Complexité des problèmes

II. Rappels d'algorithmique

Cyril Nicaud

M1 Informatique, 2014–2015

Algorithme

Définition

Un algorithme est un procédé automatique, qui résoud un problème en un nombre fini d'étapes.

Exemples :

$$\begin{array}{cccccc} & & 1 & 1 & & \\ & 4 & 3 & 7 & 8 & 1 \\ + & 1 & 1 & 5 & 3 & 4 \\ \hline & 5 & 5 & 3 & 1 & 5 \end{array}$$

```
def search(T,x):
  for y in T:
    if y == x:
      return True
  return False
```

Un problème, plusieurs solutions

Recherche dans un tableau trié

Comment tester si x est dans le tableau trié T de taille n ?

```
def search(T,x):
  for y in T:
    if y == x:
      return True
  return False
```

$\mathcal{O}(n)$

```
def binarySearch(T,x,begin,end):
    while begin <= end:
        m = (begin+end) // 2
        if T[m] == x:
            return True
        if T[m] < x:
            begin = m + 1
        else:
            end = m - 1
    return False
```

$\mathcal{O}(\log n)$

Complexité

Définition

Étudier la complexité d'un algorithme consiste à estimer la quantité de ressources utilisées par l'algorithme (en temps de calcul, en espace mémoire, ...).

Taille des entrées

- Pour définir formellement la complexité, on a besoin d'une notion de taille sur les objets
- La taille d'une chaîne de caractères est naturellement son nombre de symboles, la taille d'un tableau son nombre de cases
- La taille d'un entier peut être considérée comme 1 (si on ne manipule pas de très grands entiers), sinon n est de taille $\approx \log_2 n$ (nombre de bits pour représenter n en binaire)
- On notera $\mathcal{E}_n$ l'ensemble des entrées de taille n

Modèle d'ordinateur

- Il faut définir un langage algorithmique et un ordinateur mathématique pour se langage
- L'idée est similaire aux modèles en physique, chimie, ...

- L'un des principaux modèle d'ordinateur est la machine de Turing
- L'un des plus utilisé pour l'analyse de complexité est la Random Access Machine (RAM)
- (ne pas confondre avec la Random Access Memory qui est la mémoire vive)

- On étudiera tout ça plus tard dans le cours, pour le moment on considère qu'on peut faire toutes les instructions (et accès mémoire) du C, Java ou du Python en temps constant $\mathcal{O}(1)$

Fonction de coût

- Les instructions et accès mémoire en temps constant $\mathcal{O}(1)$
- $\mathcal{E}_n$ l'ensemble des entrées de taille n

Définition

Si E est une entrée de l'algorithme, on note $C(E)$ le coût en ressources (typiquement en temps) de son exécution pour E. La fonction C est appelée la fonction de coût.

- C'est une fonction de l'ensemble de toutes les entrées possibles, à valeurs dans $\mathbb{R}^+$
- C'est trop compliqué à décrire totalement en général
- On va en donner une description simplifiée

Types de complexités

- Les instructions et accès mémoire en temps constant $\mathcal{O}(1)$
- $\mathcal{E}_n$ l'ensemble des entrées de taille n

On considère deux types de complexités :

- La complexité dans le pire des cas :

$$\text{Pire}(n) = \max_{E \in \mathcal{E}_n} C(E)$$

- La complexité en moyenne :

$$\text{Moyenne}(n) = \sum_{E \in \mathcal{E}_n} \mathbb{P}_n(E)\, C(E),$$

où $\mathbb{P}_n$ est une probabilité sur $\mathcal{E}_n$

Formulations simples nécessaires

- Les instructions et accès mémoire en temps constant $\mathcal{O}(1)$
- $\mathcal{E}_n$ l'ensemble des entrées de taille n
- On s'intéresse à la complexité dans le pire cas

- Maintenant on a une suite $n \mapsto \text{Pire}(n)$
- C'est plus simple, mais toujours trop compliqué à décrire

Si on a deux algorithmes dont les complexités sont

$$\text{Pire}(n) = \sum_{i=1}^{n/2} \binom{n}{i}$$

et

$$\text{Pire}(n) = \frac{(2n)!}{n!^2},$$

lequel est le plus efficace ???

La notation $\mathcal{O}$

Définition

Si u_n et v_n sont deux suites positives on dit que $u_n \in \mathcal{O}(v_n)$ quand il existe une constante $C > 0$ telle que

$$u_n \leq C\, v_n, \text{ pour } n \text{ assez grand.}$$

Exemples

$$\begin{array}{ccc} n^2 \in \mathcal{O}(n^3); & n^3 \notin \mathcal{O}(n^2); & n^2 \in \mathcal{O}(\tfrac{1}{2}n^3) \\ n^2 + 100 \in \mathcal{O}(\tfrac{1}{100}\, n^2); & 2^n \notin \mathcal{O}(n^3); & 2^n \in \mathcal{O}(3^n) \\ 3^n \in \mathcal{O}(2^n); & \sqrt{n} \in \mathcal{O}(n); & \sqrt{n} \notin \mathcal{O}(\log n) \end{array}$$

Remarque : $\mathcal{O}(1)$ est l'ensemble des suites bornées

La fonction logarithme

- ln et exp sont les deux fonctions un peu avancées qu'l'on retrouve dans les complexités
- Elles sont inverses l'une de l'autre : $x = e^y \Leftrightarrow y = \ln x$
- Ce qu'il faut savoir:
  - $\ln n$ croît moins vite que n'importe quel n^α, pour $\alpha > 0$
  - e^n croît plus vite que n'importe quel n^α, pour $\alpha > 0$

Comparaison des suites usuelles

Si a, b, c, d sont des réels positifs ou nuls et

$$u_n = n^a (\ln n)^b \qquad v_n = n^c (\ln n)^d,$$

alors $u_n \in \mathcal{O}(v_n)$ si et seulement si
• ou bien $a < c$
• ou bien $a = c$ et $b \leq d$

Complexité des programmes itératifs

Principe de base

Pour estimer la complexité en temps d'un algorithme, on repère l'instruction (ou les instructions) la plus effectuée, et on majore le nombre de fois qu'elle(s) va être effectuée dans le pire des cas.

```
def search(T,x):
  for y in T:
    if y == x:
      return True
  return False
```

Le `if` est l'instruction la plus effectuée. Elle est effectuée n fois, où n est la taille du tableau. La complexité est donc $\mathcal{O}(n)$.

Complexité des programmes itératifs

Principe de base

Pour estimer la complexité en temps d'un algorithme, on repère l'instruction (ou les instructions) la plus effectuée, et on majore le nombre de fois qu'elle(s) va être effectuée dans le pire des cas.

```
def bubble(T):
  for i in range(len(T)):
    for k in range(len(T)-1):
      if T[k] < T[k+1]:
        T[k],T[k+1] = T[k+1],T[k]
```

Le `if` est encore l'instruction la plus effectuée. Elle est effectuée moins de n^2 fois, où n est la taille du tableau. La complexité est donc $\mathcal{O}(n^2)$.

Complexité des programmes itératifs

Principe de base

Pour estimer la complexité en temps d'un algorithme, on repère l'instruction (ou les instructions) la plus effectuée, et on majore le nombre de fois qu'elle(s) va être effectuée dans le pire des cas.

```
def bubble(T):
  for i in range(len(T)):
    for k in range(len(T)-1):
      if T[k] < T[k+1]:
        T[k],T[k+1] = T[k+1],T[k]
```

Le `if` est encore l'instruction la plus effectuée. Elle est effectuée moins de n^{100} fois, où n est la taille du tableau. La complexité est donc $\mathcal{O}(n^{100})$. Mathématiquement c'est correct, mais c'est comme dire que $\pi \leq 1\,000\,000$. C'est correct, mais moins informatif que $\pi \leq 4$.

Complexité des programmes itératifs

Principe de base

Pour estimer la complexité en temps d'un algorithme, on repère l'instruction (ou les instructions) la plus effectuée, et on majore le nombre de fois qu'elle(s) va être effectuée dans le pire des cas.

On essaye d'être le plus précis possible dans la majoration.

Autres notations (utiles et à connaître !):

- On dit que $u_n \in \Omega(v_n)$ quand $v_n \in \mathcal{O}(u_n)$
- On dit que $u_n \in \Theta(v_n)$ quand $u_n \in \mathcal{O}(v_n)$ et $u_n \in \Omega(v_n)$

Exemple : `bubble` est en $\Theta(n^2)$, mais pas en $\Theta(n^{100})$

Complexité des programmes récursifs

Principe de base

Soit $T(n)$ le temps dans le pire des cas pour exécuter un algorithme sur des entrées de taille n. On peut souvent exprimer $T(n)$ en fonction de $T(i)$ pour $i < n$, directement à partir du programme récursif.

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

Pour $n \geq 2$, on fait un appel pour n qui demande en plus un temps $T(n-1)$ et un temps $T(n-2)$, d'où

$$T(n) = 1 + T(n-1) + T(n-2)$$

Complexité des programmes récursifs

Principe de base

Soit $T(n)$ le temps dans le pire des cas pour exécuter un algorithme sur des entrées de taille n. On peut souvent exprimer $T(n)$ en fonction de $T(i)$ pour $i < n$, directement à partir du programme récursif.

```
def merge(T):
  if len(T) < 2:
    return T
  m = len(T)/2
  L = merge(T[:m])
  R = merge(T[m:])
  return fusion(L,R)
```

Pour $n \geq 2$, on fait un appel pour n qui demande en plus un temps $T(n/2)$ et un autre en temps $T(n/2)$. La fusion se fait en temps linéaire d'où

$$T(n) = 2\,T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

Complexité des programmes récursifs

Premier cas simple

Si $T(n) = T(n-a) + \mathcal{O}(n^b \log^c n)$, où $a > 0$ et $b \geq 0$, alors

$$T(n) = \mathcal{O}(n^{b+1} \log^c n)$$

Deuxième cas simple

Si $T(n) = T(n-a) + T(n-b) + \Omega(1)$, où $a, b > 0$ (on peut avoir $a = b$) alors T(n) est de complexité exponentielle : $\Omega(c^n)$ pour un certain $c > 1$.

Complexité des programmes récursifs

Premier cas simple

Si $T(n) = T(n-a) + \mathcal{O}(n^b \log^c n)$, où $a > 0$ et $b \geq 0$, alors

$$T(n) = \mathcal{O}(n^{b+1} \log^c n)$$

```
def fact(n):
  if n < 2:
    return 1
  return n*fact(n-1)
```

Pour $n \geq 2$, on a besoin d'un temps $T(n-1)$ plus de faire les différents calculs de l'appel, soit $\mathcal{O}(1)$:

$$T(n) = T(n-1) + \mathcal{O}(n^0 \log^0 n)$$

Et donc $T(n) = \mathcal{O}(n)$

Complexité des programmes récursifs

Deuxième cas simple

Si $T(n) = T(n-a) + T(n-b) + \Omega(1)$, où $a, b > 0$ (on peut avoir $a = b$) alors T(n) est de complexité exponentielle : $\Omega(c^n)$ pour un certain $c > 1$.

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

Pour $n \geq 2$, on a besoin d'un temps $T(n-1)$ et $T(n-2)$ plus de faire les différents calculs de l'appel, soit $\Theta(1)$:

$$T(n) = T(n-1) + T(n-2) + \Omega(1)$$

Et donc la complexité est exponentielle.

Le théorème maître

Théorème maître

Si $a > 0$, $b > 0$ et $c \geq 0$ et que

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \mathcal{O}(n^c)$$

Alors

1. Si $b^c < a$: $T(n) = \mathcal{O}(n^{\log_b a})$
2. Si $b^c = a$: $T(n) = \mathcal{O}(n^c \log n)$
3. Si $b^c > a$: $T(n) = \mathcal{O}(n^c)$

Le théorème maître – premier cas

Théorème maître

Si $a > 0$, $b > 0$ et $c \geq 0$ et que

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \mathcal{O}(n^c)$$

Alors

1. Si $b^c < a$: $T(n) = \mathcal{O}(n^{\log_b a})$

- Multiplier deux polynômes de degrés n par la méthode classique prend un temps $\mathcal{O}(n^2)$
- La méthode de Karatsuba, par une méthode diviser pour régner, permet d'avoir

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

- On est donc dans le premier cas du théorème maître et l'algorithme a une complexité $\mathcal{O}(n^{\log_b a}) \approx \mathcal{O}(n^{1.58})$

Le théorème maître – deuxième cas

Théorème maître

Si $a > 0$, $b > 0$ et $c \geq 0$ et que

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \mathcal{O}(n^c)$$

Alors

2. Si $b^c = a$: $T(n) = \mathcal{O}(n^c \log n)$

```
def merge(T):
  if len(T) < 2:
    return T
  m = len(T)/2
  L = merge(T[:m])
  R = merge(T[m:])
  return fusion(L,R)
```

On a l'équation avec $a = b = 2$ et $c = 1$

$$T(n) = 2\,T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

On est donc dans le deuxième cas, et la complexité est $T(n) = \mathcal{O}(n \log n)$