Algorithmes et complexité

Définitions, notation O

Training beOI

11 février 2016

Table of Contents

Algorithmes

Complexité

Qu'est-ce qu'un algorithme ?

- Une manière de calculer un résultat
- Une idée pour résoudre un problème
- Une suite d'instructions
- La description d'un programme

Qu'est qu'un *bon* algorithme ?

Domaine du programmeur :

- Ne crashe pas
- Se termine
- Donne la bonne réponse

Notre domaine :

- Est rapide
- Utilise peu de mémoire
- **Est accepté en concours**

Table of Contents

Algorithmes

Complexité

Mesurer l'efficacité

Idées :

- Chronométrer
- Mesurer la RAM

Mais varie selon :

- Le langage
- L'implémentation
- La machine
- L'heure de la journée

La notation O

Trouver une notion d'efficacité *intrinsèque* :

- Faisons grandir l'input
- Regardons comment la vitesse évolue

Exemple : calculer $1 + \cdots + n$. Si n multiplié par 2, le temps d'exécution :

- Reste constant : $O(1)$
- Est multiplié par 2 : $O(n)$
- Est multiplié par 4 : $O(n^2)$

Temps constant

Problème : calculer la somme $1 + 2 + \cdots + n$.

Solution 1 : Un simple calcul

```
int sum = n * (n-1) / 2;
```

- Temps ne change pas si n double
- Temps “constant”
- Complexité $O(1)$
- Temps “proportionnel à 1”

Temps linéaire

Solution 2 : Une boucle

```
int sum = 0;
for (int i = 1; i <= n; i++)
    sum += i;
```

- Temps double si n double
- Temps “linéaire”
- Complexité $O(n)$
- Temps “proportionnel à n”

Temps quadratique

Solution 3 : Deux boucles (stupide !)

```
int sum = 0;
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= i; j++)
        sum++;
```

- Temps quadruple si n double
- Temps “quadratique”
- Complexité $O(n^2)$
- Temps “proportionnel à n^2”

Puissances

Définition :

- Multiplications en chaîne
- “3 exposant n”
- $3^n = \underbrace{3 \times \cdots \times 3}_{n \text{ fois}}$

Exemples :

- $3^0 = 1$ (par définition)
- $3^1 = 3$
- $3^2 = 3 \times 3 = 9$ (carré)
- $3^3 = 3 \times 3 \times 3 = 27$ (cube)

Logarithmes : intuition (1)

Jeu à deux joueurs :

- Alice choisit entre 1 et 16
- À chaque tour :
  - Bob donne à Alice un ou plusieurs nombres
  - Alice dit si le nombre est parmi ceux-là
- Quand Bob trouve le nombre il a gagné
- Comment gagner en peu de tours ?

Logarithmes : intuition (2)

Stratégie : Donner la moitié des nombres possibles

- D'abord 8 nombres parmi les 16
- Puis 4 parmi les 8 restants
- Puis 2 parmi les 4 restants
- Puis 1 des 2 restants
- Trouvé !

Donc 4 questions suffisent.

Logarithmes : intuition (3)

De manière générale, avec n nombres au départ, combien de questions ?

Combien de fois peut-on couper en deux ?

- Si $n = 2$, une fois
- Si $n = 4$, deux fois
- Si $n = 8$, trois fois
- Si $n = 16$, quatre fois

Logarithme en base 2

La fonction qui répond à cette question est $\log_2$: le logarithme en base 2. Par exemple

- $\log_2(2) = 1$
- $\log_2(4) = 2$
- $\log_2(8) = 3$
- $\log_2(16) = 4$

La stratégie de Bob est en $O(\log_2(n)) = O(\log n)$.

En fait, le logarithme, c'est l'exposant qu'il faut mettre à 2 pour atteindre n :

$$x = \log_2(n) \Leftrightarrow 2^x = n$$

Logarithme général (bonus)

Pas seulement pour deux ! Le logarithme en base a, c'est le nombre de fois qu'on peut diviser par a. Par exemple :

- $\log_3(27) = 3$
- $\log_4(16) = 2$
- $\log_5(5) = 1$

En fait, c'est l'exposant qu'il faut mettre à a pour atteindre n :

$$x = \log_a(n) \Leftrightarrow a^x = n$$

C'est un peu "l'inverse" des puissances.

Recherche dans un tableau trié (1)

On nous donne un tableau dans l'ordre croissant :

1	4	6	9	15	23	24

Vérifier si un nombre x s'y trouve.

Solution 1 : Tout parcourir, linéaire $O(n)$

```
bool isIn(int tab[], int n, int x)
{
    for (int i = 0; i < n; i++)
        if (tab[i] == x)
            return true;
    return false;
}
```

Recherche dans un tableau trié (2)

On cherche 7.

Idée : regarder au milieu et comparer :

1	4	6	**9**	15	23	24

Trop grand ($9 > 7$), allons à gauche :

1	**4**	6	9	15	23	24

Trop petit ($4 < 7$), allons à droite :

1	4	**6**	9	15	23	24

Mais $6 \neq 7$ donc 7 n'est pas dans le tableau.

Recherche dans un tableau trié (3)

On coupe en deux à chaque fois $\Rightarrow \log_2(n)$ essais.

Solution 2 : Recherche dichotomique, logarithmique $O(\log n)$

```
bool isIn(int tab[], int n, int x)
{
    int left = 0, right = n-1;
    while (left <= right)
    {
        int middle = (left+right) / 2;
        if (x < tab[i]) right = middle - 1;
        else if (x > tab[i]) left = middle + 1;
        else return true;
    }
    return false;
}
```

Beaucoup plus rapide !

Limites pratiques

Limites sur n pour s'exécuter en quelques secondes :

Complexité	Limite de n	Exemple
$O(1), O(\log n)$	$\leq 10^{18}$	(Taille limite d'un entier)
$O(n)$	$\leq 100\,\mathrm{M}$	Parcourir un tableau
$O(n \log n)$	$\leq 1\,\mathrm{M}$	Trier un tableau
$O(n^2)$	$\leq 10\,\mathrm{k}$	Boucle dans une boucle

En concours : regarder la deuxième colonne et cela donne la complexité.