# *Algorithmique I*

Augustin.Lux@imag.fr

Roger.Mohr@imag.fr

Maud.Marchal@imag.fr

Télécom 2006/07

# *Organisation en Algorithmique*

2 séances par semaine pendant 8 semaines.
Enseignement cours + TD + un peu de tp

- 6 semaines “normales”: 2 séances CTD
- 2 semaines TD + TP (encadré - coeff 2)

Examen: 3h (coeff 18)
Documents:

- polycopié - version papier
- kiosque: documents divers, poly en html
- livres intéressants

Tronc commun avec l’Ensimag.

# *Lien avec d'autres matières*

En télécom 2a, l'algorithmique est partout présente, notamment

- projets
- compilation
- système

Pour cette raison, il n'y a pas de tp conséquent spécifique à l'algorithmique.

# *Style de travail*

En algorithmique, il y a 3 aspects:

- connaissance encyclopédique des problèmes et des méthodes - c'est très vaste "L'introduction à l'algorithmique" (Cormen et al.) fait . . . 1008 pages.
- analyse de complexité - souvent très mathématique
- savoir implanter - connaissance des langages, et des techniques de bon codage.

# *Chapitre 1: Introduction à la complexité d'algorithmes*

# *Premier exercice facile*

# *Un calcul de maximum*

La fonction suivante, écrite en C, calcule l'élément maximal d'un vecteur $T$. L'appel initial est $valmax(k)$ où $k$ est l'indice du dernier élément de T :

```
int valmax(int i)
{if(i==0)return T[0];
 else {
    if(T[i}>valmax(i-1))
      return T[i];
    else return valmax(i-1);}
}
```

# *Questions*

Un utilisateur, qui teste cette fonction sur des vecteurs de tailles croissantes, vous dit : “je ne comprends pas, la fonction marche jusqu’à 27 éléments, mais semble boucler pour 28". Comment interprétez-vous ce comportement ? Que faut-il faire pour y remédier ?

# *Questions*

- Développer l'arbre d'appels de fonctions de $valmax$ pour quelques exemples
- Mettre en equation la complexité de $valmax$ dans le cas favorable - cas défavorable.

# *Deuxième exercice facile*

# *Calcul des 2 éléments maximaux d'un vecteur*

On désire trouver les deux éléments les plus grands d'un tableau $T$. On supposera que les données sont équidistribuées. Comparer la complexité des deux versions `algo1` et `algo2` ci-dessous réalisant ce calcul. Que peut-on en conclure ?

# Algo1

```
void algo1(int n,long* x, long& m1, long& m2) {
  long max1 = x[n-1] ;
  long max2 = x[n-2] ;
  if (max1 < max2) {
    long aux = max1 ;
    max1 = max2 ;
    max2 = aux ;
  }
  for ( int i=n-3; i>=0; --i ) {
    if ( x[i] >= max1 ) {
      max2 = max1 ;
      max1 = x[i] ;
    }
    else if ( x[i] > max2 ) {
      max2 = x[i] ;
    }
  }
```

# *Algo2*

La seconde version `algo2` consiste à remplacer la partie intérieure à la boucle par :

```
if ( x[i] > max2 ) {
   if ( x[i] >= max1 ) {
      max2 = max1 ;
      max1 = x[i] ;
   }
   else {
      max2 = x[i] ;
   }
}
```

La complexité dans le cas le pire est la même pour les deux versions de l'algorithme, et correspond au cas où le tableau est trié en ordre décroissant. On a alors $2(n-2)+1 = 2n-3$ comparaisons.

Complexité en moyenne :

Version 1 : Le second test de la boucle est fait dans les cas où $x(i) < \max_1$, soit avec une probabilité $\frac{i-1}{i}$ (il y a en effet $i-1$ chances sur $i$ pour que le plus grand de $i$ éléments ne soit pas placé en dernier).

En tenant compte des $n-2$ occurences du premier test de la boucle, on a :

$$\begin{aligned} C_{moy}(n) &= 1 + n - 2 + \sum_{i=3}^{n} \frac{i-1}{i} \\ &= 2n - 2 - H_n + \frac{1}{2} \end{aligned}$$

où $H_n = \sum_{i=1}^{n} \frac{1}{i} \sim log(n)$

- Version 2 : Evaluons la probabilité pour que le second test de la boucle soit effectué : C'est le cas soit si $x(i)$ est le premier maximum parmi les i premiers éléments du tableau, soit s'il est le second. Or, ces évènements ont chacun une probabilité de $1/i$, d'où :

$$\begin{aligned} C_{moy}(n) &= 1 + n - 2 + \sum_{i=3}^{n} \frac{2}{i} \\ &= n - 4 + 2H_n \end{aligned}$$

Pour des grandes valeurs de $n$, la version 2 de l'algorithme tend donc à faire deux fois moins de comparaisons que la première. Si la fonction de comparaison est coûteuse, la version 2 sera donc environ deux fois plus efficace. Cela illustre le fait que, même pour de petits programmes très simples, il est judicieux de bien choisir l'ordre des instructions!

D'un point de vue pratique, les programmes effectuant non seulement des comparaisons mais aussi des affectations, tout dépend du coût de la comparaison et de l'architecture (et du compilateur) utilisée. Le tableau **??** compare les performances pratiques des deux programmes sur la plateforme suivante :

- Machine hardware: sun4m
- OS version: SOLARIS 5.5
- Processor type: sparc

- Hardware: sun4m
- Compilateur: gcc version 2.8.0

Le temps du programme 1 est alors environ 1.3 celui du programme 2, (à comparer au facteur 2 en nombre de comparaisons).

| Taille tableau | temps cpu Algo1 | temps cpu Algo2 | Ratio Algo |
|---|---|---|---|
| 10 | 0.00022 | 0.00021 | 1.11 |
| 50 | 0.00076 | 0.00066 | 1.22 |
| 100 | 0.0013 | 0.00098 | 1.35 |
| 1000 | 0.012 | 0.0088 | 1.37 |
| 10000 | 0.13 | 0.097 | 1.35 |
| 100000 | 1.46 | 1.12 | 1.30 |
| 1000000 | 19.2 | 15.6 | 1.23 |

Figure 1: Résultats en moyenne sur 10 exécutions de chaque algorithme. Chaque exécution est faite sur un nouveau tableau dont les éléments sont générés par `rand`. Pour chaque tableau le temps est mesuré sur 100 exécutions.

# *Les algorithmes de tri*

# *Tri par insertion*

Principe : Insérer le $i^{\text{ème}}$ élément dans la suite des $i-1$ éléments déjà triés

```
void TriInsertion( int A[], int inf, int sup){
        int i,j,val;
        for (j=inf+1; j<sup ; j++){
                val = A[j];
                i = j-1;
                while (i >= 0 && A[i] > val){
                        A[i+1] = A[i];
                        i--;
                }
                A[i+1] = val;
        }
}
```

# *Tri par sélection*

Principe : Chercher le plus grand élément et le mettre à la fin

```
void TriSelection ( int A[], int inf, int sup){
        int i,j,val,max;
        for (i=sup ; i>inf ; i--){
                max = i;
                for (j=i-1 ; j>=inf ; j--){
                        if ( A[j] > A[max])
                                max = j;
                }
                val = A[max];
                A[max] = A[i];
                A[i] = val;
        }
}
```

# *Tri par segmentation*

Principe : Trouver un pivot et ordonner chacun des sous-tableaux autour du pivot

```
void TriSegmentation(int A[], int inf, int sup){
        int i;
        if (sup > inf){
                i = Segmentation(A,inf,sup);
                TriSegmentation(A,inf,i-1);
                TriSegmentation(A,i+1,sup);
        }
}
```

# Tri par segmentation(2)

```
int Segmentation(int A[], int inf, int sup){
        int i,j,val,u;
        val = A[sup];
        i = inf-1;
        j = sup;
        for (;;){
                while (i<sup && A[++i] < val);
                while (j>inf && A[--j] > val);
                if (i >= j) break;
                u = A[i]; A[i] = A[j]; A[j] = u;
        }
        u = A[i]; A[i] = A[sup]; A[sup] = u;
        return i;
}
```

# *Tri par fusion*

Principe : Diviser en deux chaque sous-élément du tableau, trier les sous-éléments puis les fusionner

```
void TriFusion(int A[], int inf, int sup){
        int i;
        if (inf < sup){
                i = (inf + sup)/2;
                TriFusion(A,inf,i);
                TriFusion(A,i+1,sup);
                Fusionner(A,inf,i,sup);
        }
}
```

# Tri par fusion(2)

```
void Fusionner(int A[], int inf, int m, int sup){
        int aux[sup-inf+1];
        int i,j,k;
        for (i=m+1; i>inf ; i--)
                aux[i-1-inf] = A[i-1];
        for (j=m; j<sup; j++)
                aux[sup+m-j-inf] = A[j+1];
        i = inf; j= sup;
        for (k=inf; k<= sup; k++){
                if (aux[i-inf]< aux[j-inf])
                        A[k] = aux[i-inf]; i++;
                else
                        A[k] = aux[j-inf]; j--;
        }
}
```