Partie I

Procédures & fonctions

Sous programmes

- Définition

Un sous-programme est un traitement particulier appelé à s'exécuter à l'intérieur d'un autre programme

- Utilité :
  - **quand un même traitement doit être réalisé plusieurs fois** dans un programme. On écrit un sous-programme pour ce traitement et on l'appelle à chaque endroit où l'on en a besoin
  - **pour organiser le code** , améliorer la conception et la lisibilité des gros programmes .

Fonctions et procédures

- 2 sortes de sous-programmes :
  - les ***fonctions***
  - les ***procédures***
- L'appel d'une fonction est une expression, tandis que l'appel d'une procédure est une instruction :
  - une **fonction renvoie un résultat**
  - une **procédure ne renvoie rien**

Fonctions et procédures

- 2 sortes de sous-programmes :
  - les ***fonctions***
  - les ***procédures***
- L'appel d'une fonction est une expression, tandis que l'appel d'une procédure est une instruction :
  - une **fonction renvoie un résultat**
  - une **procédure ne renvoie rien**

SMI/S3 - Algorithmique 2 3

L’utilisation des procédures et des fonctions

L'utilisation d'une fonction ou d’une procédure nécessite trois parties :

- **le prototype** : c’est la déclaration nécessaire qui se place avant la fonction principale;
- **l’appel** : c’est l’utilisation d’une fonction à l’intérieur d’une autre fonction (par exemple le programme principal) ;
- **la déclaration** : c’est l’écriture proprement dite de la fonction ou procédure, en-tête et corps.

Procédures

- Une procédure est un ensemble d'instructions regroupées sous un nom, qui réalise un traitement particulier dans un programme lorsqu'on l'appelle
- Une procédure est un sous-programme qui exécute un certain nombre d’actions sans fournir de valeur de retour après son exécution.

Procédures

- Comme un programme, une procédure possède
  - un nom
  - des variables
  - des instructions
  - un début
  - une fin
- Tout comme un programme, l'écriture d'une procédure requiert un en-tête et un corps

Définition d'une procédure

Définir une procédure consiste à :

- Définir l'**ENTETE de la procédure**

  a. IDENTIFIER LA PROCEDURE
  b. DECLARER LA LISTE DES PARAMETRES

- Définir le **CORPS de la procédure :**
  a. DECRIRE L'ALGORITHME : constantes et variables, actions

Syntaxe d'une procédure sans paramètres

Syntaxe :

```
Procédure nomProcedure( )                    } En-tête
Var var1 : type,...,varN: type
Début
instruction 1
instruction 2
instruction 3                          Corps
.
.
.
instruction n
FinProcédure
```


Appel d'une procédure sans paramètres

Pour déclencher l'exécution d'une procédure dans un programme, il suffit de l'appeler :

indiquer son nom suivi de parenthèses

```
ALGORITHME test
VAR       Entier1,Entier2 : entier

DEBUT
instuctions
nomProcedure()
instuctions
FIN
```

Exemple

```
PROCEDURE Afficherchoix ()
DEBUT

ECRIRE("choix 1 : calculer la surface d'un rectangle")
ECRIRE("choix 2 : calculer la surface d'un cercle")
ECRIRE("choix 3 : quitter")

FINPROCEDURE
```


Syntaxe d'une procédure avec paramètres

Syntaxe :

```
Procédure nomProcedure(paramètres:type)      } En-tête
Var var1 : type,...,varN: type               )
Début                                        )
instruction 1                                )
instruction 2                                )
instruction 3                                )     Corps
.                                            )
.                                            )
.                                            )
instruction n                                )
FinProcédure
```

Appel d'une procédure avec paramètres

Pour appeler une procédure avec un paramètre, il faut mettre la valeur du paramètre entre parenthèses

```
ALGORITHME monAlgo
VAR        Entier1 : entier
           Chaine : chaîne
DEBUT
instuctions
nomProcedure(Entier1)
nomProcedure(3)
instuctions
FIN
```


Exemple

DEFINITION

```
PROCEDURE AfficherSomme(VAR pNb1 : ENTIER, VAR pNb2 :
ENTIER)
DEBUT
ECRIRE("la somme est ", (pNb1 + pNb2) )
finPROCEDURE
```

Appel

```
. . .
LIRE(n1, n2)
AfficherSomme(n1, n2)
. .
```

Les fonctions - Définition

• Une fonction est un sous-programme retournant un et un seul résultat au programme appelant

• Les fonctions sont appelées pour récupérer une valeur (alors que les procédures ne renvoient aucune valeur)

• L'appel des fonctions est différent de l'appel des procédures :

L'appel d'une fonction doit **obligatoirement** se trouver à l'intérieur d'une instruction qui utilise sa valeur
Le résultat d'une fonction doit obligatoirement être retourné au programme appelant par l'instruction Retourne

Les fonctions - Syntaxe

```
Fonction nomFonction(par1:type,...,parN:type):type   } En-tête
//déclaration des variables                          }
Var Valeur : type                                    }
Début                                                }
instructions/*par exemple valeur← par1+par2 */       } Corps
Retourne valeur                                      }
/*ou bien Retourne par1+par2 */                      }
FinFonction                                          }
```

Remarque:

valeur est le nom d'une variable ou constante, ou correspond à une expression

Les fonctions - Exemple

Protype et définition:
FONCTION CalculerDuree (AnDeb : ENTIER, AnFin : ENTIER) : ENTIER
VAR duree **: ENTIER**

DEBUT

duree ⟵ AnFin – AnDeb
RETOURNE duree

FINFONCTION

Appel:

```
. . .
LIRE(anNais, anCour)
ECRIRE("vous avez ", CalculerDuree(anNais,anCour), " ans")
. . .
```


Variables locales/ Variables globales

- Les variables déclarées dans un programme.

- On peut alors qualifier la variable ou la constante de GLOBALE afin de signifier qu'elle devient accessible directement par les sous-programmes, sans nécessité de la passer en paramètres.

- **CETTE PRATIQUE EST CEPENDANT DECONSEILLEE : ELLE PEUT CONDUIRE A DES EFFETS NON CONTROLES (dits « effets de bord »).**

Paramètres Et Arguments

Définition

• Un paramètre est une variable particulière qui sert à la communication entre programme appelant et sous-programme et qui a un nom et un type

Paramètres Et Arguments

Il est primordial de bien distinguer entre:

• Les paramètres formels qui se trouvent dans l'en-tête d'une procédure ou fonction lors de sa définition

• Les paramètres effectifs ou paramètres réels (ou arguments) désignent les valeurs réellement fournies lors de l'appel du sous-programme et qui sont placés entre parenthèses lors de l'appel

Paramètres formels

- placés dans la **définition** d'une procédure
- Servent à **décrire** le traitement à réaliser par la procédure indépendamment des valeurs traitées
- Ce sont des variables locales à la procédure
- Ils sont déclarés dans l'entête de la procédure

Paramètres effectifs

- placés dans **l'appel** d'une procédure
- Lors de l'appel, leurs valeurs sont transmises aux paramètres formels correspondants
- Un paramètre effectif en donnée peut être
  soit une variable du programme appelant
  soit une valeur littérale
  soit le résultat d'une expression

Modes de passage

➢Les modes de passage des paramètres définissent comment les valeurs sont passées de le programme principal (ou d'un sous-programme) au sous-programme appelé.

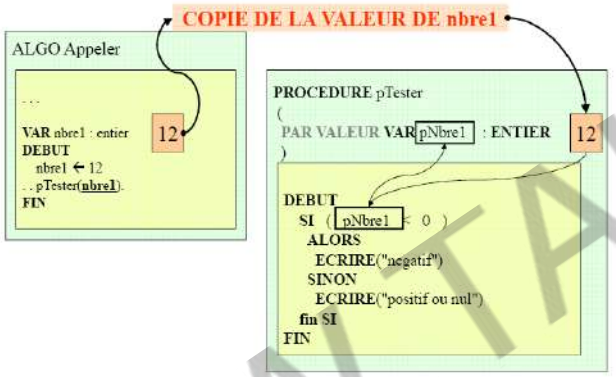
➢Le mode de passage est précisé à la déclaration de chacun des paramètres.

Passage par valeur

➢ La valeur du paramètre effectif est copié dans le paramètre formel. Le paramètre effectif ne subit aucun changement.

➢Le **passage par valeur est l'option par défaut dans la déclaration des** paramètres.

Passage par valeur(Exemple)



Passage par Référence

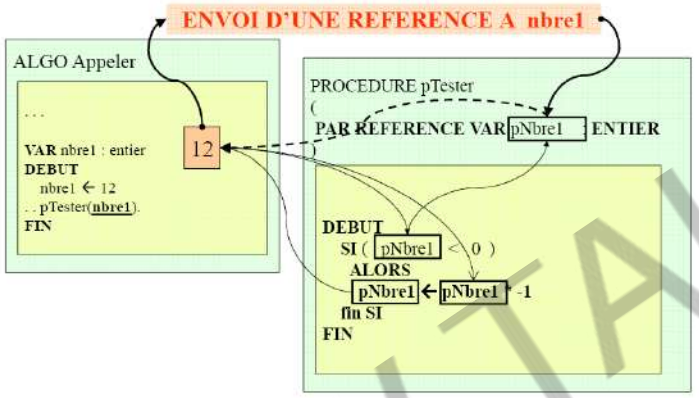
➢ toute modification sur le paramètre formel modifie le paramètre effectif. Ce dernier reçoit le résultat.

➢Dans le **PASSAGE PAR REFERENCE, le PARAMETRE FORMEL REÇOIT UNE** REFERENCE VERS LE CONTENU DU PARAMETRE Effectif : **il devient ainsi comme un alias d'un paramètre réel.**

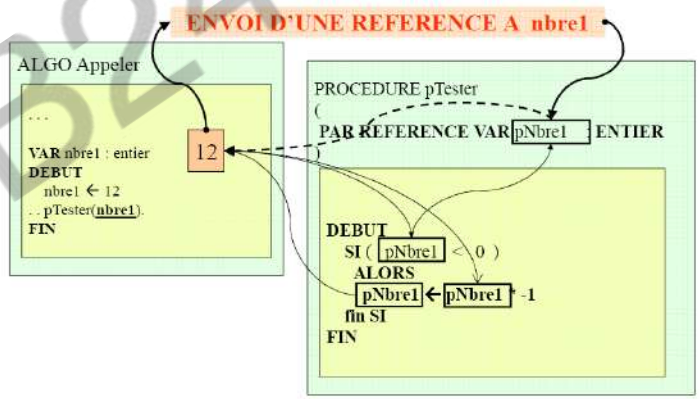
➢**Toute action effectuée sur le paramètre formel est ainsi reportée directement sur la valeur du paramètre réel.**

➢Dans ce cas, **LES VALEURS D'ORIGINE PEUVENT ETRE MODIFIEES PAR LES INSTRUCTIONS DU SOUS-PROGRAMME (sauf si l'argument est défini comme CONST** au lieu de VAR).

Passage par Référence(Exemple)



Passage par Référence(Exemple)



Modes de Passage: Exemples

```
Soit la fonction suivante:
   Fonction incremente( i:entier):
Début
     i ← i+1
     retourne i
 Fin fonction
...et on va appeler la fonction incremente
Algorithme TestPassageValeur
Var i, j:entier
Début
      i ← 0
      j← incremente(i);
     Ecrire("i=" + i);
     Ecrire("j=" + j);
  Fin
```

Quelles seront les valeurs affichés par l'algorithme pour i et j après exécution?

Modes de Passage: Exemples

Quand nous exécutons TestPassageValeur

i=0

j=1

... donc la valeur de i n'a pas été modifiée !

Modes de Passage: Exemples

On modifie la fonction comme suit:

```
   Fonction incremente( Par reference i:entier):
Début
      i ← i+1
      retourne i
 Fin fonction
...et on va appeler la fonction incremente
Algorithme TestPassageValeur
Var i, j:entier
Début
      i ← 0
      j ← incremente(i);
      Ecrire("i=" + i);
      Ecrire("j=" + j);
  Fin
```

Quelles seront les valeurs affichés par l'algorithme pour i et j après exécution?

Modes de Passage: Exemples

Quand nous exécutons TestPassageValeur

i=1

j=1

... donc la valeur de i a été modifiée !

Exercices d'application

Soit une matrice carrée. Ecrire l'algorithme qui permet de faire la somme de la diagonale de cette matrice

Exercice d'application

```
Type mat=tableau (50,50):entier

Procedure lecture ( L :mat)
Debut
Pour i ⟵ 0 à 49
Pour j ⟵ 1 à 49
Ecrire ("L (", i, j, " )")
Lire (L(i,j))
Fin Pour
Fin
```

```
Fonction somme (A : mat) :entier
Variable D : entier
Debut
D ⟵ 0
Pour i ⟵ 0 à 49 Faire
D ⟵ D+A(i,i)
Fin Pour
Retourne D
Fin
```

Exercices d'application

Algorithme somme_diagonale
Var i, j, :entier
L :mat

Début

Lecture (L)
Ecrire (" la somme de la diagonale de la matrice est ", somme (L))

Fin

Partie II

Récursivité

Récursivité

- Définition

La **RECURSIVITE désigne la démarche utilisant l'appel d'une fonction par elle-même afin de résoudre un problème calculatoire.**

- Remarque
  - Une fonction récursive doit contenir une condition terminale, spécifiant un état où l'appel récursif se termine.
  - Si la condition terminale n'est pas spécifiée l'algorithme ne s'arrête jamais.

Exemple d'une fonction récursive

Ecrire une fonction factorielle(n) qui prend un entier n puis retourne n!

Méthode non récursive

```
Fonction factorielle(n: entier) : entier
        Variable i, fact : entier
        Début
                j ← 1
                pour i = 1 à n
                        fact ← fact * i
                FinPour
                Retourne fact
        FinFonction
```

Exemple d'une fonction récursive

Or on sait que:

n! = 1 si n=0 ou n=1

n! = n * (n-1) ! Si n > 1

D'où : Méthode récursive

```
Fonction factorielle(n : entier) : entier
        Variable fact : entier
        Début
                Si n = 0 ou n = 1 Alors
                        fact ← 1
                FinSi
                Sinon
                        fact ← n * factorielle(n-1)
                FinSinon
                Retourne fact
        Finfonction
```


Procédures récursives

- Les procédures peuvent elles aussi être définies d'une manière récursive.

- Exemple:

écrire une procédure récursive qui permet d'afficher les éléments d'un tableau de 10 entiers.

- Idée:

si l'élément à afficher est le dernier dans le tableau alors l'écrire et s'arrêter

Sinon, écrire l'élément courant et afficher à partir du suivant

Exemple de procédure récursive

```
Type Tab= Tableau(10) entiers
Procédure Afficher(i: entier, T:Tab)
        Début
                Ecrire( T(i) )
                Si i < 10 Alors
                        Afficher( i+1, T)
                FinSi
        FinProcédure
```

Remarque:

Quand on fait dans l'algorithme :

Afficher(4, T)

Cela veut dire qu'on veut afficher à partir de la 4ème case

Exercices d'application

Écrire une fonction récursive qui calcule la puissance d'un nombre réel

Exercices d'application

Solution:

```
fonction puissance ( x:réel, n:entier ) : réel
Début
Si n=0
retourne  1
si n=1
retourne  x
Si n > 1
retourne x* puissance(x,n-1)
FinFonction
```


Exercices d'application

Ecrire de manière récursive une fonction qui donne le plus grand commun diviseur (pgcd) de deux entiers.

Trouver les relations de récurrence en utilisant une méthode de détermination du pgcd basée sur la soustraction

Exercices d'application

Solution:

Fonction PGCD (m, n : entier): entier
P: entier
Début
Si m = n Alors
P ⟵ n
Sinon
Si m>n alors
P ⟵ PGCD (m-n, n)
Sinon
P⟵ PGCD (n, n-m)
Fin si
Fin si
Retourne P
Fin

Exercices d'application

Reproduire l'exercice du calcul du nombre d'occurrences d'un élément donné dans un tableau sous la forme d'une fonction récursive.

Exercices d'application

Solution:

```
Fonction Occurence (i:entier, T:tableau entier, n:entier, X:entier):Entier
Var  Occ: entier
Début
Si i=n-1
      Si T(i)=X
      Occ⟵1
      Sinon
      Occ ⟵ 0
      Finsi
Sinon
      Si (T(i)=X)
        Occ⟵1 + Occurence(i+1,T,n,X)
      Sinon
        Occ⟵ Occurence(i+1,T,n,X)
      FinSi
  FinSi
Retourne Occ
FinFonction
```


Remarque: dans l'appel initialiser i à 0

Exercices d'application

On appelle palindrome une chaîne de caractère qui donne la même chaîne selon que l'on la lise de gauche à droite ou inversement. Autrement dit, le premier caractère est égal au dernier caractère, le deuxième caractère est égal à l'avant dernier caractère, etc.

Une définition récursive d'un palindrome est:
· La chaîne vide est un palindrome.
· La chaîne constituée d'un seul caractère est un palindrome.
· aXb est un palindrome si a = b et si X est un palindrome.

Ecrire une fonction récursive qui teste si une chaîne de caractères et qui renvoie vrai si elle palindrome et faux si elle ne l'est pas.

Exercices d'application

Solution:

```
Fonction palindrome(s: chaine):booléen
Var palin: booléen
début
si (longueur(s) = 0 ou longueur(s) =1)
palin ⟵ VRAI
sinon
      si (caractère(s,0) = caractère(s,longueur(s)-1)
      palin⟵ palindrome(sousChaine(s,1, longueur(s)-2)
      sinon
      palin  ⟵ FAUX
      Finsi
Finsi
Retourne palin
finfonction
```


Partie III

Enregistrements & Fichiers

Fichier

- Définition

Un FICHIER (anglais : file) est un REGROUPEMENT LOGIQUE DE DONNEES MEMORISEES SUR UN SUPPORT PERMANENT (disque dur, par exemple) afin de permettre une réutilisation ultérieure des informations qu'il contient.

Enregistrement

Définition

- Un enregistrement est un bloc de données élémentaires qui décrit une entité.
- Un enregistrement (*anglais : record*) est composé de plusieurs champs.
- Un champ est l'une des données élémentaires d'un enregistrement

Exemple:

Dans un fichier « Clients », un enregistrement correspond aux données relatives à un client.
le numéro de client, le nom, le prénom, l'adresse..., sont des champs d'un enregistrement du fichier client.

Organisation des données dans un fichier

➢L'organisation des données dans un fichier détermine comment seront placés chacun des enregistrements

➢ On cite trois types d'organisations:

- **Organisation séquentielle**
- **Organisation calculée**
- **Organisation indexée**

Organisation séquentielle

➢ Dans des fichiers séquentiels, les enregistrements sont mémorisés consécutivement dans l'ordre de leur entrée et peuvent seulement être lus dans cet ordre.

➢Si on a besoin d'un enregistrement précis dans un fichier séquentiel, il faut lire tous les enregistrements qui le précèdent, en commençant par le premier.

Organisation calculée

Dans des fichiers à placement calculé, les enregistrements sont placés à une position précise du fichier. Cette position est:

- Soit donnée comme numéro d'ordre de l'enregistrement dans le fichier,
- Soit calculée selon un algorithme de placement appliqué à une clef. (La clef correspond à l'un des champs de l'enregistrement permettant l'identification d'un enregistrement unique au sein du fichier)

Organisation indexée

Dans des fichiers à organisation indexée, au moins 2 fichiers sont nécessaires pour chaque fichier géré :

- le premier fichier contient les données
- le second fichier contient une table d'index qui à une valeur d'une clef d'un enregistrement conserve la position du fichier à laquelle il se trouve.

Modes d'accès aux enregistrements d'un fichier

- L'accès aux enregistrements d'un fichier est déterminé par l'organisation de ce fichier.
- On distingue deux modes d'accès:
  - **Accès séquentiel**
  - **Accès direct**

Accès séquentiel

- L'accès séquentiel consiste à parcourir, dans l'ordre dans lequel ils sont stockés, les enregistrements d'un fichier, sans retour arrière possible.
- Pour accéder à un enregistrement, il faut avoir lu tous les enregistrements qui le précèdent.

Accès direct

L'accès direct permet l'accès individuel à chacun des enregistrements d'un fichier, en y accédant directement :

- Soit grâce à un numéro d'ordre de placement ;
- Soit grâce à une clef.

Algorithmique – fichiers séquentiels

Le langage algorithmique (*et la plupart des langages de programmation*) met à la disposition du programmeur un ensemble d'instructions permettant la manipulation des fichiers.

Algorithmique – fichiers séquentiels

➢ **Déclaration d'une variable de type FICHIER**

VAR fichier : FICHIER

➢ **Lire un fichier**

LIRE_FICHIER(fichier, liste_des_variables)

➢ **Tester la fin du fichier**

FF(fichier)

retourne VRAI si la fin du fichier a été atteinte (il n'y a plus d'enregistrements à lire)

Algorithmique – fichiers séquentiels

➢ **Ouvrir un fichier séquentiel**

fichier ⟵ OUVRIR(nom_du_fichier ,mode_d'Ouverture)

Les modes possibles sont: LECTURE, ECRITURE,AJOUT LECTURE/ECRITURE

Mode d'ouverture	*Actions autorisée*
Lecture	En mode lecture, seules seront autorisés l'accès aux données, sans modification possible du contenu
Ecriture	En mode écriture, le fichier est vidé de son contenu, et ne sera possible que l'écriture de nouveaux enregistrements
Ajout	En mode ajout, les données présentes dans le fichier seront préservées, et il sera possible d'ajouter de nouveaux enregistrements
Lecture/écriture	En mode lecture/écriture, les données présentes dans le fichier seront préservées, et il sera possible de modifier le contenu de certains enregistrements.

Algorithmique – fichiers séquentiels

➢**Ecrire dans un fichier**
ECRIRE(fichier, liste de variables)

➢ **Fermer un fichier**
FERMER(fichier)

Algorithmique – fichiers séquentiels

Exemple:

```
VARIABLES
f1, f2 : FICHIER
num, nom, prenom, email : CHAINE
DEBUT
f1 ← OUVRIR("fichier1.txt" , LECTURE)
f2 ← OUVRIR("fichier2.txt" , ECRITURE)
// première lecture
LIRE_FICHIER(f1, num, nom, prenom, email)
TANTQUE (NON FF(f1))
ECRIRE_FICHIER(f2, num, nom, prenom, email)
// autres lectures
LIRE_FICHIER( f1, num, nom, prenom, email)
FINTANTQUE
FERMER(f2)
FERMER(f1)
FIN
```

SMI/S3 - Algorithmique 2 2

Algorithmique – fichiers séquentiels

Exercice:

On souhaite mémoriser des noms des personnes dans un fichier nommé « personne.txt », créer les sous-programmes qui suivent :

- Une procédure de création du fichier qui contient les noms des personnes.
- Une procédure d'affichage des noms de personnes.
- Une fonction qui permet de chercher un nom passe en argument et qui renvoie vrai si ce dernier est existant et faux sinon.

Ecrire le programme principal faisant appel aux différents sous-programmes.

Algorithmique – fichiers séquentiels

```
Procedure Creation(fn : Fichier)
Var  n : chaine , rep : caractere
Debut
  fn ← Ouvrir(personne.txt,ECRITURE)
Rep ←'O'
Tant que (rep = 'O') Faire
Ecrire("entrer un Nom : "),
 Lire(n)
Ecrire_Fichier(fn,n)
Ecrire("Voulezvous ajouter un autre nom (O/N) : ")
Lire(rep)
Fin Tant que
Fermer(fn)
FinProcedure
```

Algorithmique – fichiers séquentiels

```
Procedure Affichage(fn : Fichier)
Var n : chaîne
Debut
 fn ← Ouvrir(personne.txt,LECTURE)
Lire_Fichier(fn,n)
Tant que NON(FF(fn)) Faire
Ecrire(n)
Lire_Fichier(fn,n)
Fin Tant que
Fermer(fn)
FinProcedure
```


Algorithmique – fichiers séquentiels

```
Fonction Recherche(x : chaine ; fn : Fichier) : Booleen
Var n : chaine, Trouve : Booleen
Debut
  fn← Ouvrir(personne.txt,LECTURE)
Lire(fn,n)
Trouve ← (n = x)
Tant que( (Trouve=faux) ET (NON(FF(fn))) Faire
Lire(fn,n)
Trouve ← (n = x)
Fin Tant que
Si (FF(fn)) Alors
Retourne faux
Sinon
Retourne vrai
Fin Si
Fermer(fn)
FinFonction
```

Algorithmique – fichiers séquentiels

```
Algorithme personne
Var F1:Fichier
Debut
Creation(F1)
Affichage(F1)
Si Recherche("Riadh",F1) Alors
Ecrire("Riadh est existant dans le fichier")
Sinon
Ecrire("Riadh est non existant dans le fichier")
Fin Si
Fin
```


Partie IV

La complexité

Introduction à la complexité

- Il existe souvent plusieurs façons de programmer un algorithme.
- le choix de la solution s'impose lorsque le nombre d'opérations et la taille des données d'entrée sont importants.
- Deux paramètres sont déterminants : le temps d'exécution et l'occupation mémoire.

Introduction à la complexité

La notion de complexité décrit le temps et la mémoire nécessaires pour exécuter un algorithme et permet donc de caractériser son efficacité.

SMI/S3 - Algorithmique 2 2

Types de complexités

Il existe deux types de complexités**: complexité temporelle** et **complexité spatiale.**

Définition 1: la complexité temporelle d'un algorithme est le temps mis par ce dernier pour transformer les données du problème considéré en un ensemble de résultats.

Définition 2: la complexité spatiale d'un algorithme est l'espace mémoire utilisé par ce dernier pour transformer les données du problème considéré en un ensemble de résultats.

Types de complexités

Remarque: Le coût de la mémoire étant aujourd'hui relativement faible, on cherche en général à améliorer la complexité en temps plutôt que la complexité en mémoire.

Dans ce qui suit, nous allons nous concentrer beaucoup plus sur le temps d'exécution

Types de complexités

Remarque: Le coût de la mémoire étant aujourd'hui relativement faible, on cherche en général à améliorer la complexité en temps plutôt que la complexité en mémoire.

Dans ce qui suit, nous allons nous concentrer beaucoup plus sur le temps d'exécution

Exemple:Somme des n premiers entiers

- Calcul à l'aide d'une boucle :

Entrées: entier naturel n
Sorties: entier naturel somme
Somme ← 0
i ← 1
tant que (i<=n) faire
Somme ← somme+i
i ← i+1
retourner somme

Coût(A1) : 2n additions

Exemple: Somme des n premiers entiers

- Calcul à l'aide de la formule mathématique: $\sum_{1}^{n} i = \frac{n(n+1)}{2}$

Entrées: entier naturel n
Sorties: entier naturel somme
Somme ← n+1
Somme ← somme*n
Somme ← somme/2
retourner somme

Coût(A2) : 1 addition + 1 multiplication + 1 division

Exemple: Somme des n premiers entiers

Comparaison du coût des deux algorithmes en terme d'opérations arithmétiques :

- Coût(A1) : 2n opérations arithmétiques.
- Coût(A2) : 3 opérations arithmétiques.

Pour n > 1, Coût(A2) < Coût(A1).

L'algorithme A2 est plus efficace que l'algorithme A1.

Evaluation de la complexité d'un Algorithme

❖ Le temps d'exécution dépend de la longueur de l'entrée.

❖Ce temps est une fonction T(n) où n est la longueur des données d'entrée.

Evaluation de la complexité d'un Algorithme

❖ Pour l'évaluer, il faut choisir une mesure élémentaire :
- ➢ nombre de comparaisons,
- ➢ nombre d'affectations,
- ➢ nombre d'opérations arithmétiques,

. . .

❖ La complexité algorithmique s'exprime en fonction de la taille n des entrées.

❖ **Il n'y a pas un ensemble de règles standardisé pour évaluer la complexité d'un algorithme.**

Quelques Règles d'évaluation de la complexité d'un Algorithme

➢ **une séquence d'instructions $x_1;x_2;$ .....;x_n**

$$Coût(x_1;x_2; .....;x_n) = \sum Coût(x_k)$$

Exemple

somme ← n+1
Somme ← somme*n
Somme ← somme/2

Coût(A2) = coût(X1)+ coût(X2) + coût(X3)= 3

Quelques Règles d'évaluation de la complexité d'un Algorithme

➢ **La boucle simple : Tant que (i < n) faire x_i**

$$Coût(boucle) = \sum_i (Coût(comparaison) + Coût(x_i))$$

Exemple

tant que (i<=n) faire
somme ← somme+i
i ← i+1
fintantque

Coût(A) = Coût(i <= n)+Coût(somme+i) + Coût(i ← i + 1) = 3n

Quelques Règles d'évaluation de la complexité d'un Algorithme

➢ **Les instructions conditionnelles :**
Si condition faire X_{vrai} sinon faire x_{faux}

Coût(conditionnelle)<=Coût(test)+ Sup(Coût(x_{vrai});Coût(xfaux))
Exemple :
si (i/2=0)alors n ← i/2
sinon
i ← i+1
n← i/2
Coût(A)<=Coût(i/2 = 0)+Sup(Coût(n← i/2);Coût(i← i + 1;n ←i/2))
Coût(A) <= 3

Types de complexités

Soit:

- d : une entrée de taille n,
- D_n : l'ensemble des entrées d possibles,
- Coût A(d) : le coût de l'algorithme A pour l'entrée d.

On peut alors calculer trois complexités différentes :

- la complexité dans le pire des cas,
- la complexité dans le meilleur des cas,
- la complexité moyenne.

Types de complexités

Complexité dans le pire des cas:

$$Max_A = Sup(Coût\ A(d) / d \in D_n)$$

➢ C'est cette complexité qu'on utilise le plus souvent pour comparer deux algorithmes.

Types de complexités

Complexité dans le meilleur des cas:

$$Min_A = Inf\ (Coût\ A(d) / d \in D_n)$$

Types de complexités

Complexité moyenne:

$$Moy_A = \sum_{d \in Dn} p(d) Coût\ A(d)$$

➢ La plus satisfaisante mais très difficile à calculer par méconnaissance de la distribution de probabilité suivi par les entrées.

Types de complexités

Exemple:

```
fonction rechercheElement(tab:tableau entier, x:entier): booléen
entier i;
début
i <- 0
tantque (i < n)
si (tab(i) = x) alors
retourne VRAI
finsi
fintantque
retourne FAUX
finfonction
```

Types de complexités

n = la taille du tableau, a = affectation d'entier, c = comparaison d'entier.

Complexité au pire (x n'est pas dans le tableau) : a+n*(2*c) = O(n)
Complexité au mieux (x dans la 1ére case du tableau): a+2*c = O(1)
Complexité en moyenne : considérons qu'on a 50% de chance que x soit dans le tableau, et 50% qu'il n'y soit pas, et, s'il y est sa position moyenne est au milieu.
Le temps d'exécution est [(a+n*(2*c))+(a+(n/2)*(2*c))] /2, de la forme a*n+b (avec a et b constantes) = O(n)

Types de complexités

Remarque:

Il est clair que pour certains algorithmes, il n'y a pas lieu de distinguer entre ces trois mesures de complexité.

Complexité asymptotique: Introduction

- Les algorithmes s'exécutent souvent sur des entrées de grande taille n.

- le temps précis d'exécution d'un programme n'est pas intéressant, mais **l'ordre de grandeur de ce temps en fonction de la taille des données ;**

- une approximation asymptotique de la complexité est suffisante.

Complexité asymptotique: Définition

Définition:

La complexité asymptotique d'un algorithme décrit le comportement de celui-ci quand la taille n des données du problème traité devient de plus en plus grande, plutôt qu'une mesure exacte du temps d'exécution.

Complexité asymptotique:Notation O

Définition:

Soient f et g : N → R+ : f = O(g) ssi ∃c ∈ R+ ; ∃n_0 ∈ N tels que:

$$\forall n > n_0; f(n) \leq c \quad g(n)$$

Utilité:

Le temps d'exécution est borné

Signification:

Pour toutes les grandes entrées (i.e., n >= n_0), on est assuré que l'algorithme ne prend pas plus de c x g(n) étapes. (borne supérieur)

Complexité asymptotique:Notation O

Les algorithmes s'exécutent souvent sur des entrées de grande taille n.

Conséquences :

- le coût exacte d'un algorithme n'est pas intéressant,
- une approximation asymptotique de la complexité est suffisante.

Complexité asymptotique:Notation O

échelle de fonction de complexité :

	Nom de la complexité
$O(1)$	constante
$O(log(n))$	logarithmique
$O(n)$	linéaire
$O(n^2)$	quadratique
$O(n^k)$	polynomiale
$O(2^n)$	exponentielle

Notation O: Règles de simplification

Si

$$f(n) = O(g(n))$$

et

$$g(n) = O(h(n)),$$

alors

$$f(n) = O(h(n)).$$

Notation O: Règles de simplification

Si

$$f(n) = O(kg(n))$$

où $k > 0$ est une constante,

alors

$$f(n) = O(g(n)).$$

Notation O: Règles de simplification

Si

$$f_1(n) = O(g_1(n))$$

et

$$f_2(n) = O(g_2(n))$$

alors

$$f_1(n)f_2(n) = O(g_1(n)\ g_2(n))$$

Complexité asymptotique:Exemples

Exemple 1:

T(n) = c. On écrit T(n) = O(1).

Exemple 2:

T(n) = $c_1n^2 + c_2n$.

$c_1n^2 + c_2n <= c_1n^2 + c_2n^2 = (c_1 + c_2)n^2$ pour tout n >= 1.

T(n) <= cn^2 où $c = c_1 + c_2$ et $n_0 = 1$.

Donc, **T**(n) est en O(n^2)

Complexité asymptotique:Exemples

Exemple 3:

Initialiser un tableau d'entiers

```
pour i←0 à n-1
    Tab[i]=0;
Finpour
```

Il y a n itérations

Chaque itération nécessite un temps ≤c, où c est une constante (comparaison + une affectation).

Le temps est donc T(n) ≤ cn

Donc **T(n) = O(n)**

Complexité asymptotique: Exemples

Exemple 4:

```
a = b;
```

Temps constant: O(1).

Exemple 5:

```
somme←0
Pour j←1 à n
  pour i←1 à n
  somme← somme+1
Finpour
Pour k ← 0 à n-1
  A[k] = k
Finpour
```

Temps: $O(1) + O(n^2) + O(n) = O(n^2)$

Complexité asymptotique: Exemples

Exemple 6:

```
somme←0
Pour i←1 à n
  j←1
  tant que (j<=i)
    somme← somme+1
    j←j+1
  Fin tantque
Finpour
```

Temps: $O(1) + O(n^2) = O(n^2)$

Complexité asymptotique: Exemples

Exemple 7:

```
somme ← 0
k ←1
tant que (k<=n)
  Pour j ←1 à n
     somme ←  somme+1
  Finpour
  k ← k*2
Fin tantque
```

Temps: $O(n\log_2 n)$

Complexité asymptotique: Exemples

Exemple 8: cas récursif

```
Fonction factorielle(n : entier) : entier
        Début
             Si n = 0 ou n = 1
                  Retourne 1
             Sinon
                  Retourne  n*fact(n-1)
             FinSi
        Finfonction
```

Complexité asymptotique: Exemples

Si n = 1, com(n) = c donc **la complexité est d'ordre** O (1)

On pose une équation de récurrence : appelons com(n) la complexité

com(n) = c + com(n-1) + o si n ≠ 1
com(1) = c

On résoud cette équation de récurrence :

com(n) = n*c + (n-1)*o = O(n)

Temps: O (n)

Autres notations: Ω et Θ

Grand Omega

Soient f et g : N → R+ , f (n) = Ω (g(n)) s'il existe C > 0 et no > 0 tels que:

$0 < Cg(n) \leq f(n)$ pour tout $n \geq no$

(Borne inférieur)

Grand Thêta

Soient f et g : N → R+ ,f (n) = Θ(g(n)) s'il existe C1et C2 > 0 et no > 0 tels que:

$C1g(n) \leq f(n) \leq C2g(n)$ pour tout $n \geq no$

Partie IV

Preuve d'algorithme

Introduction

La question un algorithme est-il **correct**, nous emmène à deux questions :

se termine-t-il ? autrement-dit, délivre-t-il un résultat (au bout d'un nombre fini d'opérations) **: la terminaison** ;

si oui, le résultat délivré est-il celui espéré : **la correction.**

Introduction

La preuve d'un programme se fait en un ou deux temps :

1) s'il s'agit d'une boucle conditionnelle, il faut prouver qu'il se termine (la **terminaison**) ;

2) il faut alors prouver qu'il fournit bien le résultat escompté (la **correction** ou **validité**).

Méthode

La preuve d'un algorithme se fait en un ou deux temps :

1) s'il s'agit d'une boucle conditionnelle, il faut prouver qu'il se termine (la **terminaison**) ;

2) il faut alors prouver qu'il fournit bien le résultat escompté (la **correction** ou **validité**).

Méthode

De façon plus générale, considérons la boucle suivante:- ou B désigne une expression booléenne et S une suite d'instructions :

TantQue B
Faire S
FinTantQue

Méthode

Terminaison :
Pour montrer que la boucle termine, on montre que l'exécution de *S* fait décroitre strictement une valeur ce qui finira par terminer la boucle.

Pratiquement on utilise une expression, qui est un entier positif tout au long de la boucle, qui décroît strictement.

Méthode

➢lorsqu'une suite d'entiers u_i décroît strictement, il existe un rang N à partir duquel les termes u_i sont négatifs. Or dans la boucle ui > 0, l'algorithme termine nécessairement.

➢ Cette expression peut être le simple contenu d'une variable telle qu'un compteur de boucle.

Méthode

Correction :

➢Pour démontrer que l'algorithme produit l'effet attendu, on introduit une propriété annexe *P* appelée **invariant de boucle**.

➢On montre que *P* est vérifiée avant l'entrée dans la boucle et qu'elle est conservée à chaque passage dans la boucle (c'est-a-dire si *P* est vraie avant l'exécution de S elle reste vraie après).

➢Pour Ceci on utilise une preuve par récurrence.

Méthode

Définition

Un **invariant de boucle** est une relation, une propriété liant les éléments variables d'une boucle qui est vraie avant d'effectuer la première itération de la boucle, qui reste vraie à chaque itération et qui est vraie en sortie de boucle.

Méthode

Démarche

1) On définit une pré-condition qui décrit l'état des variables avant d'entrer dans la boucle ;

2) On prouve que l'invariant de boucle est vrai à chaque itération (par un raisonnement par récurrence);

3) La condition de sortie de boucle en conjonction avec l'invariant de boucle permet de trouver une post-condition qui prouve la validité de l'algorithme.

Méthode

Démarche

1) On définit une pré-condition qui décrit l'état des variables avant d'entrer dans la boucle ;

2) On prouve que l'invariant de boucle est vrai à chaque itération (par un raisonnement par récurrence);

3) La condition de sortie de boucle en conjonction avec l'invariant de boucle permet de trouver une post-condition qui prouve la validité de l'algorithme.

Algorithmes itératifs

Exemple 1 :

Soit l'algorithme suivant :

```
DEBUT
Lire a
SI a < 0
 a ⟵ -a
FINSI
b⟵a
c⟵0
TANT QUE b > 0 FAIRE
c⟵c + a
b⟵ b - 1
FIN TANTQUE
FIN
```

Algorithmes itératifs

- La première chose à vérifier est que l'on finira par sortir de la boucle.

On procède comme suit :

- Avant d'entrer dans la boucle, la valeur de *b* est un entier positif ou nul.
- La valeur de *b* est décrémentée à chaque passage dans la boucle ; C'est une suite entière positive et décroissante.
- La valeur de b finira donc par s'annuler et on sortira alors de la boucle.
- On vient de prouver la terminaison de cet algorithme.

Algorithmes itératifs

- Une fois que l'on a prouvé que la boucle se termine, il faut s'assurer que le résultat est celui escompté.

- Pour ce faire, on considère la propriété (invariant de boucle) P(n) : $a^2 = c_n + b_n \times a$.

- Il s'agit donc de vérifier que pour tout entier n, P(n) est vraie.
- On procède par récurrence :

Algorithmes itératifs

➢ On a $b_0 = a$ et $c_0 = 0$ donc $c_0 + b_0$ - a = 0 + a x a = a^2 donc P(0) est vraie.

➢ Soit n un entier et supposons P(n) vraie ;
On a $b_{n+1} = b_n$ -1 et $c_{n+1} = c_n + a$;
Donc:
$c_{n+1} + b_{n+1}$ x a = (c_n + a) + (b_n - 1) x a = $c_n + b_n$ x a = a^2
ce qui prouve que P(n + 1) est vraie.
➢Lorsqu'on sort de la boucle, on a donc $b = 0$ et *par la suite* $a^2 = c + 0$ x a = c ce qui prouve que le **Résultat du programme est a^2**.

Algorithmes itératifs

Exemple 2:

```
Fonction(n:entier):entier
x, y:entier
Début
x = n
y = n
tant que (y ≠ 0)
x ⟵ x + 2;
y ⟵ y – 1
fin tantque
Retourne x
Fin fonction
```

Algorithmes itératifs

Invariant de boucle : p(i) : $x_i + 2y_i = 3n$

1) pré-condition : $x_0 = n$, $y_0 = n$ d'où $x_0 + 2y_0 = 3n$
donc p(0) est vraie

2)si p(i) vraie alors $x_{i+1} + 2y_{i+1} = (x_i + 2) + 2(y_i - 1) = x_i + 2y_i = 3n$,
donc p(i + 1) vraie

3)condition de sortie : à la sortie de boucle, lorsque y prend la valeur 0, donc après n itérations, p(n) étant vraie, $x_n + 2y_n = 3n$ donc f renvoie le triple de son argument.

Algorithmes récursifs

Exemple1 :
Somme des premiers entiers naturels : 1 + 2 + … + *n* = S(*n*)

```
fonction somme(n:entier):entier
si n=0
retourne 0
sinon
retourne somme (n – 1) + n
finsi
fin fonction
```

Algorithmes récursifs

- On note p(*n*) la propriéte : "*somme n* se termine et renvoie S(*n*)".

- p(0) vraie car *somme 0* se termine et renvoie 0 = S(0);
- si p(*n*) vraie pour *n* ≥ 0 alors *somme (n+1)* fait un unique appel récursif à *somme n* qui se termine et renvoie S(*n*) donc *somme (n+1)* se termine et renvoie S(*n*) + (*n*+1) = S(*n*+1), donc p(*n*+1) vraie ;
- Donc p(*n*) vraie pour tout *n* ≥ 0

Donc *somme* termine sur l'ensemble des entiers et renvoie S

Algorithmes récursifs

Exemple2 : Puissance

```
FONCTION puissance(a:réel,n:entier):réel
DEBUT
SI n = 0
RETOURNE 1
SINON
SI n = 1
RETOURNE a
SINON
SI n est pair
RETOURNE puissance(a * a, n/2)
SINON
RETOURNE(a * puissance(a * a, (n - 1)/2))
FINSI
FINSI
FINSI
FinJonction
```

Algorithmes récursifs

Montrons que pour tout $n \geq 0$, cette fonction calcule a^n.

On considère donc la propriété:

P(*n*) : " pour tout réel a , puissance(a,n) termine et puissance(a,n) = a^n".

Algorithmes récursifs

Il y a deux cas ou la fonction retourne directement une valeur :

si $n = 0$, le résultat est alors $1 = a_0$
si $n = 1$, le résultat est alors $a = a_1$

La propriété P est donc vérifiée dans ces deux cas, soit P(0) et P(1) sont vraies.

Algorithmes récursifs

Soit $n \geq 1$ et supposons P(k) vraie $0 \leq k \leq n - 1$

Si n est pair, puissance(a,n) fait appel a puissance(a*a,n/2) ;

Or par l'hypothèse, puisque $1 \leq n/2 \leq n - 1$, puissance(a*a,n/2) termine et renvoie $(a*a)^{n/2}$;
Donc puissance(a,n) termine et renvoie $(a*a)^{n/2} = a^n$.
La propriété P(n) est donc encore vérifiée.

Algorithmes récursifs

Si n est impair, puissance(a,n) fait appel à puissance(a*a,(n-1)/2) ;

Or par l'hypothèse, puisque $0 \leq (n - 1)/2 \leq n - 1$, puissance(a*a,(n - 1)/2) termine et renvoie $(a*a)^{n/2} = a^n$;

Donc puissance(a,n) termine et renvoie $a*(a*a)^{(n-1)/2} = a*a^{n-1} = a^n$.

La propriété P(n) est donc encore vérifiée.
Donc la fonction puissance termine et retourne le résultat voulu