

Chapitre 18

Algorithmique de base

1. Récursivité

- Une procédure récursive est une procédure qui s'appelle elle-même en cours d'exécution.

```
int UneFonction(int a) {  
    return (a * UneFonction(a-1));  
};
```

- Si l'on appelle « UneFonction » avec « a=4 », il y aura un appel récursif infini.
- En effet, « UneFonction(4) » va appeler « UneFonction(3) » qui va appeler « UneFonction(2) » « UneFonction(1) » etc. Cet appel de fonction récursif va finir par planter un jour car l'ordinateur n'a pas une mémoire infinie.
- Toute procédure récursive doit contenir donc un « Point d'arrêt » qui permet d'arrêter l'appel récursif.

```
UneFonctionRécursive(arguments) {  
    if (Test_ARRÊT) {  
        instructions reliées à l'arrêt  
    }else{  
        instructions suite  
        UneFonctionRécursive(arguments)  
        instructions suite  
    }  
};
```

- Calcul factoriel

▪ Une approche itérative

```
int factorielle(int i){
    int resultat;
    for(resultat=1;i>1;i--) resultat*=i;
    return(resultat);
}
```

- $factorielle(n) = n * n-1 * n-2 * \dots * 1$
- $factorielle(3) = 3 * 2 * 1 = 6$

Variable i	Variable resultat	Déroulement des instructions
i=3	resultat=0	
i=3	resultat=1	resultat = resultat*i=1*3=3
i=2	resultat=3	resultat = resultat*i=3*2=6
i=1	resultat=6	i>1 donc sortir de la boucle for
		Retourner 6

- La méthode itérative nécessite l'utilisation de variables locales pour effectuer le calcul demandé.

▪ Une approche récursive

- Il est toujours possible de transformer un programme récursif en un programme itératif. L'inverse est vrai aussi.
- La récursivité permet d'énoncer des problèmes complexes de manière concise sans perte d'efficacité.
- Le coût d'implémentation est assumé par des mécanismes internes du système.
- Ces mécanismes utilisent une pile interne.
- La plupart des systèmes de programmation modernes intègrent ce genre de mécanismes.
- Il est important d'éviter d'écrire des fonctions récursives inefficaces et insolubles !
- Si le temps de calcul ou la mémoire utilisée sont prohibitifs pour de grands nombres de données il faudra traduire la procédure récursive en procédure itérative.

```
int factorielle(int i){
    if (i>1) return (i*factorielle(i-1));
    return(1);
}
```

- $\text{factorielle}(3) = 3 \times 2 \times 1 = 6$

Analyse de la pile interne

		i=1		
	i=2	i=2	i=2	
i=3	i=3	i=3	i=3	i=3
(a)	(b)	(c)	(d)	(e)

- (a) appel de $\text{factorielle}(3)$, création de i , à qui on affecte la valeur 3. comme $i > 1$ on calcule $i \times \text{factorielle}(i-1)$: $i=3, i-1=2$ on appelle $\text{factorielle}(2)$.
- (b) création i , affecté de la valeur 2, $i > 1$ donc on appelle $\text{factorielle}(1)$.
- (c) création de i , $i=1$ donc on quitte la fonction, on libère la pile de son sommet, on retourne où la fonction $\text{factorielle}(1)$ a été appelée en rendant 1.
- (d) on peut maintenant calculer $i \times \text{factorielle}(1)$, i (sommet de la pile) vaut 2, $\text{factorielle}(1)$ vaut 1, on peut rendre 2, puis on "dépile" i .
- (e) on peut calculer $i \times \text{factorielle}(2)$, i vaut 3 (sommet de la pile), $\text{factorielle}(2)$ vaut 2, $3 \times 2 = 6$, on retourne 6, la pile est vidée et retrouve son état initial.

- Calcul du PGCD (Plus Grand Commun Diviseur)
 - Le plus grand commun diviseur de deux entiers qui ne sont pas nuls, est le plus grand nombre entier qui divise les deux nombres.

```
A=18=1*2*3*3
B=21=1*3*7
PGCD (A,B) =PGCD (18,21) =1*3=3
```

Procédure itérative

```
int pgcd(int a,int b) {
    int r;
    while (b!=0) {
        r=a%b;a=b;b=r;
    }
    return a;
}
```

Procédure récursive

```
int pgcd(int a, int b){
    return (b!=0)?a:pgcd(b,a%b);
}
```

2. Techniques de Recherche

2.1 Recherche linéaire (séquentielle)

- On examine successivement tous les éléments de la table et on regarde si on trouve l'élément recherché dans la table.
- Le nombre de tests de comparaison permet de déterminer la complexité d'une recherche donnée.
- Dans ce cas, nous effectuons au pire n opérations, n étant la taille de la table.
- On dit que la complexité d'une telle recherche est de l'ordre de n ou $O(n)$.
- Dans le meilleur des cas, on tombe directement sur la valeur, n sera égale à 1.
- Dans le pire des cas, on est obligé de balayer toute la table, n sera égale à la taille de la table.
- Si les éléments sont déjà triés, on peut interrompre la recherche avant d'atteindre la fin du tableau.

2.2 Recherche dichotomique

- Si vous avez un tableau déjà trié de taille n , on peut écrire une fonction qui cherche si un élément donné se trouve dans la table.
- Il faut voir cela comme la recherche d'un mot dans un dictionnaire

- On choisit une page au milieu du dictionnaire.
- On regarde si le mot cherché est avant ou après cette page.
- On cherche maintenant dans la moitié correspondante du dictionnaire.
- On dit qu'on procède par dichotomie.
- Le processus peut-être résumé ainsi :
 - On cherche à savoir d'abord si x est dans $t[\text{deb}, \text{fin}]$. Pour cela, on calcule le milieu = $(\text{deb} + \text{fin})/2$ et on compare x à $t[\text{milieu}]$.
 - Si $x = t[\text{milieu}]$, on a gagné. On arrête la recherche.
 - Dans le cas contraire,
 - on réessaie avec $t[\text{deb}, \text{milieu}]$ si $t[\text{milieu}] > x$
 - sinon dans $t[\text{milieu} + 1, \text{fin}]$.
- Dans le pire des cas, on divise N par deux jusqu'à qu'il soit égal à 1.
- On obtient alors $\log_2 N$ opérations.

2.3 Recherche récursive

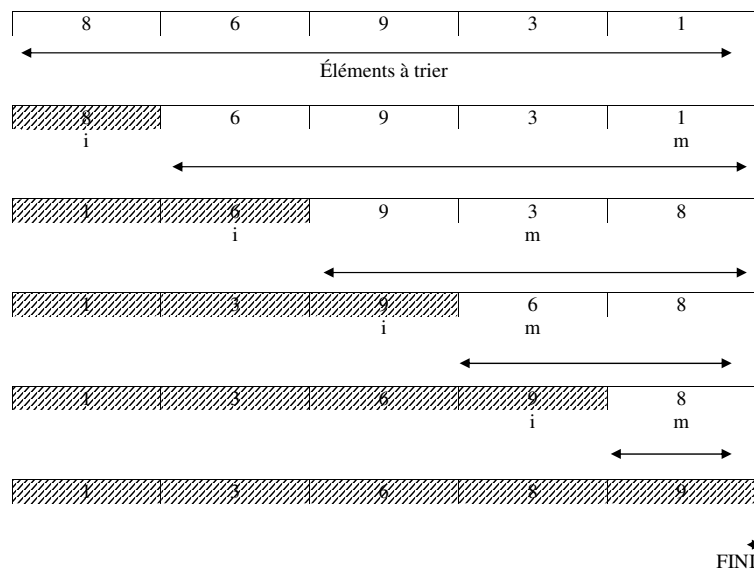
- On fait appel au même programme au cours de son déroulement en utilisant le principe de récursivité développé au début de ce chapitre.

3. Tris

- Le tri est rendu nécessaire par exemple s'il faut établir le classement des élèves, mettre en ordre certaines informations d'une application quelconque : tout cela, afin de permettre de trouver l'information recherchée de manière rapide.
- Nous allons étudier trois sortes de tris : par sélection, à bulles et par insertion.
- On suppose que nous avons un tableau d'entiers de taille N . On suppose aussi que le tableau est indexé de 0 à $N-1$.

3.1 Tri par sélection

- L'algorithme de tri associé au tri par sélection consiste à trouver l'emplacement du plus petit élément dans un tableau.
- Dès que cet élément est trouvé, nous l'interchangeons avec le premier élément du tableau ($i=0$).
- Nous recommençons l'opération pour le reste du tableau (i.e. $i = [1, N[$; N étant la taille du tableau).
- Tableau à trier : [8, 6, 9, 3, 1].
 - L'index m pointe le plus petit élément dans un tableau contenant les éléments restants à trier.



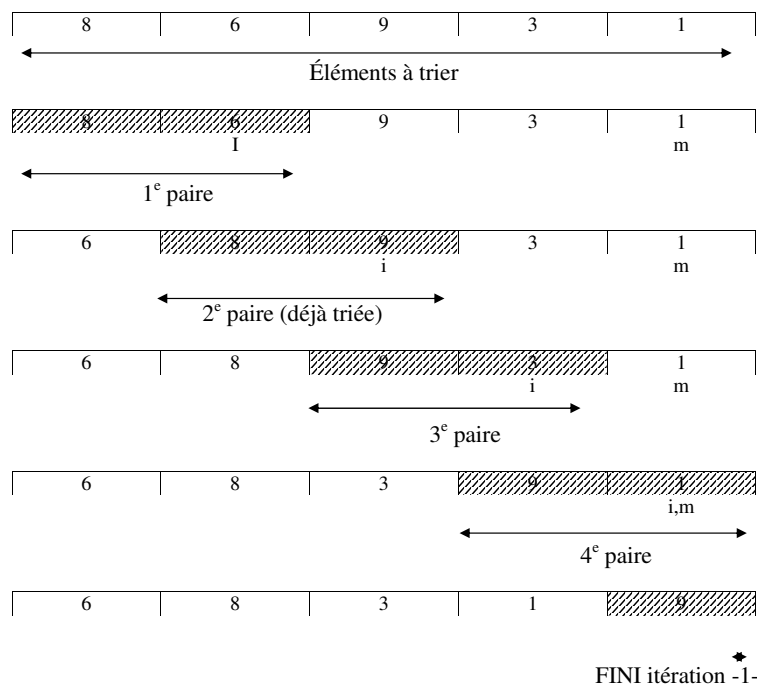
3.2 Tri à bulles

- Le tri à bulles est une variante du tri par sélection.
- Son principe consiste à échanger deux éléments consécutifs qui ne sont pas ordonnés d'un tableau donné.
- Après ce parcours, l'élément le plus grand va se retrouver en dernier.
- Nous recommençons l'opération avec les N-1 éléments du tableau [0, N-1[.
- L'algorithmique est comme suit :

```

affecter true à flag
tantQue flag=true faire
    affecter false à flag
    pourChaque (éléments de tableau)-1
        si tableau[courant]>tableau[suivant] alors
            permuter les 2 valeurs
            affecter true à flag
        fin si
    fin pourChaque
fin tantQue

```



- Ainsi prend fin la première itération avec l'élément le plus élevé qui se retrouve à la fin du tableau.
- Notons que la valeur de la variable est à « true » car nous avons permuté au moins une fois deux éléments consécutifs.
- Nous recommençons l'opération mais qu'avec les éléments non encore triés du tableau i.e : [6, 8, 3, 1].
- Nous obtenons ainsi les résultats suivants pour les autres itérations :

Itération	Combinaison	flag
2	63189	true
3	31689	true
4	13689	true
5	13689	true

3.3 Tri par insertion

- L'algorithme du tri par insertion repose sur le même principe que la technique utilisée pour trier un paquet de cartes.
- Ayant $i-1$ cartes déjà triées, on essaye de mettre la i^{e} carte à sa place dans le paquet déjà trié.
- Ainsi de suite jusqu'à $i=N$, N étant le nombre de cartes.
- La variable m représente l'index de la case où l'élément sera inséré.
- La variable i représente l'index de l'élément en cours de traitement.

