

---

TP FLIN408

Année 2010-11

Version 1.0

---

Université de Montpellier  
Place Eugène Bataillon  
34095 Montpellier Cedex 5

RODOLPHE GIROUDEAU  
161, RUE ADA  
34392 MONTPELLIER CEDEX 5  
TEL : 04-67-41-85-40  
MAIL : RGIROU@LIRMM.FR

ULIN 408  
TD – Séance n° 1

**Exercice 1 – Les suites de Syracuse**

On se propose de construire un petit programme qui permet d'étudier les suites dites de Syracuse :

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

La conjecture de Syracuse dit que quelle que soit la valeur de départ, la suite finit par boucler sur les valeurs 4, 2, 1, 4, 2, 1, ...

1. Écrire un programme qui demande une valeur de départ  $u_0$  et affiche les valeurs successives jusqu'à tomber sur la valeur 1 ;
2. Modifier le programme pour qu'il compte les itérations, sans affichage intermédiaire ;
3. Modifier le programme pour qu'il redemande éventuellement une nouvelle valeur de départ, à l'aide d'une boucle de ... `while`.

**Exercice 2 – Sous-suite de somme maximale**

On se donne un tableau  $T$  de  $n$  nombres réels. Le résultat cherché est la somme maximale des éléments d'un sous-tableau contigu de  $T$ .

1. Écrire un algorithme naïf. (calcul de toutes les sommes possibles et on garde la plus grande).
2. Améliorez l'algorithme précédent en utilisant l'égalité suivante :

$$\sum_{k=i}^j T_k = T_j + \sum_{k=i}^{j-1} T_k$$

**Exercice 3 –  $n$ -uplets de 0 et de 1 A faire si vous avez le temps**

Écrire un programme qui demande à l'utilisateur un entier  $n$  et qui affiche tous les  $n$ -uplets possibles composés de 0 et de 1. Par exemple si on rentre 3, on doit avoir en sortie (pas forcément dans cet ordre) :

000 001 010 011 100 101 110 111

**Exercice 4 – Tous les sous-ensembles de l'ensemble  $\{1, \dots, n\}$**

**A faire si vous avez le temps** Écrire un programme qui demande un entier  $n$  et qui affiche à l'écran tous les sous-ensembles de l'ensemble  $S = \{1, \dots, n\}$ . Par exemple si  $n = 3$  on doit avoir :

$\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}$

**Exercice 5 – Nombres amis**

Soit  $n$  et  $m$ , deux entiers positifs.  $n$  et  $m$  sont dits *amis* si la somme de tous les diviseurs de  $n$  (sauf  $n$  lui-même) est égale à  $m$  et si la somme de tous les diviseurs de  $m$  (sauf  $m$  lui-même) est égale à  $n$ .

Écrire une fonction qui demande à l'utilisateur deux entiers  $n$  et  $m$  et qui affiche si  $n$  et  $m$  sont ou non des nombres amis.

Écrire une fonction qui demande à l'utilisateur un entier positif  $nmax$  et qui affiche tous les couples de nombres amis  $(n, m)$  tels que  $n \leq m \leq nmax$ .

**Exercice 6 – LE PGCD**

Avant tout, rappelons quelques règles sur le *pgcd* :

- $pgcd(a, 0) = a$
- $pgcd(a, b) = pgcd(b, a)$
- $pgcd(a, a) = a$

1. Écrire un algorithme itératif qui calcule le pgcd de manière naïve en utilisant la propriété suivante :

$$pgcd(a, b) = pgcd(a, b - a) \text{ si } a < b$$

2. Écrire un algorithme itératif qui calcule le pgcd de manière plus efficace, en utilisant la propriété :

$$pgcd(a, b) = pgcd(a, b \bmod a) \text{ si } a < b$$

3. Écrire un algorithme itératif qui calcule le pgcd en utilisant les propriétés suivantes :

- $pgcd(2a, 2b) = 2 * pgcd(a, b)$
- $pgcd(2a, b) = pgcd(a, b)$  si  $b$  est impair
- $pgcd(a, b) = pgcd(a, b - a)$  si  $a < b$
- si  $a$  et  $b$  sont impairs alors  $a - b$  est pair

**Exercice 7 – Les Permutations**

Considérons un tableau d'entiers  $T$  possédant *taillemax* cases. Nous voulons le remplir avec les nombres compris entre 1 et *taillemax*, en évitant que le même nombre apparaisse plusieurs fois dans le tableau. Nous allons comparer l'efficacité de trois façons différentes de procéder.

Pour cela, on dispose de la fonction `rand`, qui donne comme résultat un nombre aléatoire  $n$  tel que  $0 \leq n < 32767$ . la ligne suivante :

1. Une première méthode simple consiste à remplir le tableau  $T$  de gauche à droite, en comparant le nombre  $n$  tiré au hasard avec tous les nombres qui le précèdent dans le remplissage du tableau. On se servira de la fonction auxiliaire dont l'en-tête est le suivant :

```
int present(int n, int k, int *T)
```

Cette fonction renvoie 1 si l'entier  $n$  est présent dans  $T$  entre les indices 0 et  $k-1$ . Écrire l'algorithme.

2. Plutôt que de revenir chaque fois en arrière pour vérifier la présence d'un nombre avant de l'insérer, on se propose d'utiliser un tableau auxiliaire  $TB$  contenant des valeurs booléennes. Au début,  $TB[i]=0$  pour  $i$  entre 0 et *taillemax-1*. Avant d'insérer un nombre  $n$  dans  $T$ , on regarde la valeur de  $TB[n]$ . Si  $TB[n]=1$ , c'est que le nombre a déjà été inséré, sinon, c'est que le nombre n'a pas été inséré. On l'insère et  $TB[n]=1$ .

3. Une troisième méthode semble plus astucieuse. On commence par initialiser  $T[i]$  à  $i$  pour tout  $i$  entre 1 et  $taillemax$ . Maintenant, puisque les nombres de 1 à  $taillemax$  sont déjà rangés dans l'ordre dans  $T$ , il suffit d'avancer dans le tableau de gauche à droite et pour chaque nombre, le permuter avec un autre choisi au hasard dans  $T[i+1..taillemax]$ . À la fin, on obtient bien une permutation des  $n$  premiers entiers. Écrire l'algorithme.

**Exercice 8 – Calcul de  $x^n$**

1. Écrire la version *itérative* de la fonction P.

```
float P (float x, int n)
```

P calcule  $x^n$ .

2. Écrire la version *réursive*, grâce à la définition abstraite suivante :

$$\begin{cases} P(x, 0) = 1 \\ P(x, n) = x \times P(x, n - 1) \quad \text{si } n \neq 0 \end{cases}$$

3. Il existe une *autre méthode réursive* pour calculer  $x^n$ . Elle s'appuie sur une nouvelle définition abstraite :

$$\begin{cases} P(x, 0) = 1 \\ P(x, n) = P(x, n \div 2)^2 & \text{si } n \text{ est pair} \\ P(x, n) = x \times P(x, n \div 2)^2 & \text{si } n \text{ est impair} \end{cases}$$

Écrire la fonction P en utilisant cette nouvelle définition.

4. Faire la trace de l'exécution de P(2, 5) avec les deux méthodes récurives des questions 2 et 3. Laquelle est la plus performante? Pourquoi?

**Exercice 9 – Calcul des  $C_n^p$**  Une définition abstraite des coefficients binômiaux  $C_n^p$  peut être :

$$\begin{cases} C_n^p = 0 & \text{si } n < p \\ C_n^p = 1 & \text{si } p = 0 \text{ ou } p = n \\ C_n^p = C_{n-1}^{p-1} + C_{n-1}^p & \text{dans les autres cas} \end{cases}$$

1. Écrire l'algorithme utilisant cette définition et dont l'en-tête est le suivant :

```
int CNP (int n, int p)
```

2. Écrire l'arbre d'appel de CNP(4, 2). Que constatez-vous? Peut-on éviter les re-calculs de CNP(2, 1), CNP(1, 0) et CNP(1, 1)?

ULIN408  
TD – Séance n° 2

---

**Exercice 1 – Analyse en moyenne** Nous avons un tableau de taille  $n$  constitué de 0 et de 1. Nous souhaitons augmenter le nombre représenté par le tableau d'une unité. Pour cela nous considérons l'algorithme 1

---

**Algorithm 1** Incrémentation d'un nombre en binaire

---

```
 $i := 0;$   
while  $i < n$  and  $a[i] = 1$  do  
   $a[i] = 0;$   
   $i ++$   
end while  
if  $i < n$  then  
   $a[i] = 1$   
end if
```

---

1. Procédez à une analyse en moyenne de cet algorithme.

**Exercice 2 – Analyse en moyenne bis**

Nous reprenons les deux algorithmes qui donnent les deux plus grands éléments.

1. Procédez à l'implémentation des deux algorithmes.
2. Vérifiez les résultats théoriques.

ULIN408  
TD – Séance n° 2

---

**Exercice 1 – Les tris vus en cours**

1. Programmez les différentes fonctions vues en cours pour programmer le tri par tas. Pour se rendre compte de la complexité de cet algorithme, faites afficher chacune des étapes de l'algorithme (chaque changement dans votre tableau). Testez cette fonction sur les tableaux suivants :

5	7	4	1	6	3	4
---	---	---	---	---	---	---

7	6	5	4	3	2	1
---	---	---	---	---	---	---

2. Faites la même chose en utilisant le tri par sélection. Que constatez-vous en terme de nombres d'étapes sur les exemples précédents.
3. Même question pour le tri rapide.

**Exercice 2 – Tri simple sur des entiers**

L'idée est de trier simplement  $n$  entiers compris entre 1 et  $\text{Max}$ . Etant donné le type suivant :  
`typedef int table[n]` L'entête de la fonction est le suivant : `int *TriSimple(table TabEntrée)`  
L'idée de l'algorithme peut être résumée par les points suivants :

- Compter le nombre d'apparitions de chaque valeur de `TabEntrée` dans un tableau auxiliaire `app`.
- Parcourir le tableau `app` et pour chaque `i` mettre `app[i]` fois la valeur `i` dans `TabSortie`.

1. Écrire la procédure `TriSimple`.
2. Justifier pourquoi cette procédure a une complexité de l'ordre de  $\mathcal{O}(n)$ .
3. Exécuter la fonction `TriSimple` sur le tableau

5	8	9	4	8	5	2	4	8	10
---	---	---	---	---	---	---	---	---	----

**Exercice 3 – Tri par énumération**

Le tri par énumération est une variante du tri par insertion, où le rang d'un élément  $e_i$  est déterminé par le nombre  $n_j$  d'éléments  $e_j$  tels que  $e_i > e_j$  pour  $j$  variant de 0 à  $n$ . On compare tous les éléments entre eux, et le rang définitif de  $e_i$  dans la liste triée est déterminé par  $n_j$ .

1. Écrivez un algorithme mettant en oeuvre le tri par énumération.
2. Faites la trace de cet algorithme sur le tableau suivant :

23, 34, 56, 6, 7, 3, 4, 36, 9, 2

**Exercice 4 – Tri fusion**

Programmer le tri fusion.

ULIN408  
TD – Séance n° 3

---

**Exercice 1 – Les Arbres Binaires de Recherche**

Nous avons vu en cours, la définition des arbres binaires de recherche, c'est-à-dire que la valeur d'un nœud est supérieure à la valeur des nœuds situés dans son sous-arbre gauche, et inférieure à la valeur des nœuds situés dans son sous-arbre droit.

On a les structures de données suivantes :

```
typedef enum {vrai=1, faux=0} booleen;  
  
typedef struct noeud {  
    int val;  
    struct noeud *sag;  
    struct noeud *sad;  
} Noeud;
```

Pour manipuler cette structure de données, vous écrirez les fonctions suivantes<sup>1</sup> :

1. **EstVide**, une fonction qui teste si un arbre est vide.
2. **CreerVide**, une fonction qui crée un arbre vide.
3. **Affiche**, une fonction qui affiche les nœuds de l'arbre dans l'ordre que vous préférez et de façon à bien voir les fils d'un nœud. Par exemple l'arbre ayant 3 nœuds, de racine 5 et de fils gauche 3 et de fils droit 7 pourra être représenté :  $\langle 5, \langle 3, \langle \rangle, \langle \rangle \rangle, \langle 7, \langle \rangle, \langle \rangle \rangle \rangle$ .
4. **InsereAuxFeuilles**, une fonction qui insère un nouveau nœud dans l'arbre avec ajout aux feuilles.
5. **InsereRacine**, une fonction qui insère un nouveau nœud à la racine de l'arbre
6. **Recherche**, une fonction qui recherche un élément dans l'arbre et renvoie vrai si l'élément est dans l'arbre, faux sinon.
7. 2 ABR sont dits *équivalents* s'ils contiennent exactement les mêmes éléments. Ecrire une fonction qui teste si 2 ABR sont équivalents.
8. Un ABR  $A$  est *contenu* dans un ABR  $B$  si tous les éléments de  $A$  sont contenus dans  $B$ . Ecrire une fonction qui teste un ABR est contenu dans un autre.

---

<sup>1</sup>vous pourrez pour certaines d'entre elles réutiliser les fonctions écrites au TD6

9. Un ABR  $A$  est dit de *domaine plus petit* qu'un ABR  $B$  si le plus petit élément de  $A$  est supérieur ou égal au plus petit élément de  $B$  et si le plus grand élément de  $A$  est inférieur ou égal au plus grand élément de  $B$ . Ecrire une fonction non récursive qui teste si un ABR est de domaine plus petit qu'un autre.

À l'aide de ces fonctions vous pourrez donc faire la suite de commandes suivantes :

```
Créer un arbre vide  
ajouter l'élément 16  
ajouter l'élément 7  
ajouter l'élément 78  
rechercher l'élément 16  
rechercher l'élément 65  
ajouter l'élément 87  
ajouter l'élément 13  
afficher l'arbre  
ajouter l'élément 56  
ajouter l'élément 1  
ajouter l'élément 5  
afficher l'arbre
```

Pour ajouter un élément dans l'arbre vous utiliserez d'une part l'insertion aux feuilles, d'autre part l'insertion à la racine et vous comparerez les arbres obtenus.

**Exercice 2 – Les arbres 2-3**

Un arbre 2-3 est un arbre de recherche tel que :

- chaque nœud interne a 2 fils ou 3 fils,
- toutes les feuilles sont au même niveau.

Chaque feuille de l'arbre contient un élément de l'ensemble représenté par l'arbre 2-3 et les éléments sont rangés de gauche à droite par ordre croissant dans les feuilles. Chaque nœud interne contient 2 valeurs : le plus grand élément du premier sous-arbre de ce nœud, puis le plus grand élément de son second sous-arbre.



1. Etant donné un arbre 2-3 contenant  $i$  noeuds internes et  $n$  feuilles et de hauteur  $h$  montrer que :

$$2^{h+1} - 1 \leq n + i \leq \frac{3^{h+1} - 1}{2}$$

et

$$2^h \leq n \leq 3^h$$

2. Ecrire un algorithme de recherche d'un élément dans un arbre 2-3 dont tous les éléments sont distincts. Donner la complexité de cet algorithme en fonction de  $n$  (nombre de feuilles).
3. Etudier l'algorithme d'insertion d'un élément qui conserve les propriétés d'un arbre 2-3.

**ULIN408**  
**TD – Séance n° 4**

---

**Exercice 1 – Table de hachage avec résolution des collisions par chaînage**

Reprendre l'exercice vu en TD, et programmer les différentes opérations sur une table de hachage.

**Exercice 2 – Table de hachage avec adressage ouvert**

Reprendre l'exercice vu en TD, et programmer les différentes opérations possibles sur une table.