UNIVERSITÉ
DE REIMS
CHAMPAGNE-ARDENNE

Algorithmique

Info0401

CARUSO Alexis
LEGENDRE Paul
2016/2017 - S4F3

Table des matières

Présentation

Dans le cadre du module d'INFO0401 (Algorithmique), nous devons rédiger un rapport contenant les algorithmes vus en cours, un exemple d'exécution pour chacun, leur complexité et le code qui leur est associé (language C).
Ces algorithmes représentent des thèmes multiples (Arithmétique, Tableau, Pile, Arbre, Graphe, ...). Les codes ont étés développés pour la plupart en C sauf pour quelques un qui ont étés rédigés en C++. Nous tenons à souligner que vous n'avons pas mis toutes les complexités pour éviter de trop grosses erreurs.
Un travail continu et régulier a été nécessaire afin d'aboutir à un rapport le plus complet possible afin de répondre aux attentes de notre professeur.
Le rapport ici présent a été rédigé en LaTeX par CARUSO Alexis et LEGENDRE Paul (L2 Informatique S4F3).

1 Arithmétique

1.1 Nombre parfait

1.1.1 Présentation

Un nombre parfait est un entier naturel qui est égal à la somme de ses diviseurs stricts. Du fait de cette règle, les nombres parfaits sont assez rare. Les 4 premiers nombres parfait sont : 6, 28, 496 et 8128.
L'algorithme permet de vérifier si un nombre donné est bien un nombre parfait.

1.1.2 Exemple d'exécution

On va choisir deux nombres, l'un parfait et l'autre non.
Prenons d'abord 10. Les diviseurs positifs de 10 sont 1, 2 et 5. En faisant la somme de ces diviseurs, on obtient : 1 + 2 + 5 = 8. Or, 8 n'est pas égal à 10 donc 10 n'est pas un nombre parfait.
Prenons ensuite 6. Les diviseurs positifs de 6 sont 1, 2 et 3. En faisant la somme de ces diviseurs, on obtient : 1 + 2 + 3 = 6. En obtenant 6, on peut en conclure que 6 est bien un nombre parfait.

1.1.3 Algorithme

```
Algorithme : Nombre parfait
Variables : n, i, somme : entiers
Debut
|  Ecrire : Entrer un nombre :
|  Lire : n
|  somme ← 0
|  Pour i allant de 1 à n/2 faire
|  |  Si n % i = 0 alors
|  |  |  somme ← somme + i
|  |  FinSi
|  FinPour
|  Si n = somme alors
|  |  Ecrire : Nombre parfait
|  Sinon
|  |  Ecrire : Pas un nombre parfait
|  FinSi
Fin
```

1.1.4 Complexité

$$\left.\begin{array}{l} Affectation : O(1+n/2) \\ Calcul : O(n/2) \\ Ecriture : O(2) \end{array}\right\} \rightarrow O(n/2) : max(O(1+n/2)/O(n/2)/O(2))$$

1.1.5 Code en C

```c
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv[])
{
    int somme=0;
    int i, n;
    printf("Saisir un nombre entier positif : ");
    scanf("%d", &n);
    for(i=1; i<=(n/2); i++)
    {
        if(n%i==0)
        {
            somme = somme+i;
        }
    }
    if(n!=somme || n==0)
    {
        printf("%d n'est pas un nombre parfait. \n", n);
    }
    else
    {
        printf("%d est un nombre parfait. \n", n);
    }
    return 0;
}
```

1.2 Fibonacci

1.2.1 Présentation

La suite de Fibonacci est une suite d'entier défini comme ceci :

$$\begin{cases} u_0 = 0 \\ u_1 = 1 \\ u_n = u_{n-1} + u_{n-2}, \forall n \geq 2 \end{cases}$$

Les deux algorithmes proposés permettent d'afficher les éléments de la suite de Fibonacci de deux manières différentes. Le premier utilise un tableau d'entier tandis que le second utilise 3 variables temporaires.

1.2.2 Exemple d'exécution

Prenons Fib(), une fonction définissant la suite de Fibonacci.
Fib(0) = 0 et Fib(1) = 1
Fib(2) = Fib(0) + Fib(1) = 0 + 1 = 1
Fib(3) = Fib(1) + Fib(2) = 1 + 1 = 2

Fib(4) = Fib(2) + Fib(3) = 1 + 2 = 3
Fib(5) = Fib(3) + Fib(4) = 2 + 3 = 5
Fib(6) = Fib(4) + Fib(5) = 3 + 5 = 8
...

1.2.3 Algorithme

Algorithme : Fibonacci avec tableau
Variables : n, i : entiers
tab : tableau d'entiers
Debut
    $tab[1] \leftarrow 0$
    $tab[2] \leftarrow 1$
    Lire : n
    Ecrire : $tab[1]$
    Ecrire : $tab[2]$
    Pour *i allant de 3 à n* **faire**
        $tab[i] \leftarrow tab[i-1] + tab[i-2]$
        Ecrire : $tab[i]$
    FinPour
Fin

Algorithme : Fibonacci avec 3 variables
Variables : a, b, c, i, n : entiers
Debut
    $a \leftarrow 0$
    Ecrire : a
    $b \leftarrow 1$
    Ecrire : b
    Pour *i allant de 2 à n* **faire**
        $c \leftarrow a + b$
        Ecrire : c
        $a \leftarrow b$
        $b \leftarrow c$
    FinPour
Fin

1.2.4 Complexité

Nous utilisons l'algorithme de Fibonacci avec variables pour calculer la complexité.

$$\left.\begin{array}{l} Affectation : O(2+(n-2)) \\ Calcul : O(n-2) \\ Ecriture : O(2+(n-2)) \end{array}\right\} \rightarrow O(n) : max(O(2+n-2)/O(n-2)/O(2+n-2))$$

1.2.5 Code en C

```c
#include <stdlib.h>
```

```c
#include <stdio.h>

int main (void)
{
   int i, taille;
   printf("Saisir un nombre entier positif : ");
   scanf("%d", &taille);
   int F[taille];
   F[0]=0;
   printf("%d \n", F[0]);
   F[1]=1;
   printf("%d \n", F[1]);

   for(i=2;i<taille;i++)
   {
      F[i] = F[i-1]+F[i-2];
      printf("%d \n", F[i]);
   }
   return 0;
}
```

```c
#include <stdlib.h>
#include <stdio.h>

int main (void)
{
   int i, taille;
   printf("Saisir un nombre entier positif : ");
   scanf("%d", &taille);
   int a = 0;
   int b = 1;
   int c, d;
   printf("%d \n", a);
   printf("%d \n", b);
   for (i=2;i<taille;i++)
   {
      c=a+b;
      printf("%d \n", c);
      a=b;
      b=c;
   }
   return 0;
}
```

1.3 Ackermann récursif

1.3.1 Présentation

La fonction d'Ackermann est définie récursivement comme cela :

$$A(m,n) = \begin{cases} n+1 & \text{si } m=0 \\ A(m-1,1) & \text{si } m>0 \text{ et } n=0 \\ A(m-1,A(m,n-1)) & \text{si } m>0 \text{ et } n>0. \end{cases}$$

L'algorithme ci-dessous est l'algorithme récursif de la fonction d'Ackermann.

1.3.2 Exemple d'exécution

Prenons Ack(), une fonction définissant la suite d'Ackermann.
Calculons par exemple Ack(1, 2) :
Ack(1, 2) = Ack(0, Ack(1, 1))
Ack(1, 2) = Ack(0, Ack(0, Ack(1, 0)))
Ack(1, 2) = Ack(0, Ack(0, Ack(0, 1)))
Ack(1, 2) = Ack(0, Ack(0, 2))
Ack(1, 2) = Ack(0, 3)
Ack(1, 2) = 4

1.3.3 Algorithme

Algorithme : Ackermann (entier m, entier n)
Debut
  Si m = *0* **alors**
    Retourner : n+1
  FinSi
  Sinon si n=*0* **alors**
    Retourner : Ackermann(m-1, 1)
  FinSi
  Sinon
    Retourner : Ackermann(m-1, Ackermann(m, n-1))
  FinSi
Fin

1.3.4 Complexité

$$\left.\begin{array}{l} Affectation : O(3n+2) \\ Calcul : O(0) \\ Ecriture : O(0) \end{array}\right\} \rightarrow O(3n) : max(O(3n)/O(0)/O(0))$$

1.3.5 Code en C

```c
#include <stdlib.h>
#include <stdio.h>

int Ackermann(int m, int n)
```

```c
{
   if (m==0)
   {
      return n+1;
   }
   else if (n==0)
   {
      return Ackermann(m-1, 1);
   }
   else
   {
      return Ackermann(m-1, Ackermann(m, n-1));
   }
}

int main (void)
{
   int m, n;
   printf("Saisir m : ");
   scanf("%d", &m);
   printf("Saisir n : ");
   scanf("%d", &n);
   int val = Ackermann(m,n);
   printf("Resultat : %d \n", val);
   return 0;
}
```

2 Tableau

Les codes concernant les tableaux ont pour la plupart étés testés avec ces fonctions :

```
void remplir(int* T, int t) {
   int i;
   for(i=0;i<t;i++)
   {
      printf("Valeur numero %d : ", i);
      scanf("%d", &T[i]);
   }
   printf("\n");
}

void remplirA(int* T, int t) {
   int i;
   printf("Remplissage aleatoire du tableau \n");
   srand(time(NULL));
   for(i=0;i<t;i++)
   {
      T[i]=(rand()%10);
   }
}

void afficher(int *T, int t) {
   int i;
   for(i=0;i<t; i++)
   {
      printf("%d \t", T[i]);
   }
   printf("\n");
}
```

2.1 Recherche d'un élément X

2.1.1 Présentation

L'algorithme ci-dessous permet de rechercher n'importe quel élément X dans un tableau d'une seule dimension à l'aide d'une variable de type booléen.

2.1.2 Exemple d'exécution

Prenons le tableau T=[2, 4, 6, 8, 10].
Nous voulons chercher 6 dans ce tableau.
L'algorithme parcourt le tableau tant que 6 n'est pas trouvé ou que le tableau ne soit pas entièrement parcouru.
2 != 6, on ne s'arrête pas.
4 != 6, on ne s'arrête pas.
6 = 6, on s'arrête car l'entier que l'on cherchait est trouvé.

2.1.3 Algorithme

Algorithme : Recherche de X dans T

Variables :
i, n : entiers
T : tableau d'entiers
$trouver$: booléen
Debut
  $trouver \leftarrow$ faux
  Pour *i allant de 1 à n* **faire**
    Si $T[i] = X$ **alors**
      $trouver \leftarrow$ vrai
    FinSi
  FinPour
  Retourner : $trouver$
Fin

2.1.4 Complexité

$$\left.\begin{array}{l} Affectation : O(0) \\ Calcul : O(n) \\ Ecriture : O(0) \end{array}\right\} \rightarrow O(nlogn) : max(O(0), O(n), O(0))$$

2.1.5 Code en C

```c
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv[])
{
   int *T;
   int i=0, n, X;
   int trouver = 1;

   printf("Saisir la dimension du tableau : ");
   scanf("%d",&n);
   T = (int*)malloc(n*sizeof(int));
   remplir(T, n);

   printf("Quelle valeur faut-il chercher ? ");
   scanf("%d",&X);

   while(T[i]!=X && i<n)
   {
      i++;
   }
   if(T[i]==X)
      {
```

```
      trouver = 0;
    }
  if (trouver == 1)
  {
    printf("%d n'est pas dans le tableau. \n", X);
  }
  else
  {
    printf("%d est dans le tableau. \n", X);
  }
  return 0;
}
```

2.2 Suppression d'un élément X

2.2.1 Présentation

L'algorithme présent permet de supprimer l'élément voulu dans un tableau donné. Le principe est très simple, il suffit de trouver la valeur à supprimer puis la remplacer par la valeur suivante avant de diminuer la taille de 1.

2.2.2 Exemple d'exécution

Prenons le tableau T=[5, 10, 15, 20, 25]
Nous souhaitons supprimer la valeur 15.
On parcourt le tableau afin de trouver cette fameuse valeur 15.
5 != 15, on continue.
10 != 15, on continue.
15 = 15, on s'arrête !
La troisième valeur est alors remplacée par la suivante (20). On fait pareil pour les valeurs suivantes et il ne reste plus qu'à décrémenter la taille.

2.2.3 Algorithme

Algorithme : Suppression de X dans T
Variables : X, N : entiers (N = taille du tableau)
T : Tableau
p, q : pointeurs d'entiers
Debut
  Ecrire : Valeur à supprimer :
  Lire : X
  $p \leftarrow T$
  $q \leftarrow T$
  TantQue $p\,!= NULL$ **faire**
    $p \leftarrow q$
    Si $p\,!= X$ **alors**
      $p \leftarrow p + 1$
    Sinon
      $N \leftarrow N$ - 1
    FinSi
    $q \leftarrow q + 1$
  FinTantQue
Fin

2.2.4 Complexité

$$\left.\begin{array}{l} Affectation : O(2n+2) \\ Calcul : O(0) \\ Ecriture : O(0) \end{array}\right\} \rightarrow O(2n) : max(O(2n+2), O(0), O(0))$$

2.2.5 Code en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
   int *p, *q, i, j, X;
   int taille;
   int *t;
   srand(time(NULL));
   printf("Saisir la dimension du tableau : ");
   scanf("%d",&taille);
   t = (int*) malloc(taille * sizeof(int));
   for(i = 0; i < taille; i++)
   {
      t[i] = rand() % 100;
   }
   afficher(t, taille);
```

```c
    printf("Saisir la valeur a supprimer : ");
    scanf("%d", &X);
    for(i=0; i<taille; i++ )
    {
        if( t[i]==X)
        {
            for(j=i; j < taille; j++)
            {
                if( j < taille-1)
                {
                    t[j] = t[j+1];
                }
                else
                {
                    t[j] = 0;
                }
            }
            --taille;
        }
    }
    afficher(t, taille);
    return 0;
}
```

2.3 Normalisation

2.3.1 Présentation

La normalisation est le fait de transformer toutes les valeurs d'un tableau d'entiers en valeurs comprises entre 0 et 1. On doit donc calculer la valeur maximal dans le but de pouvoir convertir toutes les autres valeurs en les divisant par avec le max.

2.3.2 Exemple d'exécution

Prenons par exemple le tableau T=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Pour normaliser ce tableau, il faut rechercher le maximum qui se trouve être 10.
Pour finir, on parcourt le tableau en divisant chaque élément par le maximum.
Le tableau normalisé est donc T=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.10]

2.3.3 Algorithme

Algorithme : Normalisation
Variables :
i, n : entiers
$T1$,$T2$: tableaux d'entiers
max : réel
Debut
  $max \leftarrow 0$
  Pour *i allant de 1 à n* **faire**
    Si $max < T1[i]$ **alors**
      $max \leftarrow T2[i]$
    FinSi
  FinPour
  Pour *i allant de 1 à n* **faire**
    $T2[i] \leftarrow T1[i]$ / max
  FinPour
  Pour *i allant de 1 à n* **faire**
    Ecrire : $T2[i]$
  FinPour
Fin

2.3.4 Complexité

$$\left.\begin{array}{l} Affectation : O(2n+1) \\ Calcul : O(2n) \\ Ecriture : O(n) \end{array}\right\} \rightarrow O(3n) : max(2n+1))/O(2n)/O(n))$$

2.3.5 Code en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (int argc, char* argv[])
{
   int i, taille, max;
   int *T1;
   double *T2;
   printf("Saisir la taille du tableau : ");
   scanf("%d", &taille);
   T1 = (int*) malloc(taille*sizeof(int));
   T2 = (double*) malloc(taille*sizeof(double));
   remplirA(T1, taille);

   afficher(T1, taille);
```

```
  max = 0;
  for(i=0;i<taille;i++)
  {
    if(max<=T1[i])
    {
      max = T1[i];
    }
  }

  for(i=0; i<taille; i++)
  {
    T2[i]=((double)T1[i]/(double)max);
  }

  for(i=0; i<taille; i++)
  {
    printf("%.2f \t", T2[i]);
  }
  printf("\n");

  free(T1);
  free(T2);
  return 0;
}
```

2.4 Inversion d'un tableau

2.4.1 Présentation

L'algorithme présent ci-dessous permet d'inverser toutes les valeurs d'un tableau afin de se retrouver avec un tableau inversé par rapport au tableau de base.

2.4.2 Exemple d'exécution

Prenons le tableau T=[1, 2, 3, 4, 5]
Avec l'aide d'une boucle Pour qui parcourt à la fois le tableau à l'endroit et à l'envers, les valeurs du tableau sont échangés aux places correspondantes.
T=[1, 2, 3, 4, 5]
T=[5, 2, 3, 4, 1]
T=[5, 4, 3, 2, 1]

2.4.3 Algorithme

Algorithme : InverserTableau
Variables :
i, j, $temp$, N : entiers (N = taille du tableau)
T : Tableau
Debut
  Pour *i allant de 1 à N* **faire**
    Pour *i descendant de N à 1* **faire**
      $temp \leftarrow T[i]$
      $T[i] \leftarrow T[j]$
      $T[j] \leftarrow temp$
    FinPour
  FinPour
  afficher(T)
Fin

2.4.4 Complexité

$$\left.\begin{array}{l} Affectation : O(3n) \\ Calcul : O(0) \\ Ecriture : O(n) \end{array}\right\} \rightarrow O(4n) : max(3n)/O(0)/O(n))$$

2.4.5 Code en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (int argc, char* argv[])
{
   int i, j, temp, taille;
   int *T1;
   printf("Saisir la taille du tableau : ");
   scanf("%d", &taille);
   T1 = (int*) malloc(taille*sizeof(int));
   remplirA(T1, taille);
   afficher(T1, taille);

   for (i=0,j=taille-1;i<j;i++, j--)
   {
      temp = T1[i];
      T1[i] = T1[j];
      T1[j] = temp;
   }
   afficher(T1, taille);
   free(T1);
```

```
    return 0;
}
```

2.5 Fusion de deux tableaux

2.5.1 Présentation

L'algorithme suivant permet de fusionner deux tableaux afin d'en obtenir un seul. Les tableaux de base sont supposés déjà triés. Les valeurs sont stockées dans un troisième tableau une par une en fonction de leur grandeur.

2.5.2 Exemple d'exécution

Prenons les deux tableaux T1=[2, 5, 8, 10] et T2=[1, 3, 7, 12].
On compare chaque valeur de même rang pour trouver la valeur minimale afin de la placer dans un troisième tableau T3.
2 > 1, on place 1 dans T3 : T3=[1, ...]
2 < 3, on place 2 dans T3 : T3=[1, 2, ...]
5 > 3, on place 3 dans T3 : T3=[1, 2, 3, ...]
5 < 7, on place 5 dans T3 : T3=[1, 2, 3, 5, ...]
8 > 7, on place 7 dans T3 : T3=[1, 2, 3, 5, 7, ...]
8 < 12, on place 8 dans T3 : T3=[1, 2, 3, 5, 7, 8, ...]
10 < 12, on place 10 dans T3 : T3=[1, 2, 3, 5, 7, 8, 10, ...]
On peut maintenant placer la dernière valeur, T3=[1, 2, 3, 5, 7, 8, 10, 12]

2.5.3 Algorithme

Algorithme : Fusion Tableau
Variables :
i, j, k, N, M : entiers (N et M : taille des tableaux)
$T1$, $T2$, $T3$: Tableaux (Tableaux supposés déjà triés)
Debut

```
    allouer(T1, N)
    allouer(T2, M)
    allouer(T3, N + M)
    remplir(T1)
    remplir(T2)
    remplir(T3)
    i ← 0
    j ← 0
    k ← 0
    TantQue i < N OU j < N faire
        Si T1[i] < T2[j] ET i < N alors
            T3[k] ← T1[i]
            k ← k + 1
            i ← i + 1
        Sinon
            T3[k] ← T2[j]
            k ← k + 1
            j ← j + 1
        FinSi
    FinTantQue
    afficher(T3)
```

Fin

2.5.4 Complexité

$$\left.\begin{array}{l} Affectation : O(3+(n)/2) \\ Calcul : O(n)/2 \\ Ecriture : O(n-1) \end{array}\right\} \rightarrow O : (O(3+(n)/2), O((n)/2), O(n-1))$$

2.5.5 Code en C

```c
#include <stdlib.h>
#include <stdio.h>
#define N 4

int main (void)
{
   int I[N]={1, 6, 7, 10};
   afficher(I, N);
```

```
  int J[N]={2, 5, 8, 12};
  afficher(J, N);
  int K[N+N];
  int i = 0;
  int j = 0;
  int k = 0;
  while (i < N || j < N)
  {
    if (I[i] < J[j] && i<N)
    {
      K[k]=I[i];
      k++;
      i++;
    }
    else
    {
      K[k]=J[j];
      k++;
      j++;
    }
  }
  afficher(K, N+N);
}
```

2.6 Image complémentaire d'une image binaire

2.6.1 Présentation

L'algorithme ci-dessous permet de réaliser l'image complémentaire d'une image binaire. C'est à dire qu'il transforme tout les 1 en 0 et tout les 0 en 1 dans la matrice concerné.

2.6.2 Exemple d'exécution

Prenons la matrice A suivante :
1 1 1 1
0 0 0 0
0 0 0 0
1 1 1 1
Après l'exécution de l'algorithme, tout les 1 sont devenus et des 0 et inversement. On obtient donc la matrice B suivante :
0 0 0 0
1 1 1 1
1 1 1 1
0 0 0 0

2.6.3 Algorithme

Algorithme : Image complémentaire d'une image binaire
Variables :
i, j, n : entiers
I,J : matrices binaire
Debut
  Pour *i allant de 1 à n* **faire**
    Pour *j allant de 1 à n* **faire**
      Si $I[i,j] = 1$ **alors**
        $J[i,j] \leftarrow 0$
      Sinon
        $J[i,j] \leftarrow 1$
      FinSi
    FinPour
  FinPour
  afficher(J)
Fin

2.6.4 Complexité

$$\left.\begin{array}{l} Affectation : O(n^2) \\ Calcul : O(n^2) \\ Ecriture : O(n^2) \end{array}\right\} \rightarrow O(3n^2) : max(O(n^2)/O(n^2)/O(n^2))$$

2.6.5 Code en C

```c
#include <stdio.h>
#include <stdlib.h>
#define N 4

int main (int argc, char* argv[])
{
   int Tab1[N][N]={{1, 1, 0, 0}, {1, 0, 1, 0}, {1, 0, 1, 0}, {0, 0, 1, 1}};
   int Tab2[N][N];
   int i=0;
   int j=0;

   printf("Image binaire : \n");
   for(i=0; i<N; i++)
   {
      for(j=0; j<N; j++)
      {
         printf("%d \t", Tab1[i][j]);
      }
      printf("\n");
   }
```

```
  printf("\n");

  for(i=0; i<N; i++)
  {
    for(j=0; j<N; j++)
    {
      if (Tab1[i][j]==1)
      {
        Tab2[i][j]=0;
      }
      else
      {
        Tab2[i][j]=1;
      }
    }
  }

  printf("Complement de cette image binaire \n");
  for(i=0; i<N; i++)
  {
    for(j=0; j<N; j++)
    {
      printf("%d \t", Tab2[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

2.7 Calcul du maximum de 2 images (matrices)

2.7.1 Présentation

Cet algorithme permet de trouver quel est la plus grande valeur à chaque emplacement entre deux images. Il stocke les plus grandes valeurs dans une troisième matrice qu'il affiche après coup.

2.7.2 Exemple d'exécution

Prenons les matrices A, B et C :
0 1 (A)
1 0
et
1 0 (B)
0 1
1ère case : 0 et 1 -> 1 est le max, on le stocke dans la matrice C.
2ème case : 1 et 0 -> 1 est le max, on le stocke dans la matrice C.
3ème case : 0 et 1 -> 1 est le max, on le stocke dans la matrice C.

4ème case : 1 et 0 -> 1 est le max, on le stocke dans la matrice C.
On obtient la matrice C :
1 1
1 1

2.7.3 Algorithme

Algorithme : Maximum de 2 matrices
Variables :
i, j, n : entiers
I,J,K : matrice binaire (K initialisé à 0)
Debut
  Pour *i allant de 1 à n* **faire**
    Pour *j allant de 1 à n* **faire**
      Si $I[i,j] > J[i,j]$ **alors**
        $K[i,j] \leftarrow I[i,j]$
      Sinon
        $K[i,j] \leftarrow J[i,j]$
      FinSi
    FinPour
  FinPour
  afficher(K)
Fin

2.7.4 Complexité

$$\left.\begin{array}{l} Affectation : O(n^2) \\ Calcul : O(n^2) \\ Ecriture : O(n^2) \end{array}\right\} \rightarrow O(3n^2) : max(O(n^2)/O(n^2)/O(n^2))$$

2.7.5 Code en C

```
#include <stdlib.h>
#include <stdio.h>
#define N 4

int main (void)
{
   int i, j;
   int I[N][N]={{1, 2, 3, 4}, {1, 0, 1, 0}, {0, 1, 0, 1}, {0, 1, 0, 8}};
   int J[N][N]={{1, 2, 3, 4}, {5, 6, 7, 8}, {1, 5, 1, 0}, {1, 0, 1, 8}};

   printf("Matrice I : \n");
   for(i=0; i<N; i++)
   {
      for(j=0; j<N; j++)
      {
```

```c
            printf("%d \t", I[i][j]);
        }
        printf("\n");
    }

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

    int K[N][N]={{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}};

    for (i=0;i<N;i++)
    {
        for (j=0;j<N;j++)
        {
            if (I[i][j]>J[i][j])
            {
                K[i][j]=I[i][j];
            }
            else
            {
                K[i][j]=J[i][j];
            }
        }
    }

    printf("Matrice K (maximum de chaque matrice) : \n");
    for(i=0; i<N; i++)
    {
        for(j=0; j<N; j++)
        {
            printf("%d \t", K[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

2.8 Recherche de la majorité dans un bureau de vote

2.8.1 Présentation

L'algorithme permet de trouver quel entier revient le plus souvent dans un tableau trié.

2.8.2 Exemple d'exécution

Prenons le tableau exemple suivant : T=[1, 1, 1, 1, 2, 2, 3, 4, 5, 5, 5]
On compte d'abord le nombre de fois que le premier entier sort : 4 fois. On stocke 4 dans le max.
On compte ensuite le nombre de fois que 2 sort : 2 fois. On compare avec notre max : 2<4 donc on garde 4 dans le max.
On fait pareil pour 3, 4, et 5 pour se rendre compte que c'est l'entier 1 qui sort le plus de fois.

2.8.3 Algorithme

Algorithme : Majorité dans un tableau

Variables :
i, $temp$, maj, max, N : entiers (N = taille du tableau)
Tab : Tableau (trié)

Debut
- $i \leftarrow 0$
- $temp \leftarrow 0$
- $max \leftarrow 0$
- **TantQue** $i < N$ **faire**
  - **TantQue** $Tab[i] = Tab[i+1]$ ET $i < N$ **faire**
    - $temp \leftarrow temp + 1$
    - $i \leftarrow i + 1$
  - **FinTantQue**
  - **Si** $temp > max$ **alors**
    - $maj \leftarrow Tab[i-1]$
    - $max \leftarrow temp$
  - **FinSi**
  - $temp \leftarrow 0$
  - $i \leftarrow i + 1$
- **FinTantQue**
- **Ecrire :** maj

Fin

2.8.4 Code en C

```
#include <stdlib.h>
#include <stdio.h>
#define N 10

int main (void)
```

```
{
   int k;
   int Tab[N] = {1, 1, 1, 1, 2, 3, 4, 4, 4, 5};
   for(k=0; k<N; k++)
   {
      printf("%d \t", Tab[k]);
   }
   printf("\n");
   int i=0;
   int temp=0;
   int max=0;
   int maj;
   while (i<N)
   {
      while(Tab[i]==Tab[i+1]&&i<N){
         temp++;
         i++;
      }
      if(temp>max){
         maj=Tab[i-1];
         max=temp;
      }
      temp=0;
      i++;
   }
   printf("L'entier qui revient le plus souvent est le %d \n", maj);
   return 0;
}
```

2.9 Taille d'une chaine de caractère

2.9.1 Présentation

Pour finir sur les tableaux, voici un algorithme nous donnant la taille d'une chaine de caractère.

2.9.2 Exemple d'exécution

Prenons la chaine de caractère : Bonjour.
A l'aide d'une boucle Tant Que, on incrémente une valeur tant qu'on arrive pas à la fin de ce mot.
B : i = 1.
o : i = 2.
n : i = 3.
j : i = 4.
o : i = 5.
u : i = 6.
r : i = 7.

2.9.3 Algorithme

Algorithme : TailleChaine

Variables : i : entier

s : chaine de caractère)

Debut

$i \leftarrow 0$

TantQue $s[i]\ != \backslash 0$ **faire**

$i \leftarrow i{+}1$

FinTantQue

Retourner : i

Fin

2.9.4 Complexité

$$\left.\begin{array}{l} Affectation : O(n+1) \\ Calcul : O(0) \\ Ecriture : O(0) \end{array}\right\} \rightarrow O(n) : max(O(n+1)/O(0)/O(0))$$

2.9.5 Code en C

```c
#include <stdio.h>
#include <stdlib.h>

int mystrlen(char *s) {
  int i=0;
  while(s[i]!='\0')
    ++i;
  return i;
}

int main() {
   char CH[1000];
   printf("Rentrez votre chaine : ");
   scanf("%s", CH);
   int resultat = mystrlen(CH);
   printf("%i\n", resultat);
   return 0;
}
```

3 Tri

Presque tout les tris ont étés testés avec les fonctions suivantes :

```
void remplirT(int *t, int taille)
{
   int i;
   printf("Remplissage aleatoire du tableau \n");
   srand(time(NULL));
   for(i = 0; i < taille; i++)
   {
      t[i] = rand()%100;
   }
}

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

void echanger(int t[], int i, int j)
{
   int tmp = t[i];
   t[i] = t[j];
   t[j] = tmp;
}
```

3.1 Tri par insertion

3.1.1 Présentation

Le tri par insertion est un algorithme de tri classique plutôt lent. Il faut parcourir le tableau à trier du début à la fin et pour chaque valeur, on la compare aux valeurs qui la précède pour pouvoir la mettre l'emplacement qui lui correspond dans le tableau.

3.1.2 Exemple d'exécution

Prenons le tableau T=[5, 25, 15, 10, 30, 20]
Première étape : T=[5, 25, 15, 10, 30, 20] -> T=[5, 25, 15, 10, 30, 20]
Deuxième étape : T=[5, 25, 15, 10, 30, 20] -> T=[5, 15, 25, 10, 30, 20]
Troisième étape : T=[5, 15, 25, 10, 30, 20] -> T=[5, 10, 15, 25, 30, 20]
Quatrième étape : T=[5, 10, 15, 25, 30, 20] -> T=[5, 10, 15, 25, 30, 20]
Cinquième étape : T=[5, 10, 15, 25, 30, 20] -> T=[5, 10, 15, 20, 25, 30]
Tableau trié : T=[5, 10, 15, 20, 25, 30]

3.1.3 Algorithme

Algorithme : Tri par insertion
Variables :
i, j, x, N : entiers (N = taille du tableau)
T : Tableau
Debut
  Pour *i allant de 1 à N* **faire**
    $i \leftarrow T[i]$
    Pour *j descendant de 1 à N ET $T[j-1] > x$* **faire**
      $T[j] \leftarrow T[j-1]$
    FinPour
    $T[j] \leftarrow x$
  FinPour
Fin

3.1.4 Complexité

Normalement les tris ont pour complexité : O(n^2) mais dans nos cas cela varie un petit peu.

$$\left.\begin{array}{l} Affectation : O(n^2+2n) \\ Calcul : O(0) \\ Ecriture : O(0) \end{array}\right\} \rightarrow O(n^2+2n) : max(O(2^2+2n)/O(0)/O(0))$$

3.1.5 Code en C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (void)
{
    int i,j,x,N;
    int *T;
    printf("Quelle taille pour le tableau ? ");
    scanf("%d",&N);
    T = (int*)malloc(sizeof(int)*N);
    remplirT(T,N);
    afficherT(T,N);
    for(i=1;i<N;i++)
    {
        x=T[i];
        for(j=i;j>0 && T[j-1]>x;j--)
        {
            T[j]=T[j-1];
        }
        T[j]=x;
    }
```

```
    printf("Affichage du tableau trie :\n ");
    afficherT(T,N);
    free (T);
    return 0;
}
```

3.2 Tri par sélection

3.2.1 Présentation

Le tri par sélection est un algorithme de tri par comparaison. En parcourant le tableau, on recherche à chaque fois la plus petite valeur afin de la placer au début du tableau ou après la précédente plus petite valeur.

3.2.2 Exemple d'exécution

Prenons le tableau T=[5, 25, 15, 10, 30, 20]
Première étape : T=[5, 25, 15, 10, 30, 20] -> T=[5, 10, 25, 15, 30, 20]
Deuxième étape : T=[5, 10, 25, 15, 30, 20] -> T=[5, 10, 15, 25, 30, 20]
Troisième étape : T=[5, 10, 15, 25, 30, 20] -> T=[5, 10, 15, 20, 25, 30]
Quatrième étape : T=[5, 10, 15, 20, 25, 30] -> T=[5, 10, 15, 20, 25, 30]
Cinquième étape : T=[5, 10, 15, 20, 25, 30] -> T=[5, 10, 15, 20, 25, 30]
Tableau trié : T=[5, 10, 15, 20, 25, 30]

3.2.3 Algorithme

Algorithme : Tri par sélection
Variables :
i, j, min, N : entiers (N = taille du tableau)
T : Tableau
Debut
  Pour *i allant de 1 à N* **faire**
    $min \leftarrow i$
    Pour *j allant de i à N* **faire**
      Si $T[j] < T[min]$ **alors**
        $min \leftarrow j$
      FinSi
      Si $min \mathrel{!}= j$ **alors**
        echanger($T[i]$, $T[min]$)
      FinSi
    FinPour
  FinPour
Fin

3.2.4 Complexité

$$\left.\begin{array}{l} Affectation : O(n) \\ Calcul : O(n^2) \\ Ecriture : O(0) \end{array}\right\} \rightarrow O(n^2) : max(O(n)/O(n^2)/O(0))$$

3.2.5 Code en C

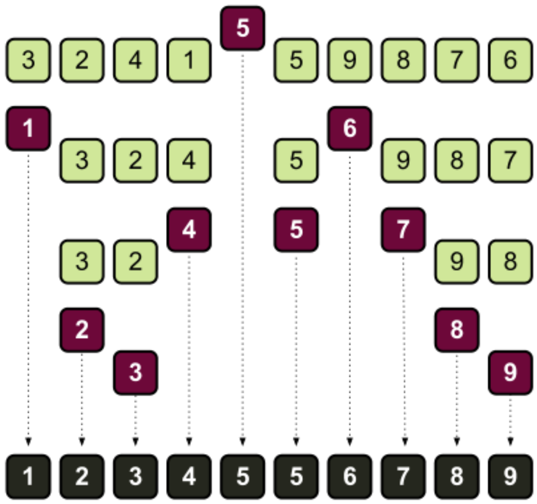
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (void)
{
    int i,j,min,tmp,N;
    int *T;
    printf("Quelle taille pour le tableau ? ");
    scanf("%d",&N);
    T = (int*)malloc(sizeof(int)*N);
    remplirT(T,N);
    afficherT(T,N);
    for(i=0;i<N;i++)
    {
        min=i;
        for(j=i;j<N;j++)
        {
            if(T[min]>T[j])
            {
                min=j;
            }
        }
        if(min!=i)
        {
            tmp = T[i];
            T[i]=T[min];
            T[min]=tmp;
        }
    }
    printf("Affichage du tableau trie :\n ");
    afficherT(T,N);
    free (T);
    return 0;
}
```

3.3 Tri Rapide

3.3.1 Présentation

Le tri rapide est fondé sur la méthode de conception "diviser pour mieux régner". Dans cet algorithme, il faut choisir un pivot afin de placer les éléments plus petit à gauche de ce pivot. Le tableau est donc divisé en deux et on répète l'action jusqu'à temps que le tableau soit trié.

3.3.2 Exemple d'exécution



3.3.3 Algorithme

Algorithme : Tri Rapide

Variables : T : Tableau

$gauche$, $droite$, $pivot$, sep : entiers

Debut

$sep \leftarrow gauche$

$pivot \leftarrow T[droite]$

Si $gauche < (sep+1)$ **alors**

triRapide(T,$gauche$,sep-1)

FinSi

Si $droite < (sep+1)$ **alors**

triRapide(T,sep,$droite$)

FinSi

Fin

3.3.4 Complexité

$$\left.\begin{array}{l} Affectation : O(nlog(n)) \\ Calcul : O(nlog(n)) \\ Ecriture : O(0) \end{array}\right\} \rightarrow O(nlog(n)) : max(O(nlog(n))/O(nlog(n))/O(0))$$

3.3.5 Code en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void TriRapide(int*, int, int);

int main (void)
{
   int G,D,N;
   int *T;
   printf("Quelle taille pour le tableau ? ");
   scanf("%d",&N);
   T = (int*)malloc(sizeof(int)*N);
   remplirT(T,N);
   afficherT(T,N);
   G=0;
   D=N-1;
   TriRapide(T,G,D);
   printf("Affichage du tableau trie :\n ");
   afficherT(T,N);
   free(T);
   return 0;
}

void TriRapide(int *tab, int debut, int fin)
{
   int gauche = debut-1;
   int droite = fin+1;
   const int pivot = tab[debut];
   if(debut >= fin)
   {
         return;
   }
   while(1)
   {
      do droite--; while(tab[droite] > pivot);
      do gauche++; while(tab[gauche] < pivot);
      if(gauche < droite)
      {
         echanger(tab, gauche, droite);
      }
      else break;
   }
   TriRapide(tab, debut, droite);
   TriRapide(tab, droite+1, fin);
}
```

3.4 Tri hollandais

3.4.1 Présentation

Le principe du tri hollandais (ou tri couleur) est de trier le tableau afin de se retrouver avec un tableau dans l'ordre des couleurs du drapeau hollandais.

3.4.2 Exemple d'exécution

Prenons le tableau T=[X, X, X, X, X, X]
Première étape : T=[X, X, X, X, X, X] -> T=[X, X, X, X, X, X]
Deuxième étape : T=[X, X, X, X, X, X] -> T=[X, X, X, X, X, X]
Troisième étape : T=[X, X, X, X, X, X] -> T=[X, X, X, X, X, X]
Quatrième étape : T=[X, X, X, X, X, X] -> T=[X, X, X, X, X, X]
Tableau trié : T=[X, X, X, X, X, X]

3.4.3 Algorithme

Algorithme : Tri hollandais
Variables :
b, w, r, N : entiers (N = taille du tableau)
T : Tableau d'entier **Debut**
$b \leftarrow 1$
$w \leftarrow 1$
$r \leftarrow N$
TantQue $w < r$ **faire**
Si $T[w] = BLANC$ **alors**
$w \leftarrow w + 1$
FinSi
Sinon si $T[w] = BLEU$ **alors**
echanger($T[b]$, $T[w]$)
$b \leftarrow b + 1$
$w \leftarrow w + 1$
FinSi
Sinon
echanger($T[w]$, $T[r]$)
$r \leftarrow r$ - 1
FinSi
FinTantQue
Fin

3.4.4 Complexité

$$\left.\begin{array}{l} Affectation : O(3+n) \\ Calcul : O(0) \\ Ecriture : O(0) \end{array}\right\} \rightarrow O(n) : max(O(3+n)/O(0)/O(0))$$

3.4.5 Code en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void remplirT(int*, int);

int main (void)
{
    int i,j,k,N;
    i = -1;
    j = -1;
    int *T;
    printf("Quelle taille pour le tableau ? ");
    scanf("%d",&N);
    k = N-1;
    T = (int*)malloc(sizeof(int)*N);
    printf("3 : BLEU | 2 : BLANC | 1 : ROUGE\n");
    remplirT(T,N);
    afficherT(T,N);
    while(k!=j)
    {
        if(T[j+1]==3)
        {
            echanger(T,j+1,i+1);
            ++i;
            ++j;
        }
        else if(T[j+1]==1)
        {
            echanger(T,j+1,k);
            --k;
        }
        else
        {
            ++j;
        }
    }
    printf("Affichage du tableau trie :\n ");
    afficherT(T,N);
    free(T);
    return 0;
}

void remplirT(int *t, int taille)
{
    int i;
    printf("Remplissage aleatoire du tableau \n");
```

```
    srand(time(NULL));
    for(i = 0; i < taille; i++)
    {
        t[i] = (rand()%3)+1;
    }
}
```

3.5 Tri à bulle

3.5.1 Présentation

Le principe du tri à bulle est de comparer chaque valeur avec la valeur suivante afin de les échanger si la seconde valeur est plus grande. Une fois le maximum du tableau à la fin de celui-ci, la taille est diminué pour recommencer le même mouvement.

3.5.2 Exemple d'exécution

Prenons le tableau T=[5, 25, 15, 10, 30, 20]
Première étape : T=[5, 25, 15, 10, 30, 20] -> T=[5, 25, 15, 10, 30, 20]
Deuxième étape : T=[5, 25, 15, 10, 30, 20] -> T=[5, 15, 25, 10, 30, 20]
Troisième étape : T=[5, 15, 25, 10, 30, 20] -> T=[5, 15, 10, 25, 30, 20]
Quatrième étape : T=[5, 15, 10, 25, 30, 20] -> T=[5, 15, 10, 25, 30, 20]
Cinquième étape : T=[5, 15, 10, 25, 30, 20] -> T=[5, 15, 10, 25, 20, 30]
Le maximum est maintenant à la fin, on recommence jusqu'à temps que le tableau soit trié en diminuant à chaque fois la taille de 1.

3.5.3 Algorithme

Algorithme : Tri à bulle
Variables : T : Tableau alloué et initialisé
i, tmp, $taille$: entiers
Debut
  TantQue *taille > 1* **faire**
    Pour *i allant de 1 à taille* **faire**
      Si $T[i] < T[i+1]$ **alors**
        $tmp \leftarrow T[i]$
        $T[i] \leftarrow T[i+1]$
        $T[i+1] \leftarrow tmp$
      FinSi
    FinPour
    $taille \leftarrow taille - 1$
  FinTantQue
Fin

3.5.4 Complexité

$$\left.\begin{array}{l} Affectation : O(3n+3) \\ Calcul : O(n) \\ Ecriture : O(1) \end{array}\right\} \rightarrow O(3n) : max(O(3n+3)/O(n)/O(1))$$

3.5.5 Code en C

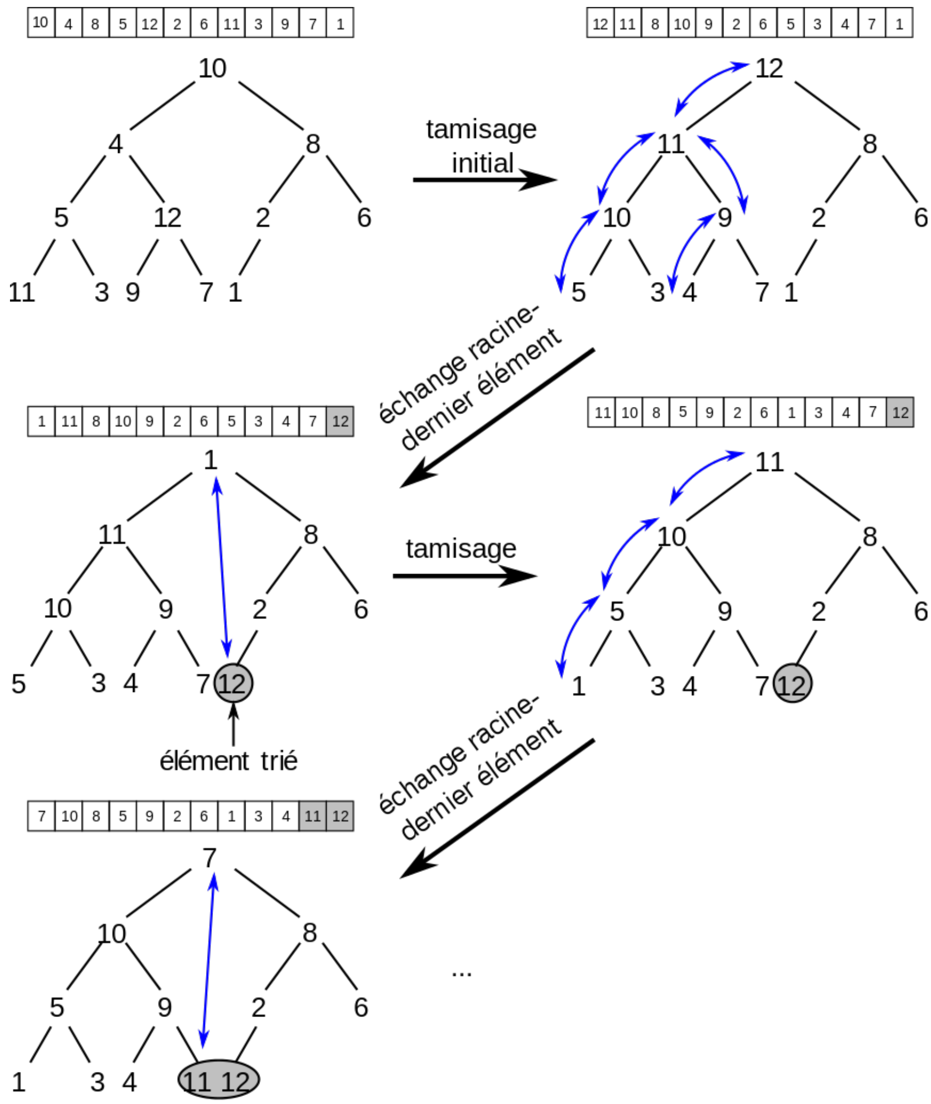
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (void)
{
   int i,j,N;
   int *T;
   printf("Quelle taille pour le tableau ? ");
   scanf("%d",&N);
   T = (int*)malloc(sizeof(int)*N);
   remplirT(T,N);
   afficherT(T,N);
   for(i=N;i>1;i--)
   {
      for(j=0;j<i-1;j++)
      {
         if(T[j+1]<T[j])
         {
            echanger(T,j+1,j);
         }
      }
   }
   printf("Affichage du tableau trie :\n ");
   afficherT(T,N);
   return 0;
}
```

3.6 Tri par tas

3.6.1 Présentation

Le tri par tas est réalisé dans un tableau à une dimension.
Les valeurs de ce tableau représentent les noeuds d'un arbre binaire.

3.6.2 Exemple d'exécution



3.6.3 Algorithme

Algorithme : Descendre $(*t,\ i,\ n)$

Variables : i, k : entiers

$*T$: Tableau d'entiers alloué ;

Debut

    $k \leftarrow 2 * i$

    Si *$(k < n)$ ET $(T[k+1] > T[k])$* **alors**

        $k \leftarrow k + 1$

    FinSi

    Si *$(k <= n)$ ET $(t[i] < t[k])$* **alors**

        Echanger(t, g, d)

        Descendre(t, k, n)

    FinSi

Fin

Algorithme : triTas

Variables : i, k : entiers
$*T$: tableau de n valeurs alloué

Debut
$k \leftarrow 2 * i$
Tas(T)
TantQue *(k > 0)* **faire**
Echanger(T, 1, k)
Descendre(T, 1, $k-1$)
$k \leftarrow k-1$
FinTantQue
Fin

Algorithme : Tas ($*T$)

Variables : i, k : entiers
$*T$: tableau de n valeurs alloué

Debut
$i \leftarrow n/2$
TantQue *($i > 0$)* **faire**
Descendre(T, 1, n)
$i \leftarrow i-1$
FinTantQue
Fin

3.6.4 Complexité

$$\left.\begin{array}{l} Affectation : O(2n * 2log(n)) \\ Calcul : O(2nlog(n)) \\ Ecriture : O(1) \end{array}\right\} \rightarrow O(2nlog(n)) : max(O(2n{*}2log(n)/O(2nlog(n))/O(1))$$

3.6.5 Code en C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void remplirT(int *t, int taille)
{
   int i;
   printf("Remplissage aleatoire du tableau \n");
   srand(time(NULL));
   for(i = 0; i < taille; i++)
   {
      t[i] = rand()%100;
   }
}
```

```c
void afficherT(int* t, int taille)
{
   int i;
   for(i = 0; i < taille; i++)
   {
      printf("%d \t", t[i]);
   }
   printf("\n");
}

void echanger(int T[], int i, int j)
{
   int echange;
   echange=T[i];
   T[i]=T[j];
   T[j]=echange;
}
void remonter(int T[], int n, int i)
{
   if (i==0) return;
   if (T[i]>T[i/2])
   {
      echanger (T, i, i/2);
      remonter (T, n, i/2);
   }
}
void redescendre(int T[], int n, int i)
{
   int imax;
   if (2*i+1>=n)
   {
      return;
   }
   if (T[2*i+1]>T[2*i])
   {
      imax=2*i+1;
   }
   else
   {
      imax=2*i;
   }
   if (T[imax]>T[i])
   {
      echanger (T, imax, i);
      redescendre (T, n, imax);
   }
}
void organiser(int T[], int n)
{
   int i;
```

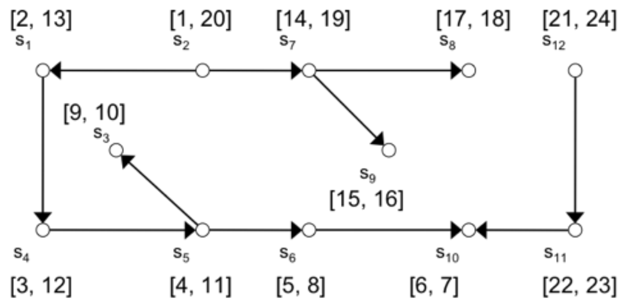
```
    for(i = 1 ; i < n ; i++)
    {
        remonter(T, n, i);
    }
}

void Tri_Arbre(int T[], int n)
{
    int i;
    organiser(T, n);
    for(i=n-1 ; i>0 ; i-- )
    {
        echanger(T, 0, i);
        redescendre(T, i, 0);
    }
}

int main(int argc, char ** argv)
{
    int *T;
    int N;
    printf("Quelle est la taille de l'arbre ? ");
    scanf("%d",&N);
    T = (int*)malloc(sizeof(int)*N);
    remplirT(T,N);
    afficherT(T, N);
    Tri_Arbre( T, N);
    printf("Affichage de l'arbre trie :\n ");
    afficherT(T, N);
    return 0;
}
```

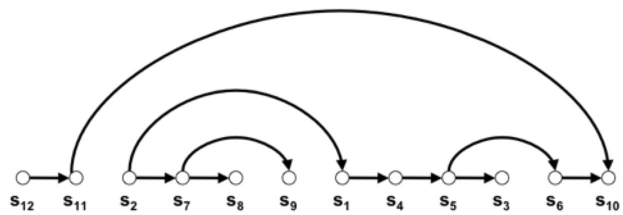
3.7 Tri Topologique

3.7.1 Présentation

Le tri topologique d'un graphe orienté acyclique G = (S, A) consiste à ordonner linéairement tous ses sommets de telle sorte que si G contient un arc entre s_i et s_j, s_i apparaisse avant sj.

3.7.2 Exemple d'exécution





3.7.3 Algorithme

Algorithme : Tri topologique

Variables : M : tableau de tableau alloués et initialisés
V^+ : tableau alloué et initialisé
i, j, k, *taille* : entiers **Debut**

- **Pour** *i allant de 1 à taille* **faire**
  - $j \leftarrow 1$
  - **TantQue** $V^+[j]$ *!=0* **faire**
    - $j \leftarrow j + 1$
  - **FinTantQue**
  - Construire liste(k,j)
  - $V^+[j] \leftarrow -1$
  - **Pour** *i allant de 1 à taille* **faire**
    - **Si** $M[j][k]$*=1* **alors**
      - $V^+[k] \leftarrow V^+[k] - 1$
    - **FinSi**
  - **FinPour**
- **FinPour**

Fin

4 Liste chaînée

4.1 Déclaration d'une liste chaînée

4.1.1 Algorithme

4.1.2 Code en C

```c
struct scellule {
   struct scellule *next;
   int nbr;
};

typedef struct scellule cellule;

typedef struct sliste {
   cellule *cpremier;
   cellule *cdernier;
}liste;
```

4.2 Création d'une liste

4.2.1 Présentation

L'algorithme suivant va nous permettre de créer une liste chaînée pour l'instant non initialisé.

4.2.2 Algorithme

Algorithme : creerListe

Variables :
l : liste
Debut
  allouer(l)
  $l \rightarrow$ cpremier $\leftarrow$ NULL
  $l \rightarrow$ cdernier $\leftarrow$ NULL
  Retourner : l
Fin

4.2.3 Code en C

```c
liste* creerListe()
{
   printf("Creation de la liste chainee\n");
   liste *l = (liste*)malloc(sizeof(liste));
   l->cpremier = NULL;
   l->cdernier = NULL;
   return l;
```

```
}
```

4.3 Affichage d'une liste

4.3.1 Présentation

L'algorithme suivant sert juste à afficher une liste afin de mieux se repérer avec toutes les fonctions disponibles.

4.3.2 Algorithme

Algorithme : afficherListe
Variables :
l : liste
c : pointeur de cellule
Debut
  $c \leftarrow l \rightarrow$ cpremier
  TantQue $c\,!= NULL$ **faire**
    Ecrire : $c \rightarrow$ nbr
    $c \leftarrow c \rightarrow$ next
  FinTantQue
Fin

4.3.3 Code en C

```c
void afficherListe(liste *l)
{
   printf("Affichage de la liste : ");
    cellule *c = l->cpremier;
    while(c != NULL)
    {
        printf("%d ", c->nbr);
        c = c->next;
    }
    printf("\n");
}
```

4.4 Saisie des éléments d'une liste au clavier

4.4.1 Présentation

Après avoir créer une liste, il faut saisir les valeurs à l'intérieur. Notre algorithme demande après chaque valeur rentré si l'utilisateur veux continuer à en rajouter ou non.

4.4.2 Exemple d'exécution

On veux rentrer les valeurs 5, 10 et 15 dans la liste préalablement créée.
On ajoute 5, l'algorithme nous demande si on veut continuer, c'est oui.
On ajoute 10, l'algorithme nous demande si on veut continuer, c'est oui.
On ajoute 15, l'algorithme nous demande si on veut continuer, non.
Notre liste est donc constitué de : 5 → 10 → 15.

4.4.3 Algorithme

Algorithme : saisirListe
Variables :
valeur, *res* : entiers
c, *cpremier*, *tmp* : pointeurs de cellule
Debut
    allouer(cpremier)
    Ecrire : Rentrez une valeur
    Lire : cpremier
    cpremier → nbr ← *valeur*
    cpremier → next ← NULL
    l → *cpremier* ← cpremier
    Ecrire : Continuer ? 1 Oui 2 Non
    Lire : *res*
    TantQue *res = 1* **faire**
        allouer(*c*)
        Ecrire : Rentrez la valeur
        Lire : *valeur*
        c → nbr ← *valeur*
        tmp → next ← *c*
        tmp ← *c*
        Ecrire : Continuer ?
        Lire : *res*
    FinTantQue
    l → cdernier ← *tmp*
Fin

4.4.4 Code en C

```
void saisirListe(liste *l)
{
    int valeur;
    int res;
    cellule *c, *cpremier, *tmp;

    cpremier = (cellule*)malloc(sizeof(cellule));

    printf("Rentrez une valeur : ");
    scanf("%d", &valeur);
```

```
    cpremier->nbr = valeur;
    cpremier->next = NULL;

    l->cpremier = cpremier;

    tmp = cpremier;

    printf("Continuer ? OUI 1 | NON 0 : ");
    scanf("%d", &res);
    while(res)
    {
        c = (cellule*)malloc(sizeof(cellule));
        printf("Rentrez la valeur : ");
        scanf("%d", &valeur);
        c->nbr = valeur;

        tmp->next = c;
        tmp = c;

        printf("Continuer ? ");
        scanf("%d", &res);
    }
    l->cdernier = tmp;
}
```

4.5 Saisie des éléments d'une liste avec un tableau

4.5.1 Présentation

Similaire à l'algorithme précédent, ici on utilise un tableau d'entier comme base pour notre liste.

4.5.2 Exemple d'exécution

Prenons un tableau d'entier standard : T=[1, 2, 3, 4] de taille 4.
On parcourt le tableau à l'aide d'une boucle pour et à chaque étape on insère l'entier dans la liste.

4.5.3 Algorithme

Algorithme : saisirListeTableau
Variables : l : Liste
$taille$,i,val : entiers
Debut
  Ecrire : nombre de valeur :
  Lire : $taille$
  Pour *i allant de 1 à taille* **faire**
    Lire : val
    Ajouter(l,val)
  FinPour
  Retourner : l
Fin

4.6 Compter le nombre de cellule dans une liste

4.6.1 Présentation

Afin de compter le nombre de cellule dans une liste donnée, il faut incrémenter une variable à chaque élément de la liste. Un fois une cellule compté, on passe à la suivante tant que la liste n'est pas vide.

4.6.2 Exemple d'exécution

Prenons la liste chaînée suivante : 5 $\rightarrow$ 10 $\rightarrow$ 15 $\rightarrow$ 20.
La première cellule est 5, on incrémente notre variable que l'on appelle a puis on passe à la cellule suivante. a=1
La cellule suivante est 10, on incrémente a et on passe à la cellule suivante. a=2
La cellule suivante est 15, on incrémente a et on passe à la cellule suivante. a=3
La cellule suivante est 20, on incrémente a et on remarque que c'est la dernière cellule, on s'arrête donc là. a=4
Notre variable a est égal à 4, il y a donc 4 cellules dans la liste chaînée.

4.6.3 Algorithme

Algorithme : nbrCellule (pointeur de liste l)
Variables :
c : pointeur de cellule
n : entier
Debut
  $n \leftarrow 0$
  $c \leftarrow l \rightarrow$ cpremier
  TantQue *c != NULL* **faire**
    $n \leftarrow n+1$
    $c \leftarrow c \rightarrow$ next
  FinTantQue
  Retourner : n
Fin

4.6.4 Code en C

```
int nbrCellule(liste *l)
{
    int n = 0;
    cellule *c = l->cpremier;
    while(c != NULL)
    {
        ++n;
        c = c->next;
    }
    return n;
}
```

4.7 Ajouter une cellule à la fin d'une liste

4.7.1 Présentation

L'algorithme suivante ajouter une cellule à la fin d'un liste préalablement créée et initialisée. Pour cela, il suffit de remplacer la dernière cellule par la valeur que l'on souhaite avant de créer une nouvelle dernière cellule.

4.7.2 Exemple d'exécution

Prenons la liste $5 \rightarrow 10 \rightarrow 15 \rightarrow 20$. Nous souhaitons ajouter 25 dans cette liste. La dernière cellule (supposée être vide) reçoit donc 25 et on peut créer une nouvelle dernière cellule vide.

4.7.3 Algorithme

Algorithme : ajouterCellule (pointeur de liste l, entier *valeur*)
Variables :
c : pointeur de cellule
Debut
    Si l==*0* **alors**
        Ecrire : Erreur : Liste non créée
    FinSi
    allouer(c)
    Si $l \rightarrow = 0$ **alors**
        $l \rightarrow$ cpremier $\leftarrow c$
    Sinon
        $l \rightarrow$ cdernier $\rightarrow$ next $\leftarrow c$
    FinSi
    $l \rightarrow$ cdernier $\leftarrow c$
    $c \rightarrow$ nbr $\leftarrow$ *valeur*
    $c \rightarrow$ next $\leftarrow 0$
Fin

4.7.4 Code en C

```
void ajouterCellule (liste *l, int val)
{
   cellule *c;
   if (l == 0)
   {
      printf("Erreur : Liste pas encore creee \n");
   }
   c = (cellule*)malloc(sizeof(cellule));
   if (l->cpremier == 0)
   {
      l->cpremier = c;
   }
   else
   {
      l->cdernier->next = c;
   }
   l->cdernier = c;
   c->nbr = val;
   c->next = 0;
}
```

4.8 Ajouter une cellule dans une liste

4.8.1 Présentation

L'algorithme qui suit à pour but d'ajouter une cellule à un rang donné dans la liste. Pour cela, on se place à ce rang donné, on ajouter notre cellule est décale toutes les cellules suivantes.

4.8.2 Exemple d'exécution

Prenons la liste : $5 \rightarrow 10 \rightarrow 15 \rightarrow 20$. Nous souhaitons ajouter 30 au troisième rang de cette liste.
On se place au troisième rang (entre 10 et 15) et on ajoute notre cellule.
Cela nous donne donc : $5 \rightarrow 10 \rightarrow 30 \rightarrow 15 \rightarrow 20$.

4.8.3 Algorithme

Algorithme : ajouterCelluleX (pointeur de liste l, pointeur de cellule c, entier a)
Variables :
p : pointeur de cellule
Debut

$i \leftarrow 1$
$p \leftarrow l \rightarrow cpremier$
TantQue *p != NULL ET i != (a-1)* **faire**
  $p \leftarrow p \rightarrow next$
  $i \leftarrow i + 1$
FinTantQue
Si *a=1* **alors**
  $l \rightarrow cpremier \leftarrow c$
FinSi
Si p=$l \rightarrow cdernier$ **alors**
  $l \rightarrow cdernier \leftarrow c$
FinSi
$c \rightarrow next \leftarrow p \rightarrow next$
$p \rightarrow next \leftarrow c$

Fin

4.8.4 Code en C

```c
void ajouterCelluleX(liste *l, cellule *c, int a)
{
   int i = 1;
   cellule *p = l->cpremier;
   while (p != NULL && i!=(a-1))
   {
      p=p->next;
      ++i;
   }
   if(a == 1)
   {
      l->cpremier = c;
   }
   if(p==l->cdernier)
   {
      l->cdernier = c;
   }
   c->next = p->next;
   p->next=c;
}
```

4.9 Supprimer une cellule dans une liste

4.9.1 Présentation

Cette algorithme sert à supprimer une cellule à un rang donné. Il faut donc se placer tout d'abord à ce rang, supprimer la cellule et mettre celles qui suivent juste après.

4.9.2 Exemple d'exécution

Prenons la liste suivante : 5 → 10 → 15 → 20 → 25. Nous souhaitons supprimer la cellule contenant la valeur 15.
On se place donc au troisième rang et on supprime cette valeur 15.
On stocke les cellules suivantes dans des cellules temporaires afin de les replacer dans notre liste initial lorsque l'on a supprimé la cellule voulu.
Notre liste se compose maintenant de : 5 → 10 → 20 → 25.

4.9.3 Algorithme

Algorithme : supprimerCelluleX (pointeur de liste l, entier a)
Variables :
c : pointeur de cellule
l : pointeur de liste$i$: entier
Debut
  Si $a = 1$ **alors**
    l → cpremier ← l → cpremier → next
  Sinon
    c ← l → cpremier
    i ← 1
    TantQue c != $NULL$ et i != a-1 **faire**
      c ← c → next
      i ← i+1
    FinTantQue
    c → next ← c → next → next
  FinSi
Fin

4.9.4 Code en C

```
void supprimerCelluleX(liste *l, int a)
{
   cellule *tmp;
   cellule *c = l->cpremier;
   int i = 1;
   while(c != NULL && i < a - 1)
   {
      c = c->next;
      ++i;
   }
   if(a == 1)
```

```
  {
    l->cpremier = c->next;
  }
  if(c == l->cdernier->next)
  {
    l->cdernier = c;
    c->next = NULL;
  }
  if(a != 1 && c != l->cdernier)
  {
    tmp = c->next;
    c->next = c->next->next;
    if(nbrCellule(l) <= 1)
    {
      l = creerListe();
    }
    else
    {
      free(tmp);
    }
   }
}
```

4.10 Fusionner deux listes

4.10.1 Présentation

L'algorithme consiste consiste à fusionner deux listes pour n'en faire qu'une.

4.10.2 Exemple d'exécution

Prenons les deux listes suivantes : $5 \rightarrow 15 \rightarrow 25$ et $10 \rightarrow 20$. On compare chaque élément pour savoir lequel placé en premier dans la liste fusionné.
$5 < 10$, on place 5 dans la liste : $5 \rightarrow$ NULL
$15 > 10$, on place 10 dans la liste : $5 \rightarrow 10 \rightarrow$ NULL
$15 < 20$, on place 15 dans la liste : $5 \rightarrow 10 \rightarrow 15 \rightarrow$ NULL
$25 > 20$, on place 20 dans la liste : $5 \rightarrow 10 \rightarrow 15 \rightarrow 20 \rightarrow$ NULL
25 est le dernier élément, on le place dans la liste : $5 \rightarrow 10 \rightarrow 15 \rightarrow 20 \rightarrow 25 \rightarrow$ NULL

4.10.3 Algorithme

Algorithme : fusionListeTriés

Variables : $p1$, $p2$: pointeurs de cellule
$l1$, $l2$, $l3$: pointeur de liste
Debut
$p1 \leftarrow l1 \rightarrow cpremier$
$p2 \leftarrow l2 \rightarrow cpremier$
Si $p1 \rightarrow val < p2 \rightarrow val$ **alors**
$l3 \leftarrow l1 \rightarrow cpremier$
$p1 \leftarrow p1 \rightarrow next$
Sinon
$l2 \leftarrow l1 \rightarrow cpremier$
$p2 \leftarrow p2 \rightarrow next$
FinSi
TantQue $p1 != NULL\ OU\ p2 != NULL$ **faire**
Si $p1 = NULL\ OU\ p1 \rightarrow val > p2 \rightarrow val$ **alors**
ajouterCellule($l3$, $p2$)
$p2 \leftarrow p2 \rightarrow next$
Sinon
ajouterCellule($l3$, $p1$)
$p1 \leftarrow p1 \rightarrow next$
FinSi
FinTantQue
Retourner : $l3$
Fin

4.10.4 Code en C

```
liste* fusionnerListe(liste *l1, liste *l2)
{
   liste *l3 = creerListe();
   cellule *c1 = l1->cpremier, *c2 = l2->cpremier;
   while (c1 && c2)
   {
      if(c1->nbr < c2->nbr)
      {
         ajouterCellule(l3,c1->nbr);
         c1 = c1->next;
      }
      else
      {
         ajouterCellule(l3,c2->nbr);
         c2 = c2->next;
      }
   }
   if(!c1)
   {
   while (c2)
```

```
      {
         ajouterCellule(l3,c2->nbr);
         c2=c2->next;
      }
   }
   else
   {
      while (c1)
      {
         ajouterCellule(l3,c1->nbr);
         c1=c1->next;
      }
   }
   return l3;
}
```

5 Pile

5.1 Déclaration d'une pile + Dépiler / Empiler

5.1.1 Présentation

Voici la déclaration d'une pile à l'aide de deux structures ainsi que les méthodes dépiler et empiler, deux méthodes fondamentales pour les piles.

5.1.2 Algorithme

Algorithme : Depiler (pointeur de pile P)

Variables :

$temp$, : pointeur d'élément

Debut

$temp \leftarrow P \rightarrow sommet$

Si $P \rightarrow sommet \mathrel{!}= NULL$ **alors**

$P \rightarrow sommet \leftarrow (P \rightarrow sommet) \rightarrow next$

FinSi

Fin

Algorithme : Empiler (pointeur de pile P, entier v)

Variables :

e, : pointeur d'élément

Debut

allouer(e)

$e \rightarrow a \leftarrow v$

Si $P \rightarrow sommte \mathrel{!}= NULL$ **alors**

$e \rightarrow next \leftarrow P \rightarrow sommet$

Sinon

$e \rightarrow next \leftarrow$ NULL

FinSi

$P \rightarrow sommet \leftarrow e$

Fin

5.1.3 Code en C

```
typedef struct selement {
  int a;
  struct selement *next;
} element;

typedef struct spile {
  element *sommet;
} Pile;

void depiler (Pile *P)
{
  element *temp = P->sommet;
```

```c
	if (P->sommet != NULL)
	{
		P->sommet = (P->sommet)->next;
	}
	free(temp);
}

void empiler(Pile *P, int v)
{
	element *e = (element*) malloc(sizeof(element));
	e->a = v;
	if(P->sommet != NULL)
		e->next = P->sommet;
	else
		e->next = NULL;
	P->sommet = e;
}
```

5.2 Remplissage d'une pile

5.2.1 Présentation

Pour tester tout les algorithmes concernant les piles, il faut d'abord remplir la pile en question. Cet algorithme permet de remplir la pile de deux façon différentes : aléatoirement ou par saisie de l'utilisateur.

5.2.2 Exemple d'exécution

Prenons l'exemple que nous voulons remplir la pile avec nos valeurs.
On empile 5 dans la pile.
On emplie 10 dans la pile.
On empile 15 dans la pile.
On empile 20 dans la pile.
On se retrouver avec cette pile : 5, 10, 15, 20.

5.2.3 Algorithme

Algorithme : remplirPile (pointeur de pile P)

Variables :

res, val, i : entier

Debut

  Ecrire : Comment voulez-vous remplir la pile ?

  Ecrire : 1 - Remplir aléatoirement

  Ecrire : 2 - Remplir avec vos valeurs

  Lire : res

  Si $res = 1$ **alors**

    Pour *i allant de 0 à 5* **faire**

      val = alea(1,100)

      Empiler(P,val)

    FinPour

  FinSi

  Si $res = 2$ **alors**

    Pour *i allant de 0 à 5* **faire**

      Ecrire : Entrez la valeur

      Lire : val

      Empiler(P,val)

    FinPour

  FinSi

Fin

5.2.4 Code en C

```
void remplirPile(Pile *P)
{
   int res, val, i;
   printf("Comment voulez-vous remplir la pile ? \n");
   printf("1 - Remplir aleatoirement\n");
   printf("2 - Remplir avec vos valeurs \n");
   scanf("%d",&res);
   if (res == 1)
   {
      for(i = 0; i < 5; i++)
      {
         srand(time(NULL) + i);
         val = rand() % 100;
         empiler(P, val);
      }
   }
   if (res == 2)
   {
      for (i = 0; i < 5; i++)
      {
         printf("Entrez la valeur : \n");
```

```
         scanf("%d", &val);
         empiler(P, val);
      }
   }
}
```

5.3 Affichage d'une pile

5.3.1 Présentation

Dans le même but que remplirPile, afficherPile permet de tester les algorithmes de pile facilement. L'algorithme répète les actions suivantes tant que la pile n'est pas vide : Afficher le sommet, empiler dans une pile temporaire et dépiler.

5.3.2 Exemple d'exécution

Prenons la pile suivante : 5, 10, 15, 20. Pour afficher, c'est très simple, on affiche les valeurs tant que la pile n'est pas vide.
On affiche 5, on dépile. (Pile : 10, 15, 20)
On affiche 10, on dépile. (Pile : 15, 20)
On affiche 15, on dépile. (Pile : 20)
Et enfin on affiche 20.

5.3.3 Algorithme

Algorithme : afficherPile (pointeur de pile P)
Variables :
tmp : pointeur de pile
Debut
  allouer(*tmp*)
  TantQue *nonVide(P)* **faire**
    Ecrire : sommet(P)
    Empiler(*tmp*, sommet(P))
    Depiler(P)
  FinTantQue
  TantQue *nonVide(P)* **faire**
    Empiler(P, sommet(*tmp*))
    Depiler(*tmp*)
  FinTantQue
Fin

5.3.4 Code en C

```
void afficherPile(Pile *P)
{
   Pile *tmp = (Pile*) malloc(sizeof(Pile));
   while(!estVide(P))
```

```
  {
    printf("%d ", sommet(P));
    empiler(tmp, sommet(P));
    depiler(P);
  }
  printf("\n");
  while(!estVide(tmp))
  {
    empiler(P, sommet(tmp));
    depiler(tmp);
  }
}
```

5.4 Obtenir la taille d'une pile

5.4.1 Présentation

L'algorithme ci-dessous nous permet d'obtenir le nombre d'éléments dans une pile donnée. Il suffit d'incrémenter une variable à chaque fois qu'on dépile un élément dans une pile temporaire. Afin de retrouver notre pile dans l'état de base, on rempile dès que l'on a fini.

5.4.2 Exemple d'exécution

Prenons la pile suivante : 5, 10, 15, 20.
On dépile 5, on incrémente la variable a = 1.
On dépile 10, on incrémente a = 2.
On dépile 15, on incrémente a = 3.
On dépile 20, on incrémente a = 4.

5.4.3 Algorithme

Algorithme : taillePile (pointeur de pile P)
Variables :
taille : entier
tmp : pointeur de pile
Debut
    $taille \leftarrow 0$
    allouer(tmp)
    TantQue *!estVide(P)* **faire**
        Empiler(tmp, sommet(P))
        Depiler(P)
        $taille \leftarrow taille + 1$
    FinTantQue
    TantQue *!estVide(tmp)* **faire**
        Empiler(P, sommet(tmp))
        Depiler(tmp)
    FinTantQue
    Retourner : *taille*
Fin

5.4.4 Code en C

```
int taillePile(Pile *P)
{
   int taille = 0;
   Pile *tmp = (Pile*) malloc(sizeof(Pile));
   while(!estVide(P))
   {
      empiler(tmp, sommet(P));
      depiler(P);
      ++taille;
   }
   while(!estVide(tmp))
   {
      empiler(P, sommet(tmp));
      depiler(tmp);
   }
   return taille;
}
```

5.5 Ackermann avec les piles

5.5.1 Présentation

Contrairement à la fonction récursive, cette fois-ci on va utiliser une pile afin de représenter la fonction d'Ackermann.

5.5.2 Exemple d'exécution

Prenons Ack(), une fonction définissant la suite d'Ackermann.
Calculons par exemple Ack(1, 2) :
Ack(1, 2) = Ack(0, Ack(1, 1))
Ack(1, 2) = Ack(0, Ack(0, Ack(1, 0)))
Ack(1, 2) = Ack(0, Ack(0, Ack(0, 1)))
Ack(1, 2) = Ack(0, Ack(0, 2))
Ack(1, 2) = Ack(0, 3)
Ack(1, 2) = 4

5.5.3 Algorithme

Algorithme : Ackermann Pile
Variables : m, n : entiers
p : Pile
Debut
  Creer(p)
  Empiler(p, m)
  Empiler(p, n)
  TantQue *(!**Vide**(p))* **faire**
    $m \leftarrow$ **Depiler**(p)
    $n \leftarrow$ **Depiler**(p)
    Si *(m = 0)* **alors**
      Empiler(p, $n + 1$)
    Sinon
      Si $n = 0$ **alors**
        Empiler(p, $n + 1$)
        Empiler(p, 1)
      Sinon
        Empiler(p, m-1)
        Empiler(p, m)
        Empiler(p, n-1)
      FinSi
    FinSi
  FinTantQue
  Retourner : Depiler(p)
Fin

5.5.4 Code en C

```
void ackermannPile(int m, int n){
  pile* p=creerP();
  empiler(p,m);
  empiler(p,n);
  afficherPile(p);
```

```
  while( !estvide(p) ){
    n=sommet(p);
    printf("%d -", n);
    depiler(p);
    if(!estvide(p)){
      m=sommet(p);
      depiler(p);
      if (m==0)
        empiler(p,n+1);
      else{
        if(n==0){
          empiler(p,m-1);
          empiler(p,1);
        }
        else{
          empiler(p,m-1);
          empiler(p,m);
          empiler(p,n-1);
        }
      }
    }
  }
  printf("\nResultat : %d\n", n);
  afficherPile(p);
  free(p);
}
```

6 Arbre

Voici le main pour tester toutes les fonctions :

```c
#include <stdio.h>
#include <stdlib.h>
#include "arbre.h"

int main(int argc, char* argv[])
{
   arbre a;
   noeud **foret = NULL;
   //**tmp = NULL;
   noeud *r = (noeud*)malloc(sizeof(noeud));

   printf("Utilisation de l'arbre exemple \n");
   foret = (noeud**)malloc(sizeof(noeud*) * 14);
   foret[0] = (noeud*)malloc(sizeof(noeud));
   foret[1] = (noeud*)malloc(sizeof(noeud));
   foret[2] = (noeud*)malloc(sizeof(noeud));
   foret[3] = (noeud*)malloc(sizeof(noeud));
   foret[4] = (noeud*)malloc(sizeof(noeud));
   foret[5] = (noeud*)malloc(sizeof(noeud));
   foret[6] = (noeud*)malloc(sizeof(noeud));
   foret[7] = (noeud*)malloc(sizeof(noeud));
   foret[8] = (noeud*)malloc(sizeof(noeud));
   foret[9] = (noeud*)malloc(sizeof(noeud));
   foret[10] = (noeud*)malloc(sizeof(noeud));
   foret[11] = (noeud*)malloc(sizeof(noeud));
   foret[12] = (noeud*)malloc(sizeof(noeud));
   foret[13] = (noeud*)malloc(sizeof(noeud));

   r->val = 1;
   r->fg = foret[0];
   r->fd = foret[1];

   foret[0]->val = 2;
   foret[0]->fg = foret[2];
   foret[0]->fd = foret[3];

   foret[1]->val = 3;
   foret[1]->fg = foret[4];
   foret[1]->fd = foret[5];

   foret[2]->val = 4;
   foret[2]->fg = foret[6];
   foret[2]->fd = foret[7];

   foret[3]->val = 5;
   foret[3]->fg = foret[8];
```

```c
foret[3]->fd = foret[9];

foret[4]->val = 6;
foret[4]->fg = foret[10];
foret[4]->fd = foret[11];

foret[5]->val = 7;
foret[5]->fg = foret[12];
foret[5]->fd = foret[13];

foret[6]->val = 8;
foret[6]->fg = NULL;
foret[6]->fd = NULL;

foret[7]->val = 9;
foret[7]->fg = NULL;
foret[7]->fd = NULL;

foret[8]->val = 10;
foret[8]->fg = NULL;
foret[8]->fd = NULL;

foret[9]->val = 11;
foret[9]->fg = NULL;
foret[9]->fd = NULL;

foret[10]->val = 12;
foret[10]->fg = NULL;
foret[10]->fd = NULL;

foret[11]->val = 13;
foret[11]->fg = NULL;
foret[11]->fd = NULL;

foret[12]->val = 14;
foret[12]->fd = NULL;
foret[12]->fd = NULL;

foret[13]->val = 15;
foret[13]->fg = NULL;
foret[13]->fd = NULL;
a = r;

printf("Parcours Prefixe : ");
parcoursPrefixe(a);
printf("\nParcours Infixe : ");
parcoursInfixe(a);
printf("\nParcours Suffixe : ");
parcoursSuffixe(a);
```

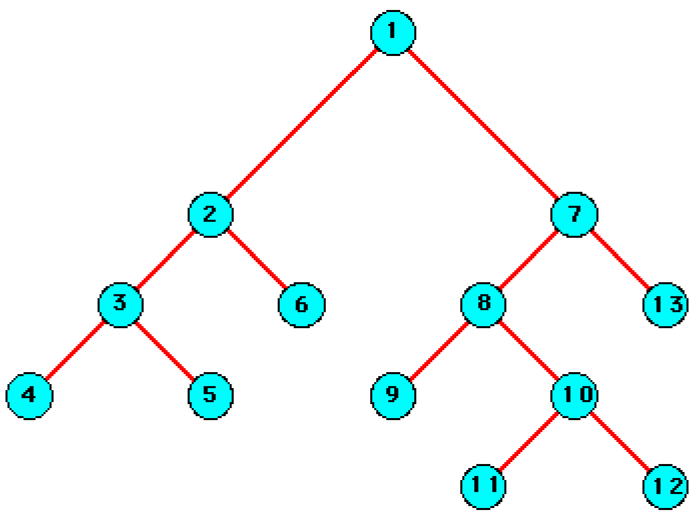
```
    printf("\n");
    printf("La taille de l'arbre est : %d \n", taille(a));
    printf("La hauteur de l'arbre est : %d \n", hauteur(a));
    printf("Il y a %d feuilles dans cet arbre. \n", nbFeuilles(a));
    printf("Il y a %d noeuds internes dans cet arbre. \n", nbNoeudsInterne(a));
    return 0;
}
```

6.1 Parcours préfixe

6.1.1 Présentation

Le parcours préfixe consiste à afficher la racine, suivi du fils gauche et enfin le fils droit. L'algorithme suivant montre comment s'y prendre.

6.1.2 Exemple d'exécution



6.1.3 Algorithme

Algorithme : Parcours préfixe (arbre a)

Debut

  Si a != $NULL$ **alors**

    Ecrire : a

    parcoursprefixe(a->filsgauche)

    parcoursprefixe(a->filsdroit)

  FinSi

Fin

6.1.4 Code en C

```
void parcoursPrefixe(arbre a)
{
```

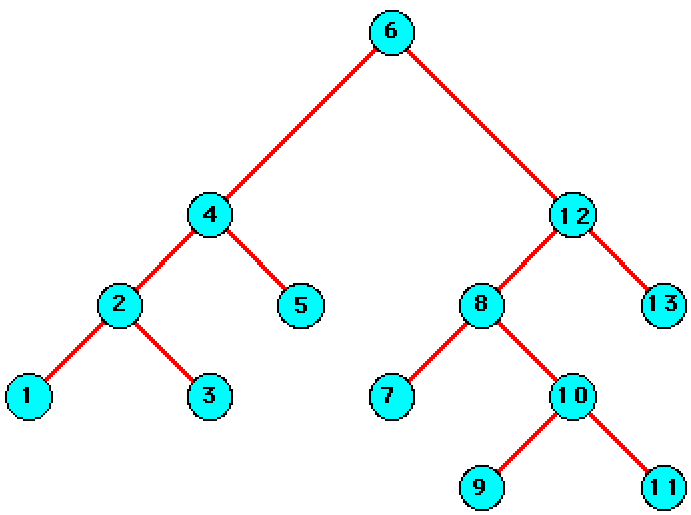
```
   if (a != NULL)
   {
      printf("%d \t", a->val);
      parcoursPrefixe(a->fg);
      parcoursPrefixe(a->fd);
   }
}
```

6.2 Parcours Infixe

6.2.1 Présentation

Le parcours infixe affiche tout d'abord le fils gauche, puis la racine et enfin le fils droit. L'algorithme ci-dessous représente ce parcours.

6.2.2 Exemple d'exécution



6.2.3 Algorithme

Algorithme : Parcours infixe (arbre a)
Debut
    Si a != $NULL$ **alors**
        parcoursinfixe(a->filsgauche)
        Ecrire : a
        parcoursinfixe(a->filsdroit)
    FinSi
Fin

6.2.4 Code en C

```
void parcoursInfixe(arbre a)
{
```

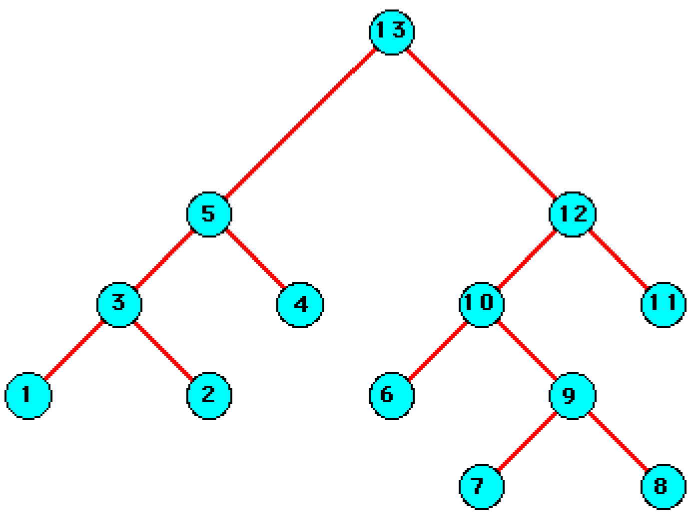
```
  if (a != NULL)
  {
     parcoursInfixe(a->fg);
     printf("%d \t", a->val);
     parcoursInfixe(a->fd);
  }
}
```

6.3 Parcours Postfixe

6.3.1 Présentation

Le parcours postfixe (ou suffixe) consiste à afficher le fils gauche, le fils droit puis la racine. Voici l'algorithme correspondant.

6.3.2 Exemple d'exécution



6.3.3 Algorithme

Algorithme : Parcours postfixe (arbre a)

Debut

  Si a != $NULL$ **alors**

    parcourspostixe(a->filsgauche)

    parcourspostfixe(a->filsdroit)

    Ecrire : a

  FinSi

Fin

6.3.4 Code en C

```
void parcoursSuffixe(arbre a)
```

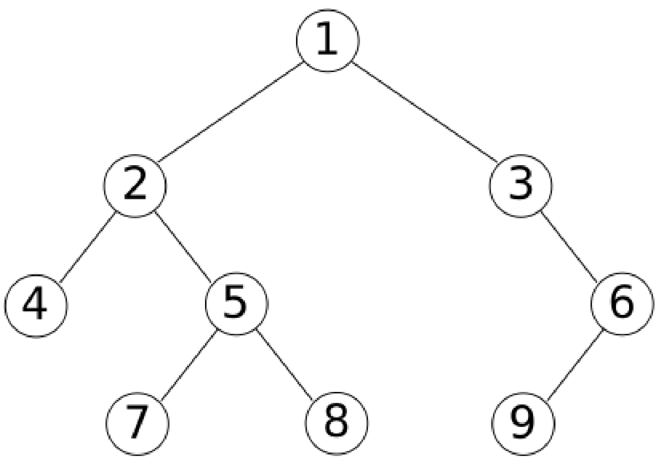
```
{
   if (a != NULL)
   {
      parcoursSuffixe(a->fg);
      parcoursSuffixe(a->fd);
      printf("%d \t", a->val);
   }
}
```

6.4 Taille d'un arbre

6.4.1 Présentation

L'algorithme suivant consiste à trouver quelle taille un arbre fait, c'est à dire combien d'élément il contient.

6.4.2 Exemple d'exécution



La taille de cet arbre est 9.

6.4.3 Algorithme

Algorithme : Taille (arbre a)
Variables :
t : entier
Debut
  Si $a = NULL$ **alors**
    $t \leftarrow 0$
  Sinon
    $t \leftarrow$ taille(a->filsgauche) + taille(a->filsdroit)
  FinSi
  Retourner : t
Fin

6.4.4 Code en C

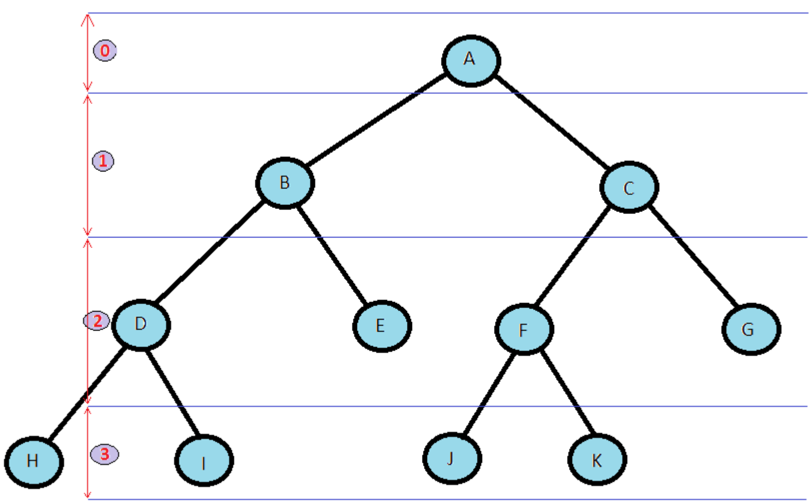
```
int taille(arbre a)
{
   int t;
   if (a == NULL)
   {
      t = 0;
   }
   else
   {
      t = 1 + taille(a->fg) + taille(a->fd);
   }
   return t;
}
```

6.5 Hauteur d'un arbre

6.5.1 Présentation

Cet algorithme consiste à calculer la hauteur d'un arbre donné.

6.5.2 Exemple d'exécution



La hauteur de cet arbre est 3.

6.5.3 Algorithme

Algorithme : Hauteur (arbre a)

Variables :

h, hg, hd : entier

Debut

Si $a = NULL$ **alors**

  Retourner : 0

Sinon

  $hg \leftarrow$ hauteur(a->filsgauche)

  $hd \leftarrow$ hauteur(a->filsdroit)

  Si $hd > hd$ **alors**

    $h \leftarrow 1 + hg$

  Sinon

    $h \leftarrow 1 + hd$

  FinSi

FinSi

Retourner : h

Fin

6.5.4 Code en C

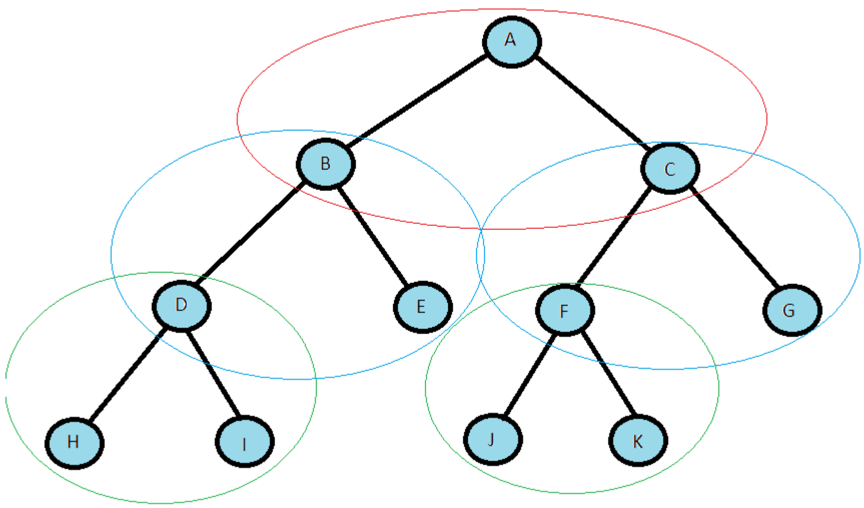
```
int hauteur(arbre a)
{
   int h, hg, hd;
   if (a == NULL)
   {
      h = 0;
   }
   else
   {
      hg = hauteur(a->fg);
      hd = hauteur(a->fd);
      if (hg > hd)
      {
         h = 1 + hg;
      }
      else
      {
         h = 1 + hd;
      }
   }
   return h;
}
```

6.6 Nombre de feuille d'un arbre

6.6.1 Présentation

L'objectif de l'algorithme suivant est de calculer le nombre de feuille de l'arbre.

6.6.2 Exemple d'exécution



Cet arbre possède 5 feuilles.

6.6.3 Algorithme

Algorithme : nbFeuille (arbre a)

Debut

  Si $a = NULL$ **alors**

    Retourner : 0

  FinSi

  Si a->*filsdroit* $= NULL$ *ET* a->*filsgauche* $= NULL$ **alors**

    Retourner : 1

  Sinon

    Retourner : nbFeuille(a->filsgauche) + nbFeuille(a->filsdroit)

  FinSi

Fin

6.6.4 Code en C

```
int nbFeuilles(arbre a)
{
  if(a == NULL)
  {
    return 0;
  }
  if(a->fd == NULL && a->fg == NULL)
  {
```

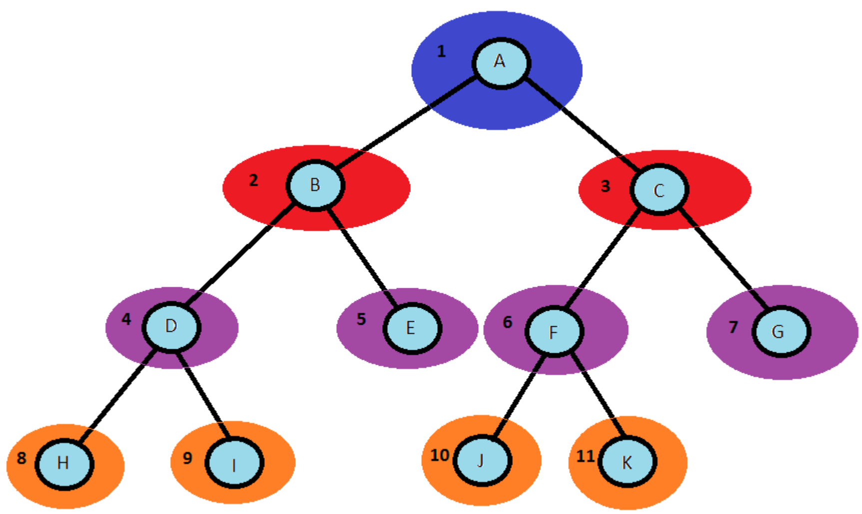
```
        return 1;
    }
    else
    {
        return nbFeuilles(a->fg) + nbFeuilles(a->fd);
    }
}
```

6.7 Nombre de noeud interne d'un arbre

6.7.1 Présentation

Pour finir, cet algorithme consiste à trouver le nombre de noeud interne de l'arbre. On incrémente une variable lorsque l'on tombe sur un fil.

6.7.2 Exemple d'exécution



Comme l'image le montre, cet arbre possède 11 noeuds internes.

6.7.3 Algorithme

Algorithme : nbNoeudInterne (arbre a)

Debut

  Si $a = NULL$ **alors**

    Retourner : 0

  FinSi

  Si $a\text{->}filsdroit\ != NULL\ OU\ a\text{->}filsgauche\ != NULL$ **alors**

    Retourner : nbNoeudInterne(a->filsgauche) + nbNoeudInterne(a->filsdroit)

  Sinon

    Retourner : 0

  FinSi

Fin

6.7.4 Code en C

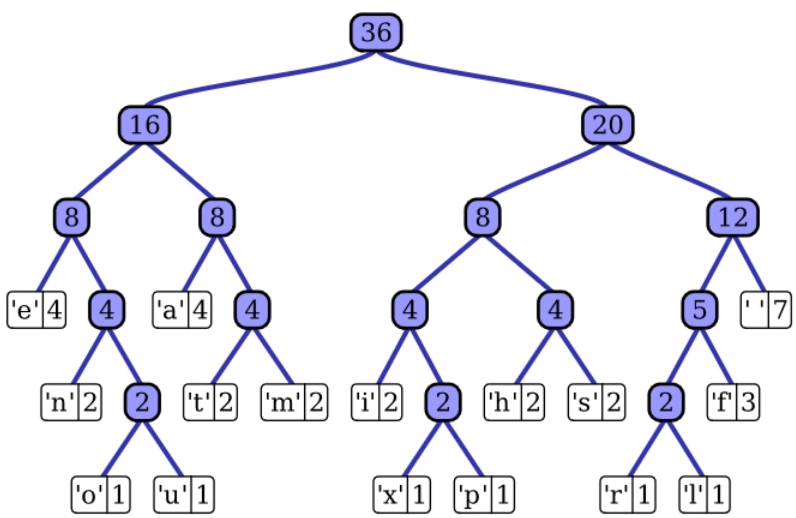
```c
int nbNoeudsInterne(arbre a)
{
   if(a == NULL)
   {
      return 0;
   }
   if(a->fd != NULL || a->fg != NULL)
   {
      return (1 + nbNoeudsInterne(a->fg) + nbNoeudsInterne(a->fd));
   }
   else
   {
      return 0;
   }
}
```

6.8 Codage de Huffman

6.8.1 Présentation

Le codage de Huffman est un algorithme de compression de données sans perte.

6.8.2 Exemple d'exécution



"This is an exemple of a huffman tree"

6.8.3 Algorithme

Algorithme : Huffman (arbre a)

Variables : *table* : table de symbole
i,n : entiers
a : arbre
$foret$: foret
Debut
  Pour i *allant de 1 à* n **faire**
    Allouer($foret[i].arbre$)
    $a \leftarrow foret[i].arbre$
    $a \rightarrow pere \leftarrow NULL$
    $a \rightarrow filsG \leftarrow NULL$
    $a \rightarrow filsD \leftarrow NULL$
    $a \rightarrow data \leftarrow table.symbole[i].data$
    $foret[i].poids \leftarrow table.symbole[i].poids$
  FinPour
  TantQue $n > 1$ **faire**
    rechercheIndice($foret$,i,j)
    $a \leftarrow arbrePere(foret[i].arbre, foret[j].arbre)$
    $foret[i].poids \leftarrow foret[i].poids + foret[j].poids$
    $foret[j].arbre \leftarrow foret[n].arbre$
    $foret[j].poids \leftarrow foret[n].poids$
    $n \leftarrow n - 1$
  FinTantQue
  Retourner : $foret[1].arbre$
Fin

Algorithme : arbrePere

Variables : $racine$,g,d : arbre
Debut
  Allouer($racine$)
  $racine.pere \leftarrow NULL$
  $racine.data \leftarrow NULL$
  $racine.filsG \leftarrow g$
  $racine.filsD \leftarrow d$
  $g.pere \leftarrow racine$
  $d.pere \leftarrow racine$
  Retourner : $racine$
Fin

6.8.4 Code en C

Pas de code disponible.

Algorithme : rechercheIndice

Variables : $foret$: foret pondérée
i,j,k : entiers
Debut

```
Si foret[1].poids < foret[2].poids alors
    j ← 1
    i ← 2
FinSi
j ← 2
i ← 1
Pour k allant de 3 à n faire
    Si foret[k].poids < foret[j].poids alors
        i ← j
        j ← k
    Sinon
        i ← k
    FinSi
FinPour
Retourner : i, j
```

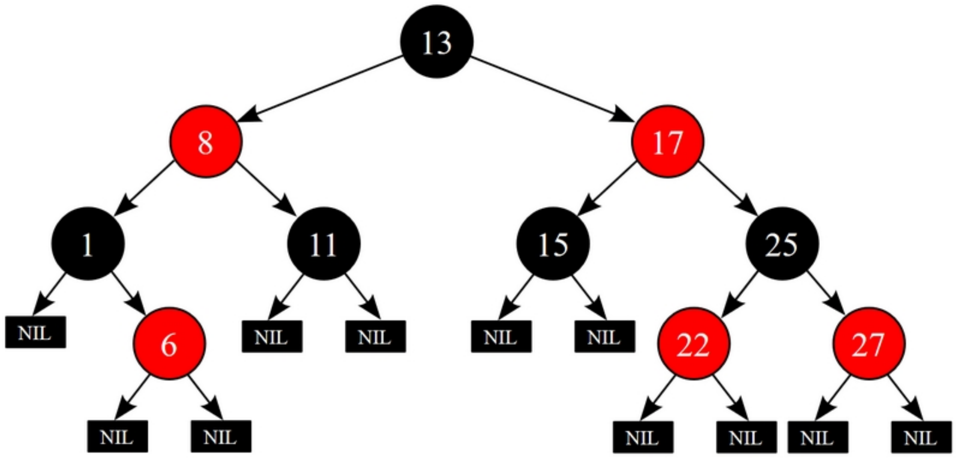
Fin

6.9 Arbre rouge-noir

6.9.1 Présentation

Un arbre rouge et noir est un arbre binaire de rechercher particulier. Cet arbre possède 4 propriétés fondamentales :
- Un nœud est soit rouge soit noir,
- La racine est noire,
- Le parent d'un nœud rouge est noir,
- Le chemin de chaque feuille à la racine contient le même nombre de nœuds noirs.

6.9.2 Exemple d'exécution



Voici un exemple d'arbre rouge-noir.

6.9.3 Code en C

Pas de code disponible.

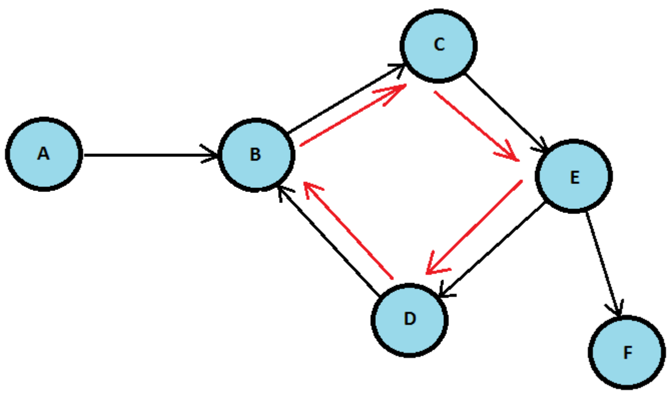
7 Graphe

7.1 Recherche de cycle

7.1.1 Présentation

L'algorithme proposé ci-dessous permet de détecter si il y a un cycle dans un graphe donné. C'est à dire si le graphe passe par un sommet déjà visité auparavant.

7.1.2 Exemple d'exécution



7.1.3 Algorithme

Algorithme : Recherche de cycle

Variables : $listeAdh$: liste d'adjacence
$tabVisites$: tableau de parcours
$cycle$:booléen
$nbSommets$:entier
cel :cellule

Debut
  Allouer($tabVisites(nbsommets)$)
  Initialiser($tabVisites(0)$)
  $cycle \leftarrow faux$
  Pour *i allant de 1 à nbSommets* **faire**
    $cel \leftarrow tab[i]$
    TantQue cel *!=NULL ET* $cel \rightarrow value > i$ *ET !cycle* **faire**
      $tabVisites[cel \rightarrow value] \leftarrow tabVisites[cel \rightarrow value] + 1$
      Si $tabVisites[cel \rightarrow value] = 2$ **alors**
        $cycle \leftarrow vrai$
      FinSi
      $cel \leftarrow cel \rightarrow next$
    FinTantQue
  FinPour
Fin

7.1.4 Code en C

```
int cycle (Liste **listeAdj, int n){
   int trouve = 1;
   int *tabVisites = (int*) malloc (sizeof (int)*n);
   for (int i=0; i<n; i++)tabVisites[i]=0;
   Cellule *courant;
   int j;
   for(int i=0;i<n;++i){
      courant=listeAdj[i]->premier;
      while(courant!=NULL && trouve != 0){
         j=courant->val;
         if(j>i){
            if((tabVisites[i]==0)||(tabVisites[j]==0)){
               tabVisites[i]++;
               tabVisites[j]++;
            }
         }
         else trouve =0;
         courant=courant->next;
      }
   }
   return trouve;
}
```

7.2 Fortement Connexe

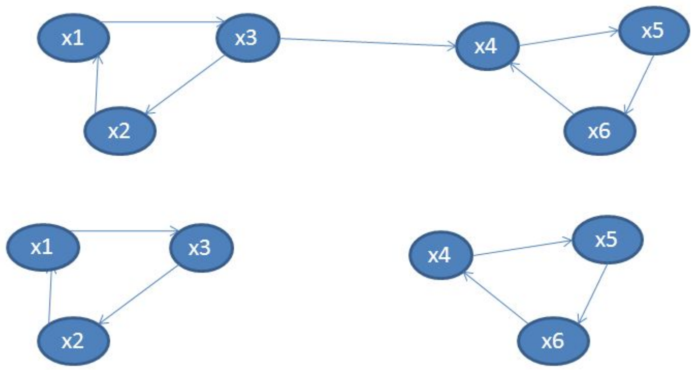
7.2.1 Présentation

En théorie des graphes, une composante fortement connexe d'un graphe orienté G est un sous-graphe de G possédant la propriété suivante :
Pour tout couple (u, v) de sommets dans ce sous-graphe, il existe un chemin de u à v.

7.2.2 Exemple d'exécution

On appelle ***composante fortement connexe*** d'un graphe orienté un sous-graphe ***fortement connexe*** maximal, c.a.d. un sous-graphe fortement connexe qui n'est pas strictement contenu dans un autre sous-graphe fortement connexe.

On appelle ***composante connexe*** d'un graphe non orienté un sous-graphe connexe maximal.



Le graphe G contient 2 composantes fortement connexes : x1x2x3 et x4x5x6

32

7.2.3 Algorithme

Algorithme : Fortement Connexe

Variables : i : entier

M : Matrice

Debut

  Pour *i allant de 1 à taille* **faire**

    Si $m[0, i]$ *= 0 OU* $m[i, 0]$ *= 0* **alors**

      Retourner : faux

    FinSi

  FinPour

  Retourner : vrai

Fin

7.2.4 Code en C

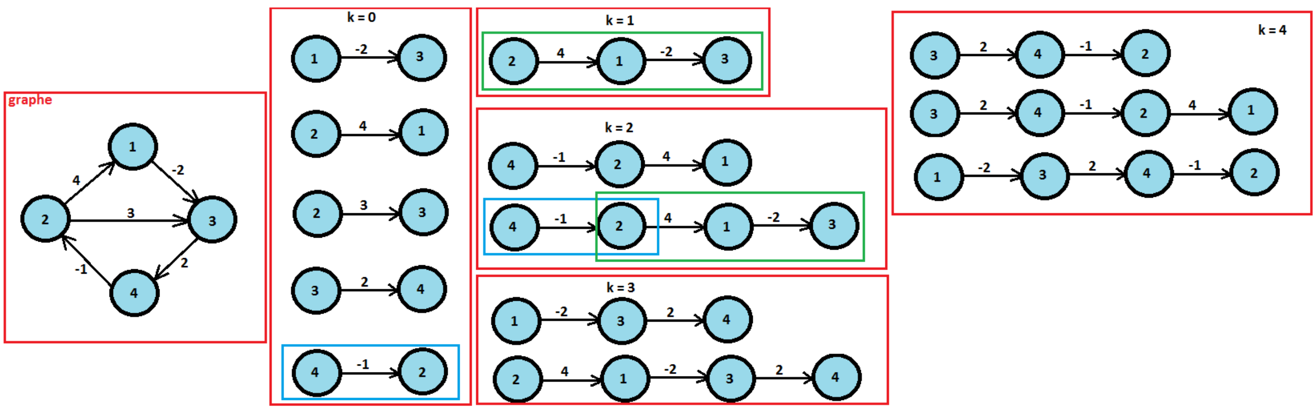
```c
int fortementConnexe(int (*matrice)[5], int taille){
   for(int i=0; i<5;++i){
      if((matrice[0][i]==0) || (matrice[i][0]==0))return 1;
   }
   return 0;
}
```

7.3 Algorithme de Floyd

7.3.1 Présentation

L'algorithme de Floyd (Floyd-Warshall) est un algorithme déterminant les distances des plus courts chemins entre toutes les paires de sommets dans un graphe orienté et pondéré.

7.3.2 Exemple d'exécution



- Pour k = 0, les seuls chemins sont les arcs directs.
- Pour k = 1, on regarde les chemins où le sommet 1 peut être un sommet intermédiaire. On trouve le chemin $2 \to 1 \to 3$ qui moins lourd que $2 \to 3$.
- Pour k = 2, maintenant, le sommet 2 peut être un sommet intermédiaire. La boîte rouge et la boîte bleu montre que le chemin $4 \to 2 \to 1 \to 3$ est la concaténation du chemin $4 \to 2$ et $2 \to 1 \to 3$ avec le sommet 2 comme sommet intermédiaire. Notons que le chemin $4 \to 2 \to 3$ n'est pas considéré car c'est $2 \to 1 \to 3$ qui un chemin le plus léger obtenu à l'itération précédente et non $2 \to 3$.
- Pour k = 3, on trouve encore d'autres chemins.
- Pour k = 4, on a trouvé des plus courts chemins entre tous les sommets.

7.3.3 Algorithme

Algorithme : Matrice prédécesseurs

Variables : i, j, k, nb_sommet : entiers
M : Matrice
P : Matrice des prédécesseurs

```
Debut
    Initialiser(P)
    Pour i allant de 1 à nb_sommet faire
        Pour j allant de 1 à nb_sommet faire
            P(i,j) ← i
        FinPour
    FinPour
Fin
```

Algorithme : Algo de Floyd

Variables : i, j, k, nb_sommet : entiers
M : Matrice
P : Matrice des prédécesseurs
Debut
  Pour *k allant de 1 à nb_sommet* **faire**
    Pour *i allant de 1 à nb_sommet* **faire**
      Pour *j allant de 1 à nb_sommet* **faire**
        Si $M(i,k) + M(k,j) < M(i,j)$ **alors**
          $M(i,j) \leftarrow M(i,k) + M(k,j)$
          $P(i,j) \leftarrow k$
        FinSi
      FinPour
    FinPour
  FinPour
Fin

7.3.4 Code en C

```c
int** floyd(int(*l)[4],int n){
  int **p=(int**) malloc (sizeof (int*)*n);
  for (int i=0; i<n; i++) p[i] = (int*) malloc (sizeof (int)*n);

  for (int i=0; i<n; i++){
    for (int j=0; j<n; j++){
      p[i][j] = i;
      for (int k=0; k<n; k++){
        if (l[i][j] > l[i][k] + l[k][j] && k!=i && k!=j){
          if(l[i][k] == 999 || l[k][j] == 999)l[i][j] = 999;
          else l[i][j] = l[i][k] + l[k][j];
          p[i][j] = k;
        }
      }
    }
  }
  return p;
}
```