es bases de l'algorithmique

Un **algorithme** est une procédure de calcul bien définie qui prend en entrée un ensemble de valeurs et qui délivre en sortie un ensemble de valeurs.

Le but de ce cours est de vous apprendre les bases de l'algorithmique.

L'auteur

M. Delest

L'article

Publié le 3 novembre 2016

DÉBUTANT

Version PDF Version hors-ligne

ePub, Azw et Mobi

I. Introduction▲

I-A. Notion d'algorithme▲

Définition 1.1. Un **algorithme** est une procédure de calcul bien définie qui prend en entrée un ensemble de valeurs et qui délivre en sortie un ensemble de valeurs.

Exemple 1.1
Problème : trier une suite de nombres entiers dans l'ordre croissant.
Entrée : suite de n nombres entiers $(a_1, a_2 \ldots a_n)$
Sortie : une permutation de la suite donnée en entrée $(a_1', a_2' \ldots a_n')$ telle que $a_1' \leq a_2' \leq \cdots \leq a_n'$. À partir de la suite (6,9,2,4), un algorithme de tri fournira le résultat (2,4,6,9).

Définition 1.2. Une valeur particulière de l'ensemble des valeurs données en entrée est appelée **instance** du problème.

Exemple 1.1 (suite)
La valeur (6,9,2,4) est une instance du problème.

Définition 1.3. Un algorithme est **correct**, si pour toute instance du problème il se termine **et** produit une sortie correcte.

Les algorithmes peuvent être spécifiés en langage humain ou tout langage informatique. Dans ce qui suit, nous utiliserons un langage proche du langage naturel. Nous donnerons une implémentation en Python (voir cours MISMI MIS 102).

Définition 1.4. Une **heuristique** est une procédure de calcul correcte pour certaines instances du problème (c'est-à-dire se termine **ou** produit une sortie correcte).

Ce cours n'aborde pas les heuristiques.

Pour qu'un algorithme puisse être décrit et s'effectue, les données d'entrées doivent être organisées.

Définition 1.5. Une **structure de données** est un moyen de stocker et d'organiser des données pour faciliter leur stockage, leur utilisation et leur modification.

De nombreux problèmes nécessitent des algorithmes :

- bio-informatique ;
- moteur de recherche sur Internet ;
- commerce électronique ;
- affectation de tâches.

Définition 1.6. L'**efficacité** d'un algorithme est mesurée par son coût (**complexité**) en temps et en mémoire.

Un problème **NP-complet** est un problème pour lequel on ne connaît pas d'algorithme correct efficace, c'est-à-dire réalisable en temps et en mémoire. Le problème le plus célèbre est le **problème du voyageur de commerce**.
L'ensemble des problèmes NP-complets ont les propriétés suivantes :

- si on trouve un algorithme efficace pour un problème NP complet alors il existe des algorithmes efficaces pour tous ;
- personne n'a jamais trouvé un algorithme efficace pour un problème NP-complet ;
- personne n'a jamais prouvé qu'il ne peut pas exister d'algorithme efficace pour un problème NP-complet particulier.

I-B. Notion de complexité▲

L'efficacité d'un algorithme est fondamentale pour résoudre effectivement des problèmes.

Exemple1.2.
Supposons que l'on dispose de deux ordinateurs. L'ordinateur A est capable d'effectuer 10^9 instructions par seconde. L'ordinateur B est capable d'effectuer 10^7 instructions par seconde. Considérons un même problème (de tri par exemple) dont la taille des données d'entrées est n. Pour l'ordinateur A, on utilise un algorithme qui réalise $2n^2$ instructions. Pour l'ordinateur B, on utilise un algorithme qui réalise $50n\log(n)$instructions. Pour traiter une entrée de taille 10^6 : l'ordinateur A prendra 2000 s et l'ordinateur B prendra 100 s. Ainsi, même si la machine B est médiocre, elle résoudra le problème 20 fois plus vite que l'ordinateur A.

Définition 1.1. La **complexité** d'un algorithme est :

- en temps, le nombre d'opérations élémentaires effectuées pour traiter une donnée de taille n ;
- en mémoire, l'espace mémoire nécessaire pour traiter une donnée de taille n.

Dans ce cours, nous considérerons que la complexité des instructions élémentaires les plus courantes sur un ordinateur ont un temps d'exécution que l'on considérera dans ce cours comme constant égal à 1. Les instructions élémentaires sont : addition, multiplication, modulo et partie entière, affectation, instruction de contrôle. Ce qui intéresse fondamentalement l'algorithmique, c'est l'ordre de grandeur (au voisinage de l'infini) de la fonction qui exprime le nombre d'instructions. Les courbes de références sont iciLes courbes « étalon ».

I-C. Langage de description d'algorithmes▲

Il est nécessaire de disposer d'un langage qui soit non lié à l'implémentation. Ceci permet une description plus précise des structures de données ainsi qu'une rédaction de l'algorithme plus souple et plus « lisible ». Le langage EXALGODescription d'algorithme - Langage EXALGO est un exemple de ce qui peut être utilisé et qui sera utilisé dans ce cours. Il est composé de chaînes de caractères alphanumériques, de signes opératoires (**+**, **-**, *****, **/**, **<**, **<=**, **>=**, **>**, **<>**, **==**, **=**, **ou, non, et**), de mot-clés réservés, et de signes de ponctuation : **''**=, **;**, **(**, **)**, **début**, **fin**, **//**. Les balises **début** et **fin** peuvent être remplacées par **{** et **}**.

Python n'utilise pas de marqueurs de fin. Le caractère « : » est le marqueur de début et quand l'indentation cesse Python considère que c'est un marqueur de fin.

II. Codage et structures de contrôle▲

II-A. Définitions▲

Définition 2.1. Un **type abstrait** est un triplet composé :

- d'un nom ;
- d'un ensemble de valeurs ;

- d'un ensemble d'opérations définies sur ces valeurs.

Les types abstraits de base de l'algorithmique sont :

entier, **caractère**, **booléen**, **réel**

que l'on écrit respectivement en EXALGO

entier, **car**, **booléen**, **réel**

Définition 2.2. Une **variable** est un triplet composé :

- d'un type (déjà défini) ;
- d'un nom (a priori toute chaîne alphanumérique) ;
- d'une valeur.

On écrit en EXALGO

Sélectionnez

```
var NomDeVariable : Type;
```

`Type` est à prendre pour l'instant dans l'ensemble {**entier**, **car**, **booléen**, **réel**}.

Définition 2.3. Les **Expressions** sont constituées à l'aide de variables déjà déclarées, de valeurs, de parenthèses et d'opérateurs du (des) type(s) de variables concernées.

Définition 2.4. L'**affectation** est l'instruction qui permet de stocker une valeur dans une variable.

On écrit :

Sélectionnez

```
NomDeVariable = ExressionDuTypeDeLaVariable;
```

Toute variable doit être déclarée et recevoir une valeur initiale.

II-B. Types de base▲

II-B-1. Booléens▲

Une variable de type **booléen** prend comme valeur **VRAI** ou **FAUX**. Les opérations usuelles sont ET, OU et NON qui sont données dans les tables qui suivent :

OU	FAUX	VRAI
FAUX	FAUX	VRAI
VRAI	VRAI	VRAI

ET	FAUX	VRAI
FAUX	FAUX	FAUX
VRAI	FAUX	VRAI

NON	
FAUX	VRAI
VRAI	FAUX

II-B-2. Entiers▲

Une variable de type **entier** peut prendre comme valeur l'ensemble des nombres entiers signés. Les opérations associées sont les opérations usuelles +,-,*,/.

II-B-3. Réels▲

Une variable de type **réel** peut prendre comme valeur l'ensemble des nombres réels. Les opérations associées sont les opérations usuelles +,-,*,/.

II-B-4. Caractères▲

Une variable de type **car** peut prendre comme valeur l'ensemble des caractères imprimables. On notera les valeurs entre guillemets. On considère souvent que les caractères sont ordonnés dans l'ordre alphabétique.

II-B-5. Attention▲

Les valeurs :

- **"1"** qui est un caractère ;
- **1** qui est un entier ;

- **1.** qui est un réel ;

sont différentes et ne seront pas codées de la même manière dans la mémoire de la machine.

II-B-6. Comparaison▲

Les opérateurs **<**, **≤**, **==**, **!=**, **>**, **≥** permettent de comparer les valeurs de type entier, réel et caractère. Le résultat de cette comparaison est une valeur booléenne.

II-C. Structures de contrôle▲

Il y a trois structures principales de contrôle qui permettent de construire des algorithmes.

- Bloc d'instructions :

Sélectionnez

```
début
  instruction1
  instruction2
  ..........
fin
```

- Alternative :
  - Alternative simple :

    Traduction Python
    Sélectionnez

    Sélectionnez

    ```
    si ExpressionBooléenne alors
      BlocInstructions1
    sinon
      BlocInstructions2
    finsi;
    ```

- Alternative multiple (traduction Python) :

  Traduction Python
  Sélectionnez

  Sélectionnez

  ```
  selon que
    cas  cas1  : BlocInstructions1
    cas  cas2  : BlocInstructions2
    ..........
    autrement  :  BlocInstruction
  finselonque
  ```

- Répétition
  L'instruction **exit** permet d'arrêter la répétition.
  - Le bloc d'instructions peut ne pas être exécuté :

    Traduction Python
    Sélectionnez

    Sélectionnez

    ```
    tant que ExpressionBooléenne faire
      BlocInstructions
    fintantque;
    ```

- Le bloc d'instructions peut ne pas être exécuté et il y a une variable indicatrice :

  Traduction Python
  Sélectionnez

  Sélectionnez

  ```
  pour VariableIndicatrice
    allant de ValeurInitiale à ValeurFinale
    par pas de ValeurPas faire
      BlocInstructions
  finpour;
  ```

- Le bloc d'instructions est exécuté au moins une fois (ne se traduit pas directement en Python) :

Sélectionnez

```
répéter
  BlocInstructions
jusqu'à ExpressionBooléenne finrépéter;
```

II-D. Fonctions▲

Une fonction est une section d'algorithme qui a un objectif bien défini et un nom. En général, elle communique avec l'extérieur par le biais de paramètres typés. Elle possède des variables locales qui ne sont pas visibles à l'extérieur de la fonction. Ces variables peuvent être des fonctions. Une fonction retourne une valeur par l'instruction simple retourne(Expression). L'expression peut être :

- vide, tout s'est bien passé, mais il n'y a pas de résultat à retourner : retourne() ;
- sans résultat, il est impossible de retourner un résultat suite à un cas de figure de l'instance : retourne(`NUL`).

II-D-1. Syntaxe▲

- Écriture de la fonction :

  Sélectionnez

  ```
  fonction NomDeFonction (ListeParamètres) :TypeRésultat;
    // déclarations des variables ou fonctions locales autres que les paramètres
    début
      // partie instruction qui contient l'appel à retourne
    fin
  finFonction
  ```

- Liste des paramètres
  Les paramètres sont passés :

  - par référence `ref`, on écrit :

    Sélectionnez

    ```
    ref ListeVariable NomDeType
    ```

    la fonction travaille directement dans la variable passée en paramètre ;

- par valeur `val`, on écrit :

  Sélectionnez

  ```
  val ListeVariable:NomDeType
  ```

  la fonction travaille sur une copie de la variable passée en paramètre.

- Le type du résultat est **vide** si la fonction ne renvoie pas de résultat.

II-D-2. Utilisation▲

Une fonction s'utilise en écrivant :

Sélectionnez

```
NomDeFonction(ListeInstanceParamètres)
```

- dans le calcul d'une expression, si la fonction retourne une valeur ;
- comme une instruction simple, si elle ne retourne pas de valeur.

II-D-3. Exemple▲

Sélectionnez

```
fonction exemple(val n :entier;ref m : entier) :vide;
  début
    n = 5;
    m = 7;
  fin
finFonction
```

Supposons que l'on ait la séquence suivante :

Sélectionnez

```
var p,q :entier;
début
  p = 1;
  q = 2;
  exemple(p,q);
fin
```

Après exécution p contiendra 1 et q contiendra 7 (Animation ici).

III. Description d'algorithme - Langage EXALGO▲

EXALGO permet de fixer les quelques règles élémentaires permettant d'écrire des algorithmes en s'affranchissant l'implémentation.

III-A. Généralités▲

Le langage EXALGO est composé de chaînes de caractères alphanumériques, de signes opératoires, de mot-clés réservés, et de signes de ponctuation : **=**, **;**,**(**,**)**, **début**, **fin**, **//**. Les marqueurs de fin, **début** et **fin** peuvent être remplacés par **{** et **}** lorsqu'il y a encombrement.

III-B. Type▲

- Types prédéfinis : **entier**, **car**, **booléen**, **réel** ;

Définition de type
Sélectionnez

```
type NomDeType = TypePrédéfini;
```

Définition d'un tableau d'entiers
Sélectionnez

```
typeNomDeType = tableau 1..limite deTypePrédéfini;
```

III-C. Variables▲

Sélectionnez

```
var NomDeVariable: TypePrédéfini;
```

III-D. Expressions▲

Constituées à l'aide de variables déjà déclarées, de parenthèses et d'opérateurs du (des) type(s) des variables concernées.

III-E. Instructions simples▲

- affectation :

  Sélectionnez

  ```
  NomDeVariable = ExressionDuTypeDeLavariable;
  ```

- sortie de calcul : exit, retourne().

III-F. Structure de contrôle▲

- Bloc d'instruction :

  Bloc d'instruction
  Sélectionnez

  ```
  Instruction1
  instruction2
  .............
  ```

- Alternative :

  Sélectionnez

  ```
  si ExpressionBooléenne alors
       BlocInstruction1
  sinon
       BlocInstruction2
  finsi;
  ```

- Alternative multiple :

  Sélectionnez

  ```
  selon que
       cas cas1 : BlocInstruction1
       cas cas2 : BlocInstruction2
       .............
       autrement : BlocInstruction
  finselonque
  ```

- Répétition : **exit** permet d'arrêter la répétition
  - le bloc d'instruction peut ne pas être exécuté

    Sélectionnez

    ```
    tant que ExpressionBooléenne faire
         BlocInstruction
    ```

```
fintantque;
```

- le bloc d'instruction peut ne pas être exécuté et il y a une variable indicatrice

  Sélectionnez

  ```
  pour VariableIndicatrice
       allant de ValeurInitiale à ValeurFinale
       par pas de ValeurPas faire
             BlocInstruction
  finpour;
  ```

- le bloc d'instruction est exécuté au moins une fois

Sélectionnez

```
répéter
     BlocInstruction
jusqu'à ExpressionBooléenne finrépéter;
```

III-G. Fonctions▲

Une fonction retourne une valeur par l'instruction simple (retourne(Expression)). Une fonction s'utilise dans le calcul d'une expression ou comme instruction simple.

- **Écriture de la fonction :**

  Sélectionnez

  ```
  fonction NomDeFonction (ListeParamètres):TypeRésultat;
    // déclarations des variables locales autres que les paramètres
    début
      // partie instruction qui contient l'appel à retourne()
    fin
  finFonction
  ```

- **Liste des paramètres .**
  Les paramètres sont passés :
  - par référence `ref`, on écrit :

    Sélectionnez

    ```
    ref ListeVariable:NomDeType
    ```

- par valeur `val`, on écrit :

  Sélectionnez

  ```
  val ListeVariable:NomDeType
  ```

- Le type du résultat est `vide` si la fonction ne renvoie pas de résultat.

III-H. Types▲

III-H-1. Type structuré▲

Un type structuré est constitué à partir de types de base ou d'autres types déclarés.

Sélectionnez

```
type NomDeType: structure
                  champ1:NomDeType1
                  champ2:NomDeType2
                  …
                finstructure
```

Après la déclaration :

Sélectionnez

```
var E:NomDeTypeEnregistrement
```

on accède aux différents champs par le nom de la variable suivi d'un **point** suivi du nom de champ (E.champ1).

III-H-2. Type pointeur▲

Si O est un objet de type T, on accède à l'objet par O**^**. Si on déclare :

Sélectionnez

```
var P:^NomDeType
```

alors on peut obtenir un objet accessible par allouer(P). Lorsqu'on n'utilise plus l'objet, il faut libérer l'espace qu'il utilise par desallouer(P).

IV. Structures de données▲

IV-A. Définition▲

Définition 3.1. Une **séquence** sur un ensemble E est une suite d'éléments (e1,e2,…en) d'éléments de E.

Une séquence peut contenir des éléments identiques de l'ensemble E.

Exemple 3.1 (3,5,8,2,12,6) : est une séquence d'éléments de N, ensemble des entiers naturels. ("a","z","T","A","a") est une séquence sur l'ensemble des caractères imprimables (char).

Il existe plusieurs variantes de séquences suivant les opérations de manipulation autorisées : accès par l'indice de l'élément ou non, accès à la fin de la séquence ou non…
On utilisera en général des noms particuliers dépendant des caractéristiques de la séquence.

Exemple 3.2 : Un **vecteur** peut être défini par une séquence dans laquelle l'accès aux éléments se fait par son indice et la taille de la séquence dépend de l'espace dans lequel on se trouve. On dit aussi qu'on a un accès direct à l'élément. Dans la plupart des langages de programmation, le vecteur existe sous le nom d'**array**.

Exemple 3.3 : Soit la procédure calculant la factorielle :

Sélectionnez

```
fonction fac(val n :entier) :entier;
  début
    si n <= 1 alors
       retourner(1)
    sinon
      retourner(n * fac(n-1))
    finsi
  fin
finfonction
```

Programme Python
Sélectionnez

La séquence des valeurs de n au cours des appels récursifs doit être mémorisée. Supposons l'appel fac(4) alors :

- il y aura appel de fac(3), la mémorisation de n se fera par la séquence L=(4) ;
- il y aura appel de fac(2), la mémorisation de n se fera par la séquence L=(3,4) ;
- il y aura appel de fac(1), la mémorisation de n se fera par la séquence L=(2,3,4) ;
- après exécution de fac(1), la valeur est supprimée en tête de séquence L=(3,4) ;
- après exécution de fac(2), n prend pour valeur la tête de la séquence et la valeur est supprimée en tête de séquence L=(4) ;
- après exécution de fac(3), n prend pour valeur la tête de la séquence et la valeur est supprimée en tête de séquence L=().

IV-B. Structure▲

Soit $F_1, F_2, \ldots, F_p$ des ensembles.

Définition 3.2. Une **structure** sur $F_1 \times F_2 \times \cdots \times F_p$ est une séquence $(f_1, f_2, \ldots, f_k)$ telle que

$\forall i \in [1..k], f_i \in F_i$.

Les structures sont des cas particuliers de séquences. En algorithmique, chaque ensemble F_i peut être un type de base ou une structure. Ce mécanisme permet de définir de nouveaux types plus complexes que les types de base. En EXALGO, on écrit :

Sélectionnez

```
nom_du_type = structure
                nom_champs_1 :type1;
                nom_champs_2 :type2;
                …
                nom_champs_k :typek;
              finstructure
```

Cela signifie que lorsqu'une variable est déclarée de ce type, elle référence k variables en même temps. Soit V une variable dont le type est une structure, on désigne un des champs par **V.** suivi du nom du champ.

Exemple 3.4 : Une date de naissance est un exemple de structure. On peut écrire :

Sélectionnez

```
dateDeNaissance = structure
                    jourDeNaissance: entier;
                    moisDeNaissance: entier;
```

```
        annéeDeNaissance: entier;
      finstructure
```

On peut définir une structure composée du sexe et de la date de naissance :

Sélectionnez

```
individu = structure
             sexe :booléen
             date :dateDeNaissance;
           finstructure.
```

Soit la déclaration :

Sélectionnez

```
var I :individu
```

alors I.sexe sera un booléen et I.date.jourDeNaissance sera un entier. Ainsi les instructions suivantes ont un sens :

Sélectionnez

```
I.date.jour = 12;
I.sexe = faux;
```

IV-C. Table d'association à clé unique▲

Définition 3.3 : Soit F un ensemble. Une **table d'association à clé unique** est une séquence d'éléments de $N \times F$ (N est l'ensemble des entiers naturels), $((c_1, f_1),\ (c_2, f_2),\ \ldots,\ (c_k, f_k))$ telle que :

$$\forall i, j \in [1..k], i \neq j, c_i \neq c_j$$

Les tables d'association sont un cas particulier de séquences d'éléments structurés. La structure se décrit en EXALGO :

Sélectionnez

```
association = structure
                cle :entier;
                valeur :type_prédéfini;
              finstructure
```

Exemple 3.5 : Lors de l'activation du compte électronique, l'étudiant de l'Université Bordeaux 1 fournit un numéro INE qui sera associé à un mot de passe. On a donc quelque part dans le système de gestion des comptes une table d'association à index unique dont l'élément de séquence est :

Sélectionnez

```
Etudiant = structure
             INE :entier;
             motDePasse :typeMotDePasse;
           finstructure
```

V. Complexité▲

V-A. Définitions▲

Définition 4.1. (Notation de Landau). On dit que $f = O(g)$ s'il existe deux nombres réels $k, a > 0$ tels que $\forall x > a, |f(x)| \leq k|g(x)|$.

Exemple 4.1. Si le nombre d'instructions est égal à $f(n) = an^2 + bn + c$ avec a,b,c des constantes réelles, alors $f(n) = O(n^2)$.

Les figuresLes courbes « étalon » permettent de comparer les fonctions usuelles utilisées pour décrire la complexité d'un algorithme en fonction de la taille n des données d'entrées. Parmi les fonctions usuelles, le log à base 2 de $n\log_2(n)$ joue un rôle important. Pour un algorithme A, notons $C_A(D)$, le coût de l'algorithme A pour une instance D.

Définition 4.2. On définit les trois complexités suivantes :

- complexité dans le pire des cas :
  $C_A^{>}(n) = \max \{C_A(d), d\ donnée\ de\ taille\ n\}$
- complexité dans le meilleur des cas :
  $C_A^{<}(n = \min \{C_A(d), d\ donnée\ de\ taille\ n\}$
- complexité en moyenne :
  $C_A(n) = \sum_{d\ instance\ de\ A} Pr(d) C_A(d)$
  où $Pr(d)$ est la probabilité d'avoir en entrée une instance d parmi toutes les données de taille n.

Soit D_n l'ensemble des instances de taille n. Si toutes les instances sont équiprobables, on a :

$$C_A(n) = \frac{1}{|D_n|} \sum_{d \text{ instance de } A} C_A(d)$$

Parfois, il est nécessaire d'étudier la complexité en mémoire lorsque l'algorithme requiert de la mémoire supplémentaire (donnée auxiliaire de même taille que l'instance en entrée par exemple).

V-B. Structures de contrôle▲

Les algorithmes font intervenir les opérations élémentaires suivantes :

- opérations élémentaires +, -, *, / ;
- test d'expression booléenne ;
- appel de fonctions.

Les complexités en temps des structures sont données ci-dessous :

- bloc d'instructions : somme des coûts des instructions ;
- Alternative :
  - Alternative simple : un test,
  - Alternative multiple :
    - complexité minimum : un test ,
    - complexité maximum : nombre de cas possible-1 ;
- Répétition
  Soit $B_T(n)$ (resp. $B_O(n)$) la complexité en nombre de tests (resp. d'opérations élémentaires) de la suite d'instructions à itérer, et k le nombre de fois où l'itération s'effectue alors la complexité sera de :
  - $kB_T(n) + 1$ pour le nombre de tests ;
  - $kB_O(n)$ pour le nombre d'opérations du « tant que » et du « répéter » ;
  - $k(B_O(n) + 1)$ pour le nombre d'opérations du « pour ».

V-C. Exemples▲

V-C-1. Somme des N premiers entiers▲

Sélectionnez

```
fonction suite(val n :entier) :entier;
  var i,s :entier;
  début
    s = 0;
    pour i allant de 1 à n faire
      s = s + i;
    finpour;
    retourner(s)
  fin
finfonction;
```

Source Python
Sélectionnez

On a :

$$C^{>}_{suite}(n) = C^{<}_{suite}(n) = C_{suite}(n) = O(n)$$

V-C-2. Apparition d'une pile dans une suite de n lancers d'une pièce▲

Entrée : un entier n
Sortie : « vrai » si on rencontre une pile, « faux » sinon.
La fonction suivante retourne « vrai » lorsque l'un des lancers est égal à 6 et « faux » sinon.

Sélectionnez

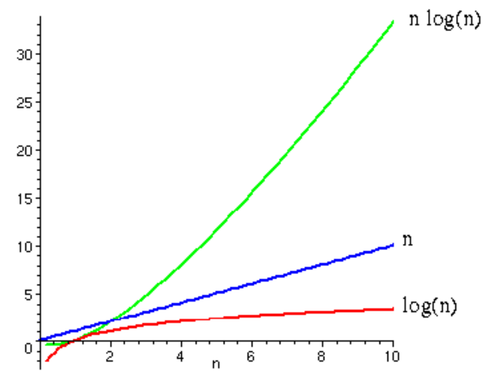
```
fonction jeuDePile(val n :integer) :booléen;
  var i : entier;
  début
    pour i allant de 1 à n faire
      f = résultat_lancer_pièce()
      si (f==pile) alors
        retourner(vrai)
      finsi
    finpour
    retourner(faux)
  fin
finFonction
```

Source Python
Sélectionnez

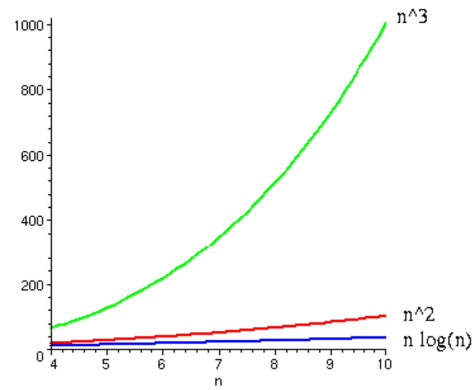
- $C^{>}_{suite}(n) = O(n)$ (On ne tire jamais de pile)
- $C^{<}_{pile}(n) = O(1)$ (On tire une pile le premier coup)
- Les faces du dé apparaissent de manière équiprobable et les tirages sont indépendants. On peut montrer que le coût moyen de l'algorithme est : $C_{pile}(n) = O(1)$.

V-D. Les courbes « étalon »▲

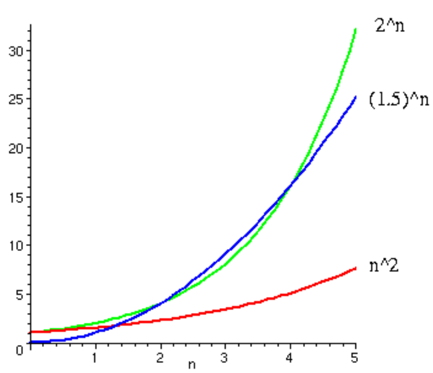
V-D-1. n, log(n), nlog(n)▲



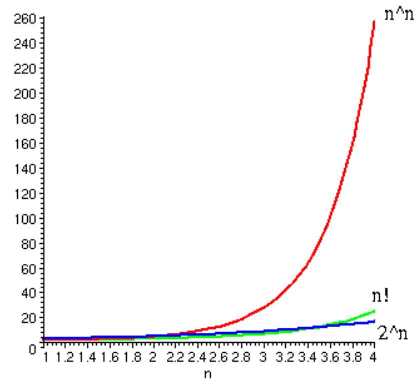
V-D-2. nlog(n), n2, n3▲



V-D-3. n2, 1.5n ,2n▲

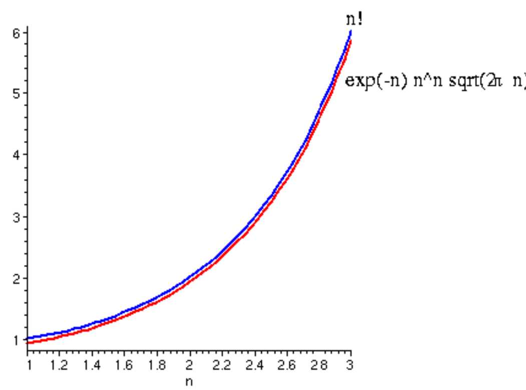


V-D-4. 2n, nn n ! ▲



V-E. Formule de Stirling▲

$$n! = e^{-n}n^n\sqrt{2\pi n}$$



VI. Tableaux▲

VI-A. Définition▲

Définition 5.1. Un **tableau** est une table d'association à clé uniqueTable d'association à clé unique telle que :

- le nombre d'éléments de la table (**dimension** ou **taille**) est constant ;
- l'accès aux éléments s'effectue directement par la clé ;
- les valeurs minimum et maximum des clés sont des constantes.

On écrit en EXALGO :

Sélectionnez

```
nom_tableau = tableau[min_indice..max_indice] de type_predefini;
```

ce qui signifie que :

- les éléments ont pour type le type_prédéfini ;
- les indices des éléments vont de min_indice à max_indice, avec min_indice < max_indice.

La taille du tableau est donc max_indice - min_indice + 1. Pour accéder à un élément d'un tableau T d'indice I, on écrit T[I]. La complexité de l'accès à un élément du tableau est $O(1)$.

Soit min_indice< i<j <max_indice, on notera T[i..j] la séquence des éléments de T (T[i],T[i+1], …,T[j]).

Beaucoup d'algorithmes peuvent être décrits sans préciser un type particulier. Dans ce cas, on écrira à la place de type_prédéfini le mot **élément** et on précisera les valeurs possibles pour élément.

Exemple 5.1. Soit deux tableaux :

- TC = `tableau[1..10] de car;` ;
- TE = `tableau[1..10] d'entiers;`.

L'algorithme qui permet de trier TC et TE est le même. Seul diffère le type de l'élément manipulé. On écrira dans ce cas un algorithme sur un tableau.

Sélectionnez

```
T = tableau[1..10] d'éléments;
```

et on précisera que l'élément est dans {car,entier}.

VI-B. Primitives▲

Les **paramètres tableaux** doivent, sauf raison majeure, être passés en **paramètre par référence** afin d'éviter la recopie.

VI-B-1. Initialisation d'un tableau▲

Sélectionnez

```
fonction init(ref T :tableau[min_indice..max_indice] d'éléments;
             val valeurInitiale :élément) :vide;
  var i :entier;
  début
    pour i allant de min_indice à max_indice faire
      T[i] = valeurInitiale
    finpour
  fin
finfonction
```

Complexité : $O(n)$

VI-B-2. Taille d'un tableau▲

Sélectionnez

```
fonction taille(ref T :tableau[min_indice..max_indice] d'éléments) :entier;
  début
    retourner(max_indice - min_indice + 1)
  fin
finfonction
```

Complexité : $O(1)$.

VI-B-3. Échange d'éléments▲

Sélectionnez

```
fonction echange(ref T :tableau[min_indice..max_indice] d'éléments;
             val indice1,indice2 : entier) :vide;
  var e :élément;
  début
    e = T[indice1];
    T[indice1] = T[indice2];
    T[indice2] = e;
  fin
finfonction
```

Complexité : $O(1)$.

VI-B-4. Copie de tableau▲

Sélectionnez

```
fonction copie(ref T1,T2 :tableau[min_indice..max_indice] d'élément;
             val indiceT1_1,indiceT1_2,indiceT2 : entier) :booléen;
   var i :entier;
  début
    si indiceT2+indiceT1_2-indiceT1_1>max_indice alors
      retourner(faux)
    sinon
      pour i allant de indiceT1_1 à indiceT1_2 faire
        T2[indiceT2] = T1[i];
        indiceT2 = indiceT2 + 1;
      finpour
      retourner(vrai)
  fin
finfonction
```

Complexité :

- minimum : $O(1)$
- maximum : $O(n)$

Source Python
Sélectionnez

VI-C. Quelques exemples d'algorithmes▲

VI-C-1. Somme des éléments d'un tableau d'entiers▲

Sélectionnez

```
fonction somme(ref T :tableau[min_indice..max_indice] d'entiers) :entier;
  var s,i :entier;
  début
    s = 0;
    pour i allant de min_indice à max_indice faire
      s = s + T[i]
    finpour
    retourner(s)
  fin
finfonction
```

Complexité : $O(n)$

VI-C-2. Recherche d'un élément▲

Propriété 5.2. Soit i,j deux entiers, i<=j. Soit T un tableau d'éléments d'indice variant entre i et j. Pour tout élément e, appartenant au tableau T, on a :

T[i] = e ou e est dans T[i+1..j]

Sélectionnez

```
fonction cherche(ref T :tableau[min_indice..max_indice] d'éléments;val e :élément) :entier;
  var i :entier;
  début
    pour i allant de min_indice à max_indice faire
      si T[i]==e alors
        retourner(i)
      finsi
    finpour
    retourner()
  fin
finfonction
```

Complexité :

- minimum : $O(1)$
- maximum : $O(n)$

Source Python
Sélectionnez

VI-C-3. Recherche de l'indice du premier élément minimum▲

On suppose que le tableau contient des éléments comparables (l'ensemble des éléments est muni d'une relation d'ordre). Choisissons ici, pour simplifier les notations, des entiers.

Propriété 5.3. Soit i,j deux entiers, i<=j. Soit T un tableau d'entiers d'indice variant entre i et j. Soit m l'élément minimum du tableau, on a :

T[i] = m ou m est dans T[i+1..j]

Sélectionnez

```
fonction minimum(ref T :tableau[min_indice..max_indice] d'entier) :entier;
  var i,sauv :entier;
  début
  sauv = min_indice;
  pour i allant de min_indice+1 à max_indice faire
    si T[i]<T[sauv] alors
      sauv = i
    finsi
  finpour
  retourner(sauv)
finfonction
```

Complexité : $O(n)$

VI-D. Matrices▲

VI-D-1. Déclaration▲

Une matrice M de dimension $n \times m$ est un tableau de dimension n dont chaque élément est un tableau de dimension m. On peut donc déclarer la matrice sous la forme suivante :

Sélectionnez

```
var M :tableau[1..n] de tableau [1..m] d'éléments;
```

VI-D-2. Initialisation▲

Sélectionnez

```
fonction initMatrice(ref M :tableau[1..n] de tableau [1..m]  d'éléments;
              val valeurInitiale :élément) :vide;
  var i,j :entier;
  début  pour i allant de 1 à n faire
    pour j allant de 1 à m faire
      M[i][j] = valeurInitiale
    finpour
  finpour
  retourner()
finfonction
```

Complexité : $O(nm)$

VI-D-3. Somme de deux matrices réelles▲

Sélectionnez

```
fonction sommeMatrice(ref M1,M2 :tableau[1..n] de tableau [1..m] de réels) :
                      tableau[1..n] de tableau [1..m] de réels;
  var i,j :entier;
  var M :tableau[1..n] de tableau [1..m] de réels;
  début
  pour i allant de 1 à n faire
    pour j allant de 1 à m faire
      M[i][j] = M1[i][j] + M2[i][j];
    finpour
  finpour
  retourner(M)
finfonction
```

Complexité : $O(nm)$

VII. Tri non récursif▲

On considérera dans tout ce chapitre que l'on manipule des entiers. L'objet du tri est d'ordonner une séquence de N entiers. On considérera que ces entiers sont rangés dans un tableau.

Sélectionnez

```
var T :tableau[1..N] d'entiers;
```

De plus, on considéra que l'ordre est croissant.

VII-A. Tri sélection▲

Ce tri est basé sur l'algorithme de recherche du minimumRecherche de l'indice du premier élément minimum On adapte cet algorithme pour pouvoir effectuer la recherche dans un sous-tableau. On a le déroulement ici.

Sélectionnez

```
fonction minimumSoustableau(ref T :tableau[1..N] d'entiers, val Imin,Imax :entier) :entier;
   var sauv :entier;
  début
    sauv = Imin;
    pour i allant de Imin+1 à Imax faire
      si T[i]<T[sauv] alors
        sauv = i;
      finsi
    finpour
    retourner(sauv);
  fin
finfonction

fonction triSelection(ref T :tableau[1..N] d'entiers) :vide;
  var i,j,indice_cle :entier;
  début
    pour i allant de 1 à N-1 faire
      indice_cle = minimumSoustableau(T,i,N);
      echange(T[i],T[indice_cle]);
    finpour
  fin
finfonction
```

Propriété 6.1. La complexité de l'algorithme triSelection sur une instance de taille N est $O(n^2)$

VII-B. Tri insertion et tri à bulle▲

Propriété 6.2. Soit T un tableau d'entiers trié d'indice variant entre i et j. Soit e un entier quelconque, alors on a l'une des propriétés suivantes :

- e ≤ T[i] ;
- il existe un unique entier k dans [i..j-1] tel que T[k] < e ≤ T[k+1] ;
- e > T[j].

On déduit de cette propriété deux algorithmes permettant de trier un tableau.

VII-B-1. Tri insertion▲

Sélectionnez

```
fonction triInsertion(ref T :tableau[1..N] d'entiers) :vide;
  var i,j,cle :entier;
  début
    pour i allant de 2 à N faire
      cle = T[i];
      j = i-1;
      tant que j>0 et T[j]>cle faire
        T[j+1] = T[j];
        j = j-1;
      fintantque
      T[j+1] = cle;
    finpour
  fin
finfonction
```

On a le déroulement ici.

Propriété 6.3. La complexité de l'algorithme triInsertion sur une instance de taille N est :

- au minimum en $O(N)$;
- au maximum et en moyenne en $O(N^2)$.

Idée de la démonstration
La boucle « pour » s'effectue systématiquement et demandera $O(N)$ opérations.
La boucle « tant que » effectue au minimum $O(1)$ opération (cas où les nombres sont déjà triés) et au maximum $O(N)$.
La boucle « tant que » effectue en moyenne $O(N/2)$ opérations.

VII-B-2. Tri à bulle▲

Sélectionnez

```
fonction triBulle(ref T :tableau[1..N] d'entiers) :vide;
  var i,j,cle :entier;
  début
    pour i allant de 1 à N-1 faire
      pour j allant de N à i+1 par pas de -1 faire
        si T[j] < T[j-1] alors
          echange(T, j, j-1);
        finsi
      finpour
    finpour
  fin
finfonction
```

On a le déroulement ici.

Propriété 6.4. La complexité de l'algorithme triBulle sur une instance de taille N est $O(N^2)$.

VII-C. Fusion de tableaux triés▲

Lorsque deux tableaux T1 et T2 sont triés, il est aisé de construire un nouveau tableau contenant la séquence triée regroupant les séquences correspondantes à T1 et T2.

Sélectionnez

```
PREMIERE VERSION
fonction fusion(ref T1 :tableau[1..N1] d'entier;
                                ref T2 :tableau[1..N2] d'entier
                                ) :tableau[1..N1+N2] d'entier;
  var I1,I2,i :entier;
  var T :tableau[1..N1+N2] d'entier;
  début
    I1 = 1;
    I2 = 1;
    pour i allant de 1 à N1+N2 faire
      si T1[I1]≤T2[I2] alors
        T[i] = T1[I1];
        I1 = I1+1;
      sinon
        T[i] = T2[I2];
        I2 = I2+1;
      finsi
    finpour
    retourner(T)
  fin
finfonction
```

Complexité : $O(n)$

Cette version ne fonctionne pas toujours. Par exemple, si I1 a dépassé N1 et vaut par exemple N1+1, on comparera T1[N1+1] à T2[I2] ce qui n'a pas de sens. Il faut donc utiliser un algorithme exactAlgorithme de fusion de deux tableaux. On a le déroulement ici.

VII-D. Tri par dénombrement▲

Soit une séquence d'éléments de [0..k], il est alors possible de réaliser l'histogramme des valeurs. Par la suite le tri des éléments de la séquence se fait en temps linéaire $O(n)$.

Sélectionnez

```
fonction triHisto(ref T :tableau[1..N] d'entiers) :vide;
  var H :tableau[0..maximum(T)] d'entier;
  var i,j,k,max : entier;
  début
    init(H,0);
    pour i allant de 1 à N faire
      H[T[i]] = H[T[i]] + 1;
    finpour
    i = 1;
    max = maximum(T);
    pour j allant de 0 à max faire
        pour k allant de 1 à H[j] faire
          T[i] = j;
          i = i+1;
        finpour
    finpour
  fin
finfonction
```

On a le déroulement ici.

VII-E. Algorithme de fusion de deux tableaux▲

VII-E-1. Aide▲

On considère une nouvelle fonction **copie** qui copie un tableau dans un autre même s'ils n'ont pas la même définition. L'en-tête de la fonction est :

Sélectionnez

```
fonction copie(ref T1:tableau[1..N1] d'élément;
               ref T2:tableau[1..N2] d'élément;
               val indiceT1_1,indiceT1_2,indiceT2: entier):vide;
```

Le schéma de la fonction fusion est alors le suivant :

Sélectionnez

```
fonction fusion(ref T1:tableau[1..N1] d'entier;
                ref T2:tableau[1..N2] d'entier
                ):tableau[1..N1+N2] d'entier;
  var I1,I2,i:entier;
  var T:tableau[1..N1+N2] d'entier;
  début
    Initialiser I1,I2,i
    tant que I1 ≤ N1 et I2 ≤ N2 faire
        // ... On compare tant qu'il reste des éléments dans T1 et T2
    fintantque
    si I1 ≤ N1 alors
        // il n'y a plus d'éléments dans T2 :copier(..........)
    sinon
        // il n'y a plus d'éléments dans T1 :copier(..........)
    finsi
    retourner(T)
  fin
finfonction
```

VII-E-2. Algorithme de fusion▲

Sélectionnez

```
fonction fusion(ref T1:tableau[D1..N1] d'entier;
                ref T2:tableau[D2..N2] d'entier
                ):tableau[1..N1+N2-D1-D2+2] d'entier;
  var I1,I2,i: entier;
  var T:tableau[1..N1+N2-D1-D2+2] d'entier;
  début
    i = 1;
    I1 = D1;
    I2 = D2;
    tant que I1 ≤ N1 et I2 ≤ N2 faire
      si T1[I1] ≤ T2[I2] alors
        T[i] = T1[I1];
        I1 = I1+1;
      sinon
        T[i] = T2[I2];
        I2 = I2+1;
      finsi
      i = i+1
    fintantque
    si I1 ≤ N1 alors
      copier(T1,T,I1,N1,i)
    sinon
      copier(T2,T,I2,N2,i)
    finsi
    retourner(T)
```

```
  fin
finfonction
```

VII-E-3. Algorithme de fusion pour des morceaux de tableaux▲

Sélectionnez

```
fonction fusion(ref T1:tableau[D1..N1] d'entier;
                ref T2:tableau[D2..N2] d'entier;
                val DT1,FT1,DT2,FT2:entier):
                tableau[1..FT1+FT2-DT1-DT2+2] d'entier;
  var I1,I2,i:entier;
  var T:tableau[1..FT1+FT2-DT1-DT2+2] d'entier;
  début
    i = 1;
    I1 = DT1;
    I2 = DT2;
    tant que I1 ≤ FT1 et I2 ≤ FT2 faire
      si T1[I1] ≤ T2[I2] alors
        T[i] = T1[I1];
        I1 = I1+1;
      sinon
        T[i] = T2[I2];
        I2 = I2+1;
      finsi
        i = i+1
    fintantque
    si I1 ≤ FT1 alors
      copier(T1,T,I1,FT1,i)
    sinon
      copier(T2,T,I2,FT2,i)
    finsi
    Retourner(T)
  fin
finfonction
```

VIII. Retour sur les fonctions, récursivité▲

VIII-A. Visibilité▲

Comme vu au chapitre Codage et structures de contrôleCodage et structures de contrôle, on peut déclarer dans une fonction des variables et des fonctions locales :

Sélectionnez

```
fonction NomDeFonction (ListeParamètres) :TypeRésultat;
  // déclarations des variables ou fonctions locales
  début
    // partie instruction qui contient l'appel à retourner
  fin
finFonction
```

La multi-imbrication possible des fonctions entraîne l'existence de problèmes de visibilité : entre les variables et entre les fonctions.

VIII-A-1. Visibilité d'une variable▲

- **Règle 1** : une variable **V** (locale ou non) est visible depuis sa déclaration jusqu'au marqueur `finFonction` de la fonction **F** où elle a été déclarée.
- **Règle 2** : si une fonction G est locale à **F** et déclare une variable **V** déjà déclarée dans **F** alors la variable originelle est momentanément cachée.

VIII-A-2. Exemple▲

Soit la fonction **P** suivante :

Sélectionnez

```
fonction P (....) :....;
  var x,y,z  : entier ;

  fonction R() :vide;
    var z,u,v  : entier ;
    début
      z = 0;
      u = 6;
      ...
    fin ;
  finFonction

  fonction Q(ref x :entier ) :....;
    var u,y  : entier ;
    début
      y = 4;
      x = x+y;
      u = 7
    fin ;
  finFonction

  début
    x = 1;
```

```
    y = 2;
    z = 3;
    R() …
    Q(z);
  fin
finFonction
```

- La fonction **P** déclare trois variables locales **x**, **y**, **z** et deux fonctions locales **Q** et **R**.
- La fonction **Q** déclare deux variables locales **u**, **y** et un paramètre **x**.
- La fonction **R** déclare trois variables locales **z**, **u** et **v**.

On a le déroulement ici.

VIII-A-3. Visibilité d'une fonction▲

Une fonction est visible depuis la fin de son entête jusqu'au finFonction de la fonction où elle a été déclarée. Cependant comme pour les variables, elle peut momentanément être cachée par une autre fonction ayant le même entête (surcharge).

VIII-A-3-a. Exemple▲

La fonction **P** suivante est annotée pour préciser la visibilité des fonctions **Q**,**R**,**T**.

Sélectionnez

```
fonction P(....) :....;
  .....
  fonction Q(....) :.....;
    .....
    fonction R(...) :.....;
      ....
      début
        ....// on peut utiliser P,Q,R
      fin
    finFonction ;
    début
      ....// on peut utiliser P,Q,R
    fin
  finFonction
  fonction T(...) :...;
    début
      ....// on peut utiliser P,Q,T mais pas R
    fin
  finFonction ;
  début
    ... //// on peut utiliser P,Q,T mais pas R
  fin
finFonction
```

VIII-B. Récursivité▲

La récursivité consiste à remplacer une boucle par un appel à la fonction elle-même. Considérons la suite factorielle, elle est définie par :

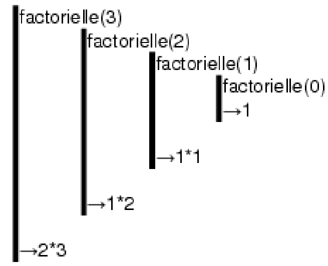
- $0! = 1$;
- $n! = n(n-1)!$.

La fonction peut s'écrire simplement :

Sélectionnez

```
fonction factorielle(val n :entier) :entier;
  début
    si (n == 0)
      retourne(1)
    sinon
      retourne(factorielle(n-1) * n)
    finsi
  fin
finfonction;
```

On a le déroulement ici. On peut décrire sur le papier les changements et les appels sous la forme suivante :



Valeur fonction	Séquence de n
	3
	3,2
	3,2,1
	3,2,1,0
1	
	3,2,1
1	
	3,2
2	
	3
6	

Plusieurs appels à la fonction peuvent être exécutés dans son corps. Soit la suite dite de Fibonacci définie par :

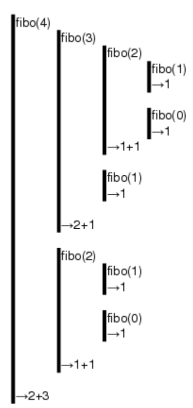
- $u_0 = 1$;
- $u_1 = 1$;
- $u_n = u_{n+1} + u_{n+2}$ pour $n > 2$.

La fonction s'écrit tout aussi simplement :

Sélectionnez

```
fonction fibo(val n :entier) :entier;
  début
    si (n == 0) ou (n == 1) alors
      retourne(1)
    sinon
       retourne(fibo(n-1) + fibo(n-2))
    finsi
  fin
finfonction;
```

On a le déroulement ici. On peut décrire sur le papier les changements et les appels sous la forme suivante :



Valeur fonction	Séquence de n
	4
	4,3
	4,3,2
	4,3,2,1
1	
	4,3,2
	4,3,2,0
1	
2	
	4,3
	4,3,1
1	
	4,3
3	
	4
	4,2
	4,2,1
1	
	4,2
	4,2,0
1	
	4,2
2	
	4
5	

VIII-C. Complexité▲

Examinons la suite définie par :

- $u_1 = 1$;
- $u_n = u_{n-1} + n$ pour $n > 1$.

Une fonction permettant le calcul de son n^e terme est :

Sélectionnez

```
fonction suite(val n :entier) :entier;
  var i,s :entier;
  début
    s = 0;
    pour i allant de 1 à n faire
      s = s+i;
    finpour;
    retourner(s)
  fin
finfonction;
```

L'exemple ci-dessus devient en algorithme récursif :

Sélectionnez

```
fonction suiteR(val n :entier) :entier;
  début
    si n == 1 alors
      retourne(1)
    sinon
      retourne(suiteR(n-1) + n)
    finsi
  fin
finfonction;
```

La complexité en nombre d'opérations de **suite** et **suiteR** est en $O(n)$. On aurait donc tendance à préférer **suiteR** pour sa lisibilité. Cependant, si on examine la complexité en mémoire, **suite** est en $O(1)$ alors que **suiteR** est en $O(n)$.
La programmation non récursive est donc plus efficace. **L'utilisation de la récursivité ne doit pas se faire au détriment de l'efficacité.**

VIII-D. Exemples▲

Chaque fois que l'on désire programmer une fonction récursive, on doit répondre aux questions suivantes :

- comment le problème au rang **n** se déduit-il de la solution à un (des) rang(s) inférieur(s) ?
- quelle est la condition d'arrêt de la récursivité ?

VIII-D-1. Recherche d'un élément dans un tableau d'entiers▲

Sélectionnez

```
fonction cherche(ref T :tableau[min_indice..max_indice] d'éléments;
                 val e : élément) :entier;
  début
    si T[min_indice] == e alors
      retourner(min_indice)
    sinon
      si min_indice == max_indice alors
        retourner(NUL)
      sinon
        retourner(cherche(T[min_indice+1..max_indice], e))
      finsi
    finsi
  fin
finfonction
```

VIII-D-2. Minimum dans un tableau d'entiers▲

Sélectionnez

```
fonction minimumTableau(ref T :tableau[1..N] d'entiers;
                        val Imin :entier) :entier;
  var sauv :entier;
  début
    si Imin == N alors
      retourner(T[N])
    sinon
      sauv = minimumTableau(T, Imin+1];
      si T[Imin] < sauv alors
        retourner(T[Imin])
      sinon
        retourner(sauv)
      finsi
    finsi
  fin
finfonction
```

IX. Diviser pour régner▲

IX-A. Dichotomie▲

La dichotomie fait partie des méthodes dites « diviser pour régner ». Elle consiste pour un objet de taille N à exécuter un algorithme de façon à réduire le problème à un objet de taille N/2. On répète alors l'algorithme de réduction sur ce dernier objet. Ainsi, il suffit de connaître la résolution pour un problème de taille faible (typiquement N=1 ou N=2) pour obtenir la totalité de la résolution.Ce type d'algorithme est souvent implémenté de manière récursive. Lorsque cette technique est utilisable, elle conduit à un algorithme très efficace et très lisible.
Il est parfois nécessaire de traiter les données avant d'appeler la fonction récursive. La fonction récursive est alors une fonction locale à la fonction d'appel.

IX-B. Exemples▲

IX-B-1. Recherche du zéro d'une fonction croissante▲

Soit g une fonction croissante sur un intervalle [a,b] et telle que f(a)≤0 et f(b)≥0. L'algorithme ci-dessous permet de trouver la valeur x de [a,b] telle que f(x)=0 avec une précision e.

Sélectionnez

```
fonction zero(ref g(val n :réel) :fonction;val a,b,e :réel) :réel;
  var M :réel;
  début
    M = g((a+b)/2);
    si |M| < e alors
      retourne((a+b)/2)
    sinon
      si M > 0 alors
        zero(g, a, (a+b)/2, e)
      sinon
        zero(g, (a+b)/2, b, e)
      finsi
    finsi
  fin
finfonction
```

IX-B-2. Trouver un élément dans un tableau ordonné▲

Nous avons déjà traité cet algorithme sous une autre forme au chapitre TableauxQuelques exemples d'algorithmes.

Propriété 8.1. T un tableau d'entiers triés d'indice variant entre d et f. Posons $m = \lfloor (d+f)/2 \rfloor$. Soit e un entier appartenant à la séquence contenue dans T. On a l'une des propriétés suivantes :

- T[m] = e ;
- e est dans la séquence contenue dans T[d..m-1] ;
- e est dans la séquence contenue dans T[m+1..f].

Sélectionnez

```
fonction cherche(ref T :tableau[1..N] d'entiers; val e :entier) :entier;
  var d,f :entier;
  fonction chercheRec(ref T :tableau[1..N] d'entiers; val d,f,e :entier) :entier;    var m;
    début
      si f == d alors
         si T[d] == e alors
           retourner(f)
        sinon
            retourner(NUL)
        finsi
      sinon
        m = partieEntiere((d+f)/2);
        si T[m] < e alors
          retourner(chercheRec(T,m+1,f,e))
        sinon
          retourner(chercheRec(T,d,m,e))
        finsi
      finsi
    fin
  finfonction
  début
    d = 1;
    f = N;
    retourner(chercheRec(T,d,f,e))
  fin
finfonction
```

Propriété 8.2. La complexité de la fonction cherche est $O(\log_2(n))$.

Idée de la preuve : la complexité de la fonction cherche est donnée par la complexité de chercheRec. Soit $f(n)$ le nombre de tests effectués par cette fonction.
On a :

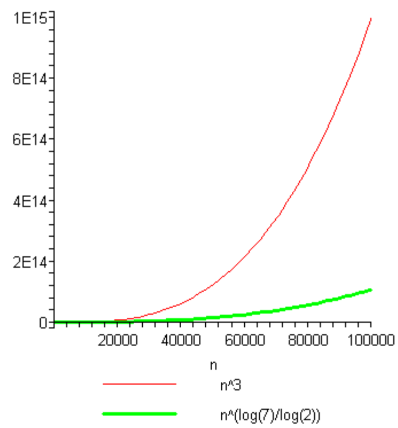
$$\begin{aligned} f(n) &= 2 + f(\lfloor n/2 \rfloor) \\ f(1) &= 2 \end{aligned}$$

Soit p tel que $2^p \leq n \leq 2^{p+1}$. On a donc $p \leq \log_2(n) \leq p+1$. De plus : $f(n) = 2\sum_{i=0}^{p} 1$ et donc $f(n) = 2 \times (p+1)$
.

IX-B-3. Remarque▲

L'algorithme de multiplication de deux matrices de dimension $n \times n$ s'implémente facilement en $O(n^3)$. Strassen a montré qu'en utilisant une méthode « diviser pour régner », la multiplication peut s'effectuer en $O\left(n^{\ln(7)/\ln(2)}\right)$.

La courbe se présente comme suit :



IX-C. Complexité▲

Un algorithme « diviser pour régner » a la structure suivante :

1. Construire une solution élémentaire pour $n \leq n0$;
2. Pour résoudre un problème de taille $n > n0$, l'algorithme consiste à décomposer le problème en sous-problèmes ayant tous la taille n/b (peut-être approximativement) ;
3. À appliquer l'algorithme à tous les sous-problèmes ;
4. À construire une solution du problème en composant les solutions des sous-problèmes.

La complexité en temps de l'algorithme est donc déterminée par une équation de récurrence de la forme :

$$C(n) = aC(n/b) + d(n)$$

qui après résolution permet de montrer que cette méthode conduit à des algorithmes plus efficaces en nombre d'opérations. Cependant, cela ne doit pas occulter l'aspect mémoire. La complexité en mémoire doit rester d'un ordre raisonnable. (cf. récursivitéRécursivité).

X. Tris récursifs▲

On considérera dans tout ce chapitre que l'on manipule des entiers. L'objet du tri est d'ordonner une séquence de N entiers. On considérera que ces entiers sont rangés dans un tableau :

Sélectionnez

```
var T :tableau[1..N] d'entiers;
```

De plus, on considéra que l'ordre est croissant.

X-A. Tri fusion ▲

Cet algorithme consiste à diviser la séquence d'entiers en deux sous-séquences, à les trier de manière récursive, puis à fusionner les deux sous-séquences triées. On utilise la fonction fusionAlgorithme de fusion de deux tableaux vue au chapitre tris non récursifsTri non récursif.

Sélectionnez

```
fonction triFusion(ref T :tableau[1..N] d'entiers) :vide;
  var d,f :entier;
  fonction fusionLocal(ref T :tableau[1..N] d'entier;
                        val d,m,f :entier) :vide;
    var C :tableau[1..f-d+1] d'entier;
    début
       C = fusion(T,T,d,m,m+1,f);
       copie(C,T,d,f,d);
    fin
  finfonction
  fonction triFusionRec(ref T :tableau[1..N] d'entiers; val d,f :entier) :vide;
    début
      si d<f alors
        m = partieEntiere((d+f)/2);
        triFusionRec(T,d,m);
        triFusionRec(T,m+1,f);
        fusionLocal(T,d,m,f);
      finsi
    fin
  finfonction
  début
    trifusionRec(T,1,N)
  fin
finfonction
```

On a le déroulement ici.

Propriété 9.1. La complexité du tri fusion pour une séquence de n éléments est $O(n\log_2(n))$.

idée de la preuve : la complexité de la fonction triFusion est donnée par la complexité de triFusionRec. Soit $f(n)$ le nombre de tests effectués par cette fonction. On a :

$$\begin{array}{ll} f(n) & = 1 + 2f(\lfloor n/2 \rfloor) + 3n \\ f(1) & = 0 \end{array}$$

Soit p tel que $2^p \leq n \leq 2^{p+1}$. On a donc $p \leq \log_2(n) \leq p+1$

On en déduit :

$$f(n) = \sum_{i=0}^{p} 2^i + 3pn$$

et

$$f(n) = 2^{p+1} + 3pn - 1$$

X-B. Tri rapide▲

Cet algorithme consiste à utiliser une valeur x de la séquence pour diviser la séquence d'entiers en deux sous-séquences :

- l'ensemble des valeurs inférieures ou égales à x ;
- l'ensemble des valeurs supérieures à x.

Puis la procédure s'effectue récursivement sur les deux sous-séquences :

Sélectionnez

```
fonction triRapide(ref T :tableau[1..N] d'entiers) :vide;
  fonction diviserSequence(ref T :tableau[1..N] d'entier;
                           val d,f :entier) :entier;
    var x,i :entier;
    début
      x = T[f];
      i = d-1;
      pour j allant de d à f-1 faire
        si T[j] ≤ x alors
          i = i+1;
          echanger(T,i,j);
        finsi
      finpour
      echanger(T,i+1,f);
      retourner(i+1);
    fin
  finfonction
  fonction triRapideRec(ref T :tableau[1..N] d'entiers; val d,f :entier) :vide;
    var p :entier;
    début
      si d<f alors
        p = diviserSequence(T,d,f);
        triRapideRec(T,d,p-1);
        triRapideRec(T,p+1,f);
      finsi
    fin
  finfonction
  début
    triRapideRec(T,1,N)
  fin
finfonction
```

On a le déroulement ici.

Propriété 9.2. La complexité du tri rapide pour une séquence de n éléments est :

- au maximum en $O(n^2)$;
- en moyenne et au minimum en $O(n\log(n))$.

XI. Une implémentation des polynômes▲

XI-A. Énoncé▲

Décrire une structure permettant de gérer des polynômes définis sur les réels. Écrire un ensemble de primitives associées permettant les principales opérations.

XI-B. Structure et primitives ▲

On note n le degré (resp. le degré maximum) du polynôme (resp. des polynômes).

XI-B-1. Structure ▲

Un polynôme peut être défini par son degré et un tableau contenant les coefficients. On définit également une primitive d'initialisation.

Sélectionnez

```
constante Taille = 200;
type polynome = structure
                    coeff:tableau[0..200] de réels;
                    degre: entier;
                finstructure

fonction init(ref p:polynome):vide;
  var i:entier;
  début
    p.degre = 0;
    pour i allant de 0 à Taille faire
      p.coeff[i] = 0;
    finpour
  fin
finfonction
```

XI-B-2. Addition ▲

On notera l'analogie avec l'algorithme de fusion de tableau.

Sélectionnez

```
fonction ajoute(ref p1,p2:polynome):polynome;
  var p:polynome;
  var m,i:entier;
  début
    m = min(p1.degre,p2.degre);
    pour i de 0 à m faire
      p.coeff[i] = p1.coeff[i] + p2.coeff[i]
    finpour;
    si m < p1.degre alors
      pour i de m+1 to p1[degre] do
        p.coeff][i] = p1.coeff][i];
      finpour:
      p.degre = p1.degre;
    sinon
      pour i de m+1 to p2[degre] do
        p.coeff[i] = p2.coeff[i];
      finpour:
      p.degre = p2.degre;
    finsi
    retourner(p);
  fin;
finfonction;
```

Complexité : $O(n)$

XI-B-3. Multiplication▲

Sélectionnez

```
fonction multiplie(ref p1,p2:polynome):polynome;
  var p:polynome;
  var i,j:entier;
  début
    init(p);
    p.degre=p1.degre + p2.degre;
    pour i de 0 à p1.degre;
      pour j de 0 à p2.degre faire
        p.coeff[i+j] = p.coeff[i+j] + p1.coeff[i] * p2.coeff[j];
      finpour
    finpour
  retourner(p);
  fin
finfonction
```

Complexité : $O(n^2)$

XI-B-4. Polynôme opposé▲

Soit :

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

Le polynôme opposé est :

$$Q(x) = \sum_{i=0}^{n} -p_i x^i$$

Sélectionnez

```
fonction moins(ref p:polynome):polynome;
  var i:entier;
  var m:polynome;
  début
    m.degre=p.degre;
    pour i de 0 à p.degre faire
      m.coeff[i] =- p.coeff[i];
    finpour
```

```
    retourner(m)
  fin
finfonction
```

Complexité : $O(n)$

XI-B-5. Multiplication par xn▲

Sélectionnez

```
fonction decale(ref p:polynome;val n:entier):polynome
  var i:entier;
  var d:polynome;
  début
    d.degre = p.degre + n;
    si n>0 then
      pour i de 0 à n-1 FAIRE
        d.coeff[i] = 0;
      finpour
      pour i de 0 à p.degre faire
        d.coeff[i+n] = p.coeff[i];
      finpour
    sinon
      pour i de -n à p.degre faire
        d.coeff[i+n] = p.coeff[i];
      finpour
    finsi
    retourner(d)
  fin
finfonction
```

Complexité : $O(n)$

XI-B-6. Dérivée▲

Sélectionnez

```
fonction deriv(ref p1:polynome):polynome;
  var p:polynome;
  var i:entier;
  début
    si p1.degre == 0 alors
      p.degre = 0;
      p.coeff[0] = 0;
    sinon
      p.degre = p1.degre-1;
      pour i de 1 à p1.degre faire
        p.coeff[i-1] = p1.coeff[i]
      finpour;
    finsi;
    retourner(p)
  fin
finfonction
```

Complexité : $O(n)$

XI-B-7. Valeur en un point▲

Sélectionnez

```
fonction valeur(ref p:polynome;val x: réel):réel;
  var f,s,i:réel;
  début
    s = 0;
    f = 1;
    pour i allant de 0 à p.degre faire
      s = s + f * p.coeff[i];
      f = f * x
    finpour
    retourner(s)
  fin
finfonction
```

Complexité : $O(n)$

XI-B-8. Intégrale définie▲

Sélectionnez

```
fonction integraleDefinie(ref p1:polynome;val x,y: réel):réel;
  var p:polynome;
  var s:réel;
  var i:entier;
  début
    p.degre = p1.degre+1;
    p.coeff[0] = 0;
    pour i allant de 0 à p1.degre faire
      p.coeff[i+1] = p1.coeff[i] / (i+1);
    finpour;
    retourner(valeur(p,y) - valeur(p,x))
```

```
  fin
finfonction
```

XI-C. Amélioration de la complexité▲

Même si en première approche, la complexité ne prend en compte que le nombre d'opérations (+, *), en seconde analyse, les multiplications sont beaucoup plus coûteuses que les additions. Cela est pris en compte dans les algorithmes ci-dessous.

XI-C-1. Valeur en un point▲

L'algorithme énoncé au paragraphe précédent effectue 2n multiplications. Le schéma d'Horner d'un polynôme permet d'effectuer n multiplications seulement. Le schéma d'Horner repose sur la propriété suivante :

Soit P(x) un polynôme de degré supérieur à 0 :

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

Alors, P(x) s'écrit :

$$P(x) = A(x)x + a_0$$

On en déduit le schéma de Horner :

$$P(x) = ((\ldots((a_n x + a_{n-1})x + a_{n-2})\ldots)x + a_1)x + a_0$$

Sélectionnez

```
fonction valeur(ref p:polynome;val x: réel):réel;
  var s:réel;
  début
    s = p.coeff[p.degre];
    pour i allant de p.degre-1 à 0 pas de -1 faire
      s = s * x + p.coeff[i]
    finpour
    retourner(s)
  fin
finfonction
```

XI-D. Multiplication ▲

Une méthode « diviser pour régner » permet d'améliorer cet algorithme. Elle est basée sur l'égalité suivante :

$$(ay + b)(cy + d) = acy^2 + ((a + b)(c + d) - ac - bd)y + bd$$

Cette égalité signifie, entre autres, que si deux polynômes sont de degré 1, il suffit de trois multiplications de réels pour obtenir leur produit. Les quantités a, b, c, d, y étant quelconque, celles-ci peuvent être elles-mêmes des polynômes. Soit le polynôme :

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

On peut écrire P(x) sous la forme :

$$P(x) = A(x)x^{1+\lfloor n/2 \rfloor} + B(x)$$

avec :

$$A(x) = \sum_{i=0}^{n-1-\lfloor n/2 \rfloor} p_i x^i$$

$$B(x) = \sum_{i=0}^{\lfloor n/2 \rfloor} p_i x^i$$

De même, on a :

$$Q(x) = \sum_{i=0}^{n} q_i x^i$$

$$Q(x) = C(x)x^{1+\lfloor n/2 \rfloor} + D(x)$$

$$C(x) = \sum_{i=0}^{n-1-\lfloor n/2 \rfloor} q_i x^i$$

$$D(x) = \sum_{i=0}^{\lfloor n/2 \rfloor} q_i x^i$$

On peut donc utiliser l'équation de départ avec : $a = A(x)$, $b = B(x)$, $c = C(x)$, $d = D(x)$ et $y = x^{1+\lfloor n/2 \rfloor}$.

De plus, on est amené à calculer des produits de polynômes de degré au plus n/2. Il s'agit donc d'une méthode : « diviser pour régner ». La complexité en nombre de multiplications est alors $O\left(n^{\log_2 3}\right)$. Comme on notera ci-dessous, l'algorithme est plus complexe à écrire, mais il est bien plus efficace aussi.
Un prétraitement permet de considérer des polynômes de même degré. Il faut donc une fonction « chapeau ». On définit de plus deux autres fonctions utiles pour le calcul :

- etend : si le degré de $P(x)$ est supérieur au degré de $Q(x)$, alors $Q(x)$ est modifié de manière à ce que les coefficients manquants soient à zéro, et les degrés des deux polynômes deviennent ainsi égaux ;
- tronque : permet d'initialiser un polynôme avec les premiers termes d'un autre polynôme (calcul de $B(x)$ et $D(x)$).

Sélectionnez

```
fonction multiplie(ref p,q:polynome):polynome;
  var p1,p2:polynome;
  fonction etend(ref p:polynome;val n:entier):polynome;
    var i:entier;
    var r:polynome;
    début
      r = decale(p,0);
      r.degre = n;
      pour i de p.degre+1 à n faire
        r.coeff[i] = 0;
      finpour;
      retourner(r);
    fin
  finfonction
  fonction tronque(ref p:polynome;val n:entier)
    var c:polynome;
    var i:entier;
    début
      c.degre = n;
      pour i de 0 à n faire
        c.coeff[i] = p.coeff[i];
      finpour;
      retourner(c);
    fin
  finfonction
  fonction multirec(ref p,q:polynome)
    var a,b,c,d:polynome;
    var c0,c1,c2:réel;
    var C0,C1,C2:réel;
    var m:entier;
    début
      selon que
        cas p.degre == 0
          retourner(polynome([p.coeff[0]*q.coeff[0]]))
        cas p.degre == 1
          c0 = p.coeff[0] * q.coeff[0];
          c2 = p.coeff[1] * q.coeff[1];
          c1 = (p.coeff[0] + p.coeff[1]) * (q.coeff[0] + q.coeff[1]);
          c1 = c1 - c0 - c2;
          retourner(polynome([c0,c1,c2]));
        autrement
          m = partieEntière(p.degre/2);
          a = decale(p, -(m+1));
          b = tronque(p, m);
          c = decale(q, -(m+1));
          d = tronque(q, m);
          C2 = multirec(a, c);
          C0 = multirec(b, d);
          C1 = multirec(ajout(a,b), ajout(c,d));
          C1 = ajout(C1, ajout(moins(C0), moins(C2)));
          C1 = decale(C1, 1+m);
          C2 = decale(C2, 2+2*m);
          C0 = ajout(C0, ajout(C1,C2));
          retourner(C0);
      finselonque:
    fin
  finfonction;
    début
      si p.degre > q.degre alors
        p1 = p;
        p2 = etend(q, p.degre);
      sinon
        p1 = q;
        p2 = etend(p, q.degre);
      finsi;
      retourner(multirec(p1,p2));
    fin
  finfonction
```

XII. Listes▲

XII-A. Définition▲

Définition 6.1. Une **liste** est une table d'association à clé uniqueTable d'association à clé unique telle que :

- le nombre d'éléments de la table (**dimension** ou **taille**) soit variable ;

- l'accès aux éléments s'effectue indirectement par le contenu de la clé qui le localise appelée **pointeur**.

La complexité de l'accès à un élément par son pointeur est $O(1)$. Si p est un pointeur vers un élément alors **contenu(p)** est l'élément lui-même. Un pointeur qui n'adresse aucun élément a pour valeur NIL. On écrit en EXALGO pour déclarer un pointeur :

Sélectionnez

```
nom_pointeur=^type_predefini;
```

On écrit en EXALGO pour déclarer une liste :

Sélectionnez

```
type_liste=liste de type_predefini;
```

La manipulation des éléments de la liste dépend des fonctions définies comme s'exécutant en temps $O(1)$.

XII-B. Liste simplement chainée▲

Définition 6.2. Une liste est dite **simplement chainée** si les opérations suivantes s'effectuent en $O(1)$:

- accès :

  Sélectionnez

  ```
  fonction premier(val L :type_liste) :^type_predefini;
  fonction suivant(val L :type_liste;
                    val P :^type_predefini) :^type_predefini;
  fonction listeVide(val L :type_liste) :booléen;
  ```

- modification :

Sélectionnez

```
fonction créer liste(ref L :type_liste) :vide;
fonction insérerAprès(val x :type_prédéfini;
                      ref L :type_liste;
                      P :^type_predefini) :vide;
fonction insérerEnTete(val x :type_prédéfini;
                      ref L :type_liste) :vide;
fonction supprimerAprès(ref L :type_liste;val P :^type_predefini) :vide;
fonction supprimerEnTete(ref L :type_liste) :vide;
```

On écrira en EXALGO listeSC pour préciser qu'il s'agit d'une liste simplement chaînée.

XII-B-1. Test de fin de liste▲

Sélectionnez

```
fonction estDernier(ref L :listeSC de type_prédéfini;
                  ref P :^type_prédéfini) :booléen;
  début
    retourner(suivant(L,P) == NIL)
  fin
finfonction
```

XII-B-2. Chercher un élément dans une liste▲

Sélectionnez

```
fonction chercher(ref L :listeSC de type_prédéfini;
                  ref E :type_prédéfini) :^type_predefini;
  début
  var p :^type_prédéfini;
  début
    si listeVide(L) alors
      retourner(NIL)
    sinon
      p = premier(L);
      tant que non(estDernier(L,p)) et (contenu(p) != e) faire
         p = suivant(L,p);
      fintantque
      si (contenu(p) != e) alors
        retourner(NIL)
      sinon
        retourner(p)
      finsi
    finsi
  fin
finfonction
```

Complexité :

- minimum : $O(1)$;

- maximum : $O(n)$.

XII-B-3. Trouver le dernier élément▲

Sélectionnez

```
fonction trouverDernier(ref L :listeSC de type_prédéfini) :^type_predefini;
  var p :^type_prédéfini;
  début
    si listeVide(L) alors
      retourner(NIL)
    sinon
      p = premier(L);
      tant que non(estDernier(L,P)) faire
          p = suivant(L,p);
      fintantque
      retourner(p)
    finsi
  fin
finfonction
```

Complexité : $O(n)$.

XII-C. Liste doublement chaînée▲

Définition 6.3. Une liste **doublement chaînée** est une liste pour laquelle les opérations en temps $O(1)$ sont celles des listes simplement chaînées auxquelles on ajoute les fonctions d'accès.

Sélectionnez

```
fonction dernier(val L :type_liste) :^type_predefini;fonction précédent(val L :type_liste;
                   val P :^type_predefini) :^type_predefini;
```

On écrira en EXALGO listeDC pour préciser qu'il s'agit d'une liste doublement chaînée.

XII-C-1. Supprimer un élément▲

Sélectionnez

```
fonction supprimer(ref L :listeDC de type_predefini;
                   val e : type_prédéfini) :booléen;
  var p,prec,suiv :^type_prédéfini;
  début
    p = chercher(L,e);
    si p == NIL alors
      retourner(FAUX)
    sinon
      si estPremier(L,p) alors
        supprimerEnTete(L)
      sinon
        prec = précédent(L,p);
        supprimerApres(L,prec);
      finsi
      retourner(VRAI)
    finsi
  fin
finfonction
```

Complexité :

- minimum : $O(1)$;
- maximum : $O(n)$.

XII-D. Quelques algorithmes▲

XII-D-1. Taille▲

Sélectionnez

```
fonction taille(val L :type_liste) :entier;
  var p :^type_prédéfini;
  var t :entier;
  début
    si listeVide(L) alors
      retourner(0)
    sinon
      retourner(1 + taille( suivant(L, premier(L)) ))
    finsi
  fin
finfonction
```

Complexité : $O(n)$.

XII-D-2. Insérer dans une liste triée▲

On suppose la liste triée doublement chaînée dans l'ordre croissant :

Sélectionnez

```
fonction insertionTrie(ref L :listeDC de type_prédéfini;
                     val e : type_prédéfini) :vide;
  var p :^type_prédéfini;
  début
    si listeVide(L) alors
      insererEnTete(e,L)
    sinon
      si contenu(premier(L))>e alors
        insererEnTete(e,L)
      sinon
        insererTrie(suivant(L,premier(L)), e)
      finsi
    finsi
  fin
finfonction
```

Complexité moyenne : $O(n)$.

XIII. Une implémentation de liste simplement chaînée de caractères▲

On considérera dans tout ce chapitre que l'on a des valeurs qui correspondent à un caractère.

XIII-A. Qu'est-ce qu'implémenter▲

Pour certaines structures de données, l'ensemble des langages de programmation proposent une traduction immédiate. Pour d'autres, il n'existe pas de traduction immédiate. Il faut alors définir explicitement l'algorithme de chacune des primitives.

Exemple - les listes. On doit définir le stockage de la liste, et en fonction de ce stockage comment s'effectue par exemple l'adjonction.

L'implémentation doit respecter la complexité des primitives à part celle d'initialisation (celle-ci ne s'exécutera qu'une fois).

Exemple - les listes. On utilise souvent les fonctions ajouter et supprimer, mais une seule fois creerListe.

XIII-B. Choix de la structure▲

Ici nous allons choisir de ranger les éléments dans un tableau « suffisamment grand ». Chaque élément du tableau est une paire (valeurElement, pointeurSuivant). Un pointeur est la valeur d'un index du tableau ; ainsi l'accès au suivant est en complexité $O(1)$. La zone de stockage peut donc être décrite par :

Sélectionnez

```
elementListe = structure
                 valeur :car;
                 suivant :entier;
               finstructure;
stockListe = tableau[1..tailleStock] d'elementListe;
```

La valeur du pointeur (champ suivant) est donc un entier compris entre 0 et tailleStock. La valeur 0 correspondant à l'absence d'élément suivant. Le premier élément doit être accessible en $O(1)$, il faut donc conserver son index. Si la liste est vide, par convention, l'index du premier sera 0. On peut donc représenter une liste par la structure suivante :

Sélectionnez

```
listeSC_Car = structure
                tailleStock :entier;
                premier :entier;
                vListe :stockListe;
              finstructure;
```

Le tableau de stockage étant grand, mais pas illimité, il faudra prévoir que l'espace de stockage puisse être saturé.

XIII-C. Primitives d'accès▲

Ces fonctions sont immédiates.

Sélectionnez

```
fonction premier(val L :listeSC_Car) :entier;
  début
    retourner L.premier;
  fin;
finfonction

fonction suivant(val L :listeSC_Car,P :entier) :entier;
  début
    retourner L.vListe[P].suivant;
  fin
finfonction

fonction listeVide(val L :listeSC_Car) :booléen;
```

```
  début
    retourner L.premier == 0;
  fin
finfonction
```

XIII-D. Gestion de l'espace de stockage▲

Pour ajouter un élément, il faut pouvoir trouver un élément « libre » dans le tableau. Une première solution consiste à marquer les éléments libres du tableau (par exemple champ suivant de l'élément a pour valeur -1). Dans ce cas, il faudra parcourir le tableau (complexité $O(n/2)$ en moyenne). Par suite, la primitive insérerAprès ne sera plus en complexité $O(1)$ puisqu'il faudra d'abord trouver un élément libre. Une solution compatible avec la complexité des primitives consiste à gérer cet espace de stockage en constituant la liste des cellules libres. On modifie donc en conséquence la description de listeSC_Car :

Sélectionnez

```
listeSC_Car = structure
                tailleStock :entier;
                premier :entier;
                premierLibre :entier;
                vListe :stockListe;
              finstructure;
```

Par convention, l'espace de stockage sera saturé lorsque l'index premierLibre vaut 0 (la liste des cellules libres est vide). On définit donc la fonction de test :

Sélectionnez

```
fonction listeLibreVide(val L :listeSC_Car) :booléen;
  début
    retourner L.premierLibre == 0;
  fin
finfonction
```

On définit deux primitives liées à la gestion du stockage :

- mettreCellule : met une cellule en tête d'une liste ;
- prendreCellule : supprime la cellule de tête d'une liste.

Les opérations sont respectivement de type insererEnTete et supprimerEnTete. Préciser la liste sur laquelle s'effectue l'opération revient à préciser le pointeur de tête sur lequel on travaille.

Sélectionnez

```
fonction prendreCellule(ref L :listeSC_Car,ref tete :entier) :entier;
  var nouv :entier;
  début
    nouv = tete;
    tete = suivant(L,nouv);
    retourner nouv;
  fin
finfonction

fonction mettreCellule(ref L :listeSC_Car,val P :entier,ref tete :entier) :vide;
  début
    L.vListe[P].suivant = tete;
    tete = P;
  fin
finfonction
```

XIII-E. Primitives de modifications▲

Sélectionnez

```
fonction créer_liste(ref L :listeSC_Car;val tailleMax :entier) :vide;
  var i :entier;
  début
    L.tailleStock = tailleMax;
    L.premier = 0;
    L.premierLibre = 1;
    pour i allant de 1 à L.tailleStock-1 faire
       L.vListe[i].suivant = i+1;
    finpour
    L.vListe[tailleStock].suivant = 0;
  fin
finfonction

fonction insérerAprès(val x :car; ref L :listeSC_Car; val P :entier) :booléen;
  var nouv :entier;
  début
    si listeLibreVide(L) ou P == 0 alors
      retourner faux;
    sinon
      nouv = prendreCellule(L,L.premierLibre);
      L.vListe[nouv].valeur = x;
      L.vListe[nouv].suivant = suivant(L,P);
      L.vListe[P].suivant = nouv;
      retourner vrai;
    finsi
```

```
  fin
finfonction

fonction insérerEnTete(val x :car;ref L :listeSC_Car) :booléen;
  var nouv :entier;
  début
    si listeLibreVide(L) alors
      retourner faux;
    sinon
      nouv = prendreCellule(L,L.premierLibre);
      L.vListe[nouv].valeur = x;
      mettreCellule(L,nouv, L.premier);
      retourner vrai;
   finsi
  fin
finfonction

fonction supprimerAprès(ref L :listeSC_Car;val P :entier) :booléen;
  var suivP :entier;
  début
    suivP = suivant(L,P);
    si P == 0 ou suivP == 0 alors
      retourner faux;
    sinon
      L.vListe[P].suivant = suivant(L,suivP);
      mettreCellule(L,suivP, L.premierLibre);
      retourner vrai;
    finsi
  fin
finfonction

fonction supprimerEnTete(ref L :listeSC_Car) :booléen;
  var tete :entier;
  début
    si listeVide(L) alors
      retourner faux;
    sinon
      tete = L.premier;
      L.premier = suivant(L,tete);
      mettreCellule(L,tete, L.premierLibre);
      retourner vrai;
    finsi
  fin
finfonction
```

XIV. Remerciements▲

Toute l'équipe de Developpez.com remercie sincèrement **M. Delest** qui nous a aimablement permis de publier son tutoriel sur notre site. Nous tenons également à remercier zoom61, Siguillaume et Winjerome pour la gabarisation, ainsi que Claude Leloup et Malick SECK pour leur relecture orthographique.

Tutoriel pour comprendre la méthode de factorisation du Crible Quadratique : une invention de Carl Pomerance