Introduction à l'algorithmique

I. Introduction

A. Présentation des composants

B. Fonctionnement général d'un PC

C. Utilisation du logiciel

II. Intro au développement

Plusieurs paradigmes de programmation

- Programmation impérative ou procédurale (L1)
- Programmation orientée objet (L2)
- Programmation fonctionnelle

Ensemble de stratégies permettant de décrire le comportement du programme

Analyse

- Exploration du domaine d'application → déduire des contraintes et des limites
- Définir les principales fonctionnalités

⇨ Modélisation des données et définition des modules de traitement

Calculabilité du problème

Se fait durant l'analyse
Peut-on résoudre le problème avec un programme informatisé ?

Méthode de construction descendante

- Décomposition → isolement des actions du problèmes, puis les développer
- Assemblage → remplacer chaque action par leur développement ➔ actions élémentaires

Validation algorithme

- Terminaison : est-ce que le programme s'arrête ?
- Complexité : ressources nécessaires et durée d'exécution
- Exactitude : le programme fait ce qu'on attend de lui ?

Compilateur et interpréteur

Compilateur : lit un programme rédigé dans un langage de programmation, le traduit dans un autre langage (binaire dans la plupart des cas), et signale les erreurs contenues dans le programme (lexicales et syntaxiques).

Différentes étapes

- ✓ Codage
- ✓ Vérification
- ✓ Maintenance : faire évoluer le programme, lui ajouter des fonctionnalités et résoudre les problèmes

Un programme doit être commenté, indenté et avoir des noms de variables cohérents et facilitant la lecture

III. Notions de base pour l'écriture d'algorithmes

A. Structure d'un algorithme

```
algo nomAlgo
      /* déclaration des variables et constantes */
début
      /* instruction à suivre pour résoudre le problème */
fin
```

B. Variables et types de variable

Les variables désignent un emplacement de mémoire qui permet de stocker une valeur. Elles se définissent par un nom unique, un type et une valeur. Elles doivent être déclarées au début de l'algorithme :

```
variables   nomVar1 : type1
            nomVar2, nomVar3 : type2
```

Le type d'une variable est très important, il permet de déterminer les valeurs que la variable peut prendre mais aussi les opérations qui lui sont applicables.

Types numériques

- Entier (de -32 768 à 32 767)
- Entier long
- Réel
- Réel double

Ici, le type définit les valeurs min et max que peuvent prendre la variable, ainsi que la précision de sa valeur (nombre décimaux)

Opérateurs applicables à ces types : + - × / div mod < ≤ > ≥ = ≠

div correspond à une division euclidienne, et mod donnera le restant d'une division euclidienne entre deux valeurs

Type caractères

- Symbole alphanumérique (lettre ou chiffre) et signes de ponctuation
- Codé à l'aide du code ASCII
- Valeurs s'écrivent entre ' ' ('a' ; '3')

Opérateurs : < ≤ > ≥ = ≠

Type booléen

Deux valeurs sont possibles pour ce type : Vrai ou Faux

Les opérateurs sont appelés logiques et sont : et ou non (AND ; OR ; NOT)

Constante

Une constante est une donnée dont la valeur est connue avant l'exécution du programme et qui ne varie pas au cours du traitement. (Ex : π)

Constante → `nomCte = valeurCte`

C. Déclaration et affectation de variable

La déclaration d'une variable réserve un espace dans la mémoire auquel on accède par le nom de cette variable. A l'issue de la déclaration, la valeur d'une variable est indéfinie. Il faut donc l'initialiser :

- Soit en affectant une valeur ou un résultat d'expression
- Soit lire une valeur

Affectation :

```
nomVar1 ← valeur
nomVar1 ← expression
```

D. Instructions lecture/écriture

Ces instructions permettent le dialogue entre l'utilisateur et la machine.

La lecture permet de communiquer des valeurs à la machine, dans ce cas là l'ordinateur range la valeur saisie au clavier dans la variable `nomVar`. La valeur saisie au clavier doit avoir le même type que celui de la variable `nomVar`.

L'écriture permet l'affichage d'une valeur/d'un texte à l'écran. Par exemple, `écrire ("le résultat est :", nomVar)` affichera à l'écran le texte entre " " suivi de la valeur contenue dans `nomVar`.

E. Structures de contrôles

- Séquence
- Instructions conditionnelles
- Instructions itératives

Séquence :

Une séquence est l'enchainement inconditionnel d'une suite d'instructions :

```
instruction₁
instruction₂
…
instructionₙ
```

Instruction conditionnelle

Permet l'exécution d'une séquence d'instruction lorsqu'une condition est vraie. Elles peuvent être imbriquées.

```
si(condition) alors
      séquence instructionsA

si(condition) alors
      séquence instructionsA
sinon
      séquence instructionsB
```

Les conditions ne peuvent prendre que des valeurs booléennes, et sont construites à partir de constantes/variables/opérateurs de comparaison et/ou logiques. Elles peuvent être simples ou composées.

Condition simple :

- Variables de type booléen
- Comparaison

Conditions composées :

- Correspond à plusieurs expressions booléennes reliées par un opérateur logique
- Opérateurs : et ou non (AND ; OR ; NOT)

Remarque

Afin d'éviter toute ambigüité, les conditions doivent être parenthésées

E_1	E_2	E_1 **et** E_2	E_1 **ou** E_2	**non**(E_1)
vrai	vrai	vrai	vrai	faux
vrai	faux	faux	vrai	faux
faux	vrai	faux	vrai	vrai
faux	faux	faux	faux	vrai

Table – Tables de vérités des opérateurs *et*, *ou*, *non*

Propriété des opérateurs logiques

Soient a, b et c trois expressions booléennes.

Double négation	$\textbf{non}(\textbf{non}(a)) = a$
Elément neutre du **ou**	a **ou** $faux = a$
Elément neutre du **et**	a **et** $vrai = a$
Elément absorbant du **ou**	a **ou** $vrai = vrai$
Elément absorbant du **et**	a **et** $faux = faux$
Idempotence du **ou**	a **ou** $a = a$
Idempotence du **et**	a **et** $a = a$
Tautologie	a **ou** $\textbf{non}(a) = vrai$
Contradiction	a **et** $\textbf{non}(a) = faux$
Commutativité du **ou**	a **ou** $b = b$ **ou** a
Commutativité du **et**	a **et** $b = b$ **et** a
Associativité du **ou**	a **ou** $(b$ **ou** $c) = (a$ **ou** $b)$ **ou** c
Associativité du **et**	a **et** $(b$ **et** $c) = (a$ **et** $b)$ **et** c
Distributivité du **et** par rapport au **ou**	a **et** $(b$ **ou** $c) = (a$ **et** $b)$ **ou** $(a$ **et** $c)$
Distributivité du **ou** par rapport au **et**	a **ou** $(b$ **et** $c) = (a$ **ou** $b)$ **et** $(a$ **ou** $c)$
Lois de Morgan	$\textbf{non}(a$ **ou** $b) = \textbf{non}(a)$ **et** $\textbf{non}(b)$ $\textbf{non}(a$ **et** $b) = \textbf{non}(a)$ **ou** $\textbf{non}(b)$

Instructions itératives

Elles permettent d'exprimer qu'une séquence d'instruction doit être exécutée un certain nombre de fois.

➢ Boucles déterministes
- Nombre d'itérations connu à l'avance
- Utilise un compteur de boucle

➢ Boucles non déterministes
- Nombre d'itérations n'est pas connu à l'avance

Il existe trois types de boucles :
- Boucle pour
- Boucle tant que
- Boucle répéter

Boucle pour

Elle est déterministe. Elle peut être croissante ou décroissante

Croissante : `pour` $\texttt{variable}_{\texttt{compteur}} \leftarrow \texttt{valeur}_{\texttt{deb}}$ `à` $\texttt{valeur}_{\texttt{fin}}$ `faire`
`séquenceInstructions`

- $\texttt{Variable}_{\texttt{compteur}}$ est initialisée à $\texttt{valeur}_{\texttt{deb}}$
- A chaque tour de boucle, $\texttt{variable}_{\texttt{compteur}}$ est augmentée de 1
- La boucle s'arrêtera lorsque $\texttt{variable}_{\texttt{compteur}}$ sera plus grande que $\texttt{valeur}_{\texttt{fin}}$

Décroissante : `pour` $\texttt{variable}_{\texttt{compteur}} \leftarrow \texttt{valeur}_{\texttt{deb}}$ `bas` $\texttt{valeur}_{\texttt{fin}}$ `faire`
`séquenceInstructions`

- $\texttt{variable}_{\texttt{compteur}}$ est initialisée à $\texttt{valeur}_{\texttt{deb}}$
- A chaque tour de boucle, $\texttt{variable}_{\texttt{compteur}}$ est décrémentée de 1
- La boucle s'arrêtera lorsque $\texttt{variable}_{\texttt{compteur}}$ est plus grande que $\texttt{valeur}_{\texttt{fin}}$

Remarque

$\texttt{variable}_{\texttt{compteur}}$ ne doit pas être modifiée dans la séquence d'instructions

Boucle tant que
Elle est non déterministe

```
tant que (condition) faire
      séquenceInstructions
```

- Les instructions internes à la boucle sont exécutées tant que la condition est vraie
- La condition est évaluée au début de chaque itération ➔ les instructions internes à la boucle peuvent ne jamais être exécutées

Remarque

Les variables intervenant dans la condition doivent être initialisées avant la boucle. Au moins une instruction interne à la boucle doit modifier les variables intervenant dans la condition.

Boucle répéter
Elle n'est pas déterministe

```
répéter
      séquence Instructions
jusqu'à (condition)
```

- Les instructions internes à la boucle sont exécutées puis la condition est évaluée → elles sont exécutées au moins une fois
- La boucle ne se termine que quand la condition est évaluée à Vrai

Remarque

Au moins une instruction interne à la boucle doit modifier les variables intervenant dans la condition.

Exemple : exercice 4 du TD1

```
…
début
      répéter
            écrire ("Donner une note")
            lire (note)
      jusqu'à (note ≥ 0) et (note ≤ 20)
…
```

IV. Les tableaux

Jusqu'à maintenant, on a vu les types scalaire/simples (`entier, réel, caractère, booléen`). Un tableau permet de manipuler un ensemble d'éléments de même type au travers d'une seule variable. La mémoire de l'ordinateur étant composée de cases numérotées (adresses), un tableau représente un ensemble de cases mémoires consécutives.

Une variable de type `tableau` permet de manipuler un nombre fini d'éléments qui :
- Sont tous de mêmes types
- Possèdent un identifiant unique (nom de variable)
- Se différencient les uns les autres dans le tableau par leur numéro d'indice

Les tableaux les plus fréquents sont à une (vecteurs) ou deux (matrices) dimensions

Pour définir un tableau, quatre éléments fondamentaux sont nécessaires :
- Son nom
- Le type des éléments qu'il contient
- Le nombre de ses dimensions
- Sa taille (nombres d'éléments max qu'il peut contenir)

Tableaux à une dimension :

```
nomVar : tableau[nbreEltMax] de TypeDesElts
```

Tableaux à deux dimensions :

```
nomVar : tableau[nbreEltMaxligne , nbreEltMaxcolonne] de TypeDesElts
```

Remarque

L'indice de la 1ère case d'un tableau est 0

Un tableau correspond à une allocation statique de la mémoire

Une allocation statique de la mémoire correspond à une allocation mémoire dont la taille ne peut pas évoluer. Une allocation dynamique, à l'inverse, correspond à une allocation mémoire dont la taille peut évoluer au cours du programme.

La manipulation d'un tableau se fait uniquement case par case. Une case de tableau contenant des éléments de type X se comporte comme une variable de type X. On accède à une case du tableau en utilisant le nom de la variable suivi de l'indice de la case :

- `nomVar[indice]` (tableau à une dimension)
- `nomVar[indiceℓ , indicec]` (tableau à deux dimensions)

Pour effectuer un traitement sur l'ensemble des cases d'un tableau, il faut utiliser une boucle. Chaque tour de boucle traitera alors une case du tableau. Dans un tableau à deux dimensions, il faudra ainsi imbriquer deux boucles. La première boucle balayera les lignes, la deuxième, imbriquée, parcourra les colonnes.

Comment distinguer les cases qui contiennent un élément d'une case qui n'en contient pas ?

On regroupe tous les éléments au début du tableau. Pour connaitre la limite entre les cases contenant un élément des cases vides, on utilise une variable (par ex, nbE) qui indique le nombre d'éléments actuellement contenus dans le tableau.

```
pour i ← 0 à nbE-1 faire
      /* traitement sur tab[i] */
```

```
pour i ← 0 à nbE-1 faire
      pour j ← 0 nbE-1 faire
            /* traitement sur tab [i,j] */
```

V. Les sous-programmes

On a la possibilité de créer nos propres sous-programmes. Cela permet de :

- Répéter une même séquence d'instructions
- Améliorer la visibilité du programme
- Favoriser la réutilisation

Un sous-programme possède la même structure qu'un programme, implémente un cas particulier et peut être appelé par le programme principal voire un ou plusieurs autres sous-programmes.

Déclaration d'un sous-programme :

```
sous-programme nomSousProgramme (paramètres)
      déclaration des variables
début
      séquenceInstructions
fin
```

Paramètres : variables particulières permettant l'échange de valeurs entre le programme appelant et le sous-programme.

Il existe3 catégories de paramètres :

- Données (DON) : paramètres dont la valeur est inchangée par le sous-programme
- Résultats (RES) :paramètres dont la valeur est calculée par le sous-programme
- Données-résultats (DONRES) : paramètres dont la valeur est modifiée par le sous-programme

On a 2 types de sous-programme :

- Procédures
- Fonctions

A. Procédures

```
procédure nomProcédure (paramètres)
      déclaration des variables locales
début
      séquenceInstructions
fin
```

Paramètres

- ( ) (peut être vide)
- (DON = par_1 ; $type_1$, … ;
  RES = par_n ; $type_n$, … ;
  DONRES = par_p ; $type_p$, …)

L'appel d'une procédure est réalisé en donnant le nom de la procédure suivie des valeurs sur lesquelles va s'appliquer le sous-programme.

Exemple

Algorithme de procédure permettant de calculer le nombre de billets de 5€, de pièces de 2 et de 1€, contenus dans une somme entière.

```
procédure décomposition (DON = som : entier;
                        RES = B5, nP2, nP1 : entier)
      variable reste : entier
début
      B5 ← som div 5
      reste ← som mod 5
      nP2 ← reste div 2
      nP1 ← reste mod 2
fin

algo appelProc
      variable S, B5, P1, P2 : entier
début
      écrire ("Donner une somme")
      lire (S)
      décomposition (S, B5, P2, P1)
      écrire ("Billets de 5 =", B5)
      écrire ("Pièces de 2 =", P2)
      écrire ("Pièces de 1 =", P1)
fin
```

- Les paramètres formels sont utilisés pour définir un sous-programme
- Les paramètres effectifs sont utilisés lors de l'appel du sous-programme

Remarque :

- Les paramètres formels et effectifs n'ont pas besoin d'avoir le même nom
- L'ordre des paramètres n'est pas quelconque : le premier paramètre effectif correspond au premier paramètre formel, et ainsi de suite

Passage des paramètres

- Paramètres DON : passage par valeur. Le sous-programme va créer une copie du paramètre en variable locale, et va travailler sur cette variable.
- Paramètres RES et DONRES : passage par référence ou adresse. A l'inverse du passage par valeur, le sous-programme travaille directement sur les variables du programme appelant.

B. Fonctions

Les fonctions sont des cas particuliers des procédures. Elles n'utilisent que des paramètres de types DON, et ne renvoient qu'un seul résultat de type simple.

```
fonction nomFonction (paramètres) : type du résultat
      déclaration des variables locales
début
      séquenceInstructions
      retourner (valeurRetournéeParLaFct)
fin
```

L'appel d'une fonction de fait toujours dans une expression

Exemple

Algorithme de fonction permettant de calculer un entier à une certaine puissance

```
fonction puissance (DON = x, n : entier) : entier
      variable i, puiss : entier
début
      puiss ← 1
      pour i ← 1 à n faire
            puiss ← puiss × x
      retourner (puiss)
fin


algo appelFct
      variable x, n, res : entier
début
      écrire ("Donner x et n")
      lire (x)
      lire (n)al :
      res ← puissance (x, n)
      écrire (res)
```

VI. Compléments sur les tableaux

Dans cette partie, on prendra un vecteur quelconque pour les exemples, c'est-à-dire sans relation d'ordre. Soit `tailleMax` une constante et un tableau `tab` contenant nbE éléments.

A. Traitement sur les vecteurs non ordonnés

Ajout d'un élément

Principe

L'élément est mis dans la première case vide du tableau, et on augmente nbE de 1 à chaque ajout.

Remarque

L'ajout n'est possible que si le tableau n'est pas plein.

```
procédure ajout   (DON = val : entier ;
                   DONRES = nbE : entier, tab : tableau [tailleMax] d'entiers)
début
      si (nbE < tailleMax) alors
            tab[nbE] ← val
            nbE ← nbE + 1
      sinon
            écrire ("Plus de place")
fin
```

Suppression d'un élément

Principe

Dans la case p de l'élément supprimé, on déplace l'élément contenu dans la dernière case non vide du tableau, et on diminue nbE de 1 à chaque tour.

Remarque

Il n'est possible de supprimer l'élément en position p que si p correspond à l'indice d'une case vide du tableau.

```
procédure suppression   (DON = p : entier ;
                         DONRES = nbE : entier, tab : tableau[tailleMax]
                         d'entiers)
début
      si (p ≥ 0) et (p ≤ nbE) alors
            tab[p] ← tab[nbE – 1]
            nbE ← nbE – 1
      sinon
            écrire ("Position incorrecte")
fin
```

Recherche du minimum et du maximum

Principe

Dans une variable `min`, on place le premier élément contenu dans le tableau, puis on parcourt le reste du tableau, en comparant chaque élément du tableau avec `min`, et on remplace si besoin par la valeur inférieure.

Dans une variable `max`, on place le premier élément contenu dans le tableau, puis on parcourt le reste du tableau, en comparant chaque élément du tableau avec `max`, et on remplace si besoin par la valeur supérieure.

```
fonction minimum  (DON = nbE : entier, tab : tableau[tailleMax] d'entiers) :
                  entier
      variable min : entier
début
      min ← tab[0]
      pour i ← 1 à nbE – 1 faire
            si (tab[i] < min) alors
                  min ← tab[i]
      return min
fin
```

Recherche d'un élément

Principe

On parcourt toutes les cases vides du tableau en comparant le contenu de la case avec l'élément recherché. Le parcours s'arrête soit quand on a trouvé l'élément cherché, soit quand toutes les cases non vides ont été parcourues.

La recherche peut être une appartenance, c'est-à-dire que le sous-programme retourne vrai ou faux, ou une position. Dans ce cas-là, le sous-programme renvoie une position non valide pour le tableau si l'élément recherché ne s'y trouve pas.

```
fonction recherche (DON = nbE : entier, tab : tableau[tailleMax] d'entier, val :
                    entier) : booléen
      variable i : entier
début
      i ← 0
      tant que (i < nbE) et (tab[i] ≠ val) faire
            i ← i + 1
      return (i < nbE)
fin
```

B. Traitement sur les vecteurs ordonnés

Ajout d'un élément

Principe

On parcourt le vecteur en partant de la fin. Soit `i` la position courante :

- Si `tab[i]` > l'élément à ajouter on décale `tab[i]` à d'une case vers la droite
- Sinon on ajoute l'élément dans `tab[i+1]`

Remarque

L'ajout n'est possible que s'il reste de la place dans le tableau

```
procédure ajout  (DON = val : entier ; DONRES = nbE : entier, tab :
                  tableau[taille Max] d'entiers)
      variable i : entier
début
      si (nbE < tailleMax) alors
            i ← nbE – 1
            tant que (i ≥ 0) et (val < tab[i])
                  tab[i+1] ← tab[i]
                  i ← i – 1
            tab[i+1] ← val
            nbE ← nbE + 1
fin
```

Suppression d'un élément en position p

Principe

- On décale les éléments des cases non vides (dont l'indice est supérieur à p) d'une position vers la gauche
- On diminue nbE de 1

Remarque

Il n'est possible de supprimer l'élément en position p que si p correspond à l'indice d'une case non vide du tableau.

```
procédure suppression  (DON = p : entier, DONRES = nbE : entier, tab :
                        tableau[tailleMax] d'entiers)
variable i : entier
début
      si (0 ≤ p) et (p < nbE) alors
             pour i ← p à nbE – 2 faire
                  tab[i] ← tab[i+1]
            nbE ← nbE – 1
fin
```

Recherche d'un élément

Principe

- On parcourt toutes les cases non vides du tableau en comparant le contenu de la case avec l'élément recherché
- Le processus s'arrête :
  - Quand on a trouvé l'élément cherché
  - Quand on a trouvé un élément plus grand
  - Quand on a parcouru toutes les cases non vides du tableau

La recherche peut être :

- Appartenance : le sous-programme renvoie vrai ou faux
- Recherche de la position : si l'élément n'est pas dans le tableau, le sous-programme renvoi une position non valide pour le tableau

```
fonction appartient    (DON = val : entier, nbE : entier, tab :
                       tableau[tailleMax] d'entiers) : booléen
      variable i : entier
début
      i ← 0
      tant que (i < nbE) et (val < tab[i]) faire
            i ← i + 1
      retourner ((i < nbE) et (tab[i] = val))
fin
```

Recherche dichotomique

- Uniquement pour les vecteurs ordonnés
- Plus efficace que la recherche classique

Principe dans l'intervalle `[deb … fin]`

- Calculer le milieu de l'intervalle de recherche
- Comparer l'élément recherché avec `tab[milieu]`
  - Si `tab[milieu]` = élément recherché → recherche s'arrête
  - Si `tab[milieu]` > élément recherché → recherche se poursuit sur l'intervalle `[deb … milieu-1]`
  - Si `tab[milieu]` < élément recherché → recherche se poursuit sur l'intervalle `[milieu+1 … fin]`
- Recommencer 1 et 2 tant que l'intervalle de recherche existe et que l'on n'a pas trouvé l'élément

```
fonction dicho    (DON = val : entier, nbE : entier, tab : tableau[tailleMax]
                  d'entiers) : booléen
      variable deb, fin, milieu : entier
             trouve : booléen
début
      i ← 0
      fin ← nbE – 1
      trouve ← faux
      tant que (deb ≤ fin) et non(trouve) faire
             milieu ← (deb + fin) div 2
             si (tab[milieu] = val) alors
                    trouve ← vrai
             sinon
                    si (tab[milieu] > val) alors
                           fin ← milieu – 1
                    sinon
                           deb ← milieu + 1
      retourner (trouve)
fin
```

VII. Trier un tableau

Processus de rangement des éléments d'un tableau selon un certain ordre, croissant ou décroissant. Cela permet de faciliter la recherche d'un élément dans le tableau.

Un problème de tri est formulé par :

- Entrée : séquence de n éléments stockés dans un tableau : $<e_1, e_2, \dots, e_n>$
- Sortie : permutation $<e'_1, e'_2, \dots, e'_n>$ de la séquence d'entrée telle que $e'_1 \leq e'_2 \leq \dots \leq e'_n$

Méthodes de tri :

- Simples : efficaces quand le nombre d'éléments à trier est petit
- Complexes : plus adaptées quand le nombre d'éléments à trier est grand

A. Tri par sélection

Principe

- Déterminer le plus petit élément du tableau
- L'échanger avec l'élément qui est en première position dans le tableau
- Recommencer avec le reste du tableau (sous-vecteur allant de 1 à `nbE – 1`)

Algorithme général

```
début
      pour i ← 0 à nbE – 2 faire
             rechercher le minimum
             échanger le minimum avec tab[i]
fin
```

B. Tri par bulle

Principe

- Comparer deux éléments consécutifs du tableau
- Les échanger s'ils ne sont pas dans l'ordre
  - Le plus grand élément « remonte » à la fin du tableau
- Recommencer avec le reste du tableau (sous-vecteur de 0 à `nbE – 1`)

Algorithme général

```
début
	pour i ← 0 à nbE – 1 bas 1 faire
		comparer 2 à 2 les éléments consécutifs du sous-vecteur [0 … 1] et
		les échanger s’ils ne sont pas dans l’ordre
fin
```

C. Tri par insertion

Principe

- Considérer un sous-vecteur ordonné réduit à un seul élément (`tab[0]`)
- Insérer `tab[1]` dans ce sous vecteur pour obtenir un sous-vecteur ordonné de deux éléments
- Recommencer le même processus pour obtenir un sous-vecteur ordonné de nbE éléments

Algorithme général

```
début
	pour i ← 1 à nbE – 1 faire
		insérer tab[i] dans le sous-vecteur [0 … i – 1]
fin
```

VIII. Temps d’exécution

Un algorithme répond à un problème. Plusieurs algorithmes peuvent répondre à un même problème. Ainsi, il faut pouvoir les comparer pour savoir lequel est le plus efficace.

A. Comparaison des algorithmes

- Utilisation d’une mesure : *complexité en temps* ou *coût*
- Temps : difficile à déterminer avec exactitude
  dépend de l’ordinateur (notamment du nombre d’opérations par seconde qu’il peut exécuter.

⇨ On détermine un ordre de grandeur pour l’exécution.

B. Temps de calcul d’un programme

- Varie en fonction de la taille n des données manipulées par l’algorithme
- Correspond au nombre d’opérations élémentaires effectuées par l’algorithme (à un facteur multiplicatif contant près) sur ces données

On définit donc cet ordre de grandeur par une fonction mathématique : $T(n)$

⇨ Analyser un algorithme consiste à évaluer le comportement asymptotique de la fonction $T(n)$.

Approximation O

$T(n)$ est de l’ordre de $O\big(f(n)\big)$. On ne considère que le terme dominant de $T(n)$

Ex : $T(n) = 3n^2 + 2n + 5$ est en $O(n^2)$

Quelques temps d’exécution

$O(1)$	→	temps constant
$O(\log(n))$	→	temps logarithmique
$O(n)$	→	temps linéaire
$O(n\log(n))$	→	temps quasi-linéaire
$O(n^2)$	→	temps quadratique
$O(n^3)$	→	temps cubique
$O(n^k)$ avec k > 3	→	temps polynomial
$O(2^n)$	→	temps exponentiel

A titre d'exemple

Si un traitement élémentaire prend un millionième de seconde

Valeur de n	$T(n) = n$	$T(n) = n^2$
10	0.01 ms	0.1 ms
100	0.1 ms	10 ms
1000	1 ms	1 s
10^4	10 ms	100 s
10^5	100 ms	3 heures
10^6	1 s	10 jours

Calculer le nombre d'opérations élémentaires

- Opérations élémentaires
  - Affectations, comparaisons, opérations arithmétiques/logiques, lecture, écriture …
  - Leur exécution prend un temps constant
- Structures de contrôles
  - Séquences : somme des coûts de chaque instruction
  - Instruction conditionnelle : coût (condition) + max (coût(seqInstrAlors, seqInstrSinon)
  - Boucle : nbreItérations × coût (seqInstrBoucle)

Ne pas oublier le coût des sous-programmes !

Exemples

```
début
      i ← 0
      tant que (i < n) et (𝑥 ≠ tab[i]) faire
            i ← i + 1
      si (i < n) faire
            écrire("∈")
      sinon
            écrire("∉")
fin
```

```
début
      lire (n)
      pour i ← 0 à n – 1 faire
            pour j ← 0 à n – 1 faire
                  A[i, j] ← 0
      pour i ← 0 à n – 1 faire
            A[i, i] ← 0
fin
```