Université Abdelmalek Essaâdi
Ecole Nationale des Sciences Appliquées
Al Hoceima

Chapitre 1: Algorithmes itératifs et récursifs

Plan du chapitre

I. Notion d’algorithme

II. Un langage pour décrire les algorithmes

III. Algorithmes récursifs

IV. Transformation de l’itératif en récursif

I. Notion d'algorithme

1. Qu'est-ce qu'un algorithme ?

- Un algorithme est une suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème.
- Un **problème algorithmique** est spécifié en décrivant :
  - L'ensemble complet des ses **instances** (**Input**) sur lesquelles il doit travailler;
  - Sa **production** (**Output**) après son exécution sur une de ces instances.
- **Exemple**: le problème algorithmique connu sous le nom du problème de tri est défini comme suit :
  - **Problème** : le tri.
  - **Input** : une liste de n clés $<a_1, \ldots, a_n>$.
  - **Output** : une permutation de la liste d'entrée $<a_1', \ldots, a_n'>$ qui soit ordonnée : $a_1' \leq \ldots \leq a_n'$.
- Une **instance** du problème de tri pourrait être :
  - Une liste de numéros comme <14, 25, 58, 34, 64, 34>;
  - Une liste de noms comme <nom1, nom2, nom3, nom4>;

2. Propriétés d'un bon algorithme

- Il y a trois propriétés désirables pour un bon algorithme :
  - **Correction;**
  - **Efficacité;**
  - **Facilité à mettre en œuvre.**
- Ces trois objectifs ne peuvent pas toujours être tous atteints simultanément.

3. Pluralité des solutions algorithmiques

- Pour un même **problème algorithmique**, il peut exister plusieurs **algorithmes différents** :
    - Certains itératifs, d'autres récursifs;
    - Certains sont plus rapides que les autres;
    - Certains utilisent moins d'espace mémoire que d'autres;
    - …etc.
- Par exemple, pour trier un tableau donné il existe **plusieurs algorithmes,** chacun est différent de l'autre : tri par **sélection**, tri par **insertion**, tri à **bulles**, tri par **fusion**, tri **rapide**, …etc.

II. Un langage pour décrire les algorithmes

1. Structure générale d'un algorithme

- Pour la définition d'un problème, on utilise un langage scientifique.
- Pour des raisons de simplicité, on utilise une langue naturelle(le français par exemple).
- **Algorithme**

```
Fonction Nom_Fonction(Input) : Output

Var …  // variables

Début

   …  // actions

Fin
```

2. Les différents éléments d'un algorithme

Type de données
– Entier;
– Réel;
– Caractère;
– Chaînes de caractères;
– Booléen (vrai / faux);
– Tableau;

Opérations de base
– + - * / %
– >, ≥, <, ≤, = et ≠
– Non, Et, Ou
– **:=** ou ←(pour l'affectation)
– Afficher
– Lire

Structures de contrôle
– Si … Alors … Sinon … FinSi
– Pour … Faire … FinPour
– Tant Que … Faire … FinTQ
– Répéter … Jusqu'à … FinRép
– Retourner …

III. Algorithmes récursifs

1. Présentation

- Un algorithme récursif est constitué d'une fonction récursive.
- Une fonction est dite récursive si elle s'appelle elle-même.
- L'appel d'une fonction à l'intérieur d'elle-même est nommé appel récursif.
- Un appel récursif doit obligatoirement être dans une instruction conditionnelle (Si(condition) Alors Retourner …. Sinon ….), sinon la récursivité est sans fin.
- Ses composants sont :
  - Cas de base **(condition d'arrêt)** : il s'agit de cas où l'algorithme ne s'appelle pas lui-même. Sinon l'algorithme ne peut pas terminer.
  - Appel récursif **(récurrence)** : c'est le cas inductif où la fonction fait appel à elle-même.
- Chaque appel récursif doit en principe se « rapprocher » d'un cas de base, de façon à permettre la terminaison du programme.

- **Exemple :** un exemple concrète d'une fonction récursive est la suite de Fibonacci qui définie $\forall n \in \mathbb{N}$ par :

$$U_n = \begin{cases} n, \quad si\ n < 2 \\ U_{n-1} + U_{n-2}, si\ n > 1 \end{cases}$$

Cas de base

Appel récursif

- **Son algorithme :**

```
Fonction  Fibo(n : Entier) : Entier;
Début
    Si (n<2) Alors
            Retourner n;
            Sinon
                Retourner Fibo(n-1) + Fibo(n-2) ;
    FinSi
Fin
```

Cas de base : c'est quand l'algorithme s'arrête

Appel récursif : fonctionnalité de répétition

2. Conception d'un algorithme récursif

- Les étapes à suivre pour concevoir une fonction récursive sont:
  - On décompose le problème en un ou plusieurs sous-problèmes du même type. On résout les sous-problèmes par des appels récursifs.
  - Les sous-problèmes doivent être de taille plus petite que le problème initial.
  - Enfin, la décomposition doit conduire à un cas élémentaire, qui, lui, n'est pas décomposé en sous-problème (condition d'arrêt).

- **Exemple** : Problème du calcul de factorielle *n!*.

$$
\begin{aligned}
(n)! &= factorielle(n)\\
(n-1)! * n &= factorielle(n-1) * n\\
(n-2)! * (n-1) * n &= factorielle(n-2) * (n-1) * n\\
(n-3)! * (n-2) * (n-1) * n &= factorielle(n-3) * (n-2) * (n-1) * n\\
&\vdots\\
0! * \ldots * (n-2) * (n-1) * n &= factorielle(0) * \ldots * (n-2) * (n-1) * n
\end{aligned}
$$

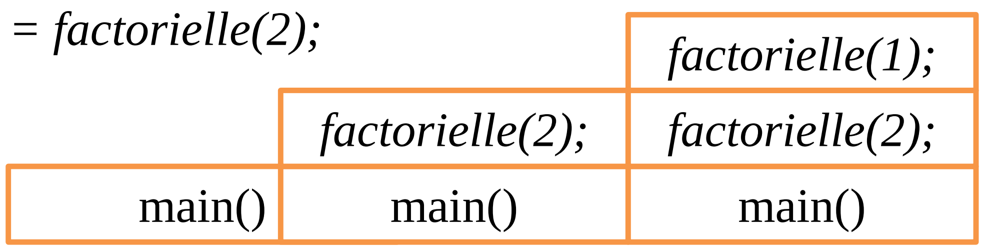
- L'algorithme récursif correspondant est donné comme suit :

```
Fonction factorielle(n : Entier) : Entier;
Début
    Si (n<2) Alors
            Retourner 1;
            Sinon
                Retourner factorielle(n-1)* n;
    FinSi
Fin
```

3. Fonctionnement de la récursivité

- Les appels récursifs sont gérés à l'aide d'une pile (stack) d'exécution qui **conserve les contextes d'appel** selon l'ordre LIFO (Last In First Out) **:**
  - à chaque appel de fonction, on **empile le contexte : lieu de l'appel, variables locales,** etc...
  - à chaque retour de fonction, on **dépile le contexte, ce qui permet de revenir au point** d'appel.
- ***Exemple :*** la pile d'exécution de la fonction factorielle pour un appel avec n=2 .

```
main() {
  resultat = factorielle(2);
}
```



1- Appel de fact avec 2 **6-** Résultat final

2- Appel de fact avec 1 **5-** Retour de l'appel avec 2

3- Exécution de condition de base **4-** Retour de l'appel avec 1

- **Exercice** : donner l'affichage de programme suivant :

```
#include<stdio.h>
#include<stdlib.h>
void f(int n){
        if(n==0)
            printf(" Terminé ");
         else {
                f(n-1);
                printf("%d\n",n);
          }
 }
int main(){
f(4);

return 0;
}
```

4. Types d'algorithmes récursifs

- ***Récursivité simple :*** Une fonction récursive contient un seul appel récursif (<u>Exemple</u>: fonction factorielle).
- ***Récursivité multiple:*** Une fonction récursive contient plus d'un appel récursif (<u>Exemple</u> : suite de Fibonacci).
- ***Récursivité mutuelle(ou croisée):*** consiste à écrire des fonctions qui s'appellent l'une l'autre. <u>Exemple</u> :

Fonction **Pair** (N : Entier) : Booléen
Début
    Si(N = 0) Retourner Vrai;
        Sinon Retourner **Impair(N-1);**
    FinSi
Fin
	Fonction **Impair** (N : Entier) : Booléen
	Début
	    Si(N = 0) Retourner Faux;
	        Sinon Retourner **Pair(N-1);**
	    FinSi
	Fin

IV. Transformation de l'itératif en récursif

1. Présentation

- Tout algorithme itératif peut être transformé en algorithme récursif sans boucle.
- Il n'existe pas méthode spécifique pour faire cette transformation.
- Pour un algorithme itératif donné, pour trouver :
  - Le cas de base : généralement, on regarde la condition d'arrête, si la boucle est en incrémentation, il faut penser à la décrémentation, car ceci permet de déterminer un simple cas de base. Exemple: Pour i:=0 à n faire ⇔ Pour i=n à 0 faire (l'algorithme s'arrête lorsque i=0) c'est le cas de base qu'on peut considérer pour la version récursive.
  - Le cas récursif : généralement, on détermine la fonction calculée par l'algorithme.

2. Exemples

- Pour montrer comment transformer un algorithme itératif en algorithme récursif, trois exemples sont utilisés :
  - **Exemple 1**: un algorithme qui calcule la somme des entiers entre 1 jusqu'à N.
  - **Exemple 2**: un algorithme qui calcule x à la puissance N.
  - **Exemple 3**: un algorithme qui permet trouver le plus grand élément d’un tableau de N entiers.

Exemple 1: Algorithme itératif

```
Fonction somme(N : Entier) : Entier;
 Var i, S : Entier;
Début
   S := 0;  // Initialisation
   Pour i allant de 1 jusqu'a N Faire // Progression
      S := S + i;  // Approximation
   Fin
   Retourner S;
Fin
```

Exemple 1: Algorithme récursif

```
Fonction Somme(N : Entier) : Entier;
Début
    Si (N=0) Alors
            Retourner N;
     Sinon
            Retourner Somme(N-1)+N;
    FinSi
Fin
```

Idée de la récursivité :

$S_N = 1 + 2 + \ldots + N$

$S_N = (1 + 2 + \ldots + N - 1) + N$

$S_N = S_{N-1} + N$ *// appel récursif*

$S_0 = 0$ *// cas de base*

- **Exemple 2: Algorithme itératif**

```
Fonction puissance(x: Réel, N : Entier) : Réel;
Var i, p : Entier;
Début
    p := 1;  // Initialisation
    Pour i allant de 1 jusqu'a  N Faire // Progression
        p := p * x;  // Approximation
    Fin
    Retourner p;
Fin
```

…. la suite

- **Exemple 2: Algorithme récursif**

```
Fonction puissance(x: Réel, N : Entier) : Réel;
Début
    Si (N=0) Alors
            Retourner 1;
    Sinon
            Retourner puissance(x, N-1) * x;
    FinSi
Fin
```

- **Exemple 3: Algorithme itératif**

```
Fonction MaxTableau(A: Tableau, N : Entier) : Entier;
Var i, max : Entier;
Début
    max := A[1];  // Initialisation
    Pour i allant de 2 jusqu'a  N Faire // Progression
        Si (A[i] > max) Alors
                max:=A[i];
        FinSi
    Fin
    Retourner max;
Fin
```

…. la suite

- **Exemple 3: Algorithme récursif**

```
Fonction MaxTableau(A: Tableau, N : Entier) : Entier;
Début
    Si (N=1) Alors
            Retourner A[1];
     Sinon
            Retourner max(MaxTableau(A, N-1) , A[N]);
    FinSi
Fin
```

Idée de la récursivité :
Cas de base : si le tableau a un seul élément (N = 1), son plus grand élément est A[1].
Appel récursif : si n > 1, le plus grand élément de A[1…N] est le plus grand entre A[N] et le plus grand élément de A[1…N-1].