Module EA4 – Éléments d'Algorithmique

Dominique Poulalhon
dominique.poulalhon@liafa.univ-paris-diderot.fr

Université Paris Diderot
L2 Informatique, Math-Info et EIDD
Année universitaire 2013-2014

Première interrogation lundi 17 février

- Amphi 10 E : groupes Info 1, Info 2 et Info 3 (A-K)
- Amphi 12 E : groupes Info 3 (L-Z), Info 4 et Math-Info

Calcul de a^n : l'exponentiation binaire

```
def puissance(a, n) :
  if n == 0 : return 1
  tmp = puissance(a, n//2)
  carre = tmp * tmp              # une multiplication
  if n%2 == 0 : return carre
  else : return a * carre        # une multiplication
```

Complexité

$\Theta(\log_2 n)$ multiplications de la forme $a^k \cdot a^\ell$, $k \in \{1, \ell\}$

Calcul de a^n : l'exponentiation binaire

Complexité

$\Theta(\log_2 n)$ multiplications de la forme $a^k \cdot a^\ell$

Calcul de a^n : l'exponentiation binaire

Complexité

$\Theta(\log_2 n)$ multiplications de la forme $a^k \cdot a^\ell$

si ces multiplications ont un coût constant, *i.e.* si les opérandes ont une taille constante, complexité en $\Theta(\log_2 n)$

c'est le cas avec l'arithmétique modulaire ou l'arithmétique flottante utilisées usuellement : tous les nombres sont codés sur exactement 32 (ou 64) bits, donc le coût d'une multiplication est constant

Complexité

$\Theta(\log_2 n)$ multiplications de la forme $a^k \cdot a^\ell$

si ces multiplications ont un coût constant, *i.e.* si les opérandes ont une taille constante, complexité en $\Theta(\log_2 n)$

sinon il faut tenir compte du coût de ces multiplications ; par exemple en arithmétique exacte sur des entiers :

valeur	taille (en bits)	coût du calcul naïf du carré
a	$\log_2 a$	$\Theta((\log_2 a)^2)$
a^k	$k \cdot \log_2 a$	$\Theta(k^2 \cdot (\log_2 a)^2)$

Application : calcul du n^e terme de la suite de Fibonacci

suite définie par $F_0 = F_1 = 1$ et $\forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

utilisation naïve de la récurrence $\Longrightarrow$ $\Theta(\alpha^n)$ additions

```
def fibo(n) :
  if n <= 2 : return 1
  return fibo(n-1) + fibo(n-2)
```

Application : calcul du n^e terme de la suite de Fibonacci

suite définie par $F_0 = F_1 = 1$ et $\forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

utilisation naïve de la récurrence $\Longrightarrow$ $\Theta(\alpha^n)$ additions

calcul itératif des premières valeurs $\Longrightarrow$ $\Theta(n)$ additions

```
def fibo(n) :
  previous, last = 1, 1
  for i in range(1, n) :
    previous, last = last, previous + last
  return last
```

Application : calcul du n^e terme de la suite de Fibonacci

suite définie par $F_0 = F_1 = 1$ et $\forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

utilisation naïve de la récurrence $\Longrightarrow$ $\Theta(\alpha^n)$ additions

calcul itératif des premières valeurs $\Longrightarrow$ $\Theta(n)$ additions

Peut-on faire encore mieux ?

Application : calcul du n^e terme de la suite de Fibonacci

suite définie par $F_0 = F_1 = 1$ et $\forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

Lemme

$$\forall n \geqslant 1,\quad \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix}$$

```
def fibo(n) :
  M = [ [1, 1], [1, 0] ]
  M = puissance_matrice_2_2 (M, n)
  return M[0][0]
```

Recherches dans une liste

`recherche(x, L)`

Étant donné une liste `L` et un élément `x`, déterminer si `x` apparaît dans `L`

recherche(x, L)

Étant donné une liste L et un élément x, déterminer si x apparaît dans L

```
def recherche_lineaire(x, L) :
  for elt in L :
    if elt == x : return True
  return False
```

recherche(x, L)

Étant donné une liste L et un élément x, déterminer si x apparaît dans L

Variante : retourner une position où x apparaît

```
def recherche_lineaire(x, L) :
  for (i, elt) in enumerate(L) :
    if elt == x : return i
  return -1
```

`occurrences(x, L)`

Étant donné une liste `L` et un élément `x`, compter les occurrences de `x` dans `L`

occurrences(x, L)

Étant donné une liste L et un élément x, compter les occurrences de x dans L

```
def occurrences(x, L) :
  res = 0
  for (i, elt) in enumerate(L) :
    if elt == x : res += 1
  return res
```

`max(L)`

Étant donné une liste `L` contenant des éléments comparables, déterminer le plus grand élément qui apparaît dans `L`

max(L)

Étant donné une liste L contenant des éléments comparables, déterminer le plus grand élément qui apparaît dans L

```
def max(L) :
  tmp = L[0]
  for elt in L :
    if elt > tmp : tmp = elt
  return tmp
```

Recherche dans un tableau *trié*

`recherche(x, T)`

Étant donné un tableau `T` *trié* et un élément `x`, déterminer si `x` apparaît dans `T`

on peut alors faire beaucoup plus efficace :

Recherche dans un tableau *trié*

recherche(x, T)

Étant donné un tableau T *trié* et un élément x, déterminer si x apparaît dans T

on peut alors faire beaucoup plus efficace :

```
def recherche_dichotomique(x, T) :
  if len(T) == 0 : return False
  n = len(T)//2
  if x == T[n] : return True
  elif x < T[n] : return recherche_dichotomique(x, T[:n])
  else : return recherche_dichotomique(x, T[n+1:])
```