Algorithmique avancé

Dr MAMBE Moïse

Sommaire

I Introduction

Le but de l'algorithmique peut être résumé comme suit : Trouver un "bon" algorithme pour un problème donné. Cela nécessite souvent des connaissances. La plupart du temps, un algorithme connu peut être adapté au problème et il vaut mieux éviter de réinventer la roue du savoir-faire.

Trouver un "bon" algorithme soulève pas mal de questions :

- Existe-t-il un algorithme pour résoudre le problème en un temps fini ? (calculabilité, indécidabilité).
- Le problème est-il un "classique"? (modélisation, connaissances)
- Comment concevoir un algorithme? Il n'y a pas de méthode miracle mais on peut identifier quelques paradigmes, patrons d'algorithmes, classes d'algorithmes.
- L'algorithme A apporte-t-il bien la réponse au problème donné? (correction des algorithmes)
- Que dire des ressources utilisées par l'algorithme A? (analyse d'algorithmes)
- L'algorithme A est-il "raisonnablement" efficace pour le problème donné? Pourrait-on faire beaucoup mieux? Que peut-on dire des ressources minima nécessaires pour résoudre le problème donné? (complexité des problèmes)

L'objectif du cours est de vous donner quelques éléments de réponse.

II Eléments de base du langage

Pour écrire un algorithme, il faut un langage algorithmique constitué d'un vocabulaire et de règles syntaxiques. Ce langage doit être :

- **spécialisé** : pour écrire des algorithmes, pas des poèmes ni des recettes de cuisine ;
- **de haut niveau** : déchargé de détails techniques (ce n'est pas un langage de programmation) ;
- **concis** :"si ça ne tient pas en une page, c'est que c'est trop long" ;
- **modulaire** ;
- **typé**.

II.1 Qualités d'un algorithme

- **Qualité d'écriture** : un algorithme doit être structuré, indenté, modulaire, avec des commentaires pertinents, etc. Il faut pouvoir comprendre la structure d'un coup d'œil rapide, et pouvoir aussi revenir dessus 6 mois plus tard et le comprendre encore ;
- **Terminaison** : le résultat doit être atteint en un nombre fini d'étapes. Il ne faut donc pas de boucles infinies, il faut étudier tous les cas possibles de données, ... ;
- **Validité** : le résultat doit répondre au problème demandé. Attention, un jeu d'essais ne prouve **jamais** qu'un programme est correct. Il peut seulement prouver qu'il est faux ;
- **Performance** : étude du coût (complexité) en temps et en mémoire.

La complexité en mémoire (c'est à dire la place mémoire prise par l'algorithme) est un problème de moins en moins primordial vu les capacités techniques actuelles.

On sait que certains problèmes n'admettent pas de solutions, et que d'autres ont des solutions qui nécessitent beaucoup de temps d'exécution (et sont donc en fait irréalisables), même avec les ordinateurs actuels.

On distingue la complexité **en moyenne** et la complexité **dans le pire des cas** (parfois on s'intéresse aussi au **meilleur des cas** mais). Ce n'est pas le même problème, mais les deux sont parlants.

II.2 L'alphabet du langage

L'alphabet du langage est constitué des caractères utilisables dans l'écriture d'un algorithme, ce sont :

- les lettres de l'alphabet : a - z ou A - Z ;
- les chiffres : 0 - 9 ;
- les caractères spéciaux : #, $, ;.

II.3 Les opérateurs

On distingue 4 types d'opérateurs :

II.3.1 Les opérateurs arithmétiques

Priorité		
-	+	Addition
	-	Soustraction
	*	Multiplication
	DIV	Division entière
	/	Division réelle
+ ▼	%	(modulo) reste d'une division entière

II.3.2 Les opérateurs relationnels

Opérateur	égalité	différence	infériorité stricte	infériorité large	supériorité	supériorité large
Symbole	=	≠	<	≤	>	≥

II.3.3 Les opérateurs logiques

Opérateur	Et logique	Ou logique	Non logique
Symbole	ET	OU	NON

II.3.4 Les opérateurs de manipulation de chaînes de caractères

Opérateur	Concaténation
Symbole	& ou +

II.4 Les identificateurs

Ce sont des mots choisis par le concepteur de l'algorithme pour représenter les objets de l'algorithme (ex. nom de l'algorithme, nom de variable, …).

Un identificateur est constitué d'une suite de caractères alphanumériques dont le premier est obligatoirement une lettre de l'alphabet ou le caractère de soulignement.

Un identificateur doit être expressif c'est-à-dire, choisi de telle sorte qu'il désigne bien l'objet qu'il représente.

Exemple : prix_TTC, valeur_max, deuxPI
Contre exemple : 2PI

Il n'y a pas de distinction entre minuscule et majuscule. Par exemple les identificateurs Prix_TTC, PRIX_TTC, prix_ttc désignent le même objet.

II.5 Les mots réservés

Ce sont des identificateurs prédéfinis du langage algorithmique ; ils ne doivent pas être utilisés comme identificateur dans un algorithme.

Exemple : ALGORITHME, FONCTION, TANTQUE, POUR

Remarque : Par convention, les mots réservés seront écrits en majuscules ainsi que le premier caractère des constantes.

II.6 Les nombres

On distingue deux types de nombre : les entiers et les réels.
Ex. 125 (entier), 3.1459, 15 E-3 (réels).

II.7 Les commentaires

Un commentaire peut être inséré dans un algorithme afin de faciliter sa lecture et sa maintenance. Il commence par une parenthèse ouvrante accolé au caractère *, suivi du texte du commentaire. Il se termine par le caractère * accolé à une parenthèse fermante.

Exemple : (* prix_TTC désigne le prix Toute Taxe Comprise *)

Remarque : Il ne peut pas avoir d'imbrication de commentaire.

III Structure d'un algorithme

III.1 Exemple d'algorithme

```
FONCTION fact(D n :ENTIER) :ENTIER
VAR i, f: ENTIER
DEBUT
  SI (n = 0) OU (n = 1) ALORS
    f ← 1
  SINON
    DEBUT
      f ← 1
      POUR i ← 2, n
        f ← f * i
    FIN
  fact ← f
FIN
```

```
ALGORITHME factoriele
VAR m, res : ENTIER
DEBUT
  ECRIRE('Entrer un entier positif :' )
  LIRE(m)
  res ← fac(m)
  ECRIRE(m, ' ! = ', res)
FIN
```

Un algorithme comprend 4 parties :

- déclaration de sous-algorithme (facultative)
- tête de l'algorithme
- déclaration des objets (facultative)
- corps de l'algorithme

III.2 Tête de l'algorithme

Il s'agit de donner un nom (ayant une relation avec le but de l'algorithme) à l'algorithme précédé du mot réservé ALGORITHME.

Ex. : ALGORITHME factorielle

Tout objet manipulé par l'algorithme devra avoir été déclaré avant son utilisation dans le corps de l'algorithme. Les différents objets sont : les types, les constantes et les variables.

III.3 Déclaration de constantes

Une constante est un espace mémoire désigné par un identificateur et utilisé pour désigner une valeur fixe d'un ensemble donnée. Cette valeur reste inchangée tout au long de l'exécution de l'algorithme.

Syntaxe :
CONST identificateur = expression

Exemples :

```
CONST Pi = 3.1459
      Nbre_mois = 12
      Ecole = 'INP-HB'
      Omega = 100 * Pi
```

III.4 Déclaration de type

Un type est constitué par un ensemble de valeurs et par des opérateurs définis sur cet ensemble.

Syntaxe
TYPE
Identificateur = nom du type

La figure présente les différentes sortes de type que l'on rencontre en algorithme.

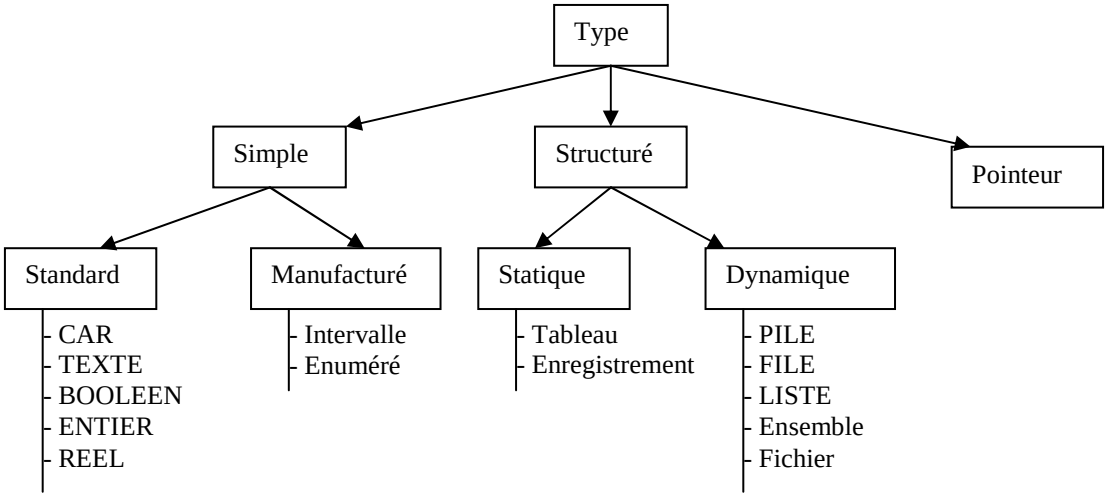


Fig. 3.1 : Les différentes sortes de types

III.5 Les types simples

On distingue deux sortes de types simples : type standard et manufacturé

III.5.1 Type standard

- Type entier (ENTIER) : c'est l'ensemble des entiers relatifs. Ses opérateurs sont les opérateurs arithmétiques (sauf l'opérateur de division des réels /), ainsi que les opérateurs relationnels.
- Type réel (REEL) : c'est l'ensemble des nombre réels. Ses opérateurs sont les opérateurs arithmétiques (sauf les opérateurs DIV et %), ainsi que les opérateurs relationnels.
- Type caractère (CAR) : c'est l'ensemble de la table ASCII. Une constante caractère est délimitée par des apostrophes. Ses opérateurs sont les opérateurs de concaténation et relationnels (il existe une relation d'ordre dans la table ASCII).

Exemple :

'a' < 'b' < 'c' < … 'z'
'a' + 'b' + 'c' donne la chaîne de caractère 'abc'

- Type chaîne de caractères (TEXTE) : c'est l'ensemble des chaînes de caractères délimitées par des apostrophes. Ses opérateurs sont les opérateurs de concaténation et relationnels.

Exemple :
'omega' < 'valeur'
'informatique' > 'Informatique' > 'Information'
'INP-' + 'HB'' donne la chaîne de caractère 'INP-HB'

- Type booléen (BOOLEEN) : c'est l'ensemble constitué des deux valeurs VRAI et FAUX. Ses opérateurs sont les opérateurs logiques.

Exemple :

A	B	A ET B	A OU B	NON A
VRAI	VRAI	VRAI	VRAI	FAUX
VRAI	FAUX	FAUX	VRAI	FAUX
FAUX	VRAI	FAUX	VRAI	VRAI
FAUX	FAUX	FAUX	FAUX	VRAI

III.5.2 Type manufacturé

Ce sont des types construits par le concepteur de l'algorithme. Leur intérêt est de définir de façon très précise le champ de validé des objets utilisés. On distingue deux sortes de types manufacturés : types énuméré et intervalle.

- Type énuméré : c'est un ensemble constitué par l'énumération des constantes symboliques.

Syntaxe :
TYPE
Identificateur = (valeur$_1$, valeur$_2$, …, valeur$_n$)

Exemple :
TYPE
Feux = (Orange, Vert, Rouge)
Banque = (BICICI, SIB, SGBCI, BIAO, BHCI, BCAO)
Jour = (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche)

Ses opérateurs sont les opérateurs relationnels.
Exemple :
Orange < Vert < Rouge

- Type intervalle : c'est un sous-type des types déjà définis. Ses opérateurs sont ceux du type père.

Syntaxe
Identificateur = borne_inférieur..borne_supérieure

Exemple :
TYPE
Chiffre = 0..9
Lettre = 'a'.. 'z'
Boulot = lundi..vendredi
Repos = samedi..dimanche

III.6 Le type tableau

Nous avons vu qu'une variable possède un nom et un type fixe et qu'elle est associée à une valeur qui peut varier pendant l'exécution d'un algorithme. Un tableau possède lui aussi un nom et un type fixe, mais il est associé à un ensemble de valeurs. Il permet donc de représenter des ensembles de valeurs ayant des propriétés communes (ex. vecteur et matrice en mathématique).

- Définition : Un tableau est une structure de données regroupant une suite d'éléments de même type rangés dans des cellules mémoires contiguës.

- Les éléments d'un tableau sont appelés variables indicées. Chaque élément du tableau est référencé par le nom du tableau suivi par un ou plusieurs indices. Chaque indice correspond à une dimension du tableau. Les valeurs des indices appartiennent à un sous-ensemble des entiers.

III.6.1 Tableau monodimension : Vecteur

Pour représenter un vecteur en mémoire, on utilisera des cellules contiguës. Si chaque élément du tableau occupe p cellules et a l'adresse de premier élément du tableau alors l'adresse d'un élément d'indice i est donnée par la formule :

a + (i - borne_inf)p

- Déclaration d'un vecteur en tant que variable : cette déclaration est à utiliser si on doit déclarer une seule instance d'un tableau.

Syntaxe

```
VAR
  Identificateur : TABLEAU[indice_inf..indice_sup] DE type T
```

- Déclaration d'un vecteur en tant que type : cette déclaration est à utiliser si on doit déclarer plusieurs instances d'un tableau ou si on doit utiliser un paramètre de type tableau dans un sous-algorithme.

Syntaxe

```
TYPE
  Identificateur = TABLEAU[indice_inf..indice_sup] DE type T
VAR
  Identificateur2 : Identificateur
```

Chaque élément du tableau est référencé dons par le nom du tableau suivi par un indice.

Exemple : tableau de 5 éléments de type entier

```
VAR
  t : TABLEAU[1..5] DE ENTIER
ou
TYPE
  Vecteur = TABLEAU[1..5] DE ENTIER
VAR
  t : Vecteur
```

Accès aux éléments de tableau t : on utilise un seul indice.

t[1] représente le 1^{er} élément du tableau t
t[2] représente le $2^{ième}$ élément du tableau t
t[i] représente le $i^{ième}$ élément du tableau t

III.6.2 Tableau à plusieurs dimensions

```
TYPE
  Identificateur = TABLEAU[indice_inf1..indice_sup1, indice_inf2..indice_sup2,
                           indice_infn..indice_supn] DE type T
VAR
```

Identificateur2 : Identificateur

Exemple : matrice de 3 lignes et 4 colonnes d'éléments de type réels c'est-à-dire 12 éléments.
VAR
  m : TABLEAU[0..2, 1..4] DE REEL
ou
TYPE
  Matrice = TABLEAU[0..2, 1..4] DE REEL
VAR
  m : Matrice

Accès aux éléments de tableau m : on utilise deux indices : un pour les lignes (1^{er} indice) et un autre pour les colonnes ($2^{ième}$ indice).
  m[0, 1] représente l'élément situé à l'intersection de la ligne 0 et de la colonne 1
  m[1, 2] représente l'élément situé à l'intersection de la ligne 1 et de la colonne 2
  m[i, j] représente l'élément situé à l'intersection de la ligne i et de la colonne j

III.7 Déclaration de variables

Une variable est un espace mémoire désigné par un identificateur et utilisé pour désigner une valeur quelconque d'un ensemble donné ; sa valeur pouvant changer tout au long de l'exécution de l'algorithme. C'est en fait une adresse où long range une valeur.

Syntaxe :
  VAR
    Identificateur : type de la variable

Exemple
VAR
    dividende, diviseur, quotient, reste : ENTIER
    prix_TTC : REEL

III.8 Corps de l'algorithme

Le corps d'un algorithme est la partie qui contient les instructions exécutables de l'algorithme. Cette partie est délimitée par les mots réservés DEBUT et FIN.

III.8.1 Affectation

L'affectation est l'instruction la plus élémentaire ; elle consiste à affecter à une variable la valeur d'une expression de même type ou de type compatible.

Syntaxe
  Identificateur ← Expression

Cette instruction se lit :
- Identificateur reçoit la valeur de Expression
- Ranger la valeur de Expression dans Identification

III.8.2 Instruction d'écriture

L'écriture est l'opération qui permet de véhiculer les données de la mémoire centrale vers l'extérieur (ex. écran).

Syntaxe

ECRIRE(v_1, v_2, …, v_n)

où les v_i sont des variables et/ou des constantes chaîne de caractères délimitées par des apostrophes.

Exemple :

```
ALGORITHME ecriture
CONST Pi = 3.14
VAR omega : REEL
DEBUT
| omega ← 100 * Pi
| ECRIRE('Valeur de omega = ', omega)
FIN
```

Remarque

L'instruction ECRIRE sans argument permet de passer à la ligne suivante de l'écran.

III.8.3 Instruction de lecture

La lecture est l'opération qui consiste à véhiculer les données de l'extérieur (ex. clavier) vers la mémoire du calculateur. C'est en fait une affectation particulière.

Syntaxe

LIRE(v_1, v_2, …, v_n)

où les v_i sont des variables.

Exemples :

```
ALGORITHME lecture
VAR i, i, res : REEL
DEBUT
| ECRIRE('Entrer deux entiers :  ')
| LIRE(i, j)
| res ← i + j
| ECRIRE(i,' + ', j, ' = ', res)
FIN
```

III.8.4 Nature séquentielle d'un algorithme

Un algorithme se déroule en séquence, c'est-à-dire que les instructions sont exécutées l'une après l'autre, sauf contre indication.

Des instructions sont séquentiellement chaînées lorsque la fin de l'exécution d'une instruction déclenche l'exécution de celle qui la suit immédiatement.

Un bloc est un ensemble d'instructions séquentiellement chaînées délimitées par les mots réservés DEBUT et FIN.

Syntaxe

DEBUT

instruction$_1$
instruction$_2$
...
instruction$_n$
FIN

III.8.5 Les structures de contrôle

Un algorithme est dit structuré s'il est écrit à l'aide des trois structures de contrôle de base :

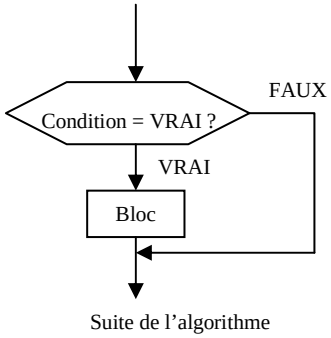
- L'affectation ;
- L'instruction conditionnelle ;
- L'instruction itérative ou boucle.

III.8.5.1 L'instruction conditionnelle

Il est nécessaire parfois dans un programme de pouvoir faire un test, c'est-à-dire évaluer une expression logique et enchaîner vers telle ou telle autre instruction selon la valeur trouvée.

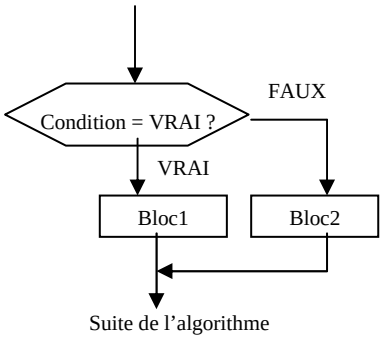
1) Syntaxe 1 : l'alternative simple

SI (condition) ALORS
Bloc



Exemple : test d'égalité de 2 nombre entiers.

```
ALGORITHME egalite
VAR a, b : ENTIER
DEBUT
  ECRIRE('Entrer deux nombres entiers : ' )
  LIRE(a, b)
  SI (a = b) ALORS
    ECRIRE('Les deux nombres sont égaux')
FIN
```



Syntaxe 2 : l'alternative complexe

SI (condition) ALORS
Bloc$_1$
SINON
Bloc$_2$

Exemple1 : test complet d'égalité de 2 nombre entiers.

```
ALGORITHME egalite
VAR a, b : ENTIER
DEBUT
  ECRIRE('Entrer deux nombres entiers : ' )
  LIRE(a, b)
  SI (a = b) ALORS
    ECRIRE('Les deux nombres sont égaux')
  SINON
    ECRIRE('Les deux nombres sont différents')
FIN
```

Exemple 2 : Déterminer le maximum de deux nombre réels saisis au clavier.

```
ALGORITHME Maximum
VAR a, b : REEL
DEBUT
  ECRIRE('Entrer deux nombres réels : ' )
  LIRE(a, b)
  SI (a = b) ALORS
    ECRIRE('Les deux nombres sont égaux')
  SINON SI (a > b) ALORS
    ECRIRE(a, ' > ', b)
  SINON
    ECRIRE(b, ' > ', a)
FIN
```

2) Le choix multiple

Le choix multiple ou aiguillage est une généralisation de l'alternative complexe. Il permet de choisir une instruction à exécuter parmi plusieurs selon la valeur d'une variable.

Syntaxe

```
CHOISIR (expression) PARMI
  valeur1 : Bloc1
  valeur2 : Bloc2
  …
  valeurn : Blocn
  AUTRE : Blocn+1
FIN
```

Si la valeur de expression est égale à une des valeurs i, le bloc i correspondant est exécuté, dans le cas contraire, le bloc n+1 est exécuté.

Exemple : Ecrire un algorithme permettant d'afficher les chiffres en toute lettre.

```
ALGORITHME Chiffre_Lettre
VAR chiffre : ENTIER
DEBUT
  ECRIRE('Entrer un chiffre : ' )
  LIRE(chiffre)
  CHOISIR (chiffre) PARMI
    0 : ECRIRE('zéro')
    1 : ECRIRE('un')
    2 : ECRIRE('deux')
    …
    9 : ECRIRE('neuf')
    AUTRE : ECRIRE('Chiffre inconnu')
  FIN
FIN
```

Remarque : La valeur de expression peut être de type : ENTIER, CAR, TEXTE, intervalle ou énuméré.

Exemple :

```
ALGORITHME Intervalle
VAR c : CAR
DEBUT
  ECRIRE('Entrer un caractère : ' )
  LIRE(c)
  CHOISIR (c) PARMI
    'A'.. 'Z', 'a'.. 'z' : ECRIRE('Lettre')
    '0'.. '9' : ECRIRE('Chiffre')
    '+',.'-', '*', '/' : ECRIRE('Opérateur arithmétique')
    AUTRE : ECRIRE('Caractère spécial')
  FIN
FIN
```

III.8.5.2 Les instructions itératives

Une instruction itérative ou boucle permet de spécifier l'exécution d'un bloc d'instruction un certain nombre de fois.
On distingue 3 types de boucles :

- POUR : on connaît le nombre d'itération à faire
- REPETER..JUSQUA : on ne connaît pas le nombre d'itération à faire
- TANTQUE : on ne connaît pas le nombre d'itération à faire

1) Boucle POUR : c'est une instruction itérative à bornes définies

Syntaxe

- Boucle POUR à pas croissant

  POUR i ← i_1, i_2, i_3
  Bloc

- Boucle POUR à pas décroissant

  POUR i ← i_2, i_1, i_3
  Bloc

Avec:

- i : variable de la boucle
- i1 : valeur initiale de i
- i2 : valeur finale de i
- i3 : appelé pas, c'est la valeur à ajouter à i à chaque itération (=1 ou -1 par défaut)

Exemple 1 : Afficher le message "Bonjour Fatou" 10 fois à l'écran.

```
ALGORITHME Salutation
CONST N = 10
VAR i : ENTIER
DEBUT
  POUR i ← 1, 10
    DEBUT
      ECRIRE('Bonjour Fatou' )
      ECRIRE
    FIN
FIN
```

Exemple 2 : Factorielle d'un nombre.

n ! = 1 x 2 x 3 x…x n

n ! = n x (n-1) x (n-2) x … x 2 x 1

```
ALGORITHME FactorielleCroissant
VAR i, n, fact : ENTIER
DEBUT
  ECRIRE('Entrer un entier positif :' )
  LIRE(n)
  SI (n = 0) OU (n = 1) ALORS
    fact ← 1
  SINON
    DEBUT
      f act← 1
      POUR i ← 2, n
        fact ← f * i
    FIN
  ECRIRE(n, ' ! = ', fact)
FIN
```

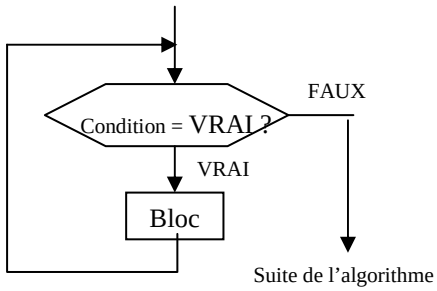
```
ALGORITHME
FactorielleDecroissant
VAR i, n, fact : ENTIER
DEBUT
  ECRIRE('Entrer un entier positif :' )
  LIRE(n)
  SI (n = 0) OU (n = 1) ALORS
    fact ← 1
  SINON
    DEBUT
      f act← 1
      POUR i ← n, 2, -1
        fact ← fact * i
    FIN
  ECRIRE(n, ' ! = ', fact)
FIN
```

2) Boucle TANTQUE

On utilise cette instruction quand on n'est pas sûr que le corps de la boucle doit être exécuté au moins une.

Syntaxe

```
TANTQUE (condition)
  Bloc
```



Tant que la condition est vérifiée, le corps de la boucle (Bloc) est exécuté.

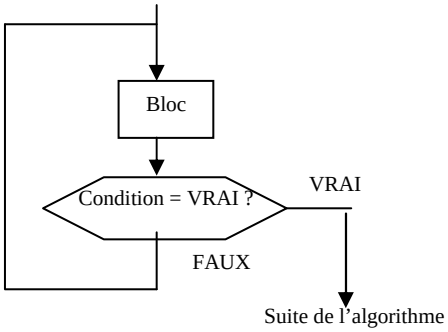
Exemple : Afficher le nom et le prénom de l'utilisateur précédé du mot Bonjour autant de fois qu'il le désire.

```
ALGORITHME Salutation
VAR nom, prenom : TEXTE
        reponse : CAR
DEBUT
  reponse ← 'o'
  TANTQUE(reponse = 'o')
    DEBUT
      ECRIRE('Entrer votre nom et votre prénom :' )
      LIRE(nom, prenom)
      ECRIRE('Bonjour  ',nom, ' ', prenom)
      ECRIRE('Continuer oui/non : ')
      LIRE(reponse)
    FIN
FIN
```

3) Boucle REPETER..JUSQUA

On utilise cette instruction quand on est sûr que le corps de la boucle doit être exécuté au moins une fois.

Syntaxe
  REPETER
    Bloc
  JUSQUA(condition)



Le corps de la boucle est exécuté jusqu'à ce que la condition soit vérifiée.

Exemple : Afficher le nom et le prénom de l'utilisateur précédé du mot Bonjour autant de fois qu'il le désire.

```
ALGORITHME Salutation
VAR nom, prenom : TEXTE
        reponse : CAR
DEBUT
  REPETER
      ECRIRE('Entrer votre nom et votre prénom :' )
      LIRE(nom, prenom)
      ECRIRE('Bonjour  ',nom, ' ', prenom)
      ECRIRE('Continuer oui/non : ')
      LIRE(reponse)
  JUSQUA(reponse = 'n')
FIN
```

Remarque
  Pour transformer une boucle TANTQUE en une boucle REPETER..JUSQUA et vice versa, il suffit de prendre la négation de la condition.

TANTQUE(condition) est équivalent à REPETER
  Bloc Bloc

IV Les sous-algorithmes

IV.1 Définition

Pour des besoins de clarté et maintenance, dès qu'un algorithme atteint une certaine complexité ou un certain volume, il devient nécessaire de le décomposer en plusieurs unités logiques ou modules plus simples que l'algorithme initial. Ces différents modules sont appelés sous-algorithmes.

Il peut arriver aussi qu'une suite d'instructions intervienne plusieurs fois dans un même algorithme ; dans ce cas, au lieu de réécrire plusieurs fois cette suite d'instructions, on l'écrit une seule fois sous forme d'un sous-algorithme qui peut être appelé n'importe où dans un algorithme (pour des besoins de réduction du code machine).

Exemple : permutation de 4 nombres entiers 2 à deux.

```
ALGORITHME permutation
VAR a, b, c, d, temp : ENTIER
DEBUT
  ECRIRE('Entrer 4 nombres entiers : ')
  LIRE(a,b,c,d)
  ECRIRE('Avant permutation : ',a, b, c, d)
  temp ← a
  a ← b    traitement 1
  b ← temp
  temp ← c
  c ← d    traitement 2

  d ← temp
  ECRIRE('Après permutation : ',a, b, c, d)
FIN
```

Les traitements 1 et traitement 2 sont identiques sauf les variables qui changent.

On distingue deux sortes de sous-algorithmes : les procédures et les fonctions.

IV.2 Procédure

Une procédure est un sous-algorithme assurant de manière autonome un traitement particulier. Ce traitement peut alors être répété dans l'algorithme principal ou dans un autre sous-algorithme par simple appel de la procédure.

La notion de procédure comporte 2 aspects qu'il est important de distinguer :

- déclaration de la procédure : c'est la définition de la procédure ;
- appel de la procédure : c'est l'utilisation de la procédure.

IV.2.1 Déclaration d'une procédure

La déclaration d'une procédure se situe en dehors d'un algorithme.

Syntaxe

```
PROCEDURE identificateur(liste de paramètres)
Déclaration d'objets locaux
DEBUT
(* corps de la procédure *)
FIN
```

IV.2.2 Structure d'une procédure

La structure d'une procédure est semblable à celle d'un algorithme ; elle comporte une tête, une partie déclaration et un corps.

La tête de la procédure commence par le mot réservé PROCEDURE suivi du nom de la procédure et d'une liste de paramètres placée entre parenthèses.
Les paramètres assurent la communication entre les différents modules d'un algorithme.

Exemple : maximum de 2 entiers.

```
PROCEDURE maximum (D x, y : ENTIER ; R max : ENTIER)
DEBUT
  SI (x ≥ y) then
    max ← x
  ELSE
    max ← y
FIN
```

IV.2.3 Notion de paramètres

On distingue 3 sortes de paramètres ou d'arguments :

- Paramètre de type Donnée : chaque paramètre est précédé de **D** ; il sert à véhiculer les données de l'extérieur (algorithme ou sous-algorithme) vers l'intérieur du sous-algorithme (ex. paramètres x et y).
- Paramètre de type Résultat : chaque paramètre est précédé de **R** ; il sert à véhiculer les résultats de l'intérieur du sous-algorithme vers l'extérieur (ex. paramètres max)..
- Paramètre de type Donnée-Résulat : chaque paramètre est précédé de **DR** ; c'est la combinaison des paramètre de type Donnée et de type Résultat. Dans un premier temps, il sert à véhiculer les données de l'extérieur (algorithme ou sous-algorithme) vers l'intérieur du sous-algorithme. Adans un deuxième temps, après le traitement, il sert à véhiculer les résultats de l'intérieur du sous-algorithme vers l'extérieur.

Exemple : échange de 2 entiers.

```
PROCEDURE echange(DR x, y : ENTIER)
VAR temp : ENTIER
DEBUT
  temp ← x
  x ← y
  y ← temp
FIN
```

Dans cet exemple, les paramètres x et y sont de type Donnée-Résultat.

IV.2.4 Applel d'une procédure

L'appel d'une procédure s'effectue à l'intérieur d'un algorithme ou d'un sous-algorithme appelant.

Exemple

```
ALGORITHME permutation
VAR a, b, c, d, temp : ENTIER
DEBUT
  ECRIRE('Entrer 4 nombres entiers : ')
  LIRE(a,b,c,d)
  ECRIRE('Avant permutation : ',a, b, c, d)
  echange(a, b) (* appel de la procédure *)
  echange(c, d)
  ECRIRE('Après permutation : ',a, b, c, d)
FIN
```

La partie déclaration et le corps de la procédure sont identiques à celles d'un algorithme.

Remarque : une procédure peut ne pas avoir de paramètres.

Exemple

```
PROCEDURE affMessage
DEBUT
  ECRIRE('Bonjour Fatou, comment vas-tu ? '
FIN
```

IV.3 Fonction

Une fonction est un sous-algorithme similaire à la procédure mais qui calcule une valeur d'un type donné ; cette valeur sera retournée à l'algorithme appelant à travers le nom de la fonction qui est considéré comme un paramètre résultat.

IV.3.1 Déclaration d'une fonction

La déclaration d'une fonction se situe en dehors d'un algorithme.

Syntaxe

```
FONCTION identificateur(liste de paramètres) : Type de la foncntion
Déclaration d'objets locaux
DEBUT
  (* corps de la procédure *)
  identificateur ← expression
FIN
```

Exemple

```
FONCTION maximum (D x, y : ENTIER) : ENTIER
VAR max : ENTIER
DEBUT
  SI (x ≥ y) then
    max ← x
  ELSE
    max ← y
  maximum ← max
FIN
```

IV.3.2 Appel d'une fonction

L'appel d'une fonction s'effectue à l'intérieur d'un algorithme ou d'un sous-algorithme appelant.

Exemple

```
ALGORITHME PlusGrand
VAR a, b, max : ENTIER
DEBUT
  ECRIRE('Entrer 2 nombres entiers : ')
  LIRE(a,b)
  max ← maximum(a, b)   (* appel de la procédure *)
  ECRIRE('Maximum de ',a, ' et ', b, ' = ', max)
FIN
```

Différences entre procédure et fonction
On note deux différences :

- au niveau de la déclaration : une fonction retourne une valeur (donc possède un type) par l'intermédiaire de son nom qui est un paramètre résultat ;
- au niveau de l'appel : le nom d'une fonction peut apparaître dans le membre gauche d'une affectation.

IV.3.3 Fonctions prédéfinies

Il existe des procédures et fonctions prédéfinies, exemples :

Procédure ou fonction	**Signification**
EXP(x)	exponentielle
SQRT(x)	racine
ABS(x)	Valeur absolue

IV.4 Mode de passage des paramètres

Les paramètres jouent deux rôles :

- transmettre au sous-algorithme, au moment de son appel, les valeurs nécessaires à son exécution ;
- au moment du retour, transmettre à l'algorithme appelant le ou les résultats du traitement effectue.

On distingue deux sortes de paramètres :

- Paramètre formel : ce sont les paramètres qui apparaissent entre parenthèses lors de la déclaration d'un sous-algorithme ; ils indiquent les types des objets qui seront transmis et permettent de réserver la place mémoire nécessaire pour les stocker (ex. n et res dans la procédure fact)

- Paramètre effectif : appelés aussi paramètres actuels, ce sont les paramètres qui apparaissent entre parenthèses à l'appel du sous-algorithme. Ils doivent avoir les mêmes types que ceux des paramètres formels, par contre ils peuvent ne pas avoir nécessairement les mêmes noms (ex. n et p dans l'algorithme combinaison).

On distingue deux modes de passages des paramètres c'est-à-dire la façon dont la valeur des paramètres effectifs est transmise aux paramètres formels.

IV.4.1 Mode passage par valeur

Dans ce mode, il y a une copie de la valeur des paramètres effectifs qui est assignée aux paramètres formels de même rang. De ce fait, toute modification des paramètres formels par le sous-algorithme n'a aucune incidence sur les paramètres effectifs.

Les paramètres de type Donnée utilisent ce mode de passage.
Exemple

```
PROCEDURE inc(D x :ENTIER)
DEBUT
| x ← x + 1
| ECRIRE('Valeur de x dans le sous-algorithme ', x)
FIN
```

```
ALGORITHME incrementation
VAR y : ENTIER
DEBUT
| y ← 0
| ECRIRE('Avant modification, valeur de y dans l'algo ', y)
| inc(y)
| ECRIRE('Après modification, valeur de y dans l'algo ', y)
FIN
```

Avant modification, valeur de y dans l'algo	0
valeur de x dans le sous-algorithme	1
Après modification, valeur de y dans l'algo	0

x et y n'ont pas le même emplacement mémoire.

IV.4.2 Mode de passage par adresse

Dans ce mode, les paramètres formels et effectifs ont le même emplacement mémoire ; donc une modification de la valeur des paramètres formels par le sous-algorithme se répercute sur la valeur des paramètres effectifs de même rang.

Les paramètres de type Résultat et Donnée-Résultat utilisent ce mode de passage.

Exemple

```
PROCEDURE inc(DR x :ENTIER)
DEBUT
| x ← x + 1
| ECRIRE('valeur de x dans le sous-algorithme ', x)
FIN
```

```
ALGORITHME incrementation
VAR y : ENTIER
DEBUT
| y ← 0
| ECRIRE('Avant modification, valeur de y dans l'algo ', y)
| inc(y)
| ECRIRE('Après modification, valeur de y dans l'algo ', y)
FIN
```

Avant modification, valeur de y dans l'algo	0
Valeur de x dans le sous-algorithme	1
Après modification, valeur de y dans l'algo	1

x et y ont le même emplacement mémoire

V Terminaison et correction des algorithmes itératifs

Puisqu'un algorithme est supposé résoudre un problème précis, il faut s'assurer de sa justesse, i.e. qu'il fait bien la tâche qu'on lui assigne. Cela peut se faire rigoureusement et c'est pour cela que l'on parle de preuve ou de correction d'un algorithme.

Un algorithme est dit totalement correct si pour tout jeu de données il s'arrête et rend le résultat attendu.

Un algorithme incorrect peut :

- Ne pas s'arrêter (on dit qu'il boucle)
- S'arrêter en rendant un mauvais résultat
- S'il termine, fournir le bon résultat, mais la terminaison n'est pas garantie (partiellement correct)

Pour certains problèmes difficiles, on peut se contenter d'un algorithme incorrect si l'on sait contrôler son taux d'erreur.

Dans le cas des algorithmes récursifs, ces méthodes sont spécifiques.

Les actions et les structures des contrôles (affectation, enchaînement, alternative, boucle), possèdent un certain nombre de propriétés qui permettent d'établir à partir d'algorithme que celui-ci conduit bien au résultat espéré en un temps fini (logique de Hoare).

V.1 Notation

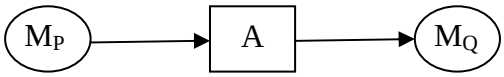
Un triplet de Hoare est un triplet noté :

{P} A {Q}

où P et Q sont des propositions logiques et A une instruction.

On dit qu'un triplet de Hoare {P} A {Q} est valide si P est vraie et qu'on exécute l'instruction A, alors, Q est vraie.

L'exécution d'un algorithme A transforme un état mémoire en un autre.



V.2 Propriété de l'affectation

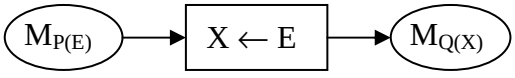
Elle s'énonce de la façon suivante :

Pour que {P} soit vrai après l'exécution de l'affectation X ← E (où E est une expression à évaluer), il suffit que la proposition {P(X ← E)} soit initialement vraie.

{P(X ← E)} est la proposition {P} dans laquelle on a remplacé toutes les occurrences de.

On note : {P(X ← E)} **X ← E** {P}

Le calcul de preuve se fait en partant de P et en calculant {P(X ← E)} par substitution X par E.



L'intérêt de cette propriété est que, si l'on veut atteindre une assertion à un état donné de l'exécution de l'algorithme, on peut par un raisonnement à reculons, construire les hypothèses et les instructions nécessaires à ce but.

Exemple

Montrons que pour obtenir {x > n+1} après l'exécution de $x \leftarrow \frac{1}{x} + n$, il faut nécessairement que {x < 1} soit initialement vrai.

Avec les notations précédentes on a :
{P} = {x > n+1}

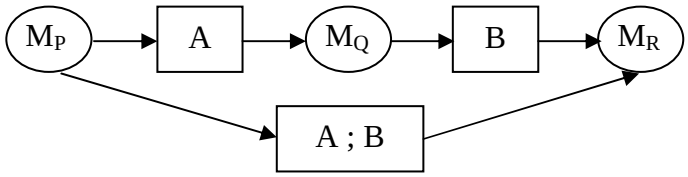
$\{P(x \leftarrow \frac{1}{x} + n)\} = \{\frac{1}{x} + n > n+1\} \Rightarrow \{\frac{1}{x} > 1\} \Rightarrow \{x < 1\}$

V.3 Propriété de l'enchaînement

La Propriété de l'enchaînement ou de la séquence est liée à la relation de Chasles et s'énonce comme suit :

- Si A est telle que son exécution, lorsque {P} est vraie, conduise à {Q} vraie ;
- Si B est telle que son exécution, lorsque {Q} est vraie, conduise à {R} ;
- Alors A suivie de B conduit à {R} vraie, lorsque {P} est initialement vraie.

Ce qui se résume par :
SI {P} A {Q} ET {Q} B {R} ALORS {P} **A ; B** {R}
Où **A ; B** signifie enchaînement séquentiel des actions A et B.



Le calcul de preuve est effectué en partant de R puis en calculant successivement Q et enfin P selon les règles qui s'appliquent pour B puis A.

Exemple : Montrons qu'il faut initialement {0 < x < 1}, pour qu'après l'enchaînement $x \leftarrow \frac{1}{x} - 1 ; x \leftarrow y + x$ on ait {x > y}.

Avec les mêmes notations on peut écrire :
{Q} = {x > 0}
{R} = {x > y}

{R(x ← y + x)} = {y + x > y} ={x > 0} = {Q}
puis

$\{Q(x \leftarrow \frac{1}{x} - 1)\} = \{\frac{1}{x} - 1 > 0\} = \{x < 1\}$

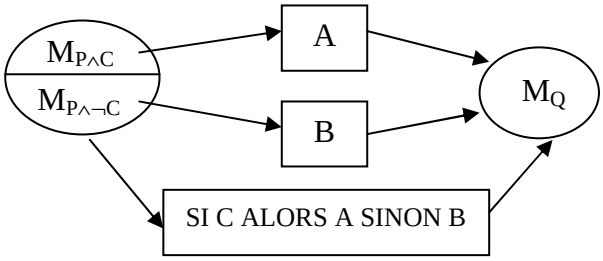
Or d'après {Q} {x > 0} donc {0 < x < 1} = {P}

V.4 Propriété de l'alternative

On exprime que l'enchainement de deux propositions {P} et {Q} par une alternative est liée à une condition {C} discriminante sur le choix de l'énoncé A ou B, ce qui se note :

SI {P ET C} **A** {Q}
ET {P ET NON C} **B** {Q}
ALORS {P} **SI {C} ALORS**
A
SINON
B
{Q}

On commence calculs de P1 = {P ET C} et P2 = {P ET NON C} tels que {P1} A {Q}
et {P2} B {Q} en suivant les règles qui s'appliquent pour A et B.



Exemple : montrons qu'avec n ≥ 0, on a la propriété suivante :
{y > x}
SI (x < 0) ALORS
y ← y + 1 – n*x
SINON
y ← y + x + 1
{y > x + 1}

Pour cela, il faut montrer les deux propriétés suivantes :
(i) {y > x ET x < 0} y ← y + 1 – n*x {y > x + 1}
(ii) {y > x ET x ≥ 0} y ← y + x + 1 {y > x + 1}

Posons
{P} = {y > x}
{Q} = {y > x+1}
{C} = {x < 0}
A : y ← y + 1 – n*x
B : y ← y + x + 1

a) {C} est réalisée
Avec les mêmes notations on peut écrire :
(i) {Q(y ← y + 1 – n*x)} = {Q(y ← y + 1 + A1)} avec A1 = -n*x > 0 car x <0 et n ≥ 0
⇒ y ← y + 1 + A1 (1)
Comme {y > x} ⇒ y = x + a avec a > 0
(explication si y > x alors on peut lui ajouter une valeur a > 0)
(1) Devient {y ← x + a + 1 + A1}
{y ← x + 1 + a + A1} comme (a + A1) >0

$\Rightarrow \{y > x + 1\} = \{Q\}$

b) NON {C} est réalisée
NON {C} = {x ≥ 0}
(ii) Q(y ← y + x + 1)} = {y + x + 1} (2)
Or {y > x ET x ≥ 0} ⇒ {y = x + a} avec a > 0
(2) Devient {Q(y ← x + a + x + 1)} = {Q(y ← x + 1 + a + x)} comme (a+x) > 0
⇒ {y > x + 1} = {Q}

V.5 Propriété de la répétition

On ne s'intéressera qu'à la boucle TANTQUE car d'un point de vue théorique, elle peut remplacer n'importe quelle autre boucle.

```
TANTQUE(C)
   A
```

La boucle TANTQUE est caractérisée par deux propriétés :

- **Propriété 1** : à la sortie de la boucle la condition est nécessairement fausse, ce qui se note : TANTQUE {C} A NON {C}
- **Propriété 2** : si {P} est un invariant pour l'action A, i.e. que si A ne modifie pas la valeur de vérité de {P}, alors {P} est un invariant pour la boucle, ce qui se note : SI {P ET C} A {P} ALORS {P} TANTQUE {C} A {P}

Pour démontrer les propriétés des algorithmes une notion clé est celle d'invariant de boucle.

Un invariant de boucle est une propriété telle que :

- initialisation : elle est vraie avant la première itération de la boucle ;
- conservation : si elle est vérifiée avant une itération quelconque de la boucle elle le sera encore avant l'itération suivante ;
- terminaison : si {P} est bien un invariant de boucle, on doit avoir automatiquement à la sortie de la boucle {NON C} et {P}.

Exemple : on calcule la somme des 100 éléments d'un tableau d'entier T

```
ALGORITHME somme
CONST N=100
VAR i, som : ENTIER
T : TABLEAU[1..N] DE ENTIER
DEBUT
  i ← 1
  som ← 0
  TANTQUE(i ≤ N)
    DEBUT
      som ← som + T[i]
      i ← i +1                 A
    FIN
FIN
```

On veut montre que $som = \sum_{i=1}^{N} T[i]$

Un invariant de cette boucle est la proposition :

$\{i \le N+1 \text{ ET } som = \sum_{j=1}^{i-1} T[j]\}$

a) Initialisation : En effet, avant l'entrée dans la boucle on a :

i = 1

som = 0

et $\{i \le N+1\ som = \sum_{j=1}^{0} T[j] = 0\}$ vrai.

b) Conservation : Examinons une itération effective pour $i = i_0$.

Nécessairement $i \le N$ à l'entrée de la boucle

Après l'exécution de l'action $i \leftarrow i + 1$, on obtient $i = (i_0 + 1) \le N + 1$

De plus, si à l'issue de l'itération précédente on a : $som = \sum_{j=1}^{i_0-1} T[j]$

L'action som ← som + T[i] conduit à

$som = \sum_{j=1}^{i_0-1} T[j] + T[i_0] = \sum_{j=1}^{i_0} T[j] = \sum_{j=1}^{i-1} T[j]$ (car $i = i_0 + 1 \Rightarrow i_0 = i - 1$)

Ce qui établit la démonstration par récurrence.

c) Terminaison

A l'issue de l'exécution de la boucle, on peut affirmer que la condition d'entrée dans la boucle est fausse et l'invariant est vrai, soit simultanément

$\{i > N\}$ ET $\{i \le N+1\}$ ET $som = \sum_{j=1}^{i-1} T[j]$

Ce qui implique $\{i = N+1\}$ ET $som = \sum_{j=1}^{N} T[j]$

Et prouve que l'algorithme calcule bien ce que l'on cherche.

$\{C\} = \{i \le N\}$

NON $\{C\} = \{i > N+1\}$

$\{P\} = \{i \le N+1 \text{ ET } som = \sum_{j=1}^{i-1} T[j]\}$

V.6 Terminaison de la boucle TANTQUE

Pour montrer la terminaison ou finitude de la boucle TANQUE, il suffit de trouver un invariant V qui s'exprime à l'aide des objets de l'algorithme et qui vérifie :

(i) $V \ge 0$ à l'entrée

(ii) $\{V \ge 0\}$ est un invariant de la boucle

(iii) V décroit

Montrons que la boucle TANTQUE ci-dessus est finie

(i) on prend V = N - i

(ii) on prend $\{i \le N + 1\}$ comme invariant de la boucle

(iii) V décroit

Prouvons (i)

V = N - i
i = 1 V = N $\Rightarrow$ V $\geq$ 0

Prouvons (ii)
N – i $\geq$ 0} invariant

Juste avant i $\leq$ N $\Rightarrow$ N+1 – i $\geq$ 0

Après l'exécution i $\leftarrow$ i +1
i $\leq$ N+1 $\Rightarrow$ N +1 – i $\geq$ 0

Donc proposition vraie avant **A** et vraie après **A**, par conséquent {v $\geq$ 0} est un invariant de la boucle TANTQUE

Prouvons (iii)
V décroit
En effet, d'une itération à une autre $V_{i+1} - V_i < 0$

Exercices 1 : Affectation
Montrer qu'une condition suffisante pour que {y $\leq$ 0} soit vraie après l'exécution de
y $\leftarrow$ 2xy – 1 est {$x^2 + y^2 \leq 1$} initialement vraie.

Exercice 2 : Enchaînement
Montrer qu'une condition suffisante pour que {y < 0} soit vraie après l'exécution de
x $\leftarrow$ y^2 + 1 suivie de l'exécution de y $\leftarrow$ x – 2 est {-1 < y < 1} initialement vraie. Montrer que cette condition est nécessaire.

Exercice 3 : Boucle
Pour un nombre positif donné x, on cherche à calculer une approximation de sa racine carrée à l'aide des deux suites suivantes :
$A_0 = x$ et $G_0 = 1$

$$A_{n+1} = \frac{A_n + G_n}{2} \quad G_{n+1} = \frac{2A_nG_n}{A_n + G_n}$$

A_n et G_n sont telles que $\lim_{n \to +\infty} A_n = \lim_{n \to +\infty} G_n = \sqrt{x}$

Questions

1. Ecrire un algorithme permettant de saisir un nombre positif x et de calculer sa racine carrée avec une précision relative ε donnée. La racine calculée sera affichée.
2. Montrer que l'algorithme écrit calcule bien une approximation de $\sqrt{x}$ en un nombre fini d'opérations.

Exercice 4 : Boucle

On considère l'algorithme suivant :

```
ALGORITHME calcul
VAR a, x, i : ENTIER
DEBUT
  LIRE(a)
  x ← 0
  i ← 0
  TANTQUE(i ≠ a)
    DEBUT
      x ← x + 2*i + 1
      i ← i +1
    FIN
    ECRIRE(x)
FIN
```

Questions

1. Montrer que l'algorithme s'exécute en un nombre fini d'opérations
2. Donner la valeur de **x** en fonction de celle de **a** à la fin de l'exécution de l'algorithme

VI Analyse des algorithmes

Deux algorithmes produisant les mêmes résultats peuvent être très différents du point de vue des méthodes utilisées, de leur complexité apparente et de leur efficacité.

La complexité apparente correspond à la difficulté qu'éprouve le lecteur à comprendre l'algorithme. Elle est liée au nombre et à la profondeur d'imbrication des structures de contrôle.

L'efficacité d'un algorithme est liée à la bonne utilisation des ressources temps de traitement et espace mémoire pour résoudre un problème donné.

Dans le cadre de ce cours on se limitera à l'efficacité d'un algorithme.

On mesure l'efficacité d'un algorithme par son coût qui est lié donc à deux paramètres essentiels :

- Le temps d'exécution appelé "Complexité temporelle" : il est liée au nombre d'opérations effectuées par l'algorithme ;
- L'espace mémoire requis appelé "Complexité spatiale" : il correspond à l'encombrement mémoire.

Problème : calculer x^n
données : x : réel , n: entier
Méthode 1 : $x^0 = 1$; $x^i = x * x^{i-1}$ avec i >0 complexité θ(n)

Méthode 2 : $x^0 = 1$;
$x^i = x^{i/2} * x^{i/2}$, si i est pair; complexité θ(log n)
$x^i = x*x^{i/2} * x^{i/2}$ si i est impair
...
résultats : y = x^n

Quelle méthode choisir? et pourquoi?
Plutôt la deuxième.

Le tableau ci-dessous montre l'évolution de la vitesse des microprocesseurs et la capacité de la mémoire centrale.

Historique	Vitesse μprocesseur	Mémoire
Fin 70	10 MHz	16 ko
Fin 90	× 40 400 MHz	× 4000 64 Mo
Fin 00	× 2.5 1 GHz	× 16 1 Go

VI.1 Complexité temporelle

En pratique, le temps d'exécution dépend de la taille et de la valeur des données. Par exemple dans le cas d'un algorithme de tri, si les valeurs sont partiellement triées ou triées à l'envers, le temps de traitement peut varier largement.

On définit alors trois sortes de complexité temporelle :

- **Complexité maximale** : c'est le temps d'exécution d'un algorithme dans le cas le plus défavorable (temps le plus long).

- **Complexité moyenne** : c'est le temps moyen d'exécution de l'algorithme appliqué à n données quelconques équiprobables (même probabilité).
  on calcule le coût pour chaque donnée possible puis on divise la somme de ces coûts par le nombre de données différentes.
- **Complexité dans le meilleur des cas** : on calcule le coût en se plaçant dans le meilleur des cas (temps le plus court).

En analyse de complexité, on étudie souvent le pire cas ce qui donne une borne supérieure de la complexité de l'algorithme.

Remarques :
- La complexité maximale et la complexité moyenne sont souvent équivalentes.
- Lorsqu'on étudie la complexité d'un algorithme, on ne s'intéresse pas au temps de calcul réel mais à un ordre de grandeur.
- Pour une complexité polynomiale, on ne s'intéresse qu'au terme de plus grand ordre.
- On exprime la complexité d'un algorithme comme une fonction f(n) de $\aleph$ dans $\Re$.

VI.1.1 Modèle fictif de calculateur

Pour évaluer le temps d'exécution d'un algorithme, on définit un modèle fictif de calculateur dont les caractéristiques sont les suivantes:

λ = temps de lecture/écriture (lambda)
τ = temps de toutes les opérations arithmétiques (tau)
μ = temps des opérations logiques (mu)

Si on a :
p opérations de L/E
q opérations arithmétiques
r opérations logiques

Le temps d'exécution est alors : $T = p\lambda + q\tau + r\mu$

Exemple 1: Ecrire un algorithme permettant de calculer la somme de N nombres réels saisis au clavier, puis donnez sa complexité temporelle.

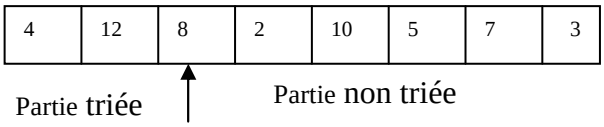
Pour chaque ligne i, on note c_i son temps d'exécution

```
ALGORITHME somme
CONST N = 10
VAR i : ENTIER
      s, x : REEL
DEBUT
 s ← 0
 POUR i ← 1, N
   DEBUT
    LIRE(x)
    s ← s + x
   FIN
 ECRIRE(s)
FIN
```

Exemple 2 : Algorithme tri par insertion

Principe

Il consiste à prendre l'élément frontière dans la partie non triée et à l'insérer à sa place dans la partie triée, puis à déplacer la frontière d'une position.



L'insertion d'un élément est effectuée par décalages ou par permutations successives.

```
PROCEDURE triInsertion(DR T : Vecteur)
VAR i, j, tmp: ENTIER
DEBUT
  POUR j ← 2, N
    DEBUT
      tmp ← T[j]
      i ← j -1

      TANTQUE(i > 0 ET T[i] > tmp)

        DEBUT

          T[i+1] ← T[i]


          i ← i – 1

        FIN
      T[i+1] ← tmp
    FIN
FIN
```

Remarques

- Pour chaque ligne **i**, on note $\mathbf{c_i}$ son temps d'exécution
- Pour chaque valeur de $\mathbf{j} \in [2, N]$, on désigne par $\mathbf{t_j}$ le nombre d'exécutions du test de la boucle TANTQUE.
- On a : $\mathbf{1 \leq t_j \leq j}$
- Chaque quantité dépend de l'état initial du tableau **T**

Complexité au meilleur : le cas le plus favorable pour l'algorithme triInserion est quand le tableau est déjà trié, Dans ce cas $t_j = 1$ pour tout j.

VI.2 Notation de Landau

Quand nous calculons la complexité d'un algorithme, nous ne calculons pas en général sa complexité exacte, mais son ordre de grandeur. Pour ce faire, nous avons besoin de notations asymptotiques.

Soient **f** et **g** deux fonctions de $\aleph$ dans $\Re$.

VI.2.1 Définition 1 : θ

On dit que **g(n)** est une borne approchée asymptotique pour **f(n)**
et l'on écrit **f(n) ∈ θ(g(n))**
s'il existe deux constantes strictement positives $\mathbf{c_1}$ et $\mathbf{c_2}$ telles que, pour **n** assez grand, on ait :

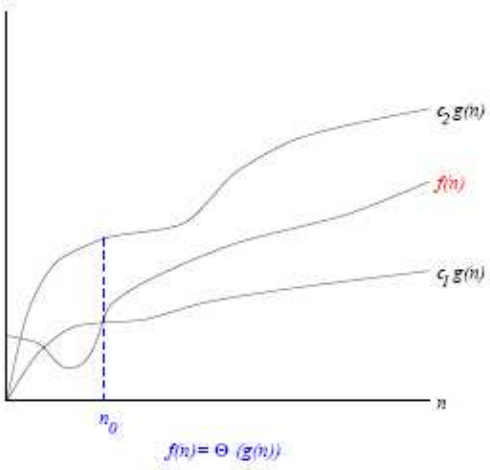
$$\mathbf{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)}$$

On dira qu'un algorithme est de complexité théorique θ(f(n)) si $\frac{f(n)}{g(n)} \to k$, c'est à dire si f(n) est équivalent à kg(n) (k étant une constante). θ signifie de l'ordre de.

Pour indiquer que **f(n) ∈ θ(g(n)),** on écrit : **f(n) = θ(g(n))**

Dans le cas du tri par insertion $T(n) = \theta(n^2)$

Illustration



Exercice

Montrer que $f(n) = \frac{1}{2}n^2 - 3n = \theta(n^2)$

On doit trouver c_1, c_2, $n_0 > 0$ telles que

$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$ pour tout $n \geq n_0$

Démonstration : on doit trouver $c_1 \leq c_2$

VI.2.2 Définition 2 : O

On dit que **g(n)** est une borne supérieure asymptotique pour **f(n)**
et l'on écrit **f(n) ∈ O(g(n))**
s'il existe une constante strictement positive **c** telle que pour **n** assez grand on ait :

0 ≤ f(n) ≤ cg(n)

Ceci revient à dire que **f(n)** est inférieure à **g(n).**à une constante près et pour **n** assez grand.
Pour indiquer que **f(n) ∈ O(g(n)),** on écrit : **f(n) = O(g(n))**

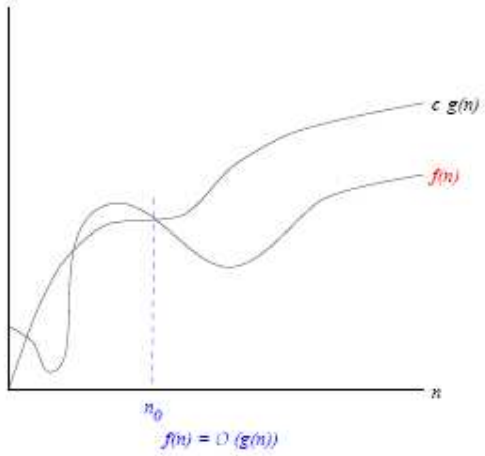
Exemples :
Monter que f(n) = n = O(n)

On doit trouver c et n_0 telles que $n \leq c\ n$
prendre n_0 =1, c=1

Monter que f(n) = 2n = O(3n)
On doit trouver c et n_0 telles que $2n \leq c\ 3n$
prendre n_0 =1, c = 2/3)

Monter que f(n) = n^2 + n -1 = O(3n)
On doit trouver c et n_0 telles que $(n^2 + n-1) \leq c\ n^2$
prendre n_0 = 1, c =1)

Illustration



VI.2.3 Définition 3 : Ω

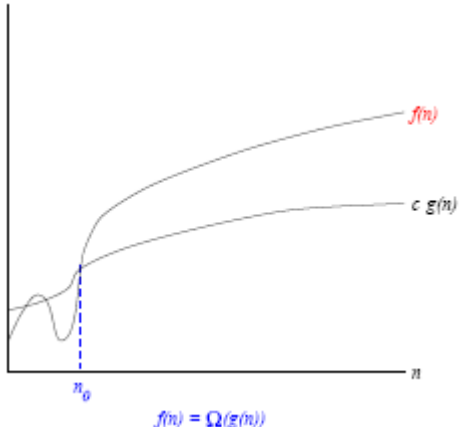
On dit que **g(n)** est une borne inférieur asymptotique pour **f(n)**
et l'on écrit **f(n) Ω(g(n))**
s'il existe une constante strictement positive **c** telle que pour **n** assez grand on ait :

$0 \leq cg(n) \leq f(n)$

Ceci revient à dire que **f(n)** est plus grande de **g(n).**à une constante près et pour **n** assez grand.

Pour indiquer que **f(n) ∈ Ω(g(n)),** on écrit : **f(n) = Ω(g(n))**

Illustration



VI.2.4 Définition 4 : o

On dit que **g(n)** est une borne supérieure non asymptotiquement approchée pour **f(n)**
et l'on écrit **$f(n) \in o(g(n))$**
si toute constante **$c > 0$** il existe une constante $\mathbf{n_0} > 0$ telle que :

$$\mathbf{0 \leq f(n) < cg(n)\ \forall n > n_0}$$

Ceci revient à dire qu'à l'infinie, la fonction **f(n)** devient négligeable par rapport à la fonction **g(n).**

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$

VI.2.5 Définition 4 : ω

On dit que **g(n)** est une borne inférieure non asymptotiquement approchée pour **f(n)**
et l'on écrit **$f(n) \in \omega(g(n))$**
si toute constante **$c > 0$** il existe une constante $\mathbf{n_0} > 0$ telle que :

$$\mathbf{0 \leq cg(n) < f(n)\ \forall n > n_0}$$

Ceci revient à dire qu'à l'infinie, la fonction **f(n)** devient arbitrairement grande par rapport à la fonction **g(n).**

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = \infty \text{ si la limite existe}$$

VI.2.6 Propriétés

- $f(n) = \theta(g(n)) \Rightarrow f(n) = O(g(n))$
- $f(n) = \theta(g(n))$ si et seulement si $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$
- Toutes les notations θ, O, Ω, o et ω sont transitives.
- Seules les notations θ, O et Ω sont réflexives.
- De plus, on a la symétrie transposée suivante :
  $f(n) \in \theta(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
  $f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$

VI.2.7 Comparaison

$T(n)$	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$
$\log\ n$	$1\,\mu s$	$1,3\,\mu s$	$1,5\,\mu s$	$1,6\,\mu s$	$1,7\,\mu s$	$1,8\,\mu$s
n	$10\,\mu s$	$20\,\mu s$	$30\,\mu s$	$40\,\mu s$	$50\,\mu s$	$60\,\mu$s
$n\ \log\ n$	$10\,\mu s$	$26\,\mu s$	$44\,\mu s$	$64\,\mu s$	$85\,\mu s$	$107\,\mu$s
n^2	$100\,\mu s$	$400\,\mu s$	$900\,\mu s$	$1,6\,ms$	$2,5\,ms$	$3,6$ ms
n^3	$1\,ms$	$8\,ms$	$27\,ms$	$64\,ms$	$125\,ms$	216 ms
n^5	$0,1\,s$	$3\,s$	$24\,s$	$1,7\,mn$	$5\,mn$	13 mn
2^n	$1\,ms$	$1\,s$	$18\,mn$	13 jours	36 ans	366 siècles
3^n	$60\,ms$	1 heure	6 ans	3900 siècles	2×10^8 siècles	$1,3 \times 10^{13}$ siècles

Remarque

- La croissance de 2^n et 3^n devient rapidement déraisonnable, on parle du phénomène d'"explosion combinatoire".

- De tels algorithmes sont inutilisables si la tille du problème devient importante.

Classe de grands θ

θ(1)		Complexité constante
θ(log n)		Complexité logarithmique
θ(n)		Complexité linéaire
θ(n log n)		Complexité quasi-linéaire
θ(n^a)	θ(n^a)	Complexité polynomiale
	θ(n^2)	Complexité quadratique
	θ(n^3)	Complexité cubique
θ(a^n)		Complexité exponentielle
θ(n!)		Complexité factorielle

θ(log n) < θ(n) < θ(n log n) < θ(n^2) < θ(n^3) < θ(a^n) <θ(n!)

Complexité	Type
$O(1)$	accéder au premier élément d'un ensemble de données
$O(\log n)$	couper un ensemble en deux puis chacun en deux, *etc.*
$O(n)$	parcourir un ensemble de n données
$O(n \log n)$	couper répétitivement un ensemble en deux et parcourir chacune des parties
$O(n^2)$	parcourir un ensemble de données une fois par élément d'un autre ensemble de même taille
$O(2^n)$	générer tous les sous-ensembles possibles d'un ensemble de données
$O(n!)$	générer toutes les permutations possibles d'un ensemble de données

VI.2.8 Recommandation pour la complexité

Il est important de connaître la complexité temporelle des algorithmes. En pratique, on distingue :

- Les bons algorithmes : les polynomiaux ;
- Les autres : qui mènent à une explosion combinatoire et qui doivent être évités.

Il faut avoir du recul par rapport à la complexité théorique quand il faut faire un choix entre un algorithme lent et simple et un algorithme plus rapide mais compliqué.

Lorsque la taille d'un algorithme est petite, un algorithme lent peut suffire.

VI.3 Complexité spatiale

C'est le coût en espace de mémoire occupé par l'exécution d'un algorithme.

Exemple de l'algorithme du calcul de la somme de N réels.

i, N : ENTIER	2 + 2 = 4 octets (un entier occupe 2 octets)
t : Vecteur	$N \times 4$ octets (un réel occupe 4 octets)
s : REEL	2 octets
	Total = $(6 + N \times 4)$ octets

Remarque:

On est le plus souvent amené à faire un compromis entre la complexité temporelle et la complexité spatiale. L'amélioration de l'une se faisant au détriment de l'autre.

Exercice 1 : PGCD

Ecrire de deux manières différente un algorithme de recherche du pcgd d'un entier N. calculer la complexité de cet algorithme.

- 1ère méthode : essai de division de N par k, k variant de N - 1 à 1
- 2ème méthode : le résultat cherché est égal à N/k si k est le plus petit diviseur > 1 de N si N n'est pas premier, $k \in [2, \sqrt{N}]$

Exercice 2 : Suite de Fibonacci

La suite de Fibonacci est définie comme suit :

$$\mathrm{Fib}(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & \text{sinon.} \end{cases}$$

1. Ecrivez un algorithme itératif calculant Fib(n).
2. Calculer sa complexité

VI.4 Calcul des complexités usuelles

VI.4.1 Fonctions exponentielles

Soit $a \neq 0$ un réel

Soient m et n des entiers, on a :

- $a^0 = 1$ $a^1 = a$ $a^{-1} = 1/a$
- $(a^m)^n = a^{mn} = a^{nm} = (a^n)^m$
- $a^m a^n = a^{m+n}$

Pour toute constante a et b telles que a > 1,

(1) $\lim_{n\to+\infty} \frac{n^b}{a^n} = 0 \Rightarrow n^b = o(a^n)$

Ainsi, une fonction exponentielle (a > 1) quelconque croit plus vite qu'une fonction polynôme strictement positive quelconque.

VI.4.2 Fonctions Polynôme

Un polynôme en n de degré est une fonction p(n) de la forme : $p(n) = \sum_{i=0}^{d} a_i n^i \quad a_d \neq 0$

Un polynôme est strictement positif asymptotiquement si et seulement si $a_d > 0$

Pour polynôme asymptotiquement positif de degré d, on $p(n) = \theta(n^d)$

On dit qu'une fonction f(n) a une borne polynomiale si $f(n) = O(n^k)$, pour une certaine constante k.

VI.4.3 Fonctions Logarithmes

Pour n > 0 et b > 1, on note $\log_b n$ le logarithme à base b de n. Cette fonction est strictement croissante.

On dit qu'une fonction a une base polylogarithmique si $f(n) = \log_b n^{O(n)}$.

En substituant **$\log_b n$** à **n** et **2^a** à **a** dans la formule(1) on obtient :

$\lim_{n\to+\infty} \frac{(\log_b n)^b}{(2^n)^{\log_b n}} = \lim_{n\to+\infty} \frac{(\log_b n)^b}{n^a} = 0 \Rightarrow (\log_b n)^b = o(n^a)$

Ainsi, une fonction polynôme strictement positive quelconque croît plus vite qu'une fonction polylogarithmique quelconque.

VI.4.4 Fonction factorielles

Pour tout $n \geq 0$

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

L'approximation de Stirling donne une borne supérieure plus approchée, ainsi qu'une borne inférieure.

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

e est la base du logarithme népérien

On peut montrer que :

- $n! = o(n^n)$
- $n! = \omega(2^n)$
- $\log(n!) = \theta(n \log n)$

Les bornes suivantes sont également valables pour tout n

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \leqslant n! \leqslant \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}$$

VI.4.5 Sommations

La somme finie $a_1 + a_2 + \ldots + a_n$ peut s'écrire $\sum_{k=1}^{n} a_k$

Si n = 0, la valeur de la somme est par convention 0.

Si la somme est infinie, on l'écrit : $\sum_{k=1}^{\infty} a_k$

a) Séries arithmétiques

$$\sum_{k=1}^{n} a_k = \frac{1}{2} n(n+1) = \theta(n^2)$$

Démonstration

$$c_1 n^2 \leq \frac{n^2}{2} + \frac{n}{2} \leq c_2 n^2 \text{ pour tout } n \geq n_0$$

$n_0 = 1$ $c_1 = \frac{1}{2}$ et $c_2 = 1$

b) Séries géométriques

$$S = \sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \quad si\ x \neq 1 = \theta(x^n)$$

c) Série emboitée

$$\sum_{k=1}^{n} a_k - a_{k-1} = a_n - a_0$$

$$\sum_{k=0}^{n-1} a_k - a_{k-1} = a_0 - a_n$$

Exemple

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = \sum_{k=1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1}\right) = 1 - \frac{1}{n}$$

Avec $a_k = \frac{1}{k}$

VI.4.6 Produit

Le produit fini $a_1 \times a_2 \times \ldots \times a_n$ peut s'écrire $\prod_{k=1}^{n} a_k$

Si n = 0, la valeur du produit est 1 par convention.

On peut convertir un produit en une somme en utilisant l'identité :

$$\log\left(\prod_{k=1}^{n} a_k\right) = \sum_{k=1}^{n} \log a_k$$

VII Algorithmes récursifs

La récursivité est une notion difficile à saisir au premier abord ; elle est illustrée aussi bien dans la vie courante qu'en informatique.

VII.1 Récursivité directe

On parle de récursivité directe d'un algorithme s'il contient un ou plusieurs appels à lui-même.

- Ce principe est parfois utilisé dans la vie courante :
  "Yao descend de Kouacou soit si Kouacou est le père (ou la mère) de Yao, soit si le père (ou la mère) de Yao descend de Kouacou".

- En mathématique de nombreuses fonctions sont définies suivant ce principe.

Exemple :

$$n! = \begin{cases} 0! = 1 \\ n! = n*(n-1)!\ \forall n > 0 \end{cases}$$

VII.1.1 Récursivité indirecte

On parle de récursivité indirecte lorsqu'un algorithme A appelle un algorithme B qui lui-même appelle A.

Exemple

PROCEDURE flop(D m : ENTIER)
DEBUT
  ECRIRE('FLOP')
  SI(m>0) ALORS
  flip(m-1)
FIN
	PROCEDURE flip(D n : ENTIER)
	DEBUT
	  ECRIRE('FLIP')
	  SI(n>0) ALORS
	  flop(n-1)
	FIN
		Exécution : flop(4)
		FLOP
		FLIP
		FLOP
		FLIP
		FLOP

VII.2 Analyse récursive

Afin d'éviter des appels infinis (bouclage) on doit prévoir une instruction telle que, dans certains cas, l'évaluation puisse se faire directement sans appel récursif. Une analyse récursive comporte trois étapes:

- **Etape 1**: Paramétrage du problème : il s'agit de déterminer les différents éléments dont dépend la solution du problème (en particulier la taille du problème).
- **Etape 2** : Recherche d'un cas trivial (test d'arrêt) : c'est l'étape clé de l'algorithme; c'est la recherche d'un cas particulier qui donne la solution directement sans appel récursif. Ce sera souvent le cas ou la taille du problème est égale à 0, ou égale à 1, etc.
- **Etape 3** : Décomposition du cas général : il s'agit de décomposer le cas général afin de le ramener à un ou plusieurs problèmes, plus simples (de tailles plus petites).

VII.3 Quelques algorithme récursifs simples

VII.3.1 Factorielle d'un nombre

Analyse du problème : soit fact la fonction factorielle

1) Paramétrage
   n: Nombre entier positif, nombre dont on veut calculer la factorielle.

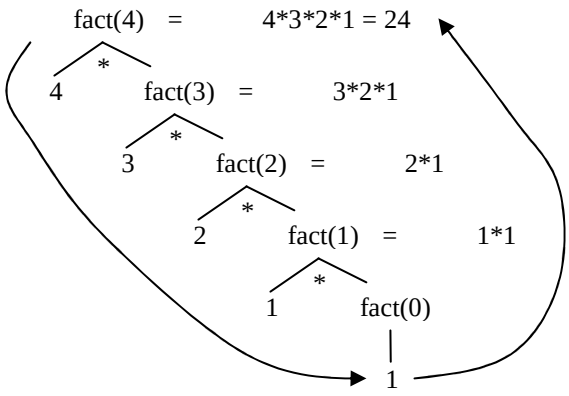
2) Recherche d'un cas trivial
   SI(n=0) ALORS fact(n) = 1

3) Décomposition
   $fact(n) = n \times fact(n-1) \quad \forall\ n > 0$

L'algorithme s'écrit alors

```
FONCTION fact(D n :ENTIER) : ENTIER
DEBUT
  SI(n=0) ALORS
    fact ← 1
  SINON
    fact ← n*fact(n-1)
  FINSI
FIN
```

Exécution: fact(4)



VII.3.2 La tour d'Hanoi

Les prêtres d'une certaine secte ont à résoudre le problème suivant : 64 disques étant posés les uns sur les autres par ordre de diamètre décroissant sur un socle A, les transférer sur un socle B, en utilisant un socle intermédiaire C.

Les seuls déplacements autorisés consistent à prendre un disque au sommet d'un socle et à le poser sur un disque plus grand ou sur un socle vide.

Selon la légende, la fin du monde coïncidera avec la fin du transfert des 64 disques.

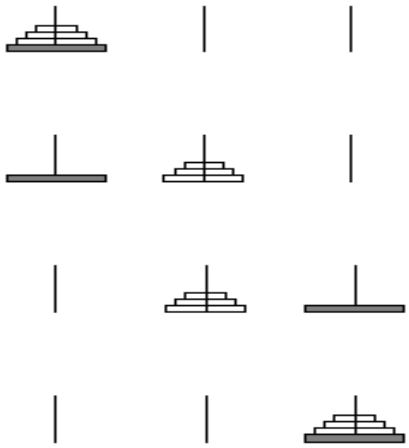
Analyse du problème
Soit hanoi le nom de la procédure récursive.
Soit deplacer la procédure de transfert d'un disque d'un socle à un autre.

1) Paramétrage
   n: Nombre de disque de type entier
   x: Socle de départ
   y: Socle d'arrivé
   z: Socle intermédiaire

2) Recherche de cas triviaux
   SI(n=0) ALORS ne rien faire
   SI(n=1) ALORS deplacer(x,y)

3) Décomposition du cas général

- Transférer récursivement (n-1) disques du sommet de x (socle de départ), sur z (socle intermédiaire), y (socle d'arrivé) sera utilisé comme socle intermédiaire.
- hanoi(n-1,x,z,y)
- Déplacer le disque qui reste en x (le plus grand) sur y.
- deplacer(x,y)
- Transférer récursivement les (n-1) disques de z en y avec x comme socle intermédiaire.
- hanoi(n-1,z,y,x)

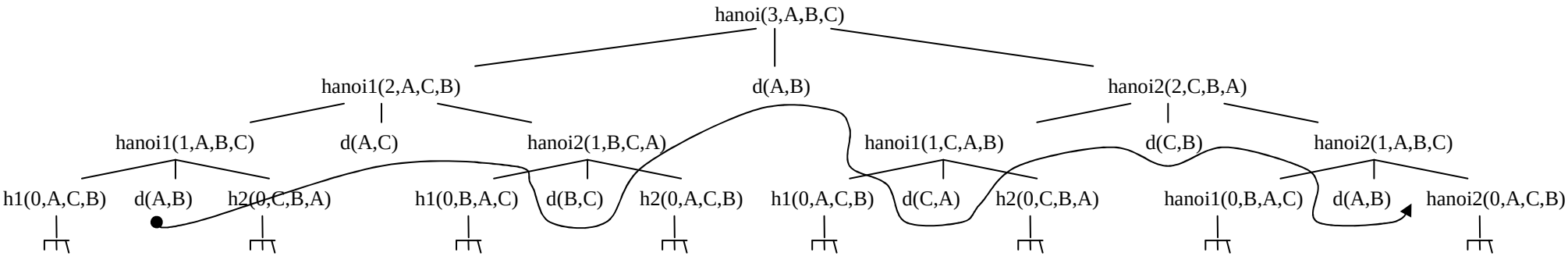


L'algorithme général s'écrit:

```
PROCEDURE hanoi(D n :ENTIER, D x, y, z : CAR)
DEBUT
  SI(n>0) ALORS
    DEBUT
      hanoi(n-1, x, z, y)
      deplacer(x, y)
      hanoi(n-1, z, y, x)
    FIN
FIN
```

```
PROCEDURE deplacer(D x, y : CAR)
DEBUT
  ECRIRE('Déplacez un disque de ',x,' à ',y)
FIN
```

Exécution : hanoi(3, A, B, C)



Solution

1. deplacer un disque de A à B
2. deplacer un disque de A à C
3. deplacer un disque de B à C
4. deplacer un disque de A à B
5. deplacer un disque de C à A
6. deplacer un disque de C à B
7. deplacer un disque de A à B

Exercice 1: Ecrire une fonction récursive permettant de rechercher de façon dichotomique une donnée dans un tableau trié par ordre croissant. Puis calculer sa complexité.

Exercice 2:
On veut réaliser des opérations d'addition et de multiplication dans un environnement qui n'offre que les 2 opérateurs inc(incrémentation) et dec(décrémentation). Ecrire sous forme récursive les algorithmes d'addition et de multiplication de 2 entiers positifs.

VII.4 Complexité des algorithmes récursifs

La complexité d'un algorithme récursif se calcule par une récurrence. A chaque algorithme récursif, on associe une complexité inconnue T(n), n étant la taille de l'algorithme.

Une récurrence est une équation ou une inégalité qui décrit une fonction à partir de sa valeur sur des entrées plus petites.

On va proposer des méthodes de résolution des équations de récurrence : Obtenir des bornes asymptotiques θ ou O pour la solution.

VII.4.1 Complexité de fact(n)

```
FONCTION fact(D n : ENTIER):ENTIER
DEBUT
  SI(n=0) ALORS                    (1)
    fact ← 1                       (2)
  SINON
    fact ← n*fact(n-1)             (3)
FIN
```

Soit T(n) la complexité de fact(n)
Les instructions (1) et (2) sont de complexité θ(1)
L'instruction (3) est de complexité θ(1) + T(n-1)
(θ(1) pour ← et *, T(n-1) pour fact(n-1))

donc $T(n) = \begin{cases} c \ si \ n = 0 \\ d + T(n-1) \ si \ n > 0 \end{cases}$

c et d sont des constantes qui traduisent une complexité θ(1)
Supposons n > 0

$$\left.\begin{array}{l} T(n) = d + T(n-1) \\ T(n-1) = d + T(n-2) \end{array}\right\} \Rightarrow T(n) = 2d + T(n-2)$$

$T(n\text{-}2) = d + T(n\text{-}3) \Rightarrow T(n) = 3d + T(n\text{-}3)$
.....
$T(n) = i \times d + T(n\text{-}i)$
si on remplace i par n on aura
$T(n) = n \times d + T(0) = n \times d + c = \theta(n)$

VII.4.2 Complexité de hanoi

```
PROCEDURE hanoi(D n : ENTIER, D x, y, z : CAR)
DEBUT
  SI(n>0) ALORS                    (1)
    DEBUT
      hanoi(n-1, x, z, y)          (2)
      deplacer(x, y)               (3)
      hanoi(n-1, z, y, x)          (4)
    FIN
FIN
```

Les instructions (1) et (3) sont de complexité θ(1)

Les instructions (2) et (4) sont de complexité T(n-1)

donc $T(n) = \begin{cases} 1 \; si \; n = 1 \\ 1 + 2T(n-1) \; si \; n > 1 \end{cases}$

Supposons n > 1

$T(2) = 1 + 2T(1) = 1 + 2 = 3 = 2^2 - 1$
$T(3) = 1 + 2T(2) = 1 + 2(2^2 - 1) = 2^3 - 1$
$T(4) = 1 + 2T(3) = 1 + 2(2^3 - 1) = 2^4 - 1$

Par généralisation on trouve comme forme récurrente

$T(n) = 2^n - 1 = \theta(2^n)$

$T(10) = 2^{10} - 1 = 1\,023$
$T(20) = 2^{20} - 1 = 1\,048\,575$
$T(32) = 2^{32} - 1 = 4\,294\,967\,295$
$T(64) = 264 - 1 = 18\,446\,744\,073\,709\,551\,615 \approx 0{,}2 \times 10^{20}$ unité de temps

Prenons comme unité de temps la seconde
1 an = 31536000 s
1 siècle $\approx 0{,}3 \times 10^{10}$ s
Pour T(64) il faudrait environ 10^{10} siècles, ce qui n'est pas loin de la fin du monde.

VII.5 Efficacité de la récursivité ?

La récursivité est légèrement moins rapide qu'un algorithme itératif équivalent (temps nécessaire à l'empilage et au dépilage des données).

La récursivité utilise plus de ressources mémoire pour empiler les contextes. Cependant la récursivité est plus « élégante » et les algorithmes récursifs sont souvent plus faciles à écrire.

Exercice 1: Recherche dichotomique
Ecrire une fonction récursive permettant de rechercher de façon dichotomique une donnée dans un tableau trié par ordre croissant. Puis calculer sa complexité.

Exercice 2 : Addition et multiplication
On veut réaliser des opérations d'addition et de multiplication dans un environnement qui n'offre que les 2 opérateurs inc(incrémentation) et dec(décrémentation). Ecrire sous forme récursive les algorithmes d'addition et de multiplication de 2 entiers positifs. Puis calculer leur complexité.

Exercice 3 : Suite de Fibonacci
La suite de Fibonacci est définie comme suit :

$$\mathrm{Fib}(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & \text{sinon.} \end{cases}$$

1. Ecrivez un algorithme récursif calculant Fib(n).
2. Montrez que la complexité (en nombre d'additions) de cet algorithme est en $\theta(2^{\frac{n}{2}})$

3. Ecrire un algorithme récursif qui calcule, pour n > 0, le couple (Fibonacci(n), Fibonacci(n - 1)).
4. Utilisez l'algorithme précédent pour écrire un nouvel algorithme calculant fibonacci(n). Qu'elle est la complexité (en nombre d'additions) de cet algorithme ?

VII.6 Terminaison et correction des algorithmes récursifs

VII.6.1 Méthode 1

Elle est basée sur le calcul du nombre d'appels de la fonction engendrée par un appel initial avec ses paramètres effectifs donné et on montre ensuite que ce nombre est fini.

```
FONCTION fact(D n : ENTIER): ENTIER
DEBUT
  SI(n=0) ALORS
    fact ← 1
  SINON
    fact ← n*fact(n-1)
FIN
```

Notons NbAp(x) le nombre d'appel de fact(x)

NbAp(0) = 0
NbAp(x) = 1 + NbAp(x-1) = x

Si x est fini alors fact(x) est fini est NbAp(x) est fini est

VII.6.2 Méthode 2

Elle est basée sur la récurrence du domaine de définition c'est-à-dire qu'on suppose que les appels récursifs engendrés par l'appel initial sont finis pour montrer que l'algorithme lui-même est fini.

fact(0) = 1 est fini
supposons que fact(n-1) soit fini, alors fact ← n*fact(n-1) est fini (multiplication de deux nombres).

Conclusion : fact(n) est fini ∀n entier.

VII.6.3 Correction des algorithmes récursifs

Pour montrer la correction des algorithmes récursifs, on utilise un raisonnement par récurrence qui est basé sur la récurrence qui a permis de construire l'algorithme.

Si n = 0 fact(0) = 1 = 0! (par convention)
Supposons que fact(n) = n!
fact(n+1) = (n+1)*fact(n)
= (n+1)*n!
= (n+1)!

Conclusion : fact(n) = n! ∀n entier

VII.7 Réalisation de la récursivité

Du fait qu'un sous-algorithme récursif s'appelle lui-même, on recommence l'exécution du sous-algorithme avec de nouvelles données avant d'avoir fini l'exécution avec les données de l'appel précédent.

Tous ces appels nécessitent la sauvegarde des résultats intermédiaires et des adresses de retour. Ce problème est résolu en utilisant une pile ; à chaque entrée dans le sous-algorithme on empile les variables et les adresses de retour à sauvegarder, à chaque sortie du sous-algorithme on dépile.

VII.8 Transformation des boucles en algorithmes récursifs

VII.8.1 Transformer une boucle en une procédure récursive

Procedure iterative :	Procedure recursive équivalente

PROCEDURE compterI
CONST N = 10
VAR i: ENTIER
DEBUT
  POUR i ← 1, N-1
  ECRIRE(i, ' ')
FIN
	PROCEDURE compterR
	CONST N = 10
	VAR i: ENTIER
	DEBUT
	  ECRIRE(i, ' ')
	  SI(i < N) ALORS
	  comterR(i+1)
	FIN

VII.8.2 **Transformer deux boucles imbriquées en une procédure récursive**

Procedure itérative :	1ère procédure récursive:	2ème procédure récursive:

PROCEDURE compterI
CONST N = 10, M = 5
VAR i, j: ENTIER
DEBUT
  POUR i ← 1, N-1
  POUR j ← 1, N-1
  ECRIRE(i * j, ' ')
FIN
	PROCEDURE compterR1(D i: ENTIER)
	CONST N = 10, M = 5
	VAR j: ENTIER
	DEBUT
	  SI(i < N) ALORS
	  DEBUT
	  POUR j ← 1, N-1
	  ECRIRE(i * j, ' ')
	  comterR(i+1)
	  FIN
	FIN
		PROCEDURE compterR2(D i, j : ENTIER)
		DEBUT
		  SI(i < N) ALORS
		  SI (j < M) ALORS
		  DEBUT
		  ECRIRE(a*b, ' ');
		  compterR2(i ,j+1)
		  FIN
		  SINON
		  compterR2(i+1,1)
		FIN

VII.9 Transformation des algorithmes récursifs en algorithmes itératifs

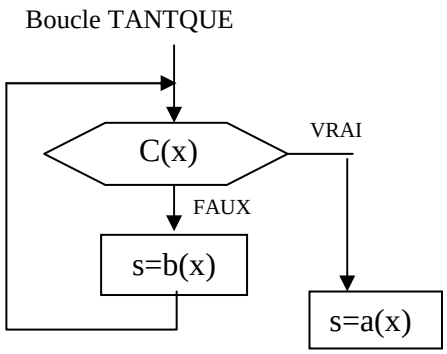
Certains langages de programmation ne permettent pas la réalisation automatique de la récursivité car leur mécanisme d'appel et de retour de sous-programmes n'utilisent pas une pile mais une zone statique de la mémoire (ex. FORTRAN et BASIC, ASSEMBLEUR). D'ou la nécessité de méthodes de transformation d'algorithmes récursifs en algorithmes itératifs.

De plus un algorithme récursif nécessite parfois un temps de calcul très long et beaucoup d'espace mémoire.

VII.9.1 Récursivité terminale

Dans la récursivité terminale, on a un seul appel récursif qui n'est suivi d'aucune autre instruction.

```
PROCEDURE f(D x : T1; R s : T2)
DEBUT
  SI(c(x)) ALORS
    s ← a(x)
  SINON
    f(b(x))
FIN
```



La récursivité terminale correspond à une boucle TANTQUE. On traduit donc f(x, s) comme suit :

```
PROCEDURE f(D x : T1; R s : T2)
DEBUT
  TANTQUE(NON (c(x))
    s ← b(x)
  s ← a(x)
FIN
```

Exemple : PGCD de 2 entiers.

```
PROCEDURE pgcd(D a, b :ENTEIR ; R p : ENTIER)
DEBUT
  SI( a=b ) ALORS
    p ← a
  SINON
    SI(a > b) ALORS
      a ← a - b
    SINON
      b ← b - a
  pgcd(a, b, p)
FIN
```

```
PROCEDURE pgcd(D a, b :ENTEIR ; R p : ENTIER)
DEBUT
  TANTQUE( a ≠ b )
    DEBUT
      SI(a > b) ALORS
        a ← a - b
      SINON
        b ← b - a
    FIN
  p ← a
FIN
```

Exercice 1 : Transformer le sous-algorithme d'addition récursif réalisé à l'aide des seules instructions inc et dec en un algorithme itératif.

```
FONCTION add(D a, b  : ENTIER): ENTIER
DEBUT
  SI(b=1) ALORS
    add ← inc(a)
  SINON
    add ← add(inc(a), dec(b))
FIN
```

```
FONCTION add(D a, b : ENTIER): ENTIER
DEBUT
  TANTQUE(b ≠ 1)
    DEBUT
      a ← inc(a)
      b ← dec(b)
    FIN
  add ← a
FIN
```

Exercice 2

Ecrire un sous algorithme récursif permettant de calculer $\sum_{i=1}^{N} i$; transformer ce sous-algorithme en un algorithme itératif.

```
FONCTION som(D n : ENTIER): ENTIER
DEBUT
  SI(n=1) ALORS
    som
  SINON
    som ← n + som(n-1)
FIN
```

```
FONCTION som(D n : ENTIER): ENTIER
VAR tmp : ENTIER
DEBUT
  tmp ← 0
    TANTQUE
  DEBUT
    tmp ← tmp + n
    n ← n – 1
  FIN
  som ← tmp
FIN
```

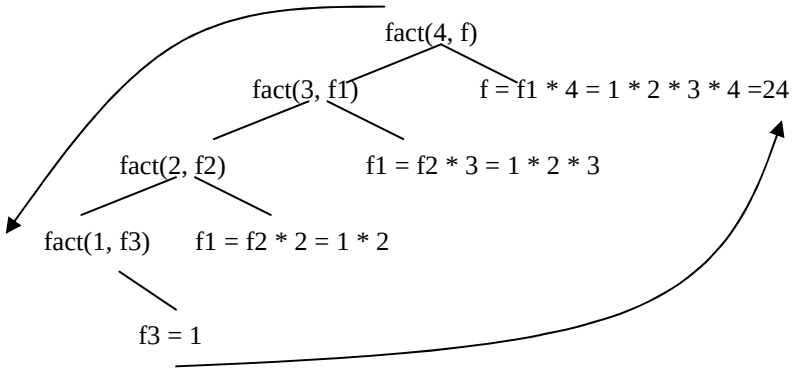
VII.9.2 Récursivité non terminale

Dans la récursivité non terminale, on a un seul appel récursif qui est suivi par une autre instruction. On se ramène à une récursivité terminale

```
PROCEDURE f(D x : T1; R s : T2)
DEBUT
  SI(c(x)) ALORS
    s ← a(x)
  SINON
    DEBUT
      a1(x)
      f(b(x))
      a2(x)
    FIN
FIN
```

Exemple

```
PROCEDURE fact(D n : ENTIER ; f : ENTIER)
DEBUT
  SI((n = 0) OU (n = 1)) ALORS
    f ← 1
  SINON
    DEBUT
      n1 ← n -1
      fact(n1, f1)
      f ← f1 * n
    FIN
FIN
```



L'arbre d'appel se présente comme suit

Pour transformer cet algorithme en un algorithme itératif, on introduit une fonction g qui nous ramène à une récursivité terminale.

$$g(n, p) = \begin{cases} g(1, 1) & \text{si } n = 1 \\ g(n-1, p*n) & \text{si } n > 1 \end{cases}$$

```
PROCEDURE gfact(D n, p : ENTIER)
DEBUT
  SI(n=1) ALORS
  p ← 1
  SINON
  gnfact(n -1, p * n)
FIN
```

```
FONCTION gnfact(D n, p : ENTIER)
DEBUT
  TANTQUE(n ≠ 1) ALORS
  DEBUT
    p ← p * n
    n ← n - 1
  FIN
FIN
```

VII.9.3 Dérécursivité dans le cas général

1) Forme générale des algorithme comportant deux appels récursif

```
SAR(x :T)
DEBUT
  a(x)
  SI(p1(x)) ALORS
    SAR(t1(x))
  b(x)
  SI(p2(x)) ALORS
    SAR(t2(p))
  c(x)
FIN
```

avec

- x : ensemble de paramètres
- a(x), b(x), c(x) : fonction de x qui ne modifient pas la valeur de x
- p1(x), p2'x) : des prédicats
- t1(x), t2(x) : fonctions des paramètres admissibles qui à x → x' de même type que x

2) Arbre binaire associé à un appel récursif

L'arbre binaire est constitué de la façon suivante :

La racine est formée de l'appel initial

Pour tout nœud x, si p1(x) est vrai, alors ce nœud possède un sous-arbre gauche constitué de l'ensemble des appels engendrés par SAR(t1(x))

Si p2(x) est vrai, alors ce nœud possède un sous-arbre droit constitué de l'ensemble des appels engendrés par SAR(t2(x))

```
SAI(x : T)
VAR u : T
      NP : CAR
DEBUT
  u←x
  NP ← 'p'
  REPETER
    CHOISIR (NP) PARMI
      'p' : DEBUT
              a(u)
              SI(p1(u)) ALORS
                DEBUT
                  b(u)
                  SI(p2(u)) ALORS
                  c(u)
                FIN
            FIN

      'g' : DEBUT
              b(u)
              SI(p2(u) ALORS
                c(u)
            FIN
    FIN

      'd' : c(u)

    CHOISIR (NP) PARMI
      'p' : DEBUT
              SI(p1(u)) ALORS
                u ← t1(u)
              SINON SI(p2(u)) ALORS
                u ← t2(u)
              SINON
                Remonter au père
            FIN

      g' : DEBUT
             SI(p2(u) ALORS
               u ← t2(u)
             SINON


           FIN

      'd' : Remonter au père
    FIN

  JUSQUA(u = fictif)
FIN
```

Exemple avec deux appels récursifs : Tour de Hanoï

```
PROCEDURE hanoi(D n, x, y, z : ENTIER)
DEBUT
  SI(n>0) ALORS
  DEBUT
    hanoi(n-1, x, z, y)
    deplacer(x, y)
    hanoi(n-1, z, y, x)
  FIN
FIN
```

```
PROCEDURE hanoi(x)
DEBUT
  SI(p1(x)) ALORS
    DEBUT
      hanoi(t1(x))
      b(x)
      hanoi(t2(x))
    FIN
FIN
```

a(x) et c(x) n'existent pas
n > 0, p1 = p2

```
hanoi(D n, x, y, z : ENTIER)
VAR tmp : ENTIER
      NP : CAR
DEBUT
  NP ← 'p'
  REPETER
    SI(NP = 'g') ALORS
      Deplacer(x, y)
    CHOISIR (NP) PARMI
      'p' : SI(n ≥ 1) ALORS
              DEBUT
                n ← n -1
                tmp ← z
                z ← y
                y ← tmp
                NP ← 'g
              FIN
            SINON
              NP ← 'p

      'g' : SI(n ≥ 1) ALORS
              DEBUT
                n ← n -1
                tmp ← z
                z ← x
                x ← tmp
                NP ← 'p
              FIN
            SINON
              NP ← 'p'
      'd' : NP ← 'p'
    FIN
  JUSQUA(n = 0)
FIN
```

VII.10 Récursivité est efficacité

L'utilisation de la récursivité permet de mettre en œuvre des méthodes puissantes et élégantes. Cependant, le problème de l'efficacité doit être examiné précisément.

En effet, au niveau de la réalisation des mécanismes de la récursivité, on se rend compte que l'on consacre beaucoup de temps pour empiler et dépiler les résultats intermédiaires.

C’est la raison pour laquelle on utilise peu la récursivité pour traiter les problèmes très sensibles au temps comme le contrôle des processus.

Pratiquement, on réalise une analyse récursive pour trouver une méthode de résolution, puis on élimine autant que possible les appels récursifs.

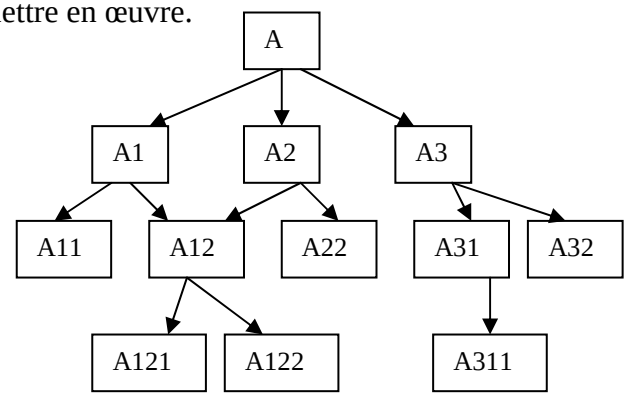
VIII Méthode d'analyse des algorithmes

Entre un problème à résoudre et un programme, l'intermédiaire indispensable est un algorithme. La phase la plus difficile de la programmation n'est pas la traduction de l'algorithme dans un quelconque langage, mais plutôt la conception de l'algorithme. On désigne cette étape sous le nom d'analyse.

Il s'agit de décomposer le problème complexe initial en des sous-problèmes dont le niveau de complexité est moins élevé ("diviser pour régner"). On distingue deux méthodes d'analyse : analyse descendante et analyse ascendante.

VIII.1 Analyse descendante

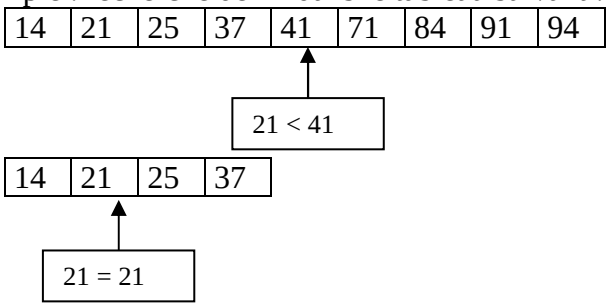
Cette méthode consiste à partir du problème initial, à descendre étape par étape vers des combinaisons de sous-problèmes de plus en plus simple à résoudre. On procède en quelque sorte par affinements successifs ; c'est la méthode la plus utilisée en programmation car naturelle et facile à mettre en œuvre.



Exemple : « Trouver un élément dans un tableau non trié par la recherche dichotomique.
La recherche dichotomique accélère la recherche d'un élément dans un tableau trié.

Principe
On compare l'élément cherché à l'élément qui se trouve au milieu du tableau ; s'il y a égalité, la recherche s'arrête ; sinon on poursuit la recherche dans la moitié inférieure ou dans sa moitié supérieure selon le résultat de la comparaison.

Exemple : recherche de 21 dans le tableau suivant :



L'algorithme se présente comme suit

```
ALGORITHME Recherche
CONST N = 10
TYPE
  Vecteur = TABLEAU[1..N] DE REEL
VAR x : REEL
      indice : ENTIER
      t : Vecteur
```

```
DEBUT
| lireVect(t)       (* Saisie des éléments du tableau t*)
| ECRIRE('Entrez l''élément cherché : ')
| LIRE(x)
| triBulles(t)      (* Tri du tableau t *)
| indice ← dicho(t, x) (* Appel de la fonction dicho *)
| SI(indice > 0) ALORS
|   ECRIRE('Elément ', x, ' a été trouvé ' , 'son indice est ', indice)
| SINON
|   ECRIRE('Elément ', x, ' n''a pas été trouvé ')
FIN
```

On détaille ci-dessous les différents sous-algorithmes.

1) Procédure de saisie des éléments d'un tableau

```
PROCEDURE lireVect(DR t : Vecteur)
VAR i :ENTIER
DEBUT
| POUR i ←1, N
|   DEBUT
|   | ECRIRE('Enter t[',i,'] = ')
|   | LIRE(t[i])
|   FIN
FIN
```

2) Procédure de tri des éléments d'un tableau par l'algorithme du tri bulles

Il s'agit de ranger les éléments du tableau dans un ordre croissant (ou décroissant)

```
PROCEDURE triBulles(DR t : Vecteur)
VAR i, j : ENTIER
      fini : BOOLEEN
DEBUT
| i ← 1
| fini ← FAUX
| TANTQUE((i < N) ET (NON fini))
|   DEBUT
|   |fini ← VRAI
|   |POUR j ← i + 1, N
|   |  SI(t[i] > t[j]) ALORS
|   |    DEBUT
|   |      echange(t[i], t[j])
|   |      fini ← FAUX
|   |    FIN
|   |i ←  i + 1
|   FIN
FIN
```

Principe

On compare chaque paire d'élément consécutif ; si ces éléments sont dans l'ordre, on passe à la paire suivante, sinon on procède à leur permutation avant de traiter la paire suivante.

On parle de tri bulles par analogie au fait que les éléments les plus petits viennent au début de la liste comme les bulles qui remontent à la surface de l'eau.

3) Procédure de permutation de deux valeurs

```
PROCEDURE echange(DR x, y : REEL)
VAR temp : REEL
DEBUT
  temp ← x
  x ← y
  y ← temp
FIN
```

4) Fonction de recherche dichotomique

```
FONCTION dicho(D t : Vecteur ;  D x : REEL) : ENTIER
VAR i, j, m : ENTIER
DEBUT
  i ← 1
  j ← N
  REPETER
    m ← (i + j) DIV 2 (* Calcul de l'indice du milieu approximatif du tableau t *)
    SI(x > t[m]) ALORS
      i ← m + 1    (* Recherche dans la moitié supérieure du tableau t *)
    SINON
      j ← m - 1    (* Recherche dans la moitié inférieure du tableau t *)
  JUSQUA((t(m) = x) OU i > j)
  SI(t[m] = x) ALORS
    dicho ← m
  SINON
    dicho ← -1
FIN
```

VIII.2 Analyse ascendante

L'analyse ascendante consiste, alors qu'on dispose d'outils (sous-algorithmes) réalisant des travaux élémentaires, à les composer pour résoudre un problème de plus haut niveau. Cette méthode est adopte une démarche bidirectionnelle :

- de haut en bas pour découper le travail en tâches plus élémentaires
- de bas en haut pour réaliser les tâches élémentaire à l'aide des outils disponibles.

VIII.3 Critique des deux méthodes

Analyse descendante

1) Avantages
   - L'approche est systématique du général au particulier ;
   - L'aspect hiérarchique de l'analyse facilite la compréhension.

2) Inconvénients
   - Elle effectue des choix conceptuels de très haut niveau qui peuvent être délicats à réaliser au niveau inférieur ;
   - On peut être amené à reprendre l'application à tous les niveaux à la suite d'une remise en cause d'un des choix initiaux (ex. structure de données dynamique et non un tableau) ;
   - Il y a un risque de redondance de certains traitements (sous-algorithmes identiques qui apparaissent dans plusieurs niveaux ;
   - On doit attendre d'écrire tous les sous-algorithmes avant de tester tout l'algorithme.

VIII.3.1 Analyse ascendante

3) Avantages
   - La réutilisation des sous-algorithmes existants ce qui accroît la vitesse de développement, réduit la taille de l'algorithme et facilite les tests ;
   - En choisissant des outils validés, on est sûr de leur opérationnalité finale (moins de tests).
   - facilité de prototypage (construction rapide d'application à partir d'outils existants même si ces outils sont trop généraux ou de performance faible)

4) Inconvénients
   - Elle ne relève pas d'une démarche systématique dans le processus d'analyse du problème ;
   - On note une dérive possible de la résolution vers celle d'un autre problème.

Conclusion : Dans la pratique, on fait un compromis entre ces deux méthodes.

Exercice : On veut produire un algorithme de résolution d'un système linéaire régulier de N équations, de la forme : $[A][X]=[B]$

On peut résoudre ce système en deux étapes :
   - d'abord on le transforme en système triangulaire par élimination progressive des variables,
   - puis on résoud le système triangulaire.

```
ALGORITHME Systeme_Lineaire
CONST N = 3
TYPE Vecteur = TABLEAU[1..N] DE REEL
       Matrice = TABLEAU[1..N, 1..N] DE REEL
VAR B, X : Vecteur
      A : Matrice
DEBUT
| Triangulation(A, B)
| Resolution(A, B, X)
FIN
```

1) Développement de l'algorithme trinagulation

On élimine la 1ère variable dans les N-1 dernière équations, on élimine la 2ème variable dans les N-2 dernières, et ainsi de suite jusqu'à la (N-1)eme variable dans la dernière équation.

```
PROCEDURE triangulation(DR A : Matrice ; D B : Vecteur)
VAR k : ENTIER
DEBUT
| POUR k ← 1, N - 1
|   elimination(k, A, B)
FIN
```

2) Développement de l'algorithme elimination

Pour simplifier les algorithmes on ne considère pas les cas de pivots nuls (A[i,i].≠ 0)

```
PROCEDURE elimination(DR A : Matrice ; DR B : Vecteur)
```

```
VAR i, j, k : ENTIER
      r : REEL
DEBUT
  POUR i ← k +1, N
    DEBUT
      r ← A[i, k]/A[k, k]
      POUR j ← k +1, N
        A[i, j] ← A[i, j] r * A[k, j]
      B[i] ←B[i] - r * B[k]
    FIN
FIN
```

3) Développement de l'algorithme résolution

Pour résoudre le système triangulaire, on part de la N^{eme} équation, on tire X_N , on reporte X_N dans la $(N-1)^{eme}$ équation, on en tire X_{N-1}, etc. plus généralement on a :

$$X_i = \left(\frac{\sum_{j=i+1}^{N} A_{ij} X_j}{A_{ii}}\right) \quad \text{avec} \quad i \text{ allant de } N \text{ à } 1$$

```
PROCEDURE resolution(DR A : Matrice ; DR B, X : Vecteur ; )
VAR i, j, k : ENTIER
      r : REEL
DEBUT
  X[N] ← B[N]/A[N,N]
  POUR i ← N-1, 1, -1
    DEBUT
      r ← 0
      POUR j ← i +1, N
        r ← r + A[i, j] * X[j]
      X[i] ←(B[i] - r) / A[i,i]
    FIN
FIN
```

IX Diviser pour régner

La méthode "Diviser pour régner" ou "Diviser pour résoudre" ou "Divide and conquer" est une méthode qui permet de résoudre un problème de taille n en le divisant en sous-problèmes plus petits de manière à ce que la solution de chaque sous-problème facilite la construction du problème entier. C'est une méthode descendante.

Chaque sous-problème est résolu en appliquant le même principe de décomposition (récursivement). Ce même procédé se répète jusqu'à obtenir des sous-problèmes suffisamment simples pour être résolus de manière triviale.

IX.1 Questions/Réponses

Q : Quand utiliser La méthode "Diviser pour régner" ?
R : Elle s'applique lorsqu'il est possible de diviser le problème en sous-problèmes indépendants, et d'en combiner les résultats de manière suffisamment efficace.

Q : Comment choisir les sous-problèmes ?
R : Il est intuitif qu'il vaut mieux choisir les sous-problèmes de taille sensiblement égale. Par exemple :

- si N est pair on peut prendre deux sous-problèmes de taille N/2
- si N est impair on peut prendre deux sous-problèmes de taille semblable soit (N-1)/2 et (N+1)/2 (c'est en général le plus efficace).

Q : Comment résoudre les sous-problèmes ?
R : Il suffit d'appliquer de manière récursive le principe de la méthode "diviser pour régner" jusqu'à ce que le sous-problème puisse être résolu de manière simple.

Les algorithmes récursifs utilisent naturellement cette technique : ils s'appellent eux-mêmes une ou plusieurs fois sur une partition du problème initial et combinent les solutions pour retrouver une solution au problème initial.

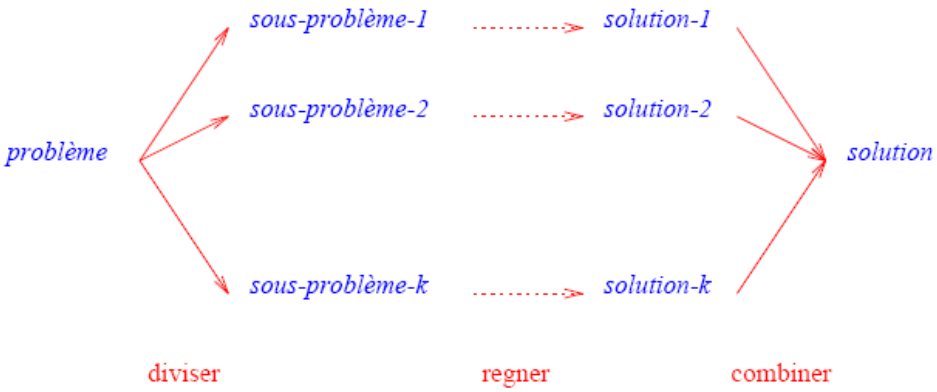
IX.2 Principe

```
FONCTION DPR(x : T) : T (* x est le problème à résoudre*)
DEBUT
| SI(x est suffisamment petit ou simple) ALORS
|   DPR ← A(x)  (* A(x) un algo connu (peut être trivial) pour résoudre x*)
| SINON
|   DEBUT
|   | Décomposer x en sous exemplaires x1, x2...xk
|   | POUR i ← 1, k
|   |   yi ← DPR(xi)  (*résoudre chaque sous-problème récursivement *)
|   | Combiner les yi pour former une solution y à x
|   | DPR ← y
|   FIN
FIN
```

Le paradigme "diviser pour régner" donne lieu à trois étapes à chaque niveau de récursivité :

- Diviser : le problème en un certain nombre de sous-problèmes ;

- Régner : sur les sous-problèmes en les résolvant récursivement ou, si la taille d'un sous-problème est assez réduite, le résoudre directement (cas trivial) ;
- Combiner : les solutions des sous-problèmes en une solution complète du problème initial.



Le schéma général des algorithmes de type "diviser pour résoudre" est le suivant :

Pour pouvoir appliquer cette méthode, on doit :
- trouver une décomposition du problème en sous-problèmes plus simples
- avoir un algorithme A(x) capable de résoudre des problèmes simples
- savoir comment combiner les solutions intermédiaires pour former la solution globale.

IX.3 Tri par fusion

IX.3.1 Principe

Une technique "diviser pour résoudre" peut être la suivante :
Pour trier une liste de n éléments, il faut trier deux sous listes de n/2 éléments chacune (les 2 moitiés de L) puis de faire l'interclassement (ou fusion) entre ces deux listes.

De la même façon, pour trier une liste de n/2 éléments il faut trier deux listes de n/4 éléments chacune puis de faire leur interclassement.

Et ainsi de suite jusqu'à l'obtention de listes d'un seul élément (dont le tri est trivial).

On peut résumer le principe comme suit :
- Diviser : diviser la liste de n éléments à trier en 2 sous-listes de n/2 éléments chacune ;
- Régner : trier les 2 sous-listes à l'aide du tri fusion (récursivité) ;
- Combiner : fusionner les 2 sous-listes triées pour produire la liste triée.

Exemple : Fusions de

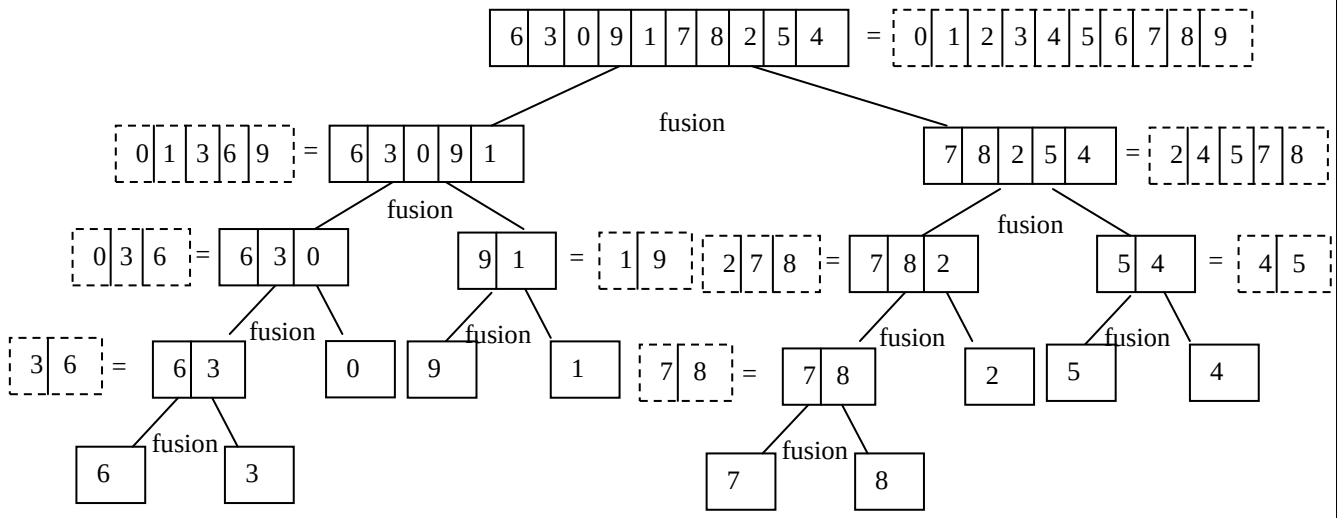
2	4	5	6

et

1	2	3	6

$\Rightarrow$

1	2	3	4	5	6	6



L'algorithme proposé ici est récursif. En effet, les deux sous tableaux seront eux même triés à l'aide de l'algorithme de tri fusion. Un tableau ne comportant qu'un seul élément sera considéré comme trié : c'est la condition sine qua non sans laquelle l'algorithme n'aurait pas de conditions d'arrêt.

Algorithme à l'aide d'un tableau peut s'énoncer de manière récursive comme suit:

```
CONST N = 10
TYPE Vecteur = TABLEAU[1..N] DE T

PROCEDURE triFusion(DR T : Vecteur,  premier, dernier : ENIER)
VAR milieu : ENTIER
DEBUT
  SI( premier ≠ dernier) ALORS (*si l'indice du 1er elt du dernier élt à traiter est différent, condition d'arrêt*)
    DEBUT
      milieu ← (premier+dernier)/2
      triFusion(T, premier, milieu)  (*tri de la 1ere moitiée du sous tableau*)
      triFusion(T, milieu+1, dernier) (*tri de la seconde moitiée du sous tableau *)
      fusion(T, premier, milieu, dernier)  (*fusion des deux sous moitiées *)
    FIN
FIN
```

```
PROCEDURE fusion(DR T : Vecteur, D premier1, dernier1, dernier2 : ENTIER)
VAR  Tbis : Vecteur
        premier2, compteur1, compteur2, i : ENTIER
DEBUT
  premier2 ← dernier1+1
  compteur1← premier1
  compteur2 ← premier2
   (*on recopie dans T les éléments du premier sous tableau*)
  POUR i ← premier1, dernier1
    Tbis[i-premier1] ← T[i]
  (*on fusionne ensuite les deux sous tableaux*)
  POUR i← premier1, dernier2
    SI(compteur1 = premier2) ALORS (*tous les éls du 1er sous tableau ont été utilisés *)
      ARRET(* tous les éléments sont donc classés *)
    SINON SI(compteur2 = (dernier2+1)) ALORS (*ts ls élts du second sous tab ont été utilisés*)
      DEBUT
        T[i] ← Tbis[compteur1-premier1] (*on recopie à la fin du tab les élts du 1er sous tab*)
        compteur1← compteur1+1
      FIN
```

```
    SINON SI(Tbis[compteur1-premier1] < T[compteur2]) ALORS (*l'élt du 1er sous tableau est le plus petit*)
      DEBUT
        T[i] ←Tbis[compteur1-premier1]  (*on ajoute un élt du premier sous tableau*)
        compteur1 ← compteur1+1  (*on progresse dans le premier sous tableau *)
      FIN
    SINON (*c'est l'élément du second sous tableau qui est le plus petit*)
      DEBUT
        T[i] ← T[compteur2]  (*on recopie cette élément à la suite du tableau*)
        compteur2 ← compteur2+1 (on progresse dans le second tableau*)
      FIN
FIN
```

```
ALGORIHME triFusionT()
VAR T : Vecteur
DEBUT
  lireVect(T)
  afficherVect(T)
  triFusion(T, 1, N)
  afficherVect(T)
FIN
```

IX.3.2 Complexité : présentation

Le tri fusion comporte 3 étapes :

- la division de l'ensemble d'éléments en deux parties,
- le tri des deux sous-ensembles
- puis la fusion des deux sous-ensembles triés.

Coût de la fusion
fusion(T, premier, milieu, dernier)
T[premier..milieu] : trié
T[milieu+1..dernier] : trié
⇒ T[premier..dernier] : trié

La complexité de la procédure fusion est en **θ(n)**
où **n = dernier – premier + 1** est le nombre d'élément à fusionner.
Temps d'exécution

- Diviser : θ(1), calcul de (premier + milieu)/2
- Régner : 2 T($\frac{n}{2}$)
- Combiner : θ(n)

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Théorème (Résolution des récurrences « diviser pour régner »).

Les résultats suivants seront admis sans preuve.

Théorème 1:

$$\begin{cases} T(1) = c, \\ n \geq 2, T(n) = aT(n/b) + cn \end{cases}$$

avec a, b, c > 0 et

a < b $\Rightarrow$ T(n) $\in$ θ(n)
a = b $\Rightarrow$ T(n) $\in$ θ(n log n)
a > b $\Rightarrow$ T(n) $\in$ θ(n $\log_b$ a)

Théorème 2 Les équations de récurrence :

$$\begin{cases} T(1) = c \\ n \geq 2, T(n) = aT(n/b) + c \end{cases}$$

avec a ≥ 1, b > 1, c > 0

a < b si a = 1 $\Rightarrow$ T(n) $\in$ θ(log n)
si a > 1 $\Rightarrow$ T(n) $\in$ θ($n^{\log_b a}$)
a = b $\Rightarrow$ T(n) $\in$ θ(n)
a > b $\Rightarrow$ T(n) $\in$ θ($n^{\log_b a}$)

Théorème 3 : généralisation

$$\begin{cases} T(1) = c \\ n \geq 2, T(n) = aT(n/b) + f(n) \end{cases}$$

Soient a ≥ 1 et b > 1 deux constantes, soit f(n) une fonction et soit T(n) une fonction définie pour les entiers positifs par la récurrence :

T(n) = aT(n/b)+ f(n);

T(n) peut alors être bornée asymptotiquement comme suit :

1. Si f(n) = O($n^{(\log_b a)-\varepsilon}$) pour une certaine constante ε > 0, alors T(n) = θ($n^{\log_b a}$).
2. Si f(n) = θ($n^{\log_b a}$), alors T(n) = θ($n^{\log_b a}$ log n) = θ(f(n) log n).
3. Si f (n) = Ω($n^{(\log_b a)+\varepsilon}$) pour une certaine constante ε > 0, et si a*f (n/b) ≤ c*f (n) pour une constante c < 1 et n suffisamment grand, alors T(n) = θ(f(n)).

Pour déterminer la complexité du tri par fusion, nous utilisons de nouveau le théorème ci-dessus.
On a **a = 2** et **b = 2**, donc **$\log_b$ a = 1** et nous nous trouvons dans le deuxième cas du théorème : f(n) = θ($n^{\log_b a}$) = θ(n). Par conséquent : T(n) = θ(n log n):

Pour des valeurs de n suffisamment grandes, le tri par fusion avec son temps d'exécution en θ(n log n) est nettement plus efficace que le tri par insertion dont le temps d'exécution est en θ(n^2).

IX.3.3 Multiplication de grands entiers

Soient X et Y deux grands nombres ayant n chiffres. (n suffisamment grand).

Pour calculer le produit de X par Y on peut programmer l'algorithme que l'on a appris au CP1 et qui consiste à multiplier chaque chiffre de Y par tous les chiffres de X en sommant les résultats intermédiaire :

Posons $x_n\ x_{n-1}\ ...\ x_3\ x_2\ x_1$ les chiffres formant X.
$y_n\ y_{n-1}\ ...\ y_3\ y_2\ y_1$ les chiffres représentant Y
La méthode habituelle demande n^2 multiplications entre chiffres et n additions de résultats intermédiaires avec des décalages vers la droite à chaque itération :

			x_n	x_{n-1} ...	x_3	x_2	x_1
		X	y_n	y_{n-1} ...	y_3	y_2	y_1
			---	---	---	---	---
			$y_1 * x_n$	$y_1 * x_{n-1}$	$y_1 * x_3$	$y_1 * x_2$	$y_1 * x_1$
		$y_2 * x_n$	$y_2 * x_{n-1}$	$y_2 * x_3$	$y_2 * x_2$	$y_2 * x_1$	0
	$y_3 * x_n$	$y_3 * x_{n-1}$	$y_3 * x_3$	$y_3 * x_2$	$y_3 * x_1$	0	0
+	....		....	....	0	0	0
	---	---	---	---	---	---	---
=	...	....	....	....	....	....	

Les additions et décalages demandes un nombre d'itérations proportionnel à n, donc cette méthode (habituelle) a une complexité de $\theta(n^2)$.

Une solution " Diviser pour résoudre" de ce problème est de scinder X et Y en deux entiers de n/2 chiffres chacun :

- Soient A la première moitié de X (les n/2 chiffres de poids fort) et B la 2e moitié de X
- Soient C la première moitié de Y (les n/2 chiffres de poids fort) et D la 2e moitié de Y

$A = x_n\ x_{n-1} ...\ x_{n/2}$ $B = x_{n/2+1}\ ...\ x_2\ x_1$
$C = y_n\ y_{n-1} ...\ y_{n/2}$ $D = y_{n/2+1}\ ...\ y_2\ y_1$

On peut écrire alors : $X = A\ 10^{n/2} + B$ et $Y = C\ 10^{n/2} + D$

Exemple : $X = 1234 \Rightarrow A = 12$ e $B = 34$
$\Rightarrow X = 12*10^{4/2} + 34 = 12*10^2 + 34 = 1200 + 34 = 1234$
Donc
$X \times Y = (A10^{n/2} + B)\ (C10^{n/2} + D) = AC10^n + (AD + BC)10^{n/2} + BD$

4 multiplications de n/2 chiffres (AC, AD, BC, BD)
3 additions en $\theta(n)$
2 décalages (multiplication par 10^n et $10^{n/2}$) en $\theta(n)$

L'équation de récurrence d'un tel algorithme serait alors :
$T(1) = a$ si n=1
$T(n) = 4T(n/2) + b\ n$ sinon

On peut trouver la solution par substitution ou bien en utilisant la technique des équations différentielles en posant par exemple $n = 2^k$, ce qui donne l'équation de récurrence :
$t_k - 4t_{k-1} = b\ 2^k$

dont la solution est connue $\theta(4^k)$ donc $\theta(n^2)$).

Cette solution (diviser pour résoudre) n'est pas plus efficace que la méthode habituelle (même complexité).

En diminuant le nombre de multiplications (donc le nombre d'appels récursifs), on peut diminuer la complexité de l'algorithme "Diviser pour résoudre" :

Remarquons que $X \times Y = AC10^n + (AD + BC)10^{n/2} + BD$
$= AC10^n + [(A-B)(D-C)+ AC + BD]10^{n/2} + BD$

3 multiplications (AC, BD, (A-B)(D-C))
4 additions en $\theta(n)$
2 soustractions en $\theta(n)$ A-B et D-C
2 décalages en $\theta(n)$

```
Fonction Mult(D X, Y, n : ENTIER) : ENTIER
DEBUT
  SI (n = 1) ALORS
    mul ← X*Y
  SINON
    DEBUT
      A ← n/2                          (*premiers chiffres de X*)
      B ← n/2                          (*derniers chiffres de X*)
      C ← n/2                          (*premiers chiffres de Y*)
      D ←n/2                           (*derniers chiffres de Y*)
      S1 ← moins(A, B)                 (* calcule A-B*)
      S2 ← moins(D, C)                 (*calcule D-C*)
      m1 ← Mult(A, C,  n/2)            (*calcule AC*)
      m2 ← Mult(S1, S2,  n/2)          (* calcule (A-B)(D-C) *)
      m3 ← Mult(B, D,  n/2)            (* calcule BD*)
      S3 ← plus( m1, m2 )              (* calcule [AC] + [ (A-B)(D-C)] *)
      S4 ← plus( S3,  m3 )             (* calcule [ AC+(A-B)(D-C)] + [BD] *)
      P1 ← Décalage(m1,  n)            (*calcule AC 10^n*)
      P2 ← Décalage( S4, n/2 )         (* calcule [AC+(A-B)(D-C)+BD] 10^n/2*)
      S5 ← plus( P1, P2 )              (* calcule [AC 10^n] + [(AC+(A-B)(D-C)+BD) 10^n/2 ] *)
      S6 ← plus( S5, m3 )              (* calcule [AC 10^n+(AC+(AB)(DC)+BD) 10^n/2 ] + [BD] *)
      Mul ← S6
    FIN
FIN
```

$T(n) = 3T(n/2) + c\,n$
Solution $\theta(n^{1,59})$, avec $1,59 = \log_2 3$, ce qui est meilleur que $\theta(n^2)$

Remarque :
Méthode asymptotiquement plus efficace mais beaucoup moins limpide de point de vue pédagogique. Efficace pour les grands nombres (> 500 chiffres)

Autre exemple Tour de Hanoï

IX.4 Recommandations générales sur "Diviser pour Résoudre"

IX.4.1 Equilibrer les sous-problèmes

La division d'un problème en sous-problème de taille sensiblement égale contribue de manière cruciale à l'efficacité de l'algorithme.

Considérer le tri par insertion.
On suppose T[1..K] trié et on insère T[K+1] par décalage

```
FONCTION Tri(DR L : Liste, D n : ENIER) : Liste
DEBUT
  Si n = 1 ALORS
    Tri ← L
  SINON
    Tri ←  Insere(Tri(n-1),T(n))
FIN
```

Division du problème en deux sous problèmes de taille différente, l'un de taille 1 et l'autre de taille n-1.
T(n) = c si n=1
= T(n-1) + n sinon (la fonction Insère est $\theta(n)$)

Ce qui conduit à $\theta(n^2)$ (plus complexe que $\theta(n\text{Log}(n))$)

Le tri par fusion subdivise le problème en deux sous problème de taille n/2. Ce qui donne une complexité égale à $\theta(n \text{ Log}(n))$.

IX.4.2 Le nombre de sous problèmes doit être indépendant de n

La décomposition doit générer un nombre de sous problème constant et indépendant de n. Sinon la solution ne sera pas efficace.

Exemple : Le calcul du déterminant d'une matrice en récursif
Det(M) = M[1,1] Si n=1 (taille de la matrice)
$= \sum_{i=1..n} (1)_{i+1} * M[1,i] * \text{Det}(M(1, j))$ Si n > 1
Avec M(i, j) désignant la matrice sans la ligne i et sans la colonne j

Pour une matrice de taille n x n, cette décomposition génère n sous-problèmes (sous-matrices de taille n-1x n-1).

T(n) = n T(n-1)
= n (n-1)T(n-2)= … = n! T(1) > $\theta(n!)$

Alors qu'il existe des algorithmes en $o(n^3)$.

Exercices
Exercice 1 : Recherche dichotomique
Ecrire une fonction récursive permettant de rechercher de façon dichotomique une donnée dans un tableau trié par ordre croissant. Puis calculer sa complexité.

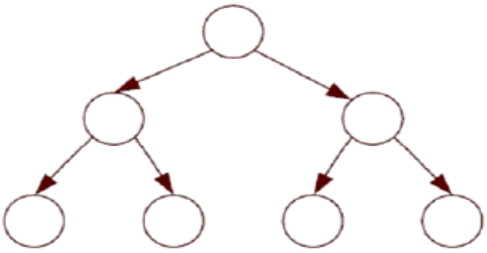
X Programmation dynamique

La programmation dynamique est similaire à la méthode "Diviser pour régner" en ce sens qu'une solution d'un problème dépend des solutions précédentes obtenues des sous-problèmes.

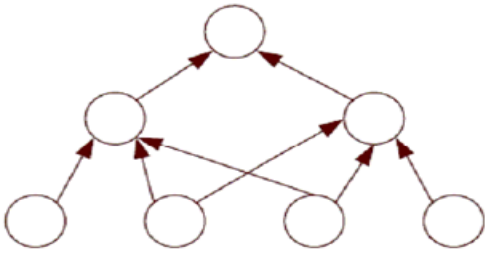
La différence significative entre ces deux méthodes est que la programmation dynamique permet d'utiliser un sous-problème dans la solution de deux sous-problèmes différents.

Tandis que l'approche diviser pour régner crée des sous-problèmes qui sont complètement séparés et peuvent être résolus indépendamment l'un de l'autre.

Un algorithme "diviser pour régner" fait donc plus de travail car Il résout plusieurs fois les sous-sous-problèmes communs.



diviser et régner



programmation dynamique

Un algorithme de programmation dynamique résout chaque sous-sous-problème une seule fois et mémorise sa solution dans un tableau afin d'évite le recalcul de la solution chaque fois que le sous-sous-problème est rencontré.

Une seconde différence entre ces deux méthodes est, comme illustré par la figure ci-dessus, est que la méthode diviser et régner est récursive, les calculs se font de haut en bas.

Tandis que la programmation dynamique est une méthode dont les calculs se font de bas en haut : on commence par résoudre les plus petits sous-problèmes. En combinant leur solution, on obtient les solutions des sous-problèmes de plus en plus grands.

La programmation dynamique est en général appliquée aux problèmes d'optimisation. Ces problèmes peuvent admettre plusieurs solutions parmi lesquelles on veut choisir une solution optimale (maximale ou minimale pour une certaine fonction de coût).

Remarque 1 : quand utiliser la programmation dynamique ?
Elle s'applique lorsque la subdivision en sous-programmes n'est pas facile à déterminer de façon optimale et les sous-programmes ne sont pas indépendants.

Remarque 1
Le style de programmation qui consiste à faire en sorte qu'une fonction "se souvienne" (enregistre) les résultats pour les valeurs d'arguments qu'elle a déjà calculer s'appelle également la "mémoisation". (cf SlidesLAlgo5.pdf)

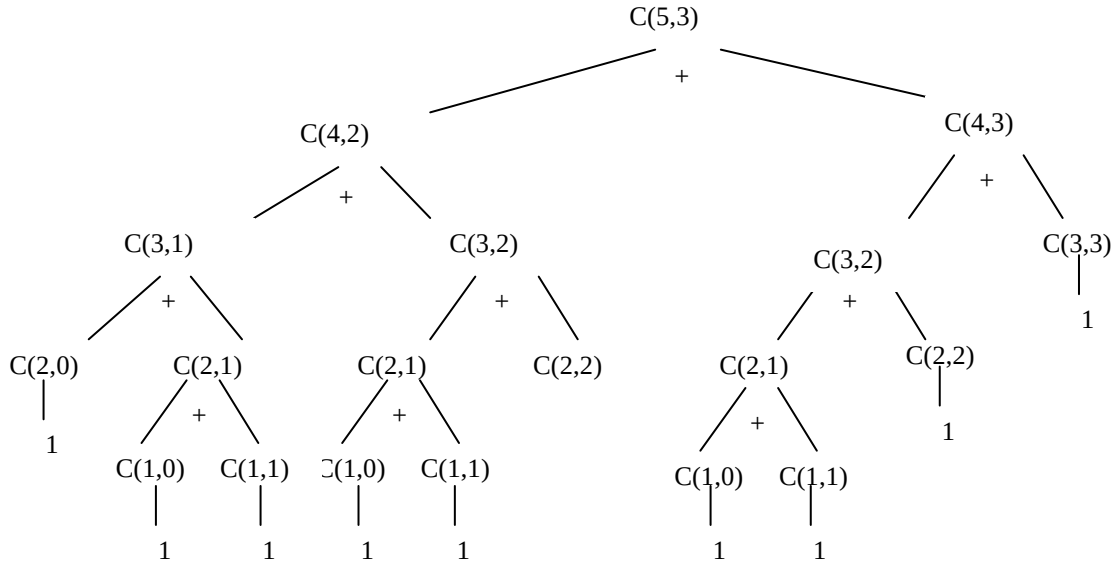
Exemple 1 : On veut calculer $C_n^p = \frac{n!}{p*(n-p)!}$

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon.} \end{cases}$$

X.1 Approche Diviser pour calculer C_n^p

FONCTION comb(n, p :ENTIER) :ENTIER
DEBUT
SI(n = 0 OU n=p) ALORS
  comb ← 1
SINON
  comb ← comb(n-1, p-1) + comb(n-1, p)
FIN
	ALGORITHME combinaison
	VAR c : ENTIER
	DEBUT
	  c ← comb(5,3)
	  ECRIPRE('Resultat=',c)
	FIN

Arbre d'appel



Le temps de calcul de cette fonction est proportionnel au nombre d'appels de la fonction.

Le nombre total d'appel récursif est : $2C_n^p - 2 = 2*10 - 2 = 18$ (-2 pour les 2 cas triviaux)

Ce nombre d'appels est donné par la récurrence :

$$N_{n,0} = N_{n,n} = 1$$
$$N_{n,k} = N_{n-1,k-1} + N_{n-1,k} + 1$$

On montre par récurrence que

$$N_{n,k} = 2 \times C_n^k - 1$$

Les temps de calcul les plus longs sont donc quand *k≈ n/2* et donc dans ce cas $\frac{n!}{k!(n-k)!} \approx \frac{n!}{k! \times k!} \approx C_n^{\frac{n}{2}}$ avec k = n/2

Complexité = θ(n!)

$$\frac{n!}{k!(n-k)!} \approx \frac{n!}{k! \times k!} \approx C_n^{\frac{n}{2}} \approx \frac{1}{\sqrt{\frac{n}{2}\pi}} \times 2^n$$

(On utilise la formule de Stirling $n! \approx \left(\frac{n}{e}\right)^n \sqrt{2n\pi}$)

X.2 Approche programmation dynamique pour calculer C_n^p

Le coût du programme précédent est du aux multiples calculs de la même valeur de C_n^p.

Pour minimiser ce coût on peut utiliser une technique de programmation dynamique qui consiste à stocker les valeurs intermédiaires pour ne pas les recalculer : c'est le triangle de Pascal.

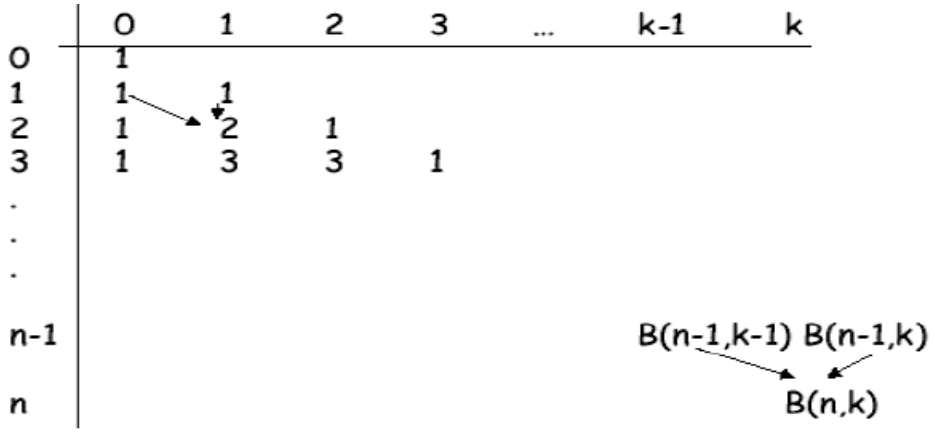
1							
1	1						
1	2	1					
1	3	3	1				
1	4	6	4	1			
1	5	10	10	5	1		
1	6	15	20	15	6	1	

Principe :

1. On rempli les k premières cases de chaque ligne de haut en bas.
2. On arrête à la ligne n
3. Temps : T(n, p) = θ(n*p)

	0	1	2	3	…	n-1	n
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
…							
n-1	1	n-1	C_{n-1}^2	C_{n-1}^3	…	1	
n	1	n	C_n^2	C_n^3		n	1

Pour éviter de calculer plusieurs fois un nombre, l'idée est de créer un tableau où on calcule tous les nombres de petites tailles, ensuite, de tailles de plus en plus grandes avant d'arriver au nombre désiré. Pour ce faire, on procède comme suit :



Pour calculer donc B(n, k), on procède comme suit :

Triangle de Pascal avec une matrice

```
TYPE Matrice = TABLEAU[0..Taille-1, 0..Taille] DE ENTIER

PROCEDURE tPascalM(D max :ENTIER)
VAR n, p : ENTIER
        B : Matrice
DEBUT
 POUR n ← 0,max-1   (* initialisation 1 à la 1ère case et celle de la diagonale*)
   DEBUT
    B[n, 0] ← 1 (* des zéros ailleurs (partie triangulaire inférieure*)
    B[n, n] ←1
    B[n, n+1] ← 0

    POUR p ← 1, n-1 (* la première ligne correspond à n=0 *)
      B[n, p] ← B[n-1, p-1] + B[n-1, p] (*calcul des termes suivant la formule :*)
                                          (* c(n, p)=c(n-1, p-1)+c(n-1, p) *)
   FIN

 (* affichage *)
 POUR n ← 0, max-1
   DEBUT
    POUR p ← 0, n
      ECRIRE(B[n, p],'  ')
   ECRIRE
 FIN

FIN
```

```
ALGORITHME combinaison
VAR c : ENTIER
DEBUT
 tPascalM(5)
FIN
```

On remplit le tableau B ligne par ligne comme suit

Complexité de l'algorithme

Cette complexité est réduite au remplissage de la moitié du tableau B. Comme le nombre d'éléments de B est de $k \times n$, la complexité de l'algorithme est par conséquent en $\theta(kn)$.

Remarque :
Il est intéressant de constater que seule la moitié de la matrice B est utilisée pour calculer B[n, k]. Autrement dit, un tableau à une dimension suffirait pour faire ces calculs. On obtient alors l'algorithme suivant :

Triangle de Pascal avec un tableau

```
TYPE Vecteur = TABLEAU[0..Taille-1] DE ENTIER

PROCEDURE tPascal3(D max : ENTIER)
VAR n, p : ENTIER
        B : Vecteur
DEBUT
  B[0] ← 1 (* initialisations *)
  B[1] ← 0;
  POUR n ←  1, max (* calcul et affichage de max lignes *)
    DEBUT
      POUR p ← n-1, 1, -1  (* c'est la ruse *)
        DEBUT
          B[p] ←  B[p] + B[p-1]
          ECRIRE(B[p], ' ')
        FIN
      ECRIRE(B[0], ' ');
      B[n] ← 0
    FIN
FIN
```

X.3 Recommandations pour programmation dynamique

Quand et comment utiliser la méthode de la programmation dynamique
La programmation est un outil général de résolution de problèmes. Toutefois, il n'y pas de règle pour affirmer que la programmation dynamique peut ou ne peut être utilisée pour résoudre tel ou tel problème.

Le gros du travail, si on veut utiliser cette méthode, réside tout d'abord dans l'obtention de l'expression récursive de la solution en fonction de celle des sous-problèmes (de taille plus petite).

Notons que dans les problèmes d'optimisation, cette manière d'obtenir la solution optimale à partir des solutions optimales des sous problèmes s'appelle le principe d'optimalité de Bellman. Il est important de souligner que ce principe, bien que naturel, n'est pas toujours applicable.

Exercices
Exercice 1 : Nombre de Fibonacci

$$\begin{cases} F_0 = 1 F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \text{ pour } n > 1 \end{cases}$$

Ecrire une fonction récursive permettant de calculer le nombre de Fibonacci et calculer sa complexité.

Ecrire le même programme avec utilisation de la programmation dynamique (mémorisation des résultats des sous-problèmes).

Exercice 2 : Problème du sac à dos

a) Variante "tout ou rien"
   Un voleur dévalisant un magasin trouve n objets, le i^e de ces objets valant v_i euros et pesant w_i kilos, v_i et w_i étant des entiers. Le voleur veut bien évidemment emporter un butin de plus grande valeur possible mais il ne peut porter que W kilos dans son sac à dos. Quels objets devra-t-il prendre ?

b) Variante fractionnaire
   Le problème est le même que le précédent, sauf qu'ici le voleur peut ne prendre qu'une fraction d'un objet et n'est plus obligé de prendre l'objet tout entier comme précédemment, ou de le laisser.

Question

1. Proposez un algorithme glouton pour la variante fractionnaire.
2. Montrez que cet algorithme est optimal.
3. Quelle est sa complexité ?
4. Montrez au moyen d'un contre-exemple que l'algorithme glouton équivalent pour la variante "tout ou rien" n'est pas optimal.
5. Proposez un algorithme de programmation dynamique résolvant la variante "tout ou rien".
6. Quelle est sa complexité ?

XI Algorithmes gloutons

Un algorithme glouton détermine une solution après avoir effectué une série de choix. Pour chaque point de décision, le choix qui semble le meilleur à cet instant est retenu. Cette stratégie ne produit pas toujours une solution optimale.

XI.1 Définition

On appelle **algorithme glouton** un algorithme qui suit le principe de faire, étape par étape, un choix localement optimal (ou choix glouton), dans l'espoir d'obtenir un résultat optimum global.

Le choix est basé sur les solutions des sous-problèmes.

Par exemple, dans le problème du rendu de monnaie (donner une somme avec le moins possible de pièces), l'algorithme consistant à répéter le choix de la pièce de plus grande valeur qui ne dépasse pas la somme restante est un algorithme glouton.

Dans les cas où l'algorithme ne fournit pas systématiquement la solution optimale, il est appelé une heuristique gloutonne.

XI.2 Comparaison avec la programmation dynamique

En programmation dynamique on fait un choix à chaque étape, mais ce choix dépend de la solution de sous-problèmes, au contraire, dans un algorithme glouton, on fait le choix qui semble le meilleur sur le moment puis on résout les sous-problèmes qui surviennent une fois le choix fait.

Une stratégie gloutonne progresse en général de manière descendante en faisant se succéder les choix gloutons pour ramener itérativement chaque instance du problème à une instance "plus petite".

Exemple 1 : Rendre la monnaie

Entrées: tableau des pièces disponibles
(P = 25;25;25;10;10;5;1;1;1;1), montant à payer (n = 67)

Sortie: ensemble des pièces utilisées (*S*=25;25;10;5;1;1), le moins de pièces possibles

Stratégie:

1. trier le tableau P des pièces disponibles par ordre de poids décroissant
2. ajouter la plus grande pièce possible au tableau S
3. tant que la somme des pièces de S n'excède pas le montant à payer refaire 2.

MonnaieI.c

```
CONST N = 4
TYPE Vecteur = TABLEAU[1..N] DE ENTIER

PROCEDURE  initiliser(DR S : Vecteur)
VAR i : ENTIER
DEBUT
  POUR(i ← 1, N)
    S[i] ← 0
FIN

PROCEDURE  monnaie(D P : Vecteur, D m : ENTIER, R S :Vecteur)
VAR S : Vecteur
        i, j : ENTIER
DEBUT
  i ← 1
  j ← 1
  trier(P) /* trier P par ordre décroissant */
  TANTQUE(m ≠ 0 ET i ≤ N)
    DEBUT
      SI(m - P[i] ≥ 0) ALORS
        DEBUT
          m ← m - P[i]
          S[j] ← P[i]
          j ← 1 + 1

        FIN
      i ← 1 + 1
    FIN
    SI(m ≠ 0)
      initialiser(S)
FIN
```

```
ALGORITHME Monnaie
VAR
DEBUT
VAR P, S: Vecteur
        m : ENTIER
DEBUT
  m ← 67
  P[1] ← 25 ; P[2] ← 25 ; P[3] ← 25 ;
  P[4] ← 10 ; P[5] =10 ; P[6] ← 5
  P[7] ← 1 ; P[8] ← 1 ; P[9] ← 1 ;
  P[10] ← 1

  initialiser(S)
  monnaie(P, m, S);
  afficher(P);
  afficher(S);
FIN
```

Analyse de quelques cas

- P = (25, 25, 25, 10, 10, 5, 1, 1, 1, 1) : toujours une solution optimale pour m < 105.
- P = (8, 4, 2) : toujours une solution optimale si elle existe pour n est pair et n ≤ 14.
- P = (5, 2) : solution existe toujours (si n ≥ 4), mais on ne la trouve pas toujours pour n= 6.
- P = (5, 4, 1) : solution existe toujours, on la trouve toujours, mais elle n'est pas toujours optimale pou n = 8.

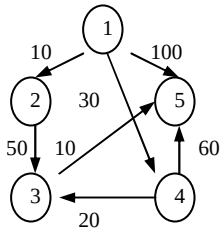
Complexité : $\theta(n^2 + n + n)$ $\theta(n^2 + 2n) \approx \theta(n^2)$
En effet :

- Algorithme de tri : $\theta(n^2)$
- Boucle TANQUE : $\theta(n)$
- Algorithme initialiser : $\theta(n)$

Exemple 2 : Algorithme de Dijkstra

L'algorithme de Dijkstra résout le problème du plus court chemin pour un graphe orienté et connexe dont le poids lié aux arcs est positif ou nul.

On peut prendre pour **exemple** le réseau routier d'une région: chaque sommet est une ville, chaque arc est une route dont le poids est en fait le kilométrage: l'algorithme de Dijkstra permet alors de trouver le plus court chemin d'une ville à une autre.



L'algorithme porte le nom de son inventeur **Edsger Dijkstra**.

```
TYPE
Matrice = TABLEAU[1..N,1..N] DE ENTIER
Vecteur = TABLEAU[1..N] DE ENTIER

Entrée
  P : MATRICE (* Matrice de poids telle que P[i,j]=poids de l'arc i→ j, P[i,j]=∞ si l'arc i→ j*)
                (* n'existe pas *)
Sortie
  C : VECTEUR (* Permet de construire le plus court chemin de la source à chaque sommet*)
  D : VECTEUR (* Contient le poids des plus courts chemins depuis le sommet 1 vers tous  *)
                (* tous les sommets *)
```

```
PROCEDURE dijkstra(D P:Matrice,R C:Vecteur, R D:Vecteur)
DEBUT
  E ← {1}
  POUR i ← 2, N
    D[i] ← P[1,i] (* initialisation de D *)
  POUR i ← 1, N-1
    DEBUT
      Choisir un sommet t de S-E tel que D[t] soit minimum
      ajouter t à E
      POUR chaque sommet s de S-E
      DEBUT
        SI(D[t]+P[t, s] < D[s]) ALORS
          C[s] ← t
        D[s] ← min(D[s], D[t] + P[t, s])
      FIN
    FIN
FIN
```

Exécution : Le sommet est 1

			S-E			
Itération	**E**	**t**	**D[2]**	**D[3]**	**D[4]**	**D[5]**
0	{1}	/	10	∞	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

Matrice P

	1	**2**	**3**	**4**	**5**
1	0	10	∞	30	100
2	∞	0	50	∞	∞
3	∞	∞	0	∞	10
4	∞	∞	20	0	60
5	∞	∞	∞	∞	0

A chaque itération, pour choisir D[i] minimum on ne considère plus les sommets de S-E déjà analysés (ex. à l'itération 2 on ne considère plus D[2]).

Les dernières valeurs de D sont les meilleurs raccourcis pour se rendre du sommet 1 aux autres sommets.

Source	Destination	Raccourci	Chemin
1	2	D[2]=10	$1 \rightarrow 2$
1	3	D[3]=50	$1 \rightarrow 4 \rightarrow 3$
1	4	D[4]=30	$1 \rightarrow 4$
1	5	D[5]=60	$1 \rightarrow 4 \rightarrow 3 \rightarrow 5$

Reconstitution des chemins

C[2]	C[3]	C[4]	C[5]
1	4	1	3

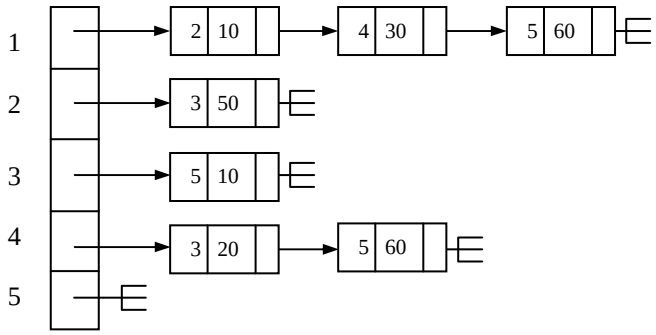
Pour trouver le plus cours chemin entre 1 et 5, on suit les prédécesseurs dans l'ordre inverse en commençant par 5
C[5]=3, C[3]=4, C[4]=1, d'où le chemin $1 \rightarrow 4 \rightarrow 3 \rightarrow 5$

Chemin le plus court entre 1 et 3
C[3]=4, C[4]=1, d'où le chemin $1 \rightarrow 4 \rightarrow 3$

Complexité de l'algorithme de Dijkstra

- Utilisation d'une matrice d'adjacence $\Rightarrow$ $\theta(n^2)$
- Utilisation d'une liste d'adjacence $\Rightarrow$ $\theta(a \log_2 n)$

Liaison aérienne entre 5 villes



L'algorithme de Dijkstra appartient à la famille des algorithmes **gloutons.**

Exercices

Exercice 1 : rendu de monnaie
Il s'agit de donner une somme avec le moins possible de pièces. L'algorithme consistant à répéter le choix de la pièce de plus grande valeur qui ne dépasse pas la somme restante. Ecrire un algorithme récursif pour faire ce travail ; On utilisera des matrices :

- La matrice P contient les pièces de monnaie de la caisse, P[i][0] contient la pièce, P[i][1] contient le nombre de pièce.

- La matrice S contient les pièces de monnaie à rendre, S[i][0] contient la pièce, S[i][1] contient le nombre de pièce.

Exercice 2 : Problème du sac à dos

a) Variante "tout ou rien"

Un voleur dévalisant un magasin trouve n objets, le i^e de ces objets valant v_i euros et pesant w_i kilos, v_i et w_i étant des entiers. Le voleur veut bien évidemment emporter un butin de plus grande valeur possible mais il ne peut porter que W kilos dans son sac à dos. Quels objets devra-t-il prendre ?

b) Variante fractionnaire

Le problème est le même que le précédent, sauf qu'ici le voleur peut ne prendre qu'une fraction d'un objet et n'est plus obligé de prendre l'objet tout entier comme précédemment, ou de le laisser.

Question

1. Proposez un algorithme glouton pour la variante fractionnaire.
2. Montrez que cet algorithme est optimal.
3. Quelle est sa complexité ?
4. Montrez au moyen d'un contre-exemple que l'algorithme glouton équivalent pour la variante "tout ou rien" n'est pas optimal.
5. Proposez un algorithme de programmation dynamique résolvant la variante "tout ou rien".
6. Quelle est sa complexité ?

Exercice 3

On considère le problème de la location d'une unique voiture. Des clients formulent un ensemble de demandes de location avec, pour chaque demande, le jour du début de la location et le jour de restitution du véhicule.

Notre but ici est d'affecter le véhicule de manière à satisfaire le maximum de clients possible (et non pas de maximiser la somme des durées des locations).

Nous disposons donc d'un ensemble *E* des demandes de location avec, pour chaque élément *e* de *E*, la date *d*(*e*) du début de la location et la date *f*(*e*) de la fin de cette location.

Nous voulons obtenir un ensemble *F* maximal de demandes satisfaites. Cet ensemble *F* doit vérifier une unique contrainte : deux demandes ne doivent pas se chevaucher dans le temps, autrement dit une location doit se terminer avant que la suivante ne commence. Cette contrainte s'écrit mathématiquement :

$$\forall e_1 \in F, \forall e_2 \in F, \quad d(e_1) \leq d(e_2) \Rightarrow f(e_1) \leq d(e_2).$$

XII Bibliographie

[ALR] : "Structure de données et algorithmes", Alfred Ano, John Hopcroft, Jeffrey Ullman, InterEdition.

[BOR] : "Initiation à la programmation", Collection Sciences et pratique de l'informatique, Bordas Informatique.

[COU] : "Initiation à l'algorithmique et aux structures de données, 1, 2 Récursivité et structures de données avancées", J. Courtin, I. Kowarski, Dunod Informatique.

[GUY] : "Arbre Table et Algorithmes", J. Guyot, C. Vial, Eyrolles.

[MEY] : "Méthodes de programmation", B. Meyer et C Baudoin, Eyrolles.

Webographie
http://brassens.upmf-grenoble.fr/IMSS/limass/algoprog/algocours.html

http://www-math.univ-fcomte.fr/pp_Annu/FLANGROGNET/ENSEIGNEMENT/Hachage_2007.pdf.

http://aqualonne.free.fr/dossiers/algo5.html

Christine PORQUET : ENSICAEN - 1ère année – 2nd semestre – Algorithmique avancée : Poly_Algo_avancee_debut.pdf

http://email.ini.dz/~hidouci/mcp/3_Divise_Resoudre.pdf

F:\Cours2008\ALGO\Avancé\http___graal.ens-lyon.fr__fvivien_Enseignement_Algo-2001-2002_index.html\graal.ens-lyon.fr_fvivien\Enseignement\Algo-2001-2002\Cours.pdf

http://www.fil.univ-lille1.fr/portail/index.php?dipl=M1&sem=S7&ue=AAC&choix=prog