Algorithmique et structures des données

Plan du cours

- Rappels
- Récursivité
- Listes
- Files
- Piles
- Arbres
- Graphes
- Fichiers
- ...

Rappels

- *Un algorithme est suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème.*

- **Trouver une méthode** de résolution du problème
- Trouver une méthode **efficace : complexité**

- **Exemple 1:**
  - **problème : calculer x^n**

    données : x : réel , n: entier

    Méthode 1 : $x^0 = 1$; $x^i = x * x^{i-1}$ $i > 0$

    Méthode 2 : $x^0 = 1$;

    $x^i = x^{i/2} * x^{i/2}$, si i est pair;

    $x^i = x * x^{i/2} * x^{i/2}$ si i est impair

    ...

    résultats : y = x^n

  - Laquelle choisir? et pourquoi?
    - Plutôt la deuxième.

=>*Analyse de la complexité des algorithmes*

$T = n$

$T = \log n$

- *La complexité d'un algorithme est la mesure du nombre d'opérations fondamentales qu'il effectue sur un jeu de données. La complexité est exprimée comme une fonction de la taille du jeu de données.*
- *Notation:*
  - *Dn: ensemble des données de taille n*
  - *C(d) le coût de l'algorithme sur la donnée d.*

- **<u>Complexité au pire</u>** : $T\max(n) = \max_{d \in Dn} C(d)$. C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à n.
  - **Avantage** : il s'agit d'un maximum, et l'algorithme finira donc toujours avant d'avoir effectué $T\max(n)$ opérations.
  - **Inconvénient** : cette complexité peut ne pas refléter le comportement « usuel » de l'algorithme, le pire cas pouvant ne se produire que très rarement, mais il n'est pas rare que le cas moyen soit aussi mauvais que le pire cas.

- **Classes de complexité**

  Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité :

  – Les algorithmes ***sub-linéaires*** dont la complexité est en général en $O(\log n)$.

  – Les algorithmes ***linéaires*** en complexité $O(n)$ et ceux en complexité en $O(n \log n)$ sont considérés comme rapides.

  – Les algorithmes ***polynomiaux*** en $O(n^k)$ pour $k > 3$ sont considérés comme lents, sans parler des algorithmes ***exponentiels*** (dont la complexité est supérieure à tout polynôme en n) que l'on s'accorde à dire impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

La récursivité

- *Une définition récursive est une définition dans laquelle intervient ce que l'on veut définir. Un algorithme est dit récursif lorsqu'il est défini en fonction de lui-même.*
- Revenons à la fonction puissance $x \to x^n$. Cette fonction peut être définie récursivement :

$$x^n = \begin{cases} 1 & \text{si } n = 0; \\ x \times x^{n-1} & \text{si } n \geq 1. \end{cases}$$

- Une définition récursive peut contenir plus d'un appel récursif

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon.} \end{cases}$$

- Des définitions sont dites *mutuellement récursives* si elles dépendent les unes des autres.

$$\text{pair}(n) = \begin{cases} \textit{vrai} & \text{si } n = 0; \\ \text{impair}(n-1) & \text{sinon;} \end{cases} \quad \text{et} \quad \text{impair}(n) = \begin{cases} \textit{faux} & \text{si } n = 0; \\ \text{pair}(n-1) & \text{sinon.} \end{cases}$$

– **Récursivité imbriquée**

La fonction d'Ackermann est définie comme suit :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1,A(m,n-1)) & \text{sinon} \end{cases}$$

- **Principe:** c’est le même principe que celui de la démonstration par récurrence en mathématiques. On doit avoir :
  - un certain nombre de cas dont la résolution est connue, ces « cas simples » formeront les cas d’arrêt de la récursion ;
  - un moyen de se ramener d’un cas « compliqué » à un cas « plus simple ».

- Intérêt: La récursivité permet d’écrire des algorithmes concis et élégants

- **Difficultés :**
  - la définition peut être dénuée de sens :

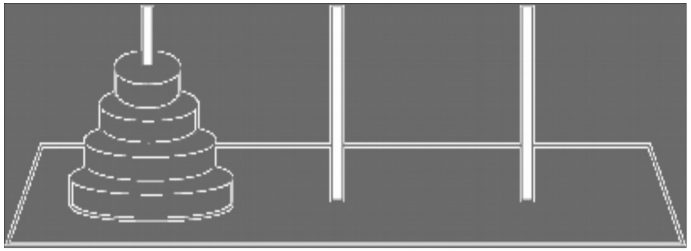
    fonction A(n)

    renvoyer A(n)
  - il faut être sûr que l'on retombera toujours sur un cas connu, c'est-à-dire sur un cas d'arrêt ; il nous faut nous assurer que la fonction est complètement définie, c'est-à-dire, qu'elle est définie sur tout son domaine d'application

Exemple: la fonction puissance

```
int puissance (int x, int n)
{
  if (n==0)
     return 1;
  else return x* puissance (x, n-1);
 }
```

Exemple: les tours de Hanoï



Les tours de hanoï est un jeu solitaire dont l'objectif est de deplacer les disques qui se trouvent sur une tour (par exemple ici la premiere tour, celle la plus a gauche) vers une autre tour (par exemple la derniere, celle la plus a droite) en suivant les regles suivantes :

- on ne peut deplacer que le disque se trouvant au sommet d'une tour ;
- on ne peut deplacer qu'un seul disque a la fois ;
- un disque ne peut pas être pose sur un disque plus petit.

```
void hanoi(int n, int i, int j, int k){
/*Affiche les messages pour déplacer n disques
de la tige i vers la tige k en utilisant la tige j */
if (n > 0)
  {
    hanoi(n-1, i, k, j)
    printf (‘Déplacer %d vers %d’, i,k);
    hanoi(n-1, j, i, k)
    }
 } /* fin de la fonction */
```

Le paradigme « diviser pour régner »

- **Principe**

  Nombres d'algorithmes ont une structure récursive : pour résoudre un problème donné, ils s'appellent eux-mêmes récursivement une ou plusieurs fois sur des problèmes très similaires, mais de tailles moindres, résolvent les sous problèmes de manière récursive puis combinent les résultats pour trouver une solution au problème initial.

  Le paradigme « diviser pour régner » donne lieu à trois étapes à chaque niveau de récursivité :

- **Diviser :** le problème en un certain nombre de sous-problèmes ;
- **Régner :** sur les sous-problèmes en les résolvant récursivement ou, si la taille d'un sous-problème est assez réduite, le résoudre directement ;
- **Combiner :** les solutions des sous-problèmes en une solution complète du problème initial.

- **Exemple : multiplication naïve de matrices**
  - Nous nous intéressons ici à la multiplication de matrices carrés de taille n.
  - **Algorithme naïf**

```
MULTIPLIER-MATRICES(A, B)
Soit n la taille des matrices carrés A et B
Soit C une matrice carré de taille n
Pour i ← 1 à n faire
   Pour j ← 1 à n faire
    cij ← 0
    Pour k ← 1 à n faire
          cij ← cij + aik * bkj
renvoyer C
```

Cet algorithme effectue $O(n^3)$ multiplications et autant d'additions

- Nous supposons que n est une puissance exacte de 2. Décomposons les matrices A, B et C en sous-matrices de taille $n/2 * n/2$. L'équation $C = A * B$ peut alors se récrire :

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

- En développant cette équation, nous obtenons :

$$r = ae + bf, \quad s = ag + bh, \quad t = ce + df \quad \text{et} \quad u = cg + dh.$$

- Chacune de ces **quatre** opérations correspond à **deux** multiplications de matrices carrés de taille $n/2$ et une addition de telles matrices. À partir de ces équations on peut aisément dériver un algorithme « diviser pour régner » dont la complexité est donnée par la récurrence :

$$T(n) = 8T(n/2) + \Theta(n^2).$$

- l'addition des matrices carrés de taille $n/2$ étant en $\Theta(n^2)$.

Dérécursivation

- **Dérécursiver**, c’est transformer un algorithme récursif en un algorithme équivalent ne contenant pas d’appels récursifs

- *Un algorithme est dit récursif terminal s'il ne contient aucun traitement après un appel récursif.*

- Les programmes itératifs sont souvent plus efficaces,
- mais les programmes récursifs sont plus faciles à écrire.
- Les compilateurs savent, la plupart du temps, reconnaître les appels récursifs terminaux, et ceux-ci n’engendrent pas de surcoût par rapport à la version itérative du même programme.
- Il est toujours possible de dérécursiver un algorithme récursif.