**Si** e->fd == NULL **Alors**

        allouer(ne)

        ne->valeur ← v;

        ne->top ← e;

        e->fg ← ne;

**Sinon**

**Si** e->fg->valeur >= e->fd->valeur **Alors**

            ajoutLogiqueArbre(e->fg, v);

**Sinon**

            ajoutLogiqueArbre(e->fd, v);

**FinSi**

**FinSi**

**FinSi**

**Fin**

```
int ajoutLogiqueArbre(Node *elem, int v){

    if(elem->lft == NULL){
        Node *new_elem = malloc(sizeof(Node));
        new_elem->value = v;
        new_elem->top = elem;
        elem->lft = new_elem;
    }
    else{
        if(elem->rft == NULL){
            Node *new_elem = malloc(sizeof(Node));
            new_elem->value = v;
            new_elem->top = elem;
            elem->rft = new_elem;
        }
        else{
            if(elem->lft->value >= elem->rft->value){
                ajoutLogiqueArbre(elem->lft, v);
            }
            else{
                ajoutLogiqueArbre(elem->rft, v);
            }
        }
    }
}
```

## Ajout en Largeur

L'Arbre est supposé équilibré

**Algorithme :** ajoutLargeur

**Données :**

*p1, *p2 :	Pile de Nœuds
done, v :	Entier
tmp, b :	Nœud
a :	Arbre

**Début**

```
done ← 0 ;
p1 = creerPile() ;
p2 = creerPile() ;
empilerPile(p1, a->racine) ;
allouer(b) ;
b->valeur ← v ;
```

**TantQue** done != 1 **Faire**

```
done ← 1
```

```
//depiler et stocker nouveaux fils dans une pile
```

**TantQue** sommet(p1) != NULL **Faire**

```
empilerPile(p2, sommet(p1)->valeur->fg) ;
```

```
    Si (sommet(p2) != NULL) Alors
        done ← 0 ;
    Sinon
        sommet(p1)->valeur->fg ← b;
        b->top ← sommet(p1)->valeur;
    FinSi
    Si
        empilerPile(p2, sommet(p1)->valeur->fd) ;
    Si (sommet(p2) != NULL) Alors
        done ← 0 ;
    Sinon
        sommet(p1)->valeur->fd ← b;
        b->top ← sommet(p1)->valeur;
    FinSi
    depilerPile(p1) ;
FinTantQue
//répéter tant que pile pas vide au départ (= pas de fils). Stockage des null, donc arrêt quand un
parcours de pile = que des null
    p1 ← p2 ;
    viderPile(p2) ;
FinTantQue
viderPile(p1);
viderPile(p2);
libérer(p1);
libérer(p2) ;
Fin
```

```
int ajoutLargeur(Arbre *a, int v){

    Pile *p1 = creerPile();
    Pile *p2 = creerPile();
    Node * tmp;
    Node * b = malloc(sizeof(Node));
    b->lft = NULL;
    b->rgt = NULL;
    b->top = NULL;
    b->value = v;

    int done = 0;
    if(a->root != NULL){
        //prendre racine
        empilerPile(p1, a->root);

        while(done != 1){
            done = 1;
            //depiler et stocker nouveaux fils dans une pile
            while(p1->first != NULL){

                empilerPile(p2, sommet(p1)->value->lft);
                tmp = sommet(p1)->value->lft;

                if(tmp != NULL){
                    done = 0;
                }
                else{
                    sommet(p1)->value->lft = b;
                    b->top = sommet(p1)->value;
                    done=1;
                    break;
                }

                empilerPile(p2, sommet(p1)->value->rgt);
                tmp = sommet(p1)->value->rgt;

                if(tmp != NULL){
                    done = 0;
                }
                else{
                    sommet(p1)->value->rgt = b;
                    b->top = sommet(p1)->value;
                    done=1;
                    break;
                }

                depilerPile(p1);
            }
            //répéter tant que pile pas vide au départ
            DeplacementPile(p1,p2);
            p2->first = NULL;
        }

        viderPile(p1);
        viderPile(p2);
        free(p1);
        free(p2);
    }
    else{
        a->root = b;
    }
}
```



## Equilibrage de l'arbre

**Algorithme :** equilibrageArbre

<b>Données :</b>	*a :	Structure Arbre
	*p1, *p2, *p3 :	Structure Pile
	*tmp :	structure Nœud
	done :	Booléen

### Début

```
p1 = creerPile();  
p2 = creerPile();  
p3 = creerPile();  
done ← Faux.  
empilerPile(p3, a->racine) ;
```

```
//prendre racine  
empilerPile(p1, a->racine);
```

### TantQue done == Faux Faire

```
done = Vrai
```

```
//depiler et stocker nouveaux fils dans une pile
```

#### TantQue sommet(p1) != NULL Faire

```
empilerPile(p2, sommet(p1)->valeur->fg) ;  
tmp = sommet(p1)->valeur->fg ;
```

#### Si tmp != NULL Alors

```
done ← Faux;  
empilerPile(p3, sommet(p1)->valeur->fg);
```

#### FinSi

```
empilerPile(p2, sommet(p1)->valeur->fd);  
tmp ← sommet(p1)->valeur->fd;
```

#### Si tmp != NULL Alors

```
done ← Faux;  
empilerPile(p3, sommet(p1)->valeur->fd);
```

#### FinSi

```
depilerPile(p1);
```

#### FinTantQue

```
//répéter tant que pile pas vide au départ (= pas de fils). Stockage des null, donc arrêt quand un  
parcours de pile = que des null
```

```
p1 ← p2;  
viderPile(p2);
```

### FinTantQue

```
vidage(a->racine);  
a->racine ← depilerPile(p3);
```

```
TantQue sommet(p3) != NULL Faire  
    ajoutLargeur(a, depilerPile(p3)->valeur);
```

```
FinTantQue
```

```
viderPile(p1);  
viderPile(p2);  
viderPile(p3);  
liberer(p1);  
liberer(p2);  
liberer (p3);
```

```
Fin
```

```
int afficheLargeur(Arbre *a){

    Pile *p1 = creerPile();
    Pile *p2 = creerPile();
    Node * tmp;
    int i, h = calculHauteur(a->root);
    h++;

    //prendre racine
    if(a != NULL && a->root != NULL){
        empilerPile(p1, a->root);
        for(i=0;i<(3*h+4);i++){printf("_");}
        printf("%d", a->root->value);
        printf("\n");
        while(p1->first != NULL){
            h--;
            for(i=0;i<(h);i++){printf("__");}
            //depiler et stocker nouveaux fils dans une pile
            while(p1->first != NULL){

                tmp = sommet(p1)->value->lft;

                if(tmp != NULL){
                    empilerPile(p2, sommet(p1)->value->lft);
                    for(i=0;i<(h);i++){printf("_");}
                    printf("%d", sommet(p2)->value->value);
                }
                else{
                    printf("_[]");
                }

                tmp = sommet(p1)->value->rgt;

                if(tmp != NULL){
                    empilerPile(p2, sommet(p1)->value->rgt);
                    for(i=0;i<(h);i++){printf("_");}
                    printf("%d", sommet(p2)->value->value);
                }
                else{
                    printf("_[]");
                }

                depilerPile(p1);
            }
            printf("\n");
            //répéter tant que pile pas vide au départ (= pas de fils).
            DeplacementPile(p1,p2);
            p2->first = NULL;
        }
    }
    printf("\n");

    viderPile(p1);
    viderPile(p2);
    free(p1);
    free(p2);
}
```

Exemple d'Arbre :

```
Saisir n : 10
Ajout en largeur v=1
Ajout en largeur v=2
Ajout en largeur v=3
Ajout en largeur v=4
Ajout en largeur v=5
Ajout en largeur v=6
Ajout en largeur v=7
Ajout en largeur v=8
Ajout en largeur v=9
Ajout logique v=10
Affichage Prefixe :
12489536770
Affichage Infixe :
84925163707
Affichage Posfixe :
89452670731
Affichage en largeur :

          1
        2   3
       4 5 6 7
      8 9 N N N N 70 N
     NN NN NN

Hauteur = 4
```

# Algorithmes de Huffman

Dans cette partie, nous traiterons différemment l'algorithme d'Huffman par rapport au cours (Cf TP et Mail envoyés). Nous n'utiliserons ainsi pas de forêt, ce qui nous permettra d'être plus efficace et d'avoir un arbre de priorité pouvant être dynamiquement parcouru.

## Définition des structures

**Définir :** Structure données

**Début**

symbole :                caractère  
poids :                entier

**Fin**

**Nommer** Données

```
/*Un element pour définir une priorité dans l'arbre d'Huffman*/
typedef struct Donnee Donnee;
struct Donnee
{
    char symbol;
    int poids;
};
```

## Recherche du maximum

Recherche la priorité maximale dans la table de priorité. Retourne le symbole correspondant et place le symbole à la fin. Il ne sera plus visité (n est diminué de 1 au prochain appel)

**Algorithme :** rechercherMax

**Données :**

\*p:                Tableau de données  
tmp :            Structure données  
n, i, imax ;            entiers

**Début**

i, imax  $\leftarrow$  0 ;

//n est réduit n à chaque appel

**TantQue** i < n **Faire**

    //On stocke l'indice où la valeur est maximale.

**Si** p[i].poids >= p[imax].poids **Alors**  
        imax  $\leftarrow$  i;

**FinSi**

    i  $\leftarrow$  i+1

**FinTantQue**

    //On échange les valeurs entre la case où la priorité est maximale et la dernière case.

```
tmp ← p[imax];  
p[imax] ← p[n];  
p[n] ← tmp;
```

**retourner** tmp.symbol; //On retourne le symbole

**Fin**

```
/*Recherche la priorité maximale dans la table de priorité.  
**Retourne le symbole correspondant et place le symbole à la fin.  
**Il ne sera plus visité (n est diminué de 1 au prochain appel*/  
char rechercherMax(Donnee *p, int n){  
  
    int i = 0;  
    int imax = 0;  
    Donnee tmp;  
  
    //n est réduit n à chaque appel  
    while(i < n){  
        //On stocke l'indice où la valeur est maximale.  
        if(p[i].poids >= p[imax].poids){  
            imax = i;  
        }  
        i++;  
    }  
  
    //On échange les valeurs entre la case où la priorité est maximale et la dernière case.  
    tmp = p[imax];  
    p[imax] = p[n];  
    p[n] = tmp;  
  
    return tmp.symbol; //On retourne le symbole  
}
```

## Transformation de la table de priorité

Transforme un tableau de priorité en un arbre d'Huffman. On crée l'arbre à partir du haut, donc on recherche la priorité maximale, on place son symbole sur le fils droit, puis on place notre pointeur de parcours sur le fils gauche. Si on est à la fin de la table, on place directement sur le dernier fils gauche.

**Algorithme :** transformerPrio

**Données :**

*a:	Structure Arbre
*b, *d :	Structure Nœud d'Arbre
n, nn :	Entiers
c :	Caractères

**Début**

```
allouer(a);  
allouer(a->racine);
```

d ← a->racine ; //On commence au sommet de l'arbre

**TantQue**  $nn \neq 0$  **Faire**

$c \leftarrow \text{rechercherMax}(p, nn);$

$nn \leftarrow nn - 1;$  //A chaque parcours du tableau, on réduit de 1 (la valeur à la case n a été traitée)

$\text{allouer}(b);$

$b \rightarrow \text{valeur} \leftarrow c;$

$d \rightarrow fd \leftarrow b;$

**Si**  $n \neq 0$  **Alors** //On est à la fin de la création de l'arbre, on place donc le dernier symbole dans le dernier fils gauche.

$\text{allouer}(d \rightarrow fg);$

$d \leftarrow d \rightarrow fg;$

**Sinon**

$d \rightarrow \text{valeur} \leftarrow c;$

**FinTantQue**

**retourner**  $a;$  //On retourne l'Arbre d'Huffman

**Fin**

```
/*Transforme un tableau de priorité en un arbre d'Huffman
**On crée l'arbre à partir du haut, donc on recherche la priorité maximale, on place son symbole sur le fils droit, puis c
**Si on est à la fin de la table, on place directement sur le dernier fils gauche*/
Arbre * transformerPrio(Donnee *p, int n){

    Arbre *a = malloc(sizeof(Arbre));
    Node* b, *d;
    int nn = n;//On ne peut pas modifier directement des paramètres de fonction
    char c;

    a->root = malloc(sizeof(Node));
    d = a->root;//On commence au sommet de l'arbre

    while(nn != 0){

        c = rechercherMax(p, nn);
        nn = nn - 1;//A chaque parcours du tableau, on réduit de 1 (la valeur à la case n a été traitée)

        b = malloc(sizeof(Node));

        b->value = c;
        d->rgt = b;

        if(nn != 0){//On est à la fin de la création de l'arbre, on place donc le dernier symbole dans le dernier
            d->lft = malloc(sizeof(Node));
            d = d->lft;
        }
        else{
            d->value = c;
        }
    }

    return a;//On retourne l'Arbre d'Huffman

}
```

## Décodage de Huffman

Grace à l'arbre de décodage obtenu à l'aide la table de priorité, on décode une chaine de caractères composée de 0 et de 1 (binaire). On parcourt dynamique l'arbre.

**Algorithme :** decodageHuffman

### Données :

*p:	Table de Données
*s :	Chaine de caractères
i, n, nn :	Entiers
*a :	Structure Arbre
*b :	Structure Noeud d'Arbre

### Début

i = 0;



/\*Transformer prio retourne l'arbre d'Huffman à partir de la table de priorités\*/

\* a  $\leftarrow$  transformerPrio(p, nn);

Node \*b  $\leftarrow$  a->racine;

/\*On parcourt la chaine de caractère en caractère.\*/

**Pour i allant de 0 à n Faire**

**Si** s[i] == '1' **Alors**

        Ecrire(b->rgt->value);

        b  $\leftarrow$  a->racine;

**Sinon**

        //Si on n'a pas 1, on passe au caractère suivant dans l'arbre de priorité d'Huffman

**Si** b->fg == NULL) **Alors**

            Ecrire(b->valeur) ;

            b  $\leftarrow$  a->racine ;

**Sinon**

            b  $\leftarrow$  b->fg;

**FinSi**

**FinSi**

**FinPour**

**Fin**

```
/*Decode une chaîne de caractères de longueur n*/
void decodage(Donnee * p, char * s, int n, int nn){
    int i = 0;

    /*Transformer prio retourne l'arbre d'Huffman à partir de la table de priorités*/
    Arbre* a = transformerPrio(p, nn);

    Node *b = a->root;

    /*On parcourt la chaîne de caractère en caractère.*/
    for(i=0;i<n;i++){
        if(s[i] == '1'){
            printf("%c", b->rgt->value);
            b = a->root;
        }
        else{//Si on n'a pas 1, on passe au caractère suivant dans l'arbre de priorité d'Huffman
            if(b->lft == NULL){
                printf("%c", b->value);
                b = a->root;
            }
            else{
                b = b->lft;
            }
        }
    }
    printf("\n");
}
```

## Encodage de Huffman

Grace à l'arbre obtenu à l'aide la table de priorité, on encode une chaîne de caractères en une chaîne composée de 0 et de 1 (binaire). On parcourt dynamique l'arbre d'Huffman.

**Algorithme :** encodageHuffman

### Données :

*p:	Table de Données
*s :	Chaîne de caractères
i, n, nn, j :	Entiers
*a :	Structure Arbre
*b :	Structure Noeud d'Arbre

### Début

```
/*Transformer prio retourner l'arbre d'Huffman à partir de la table de priorités*/
a ← transformerPrio(p, nn);
b ← a->racine;
j ← 0;
```

```
Pour i allant de 0 à n Faire
    //On regarde le fils de gauche. S'il est nul, on est arrivé tout en bas à gauche de l'arbre, donc au dernier
    //symbole. Si on a la valeur sur le fils droit, on est arrivé au bon caractère
    //j permet de compter le nombre de zéros à insérer
    TantQue b->fg != NULL && b->fd->valeur != s[i]
        b ← b->fg;
        j ← j+1 ;
    FinTantQue

    Si b->fg == NULL //On n'affiche que des 0
        TantQue j != 0 Faire
            Ecrire('0') ;
            j ← j - 1 ;
        FinTantQue
    Sinon //On affiche 1 zéro par j et le 1 une fois tous les 0 insérés
        TantQue j != 0 Faire
            Ecrire('0') ;
            j ← j - 1 ;
        FinTantQue
        Ecrire('1') ;
    FinSi
    b ← a->racine
    j ← 0;
FinPour

Fin
```

```
/*Decode une chaîne de caractères de longueur n*/
void encodage(Donnee * p, char * s, int n, int nn){
    int i;

    /*Transformer prio retourner l'arbre d'Huffman à partir de la table de priorités*/
    Arbre* a = transformerPrio(p, nn);

    Node *b = a->root;
    int j = 0;

    for(i=0;i<n;i++){
        //On regarde le fils de gauche. S'il est nul, on est arrivé tout en bas à gauche de l'arbre, donc au dernier
        //j permet de compter le nombre de zéros à insérer
        while(b->lft != NULL && b->rgt->value != s[i]){
            b = b->lft;
            j++;
        }
        if(b->lft == NULL){//On n'affiche que des 0
            while(j != 0){
                printf("0");
                j--;
            }
        }
        else{//On affiche 1 zéro par j et le 1 une fois tous les 0 insérés
            while(j != 0){
                printf("0");
                j--;
            }
            printf("1");
        }
        b = a->root;
        j = 0;
    }
    printf("\n");
}
```

## Algorithmes d'Arbre R&N

### Vérifier si un arbre R&N respecte les règles

**Algorithme :** EstRougeEtNoir

**Données :**     n, hn, h : entiers;  
                  test : booleen;  
                  \* a : Arbre;  
                  \* n : Nœud;  
                  hn  $\leftarrow$  -1;  
                  h  $\leftarrow$  0;  
                  n  $\leftarrow$  a  $\rightarrow$  root;

#### Début

```
test=vrai;
// Test Rouge Et Noir
Si a  $\rightarrow$  racine  $\rightarrow$  valeur = rouge ET a  $\rightarrow$  racine  $\rightarrow$  fd = NULL OU a  $\rightarrow$  racine  $\rightarrow$  fg = NULL OU a  $\rightarrow$  racine  $\rightarrow$  fg  $\rightarrow$ 
valeur = rouge OU a  $\rightarrow$  racine  $\rightarrow$  fd  $\rightarrow$  valeur = rouge Alors
    test=faux;
Sinon Si a  $\rightarrow$  racine  $\rightarrow$  fd = NULL ET a  $\rightarrow$  racine  $\rightarrow$  fg = NULL Alors
    test=faux;
Sinon
    // Test les branches Rouge Et Noir
    Debut
        Si n  $\rightarrow$  valeur  $\rightarrow$  noir Alors
            h  $\leftarrow$  h+1;
            Si n  $\rightarrow$  fg = NULL ET n  $\rightarrow$  fd = NULL Alors
                Si  $\rightarrow$  hn = -1 Alors
                    hn  $\leftarrow$  h;
                    retourner vrai;
                Sinon Si hn != h Alors
                    FinSi
                FinSi
            Sinon
                EstRougeEtNoir(n->fg);
                EstRougeEtNoir(n->fd);
            FinSi
            h  $\leftarrow$  h-1;
        FinSi
    Fin
FinSi
FinSi
retourner(test);
Fin
```