Introduction à la complexité

Julien Romero

Décrire les performances des algorithmes

- Quand on a des algorithmes, on veut souvent analyser leurs performances :
  - D'un point de vue empirique: temps de calculs, usage de la mémoire moyenne, ...
  - D'un point de vue théorique: la complexité en instructions et en mémoire
- La complexité peut se décliner sur plusieurs cas
  - Dans le pire de cas (le plus fréquent, celui de ce cours): on calcule le nombre maximum d'instructions ou de mémoire utilisées
  - En moyenne: on calcule le nombre moyen d'instructions ou de mémoire utilisées. Souvent plus compliquée à calculer.
  - Dans le meilleur des cas (très rare): on calcule le nombre minimum d'instructions ou de mémoire utilisées

Mesurer les complexités

- Les complexités dans les cas asymptotiques sont exprimées en ordre de grandeur en fonction des entrées, souvent en utilisant des gros O: $\mathcal{O}(\dots)$
- Les complexités sont alors catégorisées en grands ensembles:
  - Constante: $\mathcal{O}(1)$
  - Logarithmique: $\mathcal{O}(log(n))$
  - Linéaire: $\mathcal{O}(n)$
  - Quadratique: $\mathcal{O}(n^2)$
  - Polynomiale: $\mathcal{O}(n^k)$
  - Exponentielle: $\mathcal{O}(k^n)$
- On simplifie au maximum les complexités !

$$\mathcal{O}(2n) \longrightarrow \mathcal{O}(n)$$

La notation grand O

- Soit n la taille des entrées on dit que $f(n)$ est en $\mathcal{O}(g(n))$ si:

$$\exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}, \forall n \geq n_0 : \ |f(n)| \leq c \cdot |g(n)|$$

- En d'autres mots, à partir d'un certain moment, la fonction g domine la fonction f.
  - Le temps d'exécution de la fonction sera bornée.
- En pratique, on va surtout avoir besoin de simplifier les $\mathcal{O}(\dots)$
  - On enlève les constantes multiplicatives $\mathcal{O}(2n) \longrightarrow \mathcal{O}(n)$
  - En ne gardant que le pire exposant/la pire fonction **pour chaque variable d'entrée** (ici n et m)

$$\mathcal{O}(1 + n + n^2) \longrightarrow \mathcal{O}(n^2)$$

$$\mathcal{O}(n^2 + 2^n) \longrightarrow \mathcal{O}(2^n)$$

$$\mathcal{O}(n^2 + 2^n + m + m^2) \longrightarrow \mathcal{O}(2^n + m^2)$$

$$\mathcal{O}(3) \longrightarrow \mathcal{O}(1)$$

$$\mathcal{O}(x) + \mathcal{O}(y) \longrightarrow max(\mathcal{O}(x), \mathcal{O}(y))$$

$$\mathcal{O}(x) \cdot \mathcal{O}(y) \longrightarrow \mathcal{O}(x \cdot y)$$

Comment calculer la complexité (dans le pire des cas)

- On parcourt notre algorithme et on ajoute la complexité de chaque ligne
  - Si on a une déclaration simple
    - La complexité est par défaut 1
    - Si on appelle une ou plusieurs fonctions, la complexité est la somme des complexités des appels de fonctions
  - Si on a une boucle:
    - La complexité est la complexité du contenu multipliée par le nombre de répétitions maximal dans le pire des cas.
  - Si on a une condition, on calcule la complexité de chaque branche et on prend la pire.
    - Cas spécial: La condition permet d'arrêter une récursion (voir plus loin)
  - Pour les boucles et les conditions, il faut aussi compter la vérification des conditions.
- À la fin, on combine toutes les complexités et on les simplifie pour obtenir un $\mathcal{O}(\dots)$
- On peut montrer que notre complexité est optimale en donnant un exemple d'entrée ayant la pire complexité.

Exemple 1 - Factorielle (pseudo-code)

```
factorielle(n):
  fact = 1;                    # 1
  for(i=1; i <= n; i++):       # 1 à chaque iter et Max iter: n
    fact = fact * i;           # 1
  return fact;
```

Complexité algorithmique: $\mathcal{O}(1 + n \cdot 1) = \mathcal{O}(n)$

Complexité mémoire: $\mathcal{O}(1)$ (2 variables créés)

Exemple 2 - Factorielle récursive (pseudo-code)

```
factorielle(n):
  if (n == 0):    # 1
      return 1;   # 1
  else:           # Pire des cas
      return n * factorielle(n - 1);   # 1 + C(n-1)
```

Complexité algorithmique:

- On l'appelle $C(n)$
- On a $C(n) = \mathcal{O}(1) + C(n-1)$ et $C(0) = \mathcal{O}(1)$
- On retrouve bien n fois $\mathcal{O}(1)$ (attention, on a bien un multiplication ici):

$$\mathcal{O}(1) + \ldots + \mathcal{O}(1) = \mathcal{O}(n)$$

Exemple 3 - Fibonacci (pseudo-code)

```
fibo(n):
  if (n == 0){ # 1
    return 0;  # 1
  else if (n == 1){ # 1
    return 1;       # 1
  else:             # Pire des cas
      return fibo(n - 1) + fibo(n - 2);  # C(n-1) + C(n-2)
```

Complexité algorithmique:

- On a $C(n) = C(n-1) + C(n-2) + \mathcal{O}(1)$ et $C(0) = \mathcal{O}(1)$ $C(1) = \mathcal{O}(1)$

Exemple 3 - Fibonacci - Calcul abbrégé

$$
\begin{aligned}
C(n) &= C(n-1) + C(n-2) + \mathcal{O}(1) \\
&= 2 \cdot C(n-2) + C(n-3) + 2 \cdot \mathcal{O}(1) \\
&= 3 \cdot C(n-3) + 2 \cdot C(n-4) + 4 \cdot O(1) \\
&= 5 \cdot C(n-4) + 3 \cdot C(n-5) + 7 \cdot O(1) \\
&= \dots \\
&= Fibo(k+1) \cdot C(n-k) + Fibo(k) \cdot C(n-k-1) + \left(\sum_{i=1}^{k} Fibo(i) \right) \mathcal{O}(1) \\
&= \dots = Fibo(n+1)C(1) + Fibo(n)C(0) + \left(\sum_{i=1}^{n} Fibo(k) \right) \mathcal{O}(1) \\
&= Fibo(n+1) \cdot \mathcal{O}(1) + Fibo(n) \cdot \mathcal{O}(1) + Fibo(n+2) \cdot \mathcal{O}(1) \\
&= \mathcal{O}(\phi^n)
\end{aligned}
$$

c.f la prépa

Exemple 4 - Fibonacci 2 (pseudo-code)

```
fibo2(n):
  if (n == 0){ # 1
    return 0;  # 1
  else if (n == 1){ # 1
    return 1;       # 1
  else:             # Pire des cas
    fibo_tmp = new int[n+1]; # n
    fibo_tmp[0] = 1; fibo_tmp[1] = 1; # 1
    for(i=2; i<=n; i++): # 1, Max iter: n
      fibo_tmp[i] = fibo_tmp[i-1] + fibo_tmp[i-2]; # 1
    return fibo_tmp[n] # 1
```

Complexité: $\mathcal{O}(n)$

Example 5 - Écart Type

```
ecartType(nombres):
  moyenne = mean(nombres);   # complexité mean=len(nombres)=n
  variance = 0;              # 1
  for (i=0; i < len(nombres); i++):          # 1, Max iter:n
    variance += (nombres[i] - moyenne) ** 2; # 1
  variance /= n;           # 1
  return sqrt(variance)    # 1
```

$$C(n) = \mathcal{O}(n + n \cdot 1 + 1) = \mathcal{O}(n)$$

Exemple 6 - Dichotomie

```
dichotomie(element, tableau_trie):
  if (tableau_trie.length <= 1):   # 1
    return 0;                      # 1
  m = tableau_trie.length // 2     # 1
  if tableau_trie[m] == element:   # 1
    return m                       # 1
  else if tableau_trie[m] > element :          # 1
    return dichotomie(element,tableau_trie[:m])# C(n//2)
  else:
    return m + dichotomie(element,tableau_trie[m:])# C(n//2)
```

$$
\begin{aligned}
C(n) &= \mathcal{O}(1) + \mathcal{C}(n/2) \\
&= \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{C}(n/4) \\
&= \ldots = \log_2(n) \cdot \mathcal{O}(1) \\
&= O(\log_2(n))
\end{aligned}
$$