COURS ALGORITHMIQUE AVANCEE
PARTIE I & II

I Introduction

Ce document vient en supplément aux diapositives vues en cours. C'est un recueil vous permettant d'avoir à disposition les choses inutiles à connaître par cœur, et un résumé succinct du cours. **Attention ce document n'a pas pour vocation de remplacer la prise de notes** que vous effectuerez en cour mais est simplement là comme support et vérification notamment de formules mathématiques, des algorithmes …

II Plan générique du cours

Le plan est donné ci-dessous :

III Partie I : Algorithmique fondamentale

III.1 Retour sur le C :

Notion de programme : on réalise une application informatique en traduisant notre intention dans un premier temps en un langage très structuré (le langage de programmation) puis le compilateur transformera celui-ci en langage binaire (ou plutôt en assembleur).

Structure d'un programme :
On trouve dans un programme, par ordre de complexité :

1. Litteraux : constantes
2. Variables : quatre caractéristiques : nom, type, valeur et adresse
3. Expressions : composé de litteraux, variables et opérateurs. Brique de base de la programmation. Opérateurs logique, arithmétiques, comparaison et d'affectation
4. Instructions : Soit des expressions, soit des instructions de contrôle : tests et boucles.
5. Méthodes : Bloc d'instructions agissant sur des données paramétrables. Attention à ne pas confondre utilisation et définition de méthode.

Allocation dynamique :
Protéger les données notamment de leur destruction (allocation) et réserver l'espace mémoire juste nécessaire (dynamique).

III.2 Algorithmique et programme – Récursivité :

Conception d'application :
Trois niveaux : spécifications (algorithmique informelle), expression (algorithme, pseudo code) et implémentation (programme).

Bibliographie :

- Beauquier, Berstel, Chrétienne : *Éléments d'algorithmique*, Masson, 1993
- Sedgewick : *Algorithmes en C*, InterÉditions, 1992
- Aho, Hopcroft, Ullman :*Structures de données et algorithmes*, InterÉditions, 1987
- Cormen, Leiserson, Rivest : *Algorithmes*, Dunod, 1994

Exemple : tri à bulle 1 :

```
répéter {
  i : = 1 ;
  tant que i < n et s_i ≤ s_i+1
  faire i := i + 1 ;
  si  i < n  alors {
      échanger (s_i, s_i+1);
      inversion := vrai ;
  }
  sinon
      inversion := faux ;
}
tant que  inversion = vrai ;
```

Complexité :

- T(algo,d) : Calcul du temps d'un algorithme algo pour une donnée d
- T(expression) = constante
- T(si C alors I sinon J) ≤ T(C) + max(T(I),T(J))
- T(pour i := e1 à e2 faire Ii) = T(e1) + T(e2) + somme(T(Ii))
- T(fonction récursive) = équation de récurrence

Trois mesure de complexité : min, moy et max
Grand tho : $T(n) = O(f(n))$ ssi $\exists\ c > 0\ \ \exists\ N > 0\ ,\ \forall n > N\ , T(n) \leq c.f(n)$Grand omega : $T(n) = \Omega(f(n))$ ssi $\exists\ c > 0\ \ \exists\ N > 0\ \ \forall\ n > N\ \ \ T(n) \geq\ c.f(n)$
Grand theta : $T(n) = \Theta(f(n))$ ssi $T(n) = O(f(n))$ et $T(n) = \Omega(f(n))$

Ordre de grandeur de complexité : f = 1 , log(n), n, n log(n), n^2, n^3 … 2^n

Exemple : tri à bulle 2 :

```
pour  j  :=  n - 1  à  1  pas -1  faire
       pour  i  := 1  à  j  faire
              si si > si+1  alors  échanger(si, si+1) ;
```

Exemple : PGCD :
Algorithme pseudo code récursif :

```
procedure PGCD(n,m: integer): integer;
       {n>=0, m>=0}
       begin
              si m = 0 alors retour (n)
              sinon retour (PGCD(m, n mod m));
       end;
```

Algorithme pseudo code itératif :

```
procedure PGCD(n,m: integer): integer;
       {n>=0, m>=0}
       begin
              temp := 0;
              tant que m > 0
              faire
                     temp := m;
                     m := n mod m;
                     n := temp;
              fin tant que;
              return (n);
       end;
```

Implémentation récursive :

```
int PGCD(int n, int m) {
       /* n>=0, m>=0 */
       if (m == o) return (n);
       else return ( PGCD(m, n % m));
}
```

Implémentation itérative :

```
int PGCD(int n, int m){
       /* n>=0, m>=0 */
       int temp;
       while (m > 0){
              temp = m;
              m = n % m;
              n = temp;
       }
       return (n);
}
```

Elimination de la récursivité :
Passer de :

```
fonction  F(x) ;
début
       si  C(x)  alors  { I;  retour (y); }
       sinon  {J;  retour (F(z)); }
fin
```

à :

```
fonction  F(x) ;
début
       tant que non C(x) faire
              {J;  x := z; }
       I;
       retour (y);
fin
```

Exemple : Tour de Hanoï :

```
fonction H(n, x, y, z) ;
début
  si  n = 1  alors   écrire (x-y) ;
  sinon {
        H(n-1, x, z, y);
        écrire(x-y);
        H(n-1, z, y, x);
  }
fin
```

Exemple : Problème du sac à dos :

```
fonction CHIFFRES (t, i) ;
/ * prend la valeur vrai ssi il
    existe une partie de (i, i+1, ..., n)
    qui donne la somme t ; écrit les nombres correspondants */
début
       si (t = 0 ) alors retour ( vrai )
       sinon si ( t < 0 ) ou ( i > n ) alors retour ( faux )
       sinon si CHIFFRES ( t - w_i, i+1) alors
       { écrire (w_i ) ; retour ( vrai ) ; }
       sinon retour (CHIFFRES ( t, i+1) ) ;
fin
```

IV Partie II : Types abstraits simples et tris

IV.1 Types abstraits simple :

Types abstraits :

Façon d'ordonner les données. Ne dépend pas de la façon informatique de les représenter. Deux caractéristiques : son interface i.e. l'ensemble des outils manipulant le concept, et son implémentation.

Exemples de types abstraits :

Ensemble, Vecteur, Liste, Arbre, Pile…

Listes :

Accès séquentiel d'une suite d'élément L = (e_1,e_2,…,e_n).

Quelques opérations sur les listes :

- Longueur d'une suite L = (e_1,e_2,…,e_n) : n
- Liste vide ()
- Position de e_i dans L : i
- Successeur de e_i : e_{i+1} si il existe
- Tête de L : e_1 si il existe

Exemple d'implémentation : opérateur unique :

```
fonction UNIQUE ( L  liste ) : liste ;
/* supprime de L les copies d'éléments */
 p,q sont des adresses
début
  p ← Tête ( L ) ;
  tant que p défini faire
    q ← p ;
    tant que Succ ( q, L ) défini faire
      si  Val ( p, L ) = Val ( Succ (q, L ))
      alors Enlever ( Succ (q, L ))
      sinon  q ← Succ ( q, L ) ;
             p ← Succ ( p, L ) ;
       retour  L ;
fin
```

Opérateurs fondamentaux sur les listes :

- Liste_vide : → Liste
- Tête : Liste → Adresse
- Fin : Liste → Liste
- Cons : Élément x Liste → Liste
- Premier : Liste → Élément
- Elt : Adresse → Élément
- Succ : Adresse → Adresse

Exemple d'implémentation de listes : par tableau

```
struct {
  int  long ;
  Element  table [MAX] ;
}  Liste;
```

```
fonction  ajouter (L  Liste, p  Adresse, e  Element) : Liste ;
début
    si L.long = MAX  alors
       erreur (“opération impossible”)
    sinon  si  p < -1  ou  p ³ L. long  alors
       erreur (“opération impossible”)
    sinon
       pour  k ← L. long - 1  à  p + 1  pas – 1
       faire L.table[k + 1] ← L.table[k];
    L.table [p + 1] ← e ;
    L.long ← L.long + 1 ;
    retour  (L) ;
fin
```

Exemple d’implémentation de listes : par pointeur

```
typedef struct CELLULE {
       element elt ;
       struct CELLULE  *  succ ;
} cellule ;
typedef  cellule  *  adresse ;
typedef  adresse  liste ;

liste ajouter ( liste L, adresse p, element e ) {
       temp = (adresse) malloc(sizeof(cellule)) ;
       if ( temp == NULL ) erreur() ;
       else {
              temp -> elt = e ;
              temp -> succ  =  p -> succ ;
              p -> succ  =  temp ;
       }
       return L ;
}
```

Implémentation par curseur :

```
type  Liste = Adresse = -1 .. MAX ;
Cellule =  structure {
       elt : Element ;  succ : Adresse ;
}
Var   Espace :  table [0 .. MAX]  de  cellule ;
```

Transfert de cellule entre liste : permet de gérer efficacement les cellules disponibles

```
fonction  TRANSFERT (D, S : Listes;  p, q  : Adresses) : Adresse ;
début
       temp ← q->succ ;
       q->succ ← temp->succ ;
       temp->succ ← p->succ ;
       p->succ ← temp ;
       retour  ( temp ) ;
fin
```

Exemple d’ajout et de retrait :

```
Listes possédant une tête de liste
LIBRE : liste des cellules disponibles

fonction  AJOUTER (L  Liste, p  Adresse, e  Element) : Liste ;
début
       si  Vide (LIBRE)  alors
              erreur (“mémoire saturée”)
       sinon
              temp ← TRANSFERT (L, LIBRE, p, LIBRE) ;
              temp->elt ← e ;
       retour  ( L ) ;
fin
```

```
fonction  ENLEVER (L  Liste, p  Adresse) : Liste ;
début
      si  p->succ  est défini  alors
            TRANSFERT (LIBRE, L, LIBRE, p) ;
      retour  (L) ;
fin
```

Piles :

Accès au sommet de la pile (tête de liste)

Opérations standards :

Pile_vide : → Pile

Empiler : Pile x Elément → Pile

Dépiler : Pile → Pile

Sommet : Pile → Elément

Vide : Pile → Booléen

Implémentation :

```
Type  Pile = structure {
      somm : -1 .. MAX ;
      elt : table [0 .. MAX]  of  Element ;
}

fonction  Pile_vide (P Pile) :  Pile ;
début
      P.somm ← -1 ;  retour(P);
fin

fonction Empiler (P Pile, e Element) : Pile ;
début
      si  P.somm = MAX  alors erreur()
      sinon  début
            P.somm ← P.somm + 1 ;
            P.elt [P.somm] ← e ;
            retour(P)
      fin
fin

fonction  Dépiler (P Pile) : Pile ;
début
      si  Vide (P)  alors  erreur ;
     sinon début
            P.somm  ←  P.somm - 1 ;
            retour (P) ;
      fin
fin

fonction  Sommet (P Pile) : Element ;
début
      si  Vide (P)  alors  erreur
      sinon  retour (P.elt [P.somm]) ;
fin

fonction  Vide (P Pile) : booléen ;
début
      retour (P.somm = -1);
fin
```

Utilisation des piles pour simuler la récursion :

```
fonction  H (n, x, y, z) ;   /* version itérative */
début
      P  ←  Pile_vide ;   Empiler (P, (n, x, y, z)) ;
      tant que non  Vide (P)  faire
      début
            e  ←  Sommet (P);   Dépiler (P);
            si  e = x'-y' ou e = (1, x', y', z')  alors
                  écrire (x'-y')            sinon
            début     /*  e = (n, x', y', z') */
                  Empiler (P, (n-1, z', y', x')) ;
                  Empiler (P, x'-y') ;
                  Empiler (P, (n-1, x', z', y')) ;
            fin
      fin
fin
```

Files :

Suite dont on insère les éléments à un bout et ressort les éléments de l'autre.

Opérations :

File_vide : → File

Ajouter : File x Element → File

Enlever : File → File

Premier : File → Element

Vide : File → Booléen

Implémentation par file chaînée :

```
Types File = cellule *;
cellule = structure {
      elt  : Element;   succ  : File ;
}

fonction  File_vide (F File)  : File ;
début
      créer une cellule d'adresse F ;
      F->succ  ←  F ;  retour ( F ) ;
fin

fonction Ajouter (F File, e Element)  : File ;
début
      F->elt  ←  e ;
      Transfert (F, LIBRE, F, Tête (LIBRE)) ;
      F  ←  F->succ ;   retour ( F)
fin

fonction Enlever (F File)  : Element ;
début
      Transfert ( LIBRE, F, Tête (LIBRE), F ) ;
      retour (F -> elt) ;
fin

fonction  Premier ( F File )  : Element ;
début
      retour ( F -> succ -> elt ) ;
fin

fonction Vide( F File )  : Booléen ;
début
      retour ( F = F -> succ ) ;
fin
```

IV.2 Tris et type abstrait simple :

Tri par sélection :

```
fonction Tri_Selection(t table [1…n]) : table ;
début
      pour  i ← 1  à  n-1  faire {
        min ← i ;
        pour  j ← i + 1  à  n  faire
            si  t [ j ]  < t [ min ]  alors  min ← j ;
        temp ← t  [ i ]  ;
        t [ i ] ← t [ min ] ;
        t [ min ] ← temp ;
      }
      retour (t);
fin
```

Tri par insertion :

```
fonction Tri_Insertion(t table [1…n]) : table ;
début
      t [ 0 ] ←−∞ ;
      pour  i ← 2  à  n  faire
      {   k ← i - 1;  temp ← t [ i ] ;
          tant  que  temp < t [ k ] faire        {
             t [ k + 1] ← t [ k ] ;
             k ← k - 1 ;
          }
          t [ k + 1] ← temp ;
      }
      retour (t) ;
fin
```

Tri à bulle (final) :

```
fonction  Tri_Bulles ( t  table [1…n] ) : table ;
début
      i ←1 ;
      tant que  i ≤n - 1  faire {
         dernier_échange ← n ;
         pour k ← n  à  i + 1  pas - 1  faire
             si  t [ k - 1]  >  t [ k ]  alors {
               temp ← t [ k - 1 ] ;
               t [ k - 1 ] ← t [ k ] ;
               t [ k ] ← temp ;
               dernier_échange ← k ;
             }
         i ← dernier_échange ;
      }
      retour(t);
fin
```

Tri par partage / fusion :

Idée : Liste à trier ⇒ partagée en 2 sous liste plus petites ⇒ Tri des listes plus petites ⇒ Fusion des petites listes triées

Complexité : T_{max} = o(n log(n)) si les deux sous listes ont une taille avoisinant la moitié de la liste initiale.

Quicksort / tri rapide :

```
fonction Tri_Rapide ( t  table [1…n] ) : table ;
début
      appliquer TR (1, n )  à  t ;
      retour( t ) ;
fin

procédure  TR(i, j)
/*  classe la partie t [ i...j ] de t  */
début
      si i < j  alors
      {   p ← choix (i, j) ;
          k ← PARTAGE (i, j, p) ;
          TR (i, k - 1) ; TR (k + 1, j) ;
      }
fin

fonction  PARTAGE ( i, j, p ) ;
/*   partage t suivant le pivot t [ p ]    */
début
      g ← i; d ← j;
      échanger (t [ p ], t [ j ] ); pivot = t [ j ] ;
      répéter
        tant que  t[g] < pivot  faire g ← g + 1 ;
        tant que d ≥ g et t[d] ≥ pivot  faire  d ← d - 1 ;
        si  g < d  alors {
            échanger (t [ g ] , t [ d ] ) ;
            g ← g + 1 ; d ← d - 1 ;
        }
      jusqu 'à ce que  g > d ;
      échanger  ( t [ g ], t [ j ] ) ;
      retour  ( g ) ;
fin
```

Choix i,j : aléatoire ou alors :

```
fonction choix (i, j) : indice ;
début
     pour k ← i  à  j - 1 faire
         si  t [k] > t [k + 1] alors
              retour  k ;
     retour  -1 ;  /* cas t [i] ≤ t [i+1] ≤ ... ≤ t [j] */
fin
```

Version itérative :

```
fonction Tri_rapide_iteratif ( t  table [1…n] ) : table ;
début
     P ← empiler(Pile_vide, (1,n)) ;
     tant que  non vide(P)  faire
         (i, j) ← sommet(P) ;  dépiler(P) ;
         si  i < j  alors
           p ← choix (i, j) ;
           k ← PARTAGE (i, j, p) ;
           P ← empiler(P, (k +1, j)) ;
           P ← empiler(P, (i,  k -1)) ;
     retour t;
fin
```

IV.3 Tris et tables de hachage :

Hachage :

Idée : Établir une relation entre un élément et l'adresse à laquelle il est rangé en mémoire

Théorème :

Si les éléments de *table* [1 … *n*] et *x* sont choisis uniformément dans un intervalle [*a*,*b*], le temps moyen d'exécution de la recherche par interpolation est O (log log *n*)

Table de hachage :

Table dont les indices sont dans [0 .. *B*-1]

Fonction de hachage :

h : éléments → [0 .. *B*-1] non injective en général

Et si même clé :

Hachage ouvert : avec listes, triées ou non.

Hachage fermé : linéaire, quadratique, aléatoire, uniforme, double, ...

Hachage ouvert :

Exemple de fonction de hachage :

```
fonction h(x : mot) : indice ;
début
      somme := 0 ;
      pour i ← 1 à longueurmaxi  faire
            somme  ←  somme + ord(x [i]) ;
      retour (somme mod B) ;
fin
```

Implémentation :

```
const B = {constante ad hoc};
type  liste = ↑ cellule ;
cellule = struct
      elt : élément ;
      suivant : liste
end;
dictionnaire = array [0 .. B-1] of liste ;

fonction VIDER() : dictionnaire
début
      pour i ← 0 à B-1  faire A[i] ← NULL ;
      retour A ;
fin

fonction  ELEMENT (x élément,A dictionnaire) : booléen
début    p ← A [ h(x) ] ;
      tant que p ≠ NULL  faire
            si p->elt = x  alors retour vrai
            sinon p ← p->suivant ;
      retour  faux ;
fin

fonction AJOUTER(x élément,A dictionnaire);
début
      si non ELEMENT (x, A)  alors {
            i  ← h (x) ;
            p ← A [ i ] ;
            allouer  A[i];
            A[i]->elt  ← x;
            A[i]->suivant  ← p;
      }
      retour A ;
fin
```

Hachage fermé :

• liens implicites : gain de place

• taille limitée

• ré-allocation en cas de débordementImplémentation :

```
const B = {constante ad hoc} ;
vide       =  {constantes particulières} ;
disponible = {distinctes des éléments mais de même type} ;
type  dictionnaire = array[0… B-1]  of  éléments ;

fonction VIDER() : dictionnaire ;
début
      pour  i ← 0  à  B-1  faire
            A[i] ← vide ;
      retour A;
fin

fonction  POSITION (x élément,A dictionnaire) : indice ;
/* calcule la seule position possible où ajouter x dans A */
début  i ← 0;
   tant que ((i < B) et  (A[hi(x)] ≠ x)
            et  (A [hi(x)] ≠ vide)
            et  (A[hi(x)] ≠ disponible))
   faire  i ← i + 1 ;
   retour (hi(x)) ;
fin
fonction  POSITIONE (x élément,  A  dictionnaire) : indice
début
      hi ← h(x) ;
      dernier ← (hi + B-1)  mod  B ;
      tant que  hi ≠ dernier et (A [hi] ∉ {x, vide}) faire
            hi ← (hi + 1)  mod  B  ;
      retour  (hi)
fin
fonction  ELEMENT (x  élément,  A  dictionnaire) : boolean
début
      si  A [POSITIONE (x, A)] = x  retour  vrai ;
      sinon  retour  faux ;
fin
fonction ENLEVER (x  élément,  A  dictionnaire) : dictionnaire
début
      i ← POSITIONE (x, A) ;
      si A[i] =  x  alors  A[i] ← disponible ;
      retour A ;
fin
fonction  POSITIONA (x  élément,  A dictionnaire) : indice
début  hi ← h (x) ;  dernier := (hi + B-1)  mod  B ;
      tant que ((hi ≠ dernier)  et
               (A [hi] ∉ {x, vide, disponible}))
      faire
            hi ← (hi + 1)  mod  B  ;
      retour  hi ;
fin
fonction  AJOUTER  (x  élément,  A dictionnaire) : dictionnaire
début
      i ← POSITIONA(x, A) ;
      si  A [ i ] ∉ {vide, disponible}  alors {
            A [ i ] ← x ;
            retour A;
      }
      sinon si A [i] ≠ x  alors erreur (" table pleine " ) ;
fin
```