# Programmation en Java

IPA 1999

Michel Mauny

PULV & INRIA-Rocquencourt
Michel.Mauny@devinci.fr

## Plan

Corrigé du second problème du TD 1

Récursivité

Complexité

Tris



## Corrigé du second problème du TD 1

Calculer le jour de la semaine correspondant à une date donnée.

```
public class Jour {

    static int[] aujourdhui = { 27, 10, 1999 }; // la date du TD
    static int jourDAujourdhui = 3; // le jour de la semaine

    static String jourToString (int jour) {
        switch (jour) {
        case 0: return "dimanche";
        case 1: return "lundi";
        case 2: return "mardi";
        case 3: return "mercredi";
        case 4: return "jeudi";
        case 5: return "vendredi";
        case 6: return "samedi";
        default: return "jour inconnu";
        }
    }
```



## Corrigé TD1 (suite)

```
    static int compareDates(int[] date1, int[] date2) {
        // retourne -1 si date1 < date2
        //           0 si date1 = date2
        //           1 si date1 > date2
        if (date1[2] > date2[2]) return 1;
        if (date1[2] < date2[2]) return -1;
        // les annees sont egales
        if (date1[1] > date2[1]) return 1;
        if (date1[1] < date2[1]) return -1;
        // les mois sont egaux
        if (date1[0] > date2[0]) return 1;
        if (date1[0] < date2[0]) return -1;
        return 0;
    }

    static boolean memeDate(int[] date1, int[] date2) {
        // inutilisee: ici pour l'exemple
        return (compareDates(date1, date2) == 0);
    }
```

```
static String dateToString(int[] date) {
    return date[0] + "/" + date[1] + "/" + date[2];
}

static boolean bissextile(int annee) {
    return ((((annee % 4) == 0) && (!((annee % 100) == 0)))
            || ((annee % 400) == 0));
}
```





```
static int dernierJourDuMois(int mois, int annee) {
    switch (mois) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        return 31;
    case 2 :
        if (bissextile (annee))
            return 29;
        else
            return 28;
    case 4: case 6: case 9: case 11 :
        return 30;
    default: return -1;
    }
}
```





```
static int[] lendemain(int[] date) {
    // calcule le date du lendemain
    int jour = date[0];
    int mois = date[1];
    int annee = date[2];
    int[] resultat = new int[3];

    for (int i=0; i < 3; i++)
        resultat[i] = date[i];

    if (jour < dernierJourDuMois(mois, annee))
        resultat[0]++;
    else if (mois < 12) {
        resultat[0] = 1;
        resultat[1]++;
    } else {
        resultat[0] = 1;
        resultat[1] = 1;
        resultat[2]++;
    }
    return resultat;
}
```

```
static int[] veille(int[] date) {
   // calcule le date de la veille
   int jour = date[0];
   int mois = date[1];
   int annee = date[2];
   int[] resultat = new int[3];

   for (int i=0; i < 3; i++)
       resultat[i] = date[i];

   if (jour > 1)
       resultat[0]--;
   else if (mois > 1) {
       resultat[0] =
          dernierJourDuMois(mois-1, annee);
       resultat[1]--;
   } else {
       resultat[0] = 31;
       resultat[1] = 12;
       resultat[2]--;
   }
   return resultat;
}
```





```
static String jourDeLaSemaine(int[] cible) {
     int[] date = aujourdhui;
     int jour = jourDAujourdhui;

     switch(compareDates(date, cible)) {
     case 1: // la cible est dans le passe
         while (! (memeDate(date, cible))) {
             jour = (jour + 6) % 7;
             date = veille(date);
         };
         break;
     case -1:
         while (! (memeDate(date, cible))) {
             jour = (jour + 1) % 7;
             date = lendemain(date);
         };
         break;
     }
     return jourToString(jour);
}
```

```
    public static void main (String args[]) {
        if (args.length != 3) {
            System.err.println ("Il faut 3 arguments.");
            System.exit(1);
        }
        int[] date = new int[3];
        date[0] = Integer.parseInt(args[0]);
        date[1] = Integer.parseInt(args[1]);
        date[2] = Integer.parseInt(args[2]);

        System.out.println ("Le " + dateToString(date) +
                            " est un " + jourDeLaSemaine(date));
    }
}
```





Une fonction/procédure/méthode peut s'appeler elle-même à l'intérieur de sa définition.

Identique aux définitions par récurrence en mathématiques.

Exemple : la suite de Fibonnacci.

$u_0 = u_1 = 1$

$u_n = u_{n-1} + u_{n-2}$

Ce qui donne :

1, 1, 2, 3, 5, 8, 13, ...





```
static int fib(int n) {
  if (n <= 1)
      return 1;
  else
      return fib (n-1) + fib (n-2);
}
```

La fonction factorielle :

```
static int fact(int n) {
  if (n <= 0) return 1;
  else return n * fact(n-1);
}
```

Plus court

```
static int fact(int n) {
  return (n <= 0) : 1 ? (n * fact(n-1));
}
```



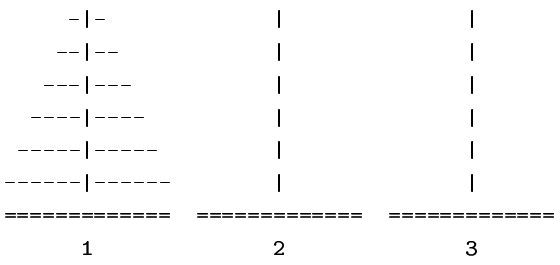


```
static int ack(int m, int n) {
  if (m == 0)
      return n + 1;
  else
      if (n == 0)
          return ack (m - 1, 1);
      else
          return ack (m - 1, ack (m, n - 1));
}
```

Cette fonction termine (lentement), mais c'est plus difficile à voir...

Les tours de Hanoi :

*n disques à faire passer un à un du poteau 1 vers le poteau 3 sans jamais déposer un disque plus grand sur un plus petit.*




Problème général : `hanoi(n, i, j)`

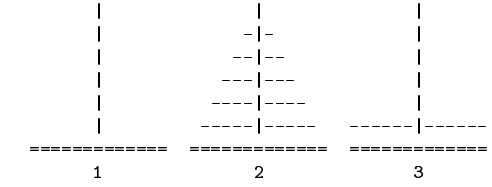




Cas $n \leq 0$ : trivial.

Si $n > 0$ :

1. faire `hanoi(n-1, 1, 2)`
2. puis $1 \rightarrow 3$
3. puis `hanoi(n-1, 2, 3)`




Mais on se trouve ramené à un problème légèrement différent.





`hanoi(n, i, j)` =

– Cas $n \leq 0$ : trivial.

– Si $n > 0$ :

1. faire `hanoi(n-1, i, 6-(i+j))`
2. puis $i \rightarrow j$
3. puis `hanoi(n-1, 6-(i+j), j)`




On est ramené au même problème que précédemment (6-(2+3) = 1), mais « strictement plus petit ».





```
static void hanoi(int n, int i, int j)  {
  if (n > 0) {
      hanoi (n-1, i, 6-(i+j));
      System.out.println (i + " -> " + j);
      hanoi (n-1, 6-(i+j), j);
  }
}
```

Comment dépend le temps d'un calcul par rapport à la taille du « sujet » à traiter.

Comptes d'apothicaires (instructions, appels, occuppation mémoire, ...)

On travaille à un facteur d'échelle près.

On mesure :
– ou bien la borne supérieure (pire des cas)
– ou alors le comportement en moyenne.





Techniques de calcul :
– récurrence
– séries formelles
– ...(techniques mathématiques sophistiquées)

Exemples ($n$ est la « taille » du problème) :
– $\log(n)$ : très bonne complexité, sub-linéaire.
– $n$ : linéaire.
– $n\log(n)$ : quasi-linéaire.
– $n^2$, $n^3$, ... : polynômial.
– $x^n$ : exponentiel.
– ...





Tri par sélection :

1. on trouve l'élément le plus petit du tableau
2. on l'échange avec le premier élément
3. on recommence avec le « sous-tableau » final restant

De l'ordre de $n^2$ comparaisons et $n$ échanges





Tri par insertion :

1. on suppose les $i-1$ premiers éléments triés,
2. et on classe le $i$ème élément à sa place parmi les $i-1$ premiers éléments, en décalant vers la droite les éléments qui sont plus grands que lui.

```
static void triInsertion() {
  int j, v, tmp;
  for (int i = 1; i < tab.length; i++) {
      tmp = tab[i]; j = i;
      while (j > 0  &&  tab[j-1] > tmp) {
          tab[j] = tab[j-1];
          j = j-1;
      }
      tab[j] = tmp;
  }
```

Tri par insertion : au plus $n^2$ comparaisons.

On peut démontrer qu'un tri nécessite au moins $\log(n)$ comparaisons (sans information sur la distribution des éléments à trier).





Au lieu de se ramener à des problèmes juste « un peu plus petits » ($n-1$ au lieu de $n$, il est préférable de « diviser » le problème (en 2, généralement).

Si on a un seul sous-problème, et que le travail de division est de coût constant, alors on a une solution en $\log(n)$.

– $n+(n-1)+\ldots+1 = n(n-1)/2$, c'est-à-dire de l'ordre de $n^2$.
– La hauteur d'un arbre binaire à $n$ feuilles est de l'ordre de $\log(n)$

Si le travail de division a un coût de l'ordre de $n$, alors on a une solution de l'ordre de $n\log(n)$.





Quicksort [Hoare 1960]

Trier le tableau `tab` entre les indices `g` et `d`.

1. On prend un élément $v$ au hasard dans le tableau.
2. On réorganise le tableau de sorte que les éléments plus petits que $b$ soient à sa gauche, et les autres à droite ($v$ est donc à sa place définitive).
3. On travaille récursivement sur les 2 sous-tableaux de droite et de gauche.

```
static void qSort(int g, int d)  {
  if (g < d) {
      int m;
      Partitionner le tableau autour de la valeur v à l'indice m
      qSort (g, m-1);
      qSort (m+1, d);
  }
}
```





```
static void qSort(int g, int d)  {
  if (g < d) {
     int m = partition(g, d);
     qSort(g, i-1);
     qsort(i+1, d);
  }
}
static int partition(int g, int d) {
    int i, m, temp, v;
    // on prend tab[g] comme pivot
    v = tab[g]; m = g;
    for (i = g+1; i <= d; i++)
        if (tab[i] < v) {
           // on avance m et on echange tab[m] et tab[i]
           m++; temp = tab[m]; tab[m] = tab[i]; tab[i] = temp;
        }
    // on echange tab[m] et tab[g]
    temp = tab[m]; tab[m] = tab[g]; tab[g] = temp;
    return m;
}
```

Complexité ? En moyenne, $1.38 \times n\log(n)$, très bon.

Tri par fusion, ou par interclassement.

Si on dispose de 2 suites triées, il est facile de les fusionner en une seule suite triée.

Pour trier 1 tableau, on a besoin d'un tableau auxiliaire.

Un tableau ou bien vide, ou bien avec un seul élément est déjà trié.

Pour trier `tab` entre `g` et `d`, il faut :

1. trier récursivement chacune des moitiés de `tab`,
2. et les fusionner dans le tableau auxiliaire.