Kajy University: Informatique

Algorithmes et structures de données

Author: Dimby Rabearivony

Date: 10 mai 2024

Version: 1.0

"Ny fahalalana no valamparihiko."

Table des matières

Chapitre Complexité algorithmique

La complexité algorithmique permet d'évaluer l'efficacité des algorithmes en termes de temps d'exécution et de consommation de ressources. Cela aide à comprendre combien de temps un algorithme prendra pour des entrées de tailles différentes. Les notations couramment utilisées pour décrire la complexité algorithmique sont O, Ω, et θ.

3.1 Les bornes asymptotiques

Les bornes asymptotiques, ou "asymptotic bounds," fournissent une approximation de la performance d'un algorithme pour des tailles d'entrée importantes. Elles aident à évaluer le comportement des algorithmes sous différents scénarios : meilleur cas, pire cas, et cas moyen.

- **Borne supérieure (Big O)** : Représentée par la notation $O(f(n))$, cette borne donne une estimation de la complexité maximale d'un algorithme, indiquant qu'il n'aura jamais une complexité supérieure à $O(f(n))$. Mathématiquement, cela signifie qu'il existe des constantes c et n_0 telles que :

$$T(n) \leq c \cdot f(n), \quad \text{pour tout } n \geq n_0.$$

- **Borne inférieure (Big Omega)** : Représentée par la notation $\Omega(f(n))$, cette borne donne une estimation de la complexité minimale, indiquant que l'algorithme ne peut pas être plus rapide que cette borne. Cela signifie qu'il existe des constantes c et n_0 telles que :

$$T(n) \geq c \cdot f(n), \quad \text{pour tout } n \geq n_0.$$

- **Borne moyenne (Big Theta)** : Représentée par la notation $\theta(f(n))$, cette borne indique une complexité moyenne ou attendue. Cela signifie que l'algorithme a une borne supérieure et inférieure qui convergent vers la même fonction. Mathématiquement, cela signifie qu'il existe des constantes c_1, c_2, et n_0 telles que :

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n), \quad \text{pour tout } n \geq n_0.$$

3.2 La notation O

La notation "O" (grand O) indique la complexité maximale d'un algorithme en termes de temps ou d'espace. Elle fournit une estimation du nombre maximum d'opérations nécessaires en fonction de la taille de l'entrée. Par exemple, la recherche binaire a une complexité de ‘O(log n)‘, car chaque étape réduit le problème de moitié.

Algorithm 12 Recherche binaire

```
1:  function RECHERCHE_BINAIRE(tableau, valeur)
2:      gauche := 0
3:      droite := taille(tableau) - 1
4:      Tant que gauche <= droite Faire
5:          milieu := (gauche + droite) / 2
6:          Si tableau[milieu] == valeur Alors
7:              Retourner milieu
8:          Sinon Si tableau[milieu] < valeur Alors
9:              gauche := milieu + 1
10:         Sinon
11:             droite := milieu - 1
12:         Fin Si
13:     Fin Tant que
14:     Retourner -1
15: Fin function
```

3.3 Les notations Ω et θ

Ces notations complètent la notation "O" pour donner une vision complète de la complexité algorithmique.

- **Borne inférieure (Ω)** : Représente la complexité minimale. Par exemple, la complexité de l'algorithme de somme de tableau est $\Omega(n)$, car chaque élément doit être parcouru.
- **Borne moyenne (θ)** : Indique la complexité moyenne ou attendue, définissant à la fois une borne inférieure et une borne supérieure qui convergent. Pour le même algorithme de somme de tableau, la complexité est $\theta(n)$.

Voici un exemple de la somme des éléments d'un tableau :

Algorithm 13 Somme de tableau

```
1: function SOMME(tableau)
2:     somme := 0
3:     Pour i de 0 à taille(tableau) - 1 Faire
4:         somme := somme + tableau[i]
5:     Fin Pour
6:     Retourner somme
7: Fin function
```

3.4 Complexité de certains algorithmes

Voici quelques exemples d'algorithmes qui illustrent des complexités courantes, allant du temps constant à l'exponentiel.

3.4.1 Temps constant (O(1))

Les algorithmes à temps constant ont une complexité qui ne dépend pas de la taille de l'entrée. Par exemple, accéder à un élément spécifique d'un tableau par son index a une complexité 'O(1)'. Cela signifie qu'il y a toujours une seule opération, quelle que soit la taille du tableau.

Algorithm 14 Accès direct à un élément du tableau

```
1: function ACCÈS_DIRECT(tableau, index)
2:     Retourner tableau[index]
3: Fin function
```

3.4.2 Temps linéaire (O(n))

Les algorithmes à temps linéaire ont une complexité qui augmente proportionnellement avec la taille de l'entrée. Par exemple, le calcul de la somme des éléments d'un tableau a une complexité 'O(n)', car il faut parcourir tous les éléments.

Algorithm 15 Calcul de la somme des éléments d'un tableau

```
1: function SOMME(tableau)
2:     somme := 0
3:     Pour i de 0 à taille(tableau) - 1 Faire
4:         somme := somme + tableau[i]
5:     Fin Pour
6:     Retourner somme
7: Fin function
```

3.4.3 Temps quadratique (O(n^2))

Les algorithmes à temps quadratique ont une complexité qui augmente avec le carré de la taille de l'entrée. Par exemple, le tri par insertion a une complexité 'O(n^2)', car chaque élément doit être comparé avec tous les éléments précédents.

Algorithm 16 Tri par insertion

```
1: Pour i de 1 à n - 1 Faire
2:     clé := tableau[i]
3:     j := i - 1
4:     Tant que j >= 0 et tableau[j] > clé Faire
5:         tableau[j + 1] = tableau[j]
6:         j = j - 1
7:     Fin Tant que
8:     tableau[j + 1] = clé
9: Fin Pour
```

3.4.4 Temps exponentiel (O(2^n))

Les algorithmes à temps exponentiel ont une complexité qui augmente de manière exponentielle avec la taille de l'entrée. Par exemple, un algorithme récursif pour résoudre le problème de la tour de Hanoï a une complexité 'O(2^n)', car chaque mouvement peut générer de multiples autres mouvements.

Algorithm 17 Tour de Hanoï

```
1: function HANOI(n, source, cible, intermédiaire)
2:     Si n == 1 Alors
3:         Déplacer un disque de la source vers la cible
4:     Sinon
5:         HANOI(n - 1, source, intermédiaire, cible)
6:         Déplacer un disque de la source vers la cible
7:         HANOI(n - 1, intermédiaire, cible, source)
8:     Fin Si
9: Fin function
```

3.5 Exercices

1. Quelle est la complexité temporelle des opérations suivantes ?
   (a). Parcourir un tableau d'entiers de longueur n.
   (b). Multiplier deux matrices de taille $n \times n$.
   (c). Effectuer une recherche binaire sur un tableau trié.
2. Quelle est la complexité temporelle du code suivant ?

```
1  for (int i = 0; i < n; i++) {
2    for (int j = 0; j < n; j++) {
3      printf("%d␣", i * j);
4    }
5  }
```

   Comment pouvez-vous améliorer cet algorithme pour réduire sa complexité ?
3. Analysez le code suivant pour déterminer sa complexité temporelle :

```
1  void operationSurTableau(int arr[], int n) {
2    for (int i = 0; i < n; i++) {
3      arr[i] = arr[i] * 2;
4    }
5
6    for (int i = 0; i < n; i++) {
7      arr[i] = arr[i] + 1;
8    }
```

```
9       }
```

4. Expliquez ce que signifie $O(n \log n)$. Donnez un exemple d'algorithme qui a une complexité de $O(n \log n)$ et expliquez pourquoi. (*Indication :* Tri fusion)
5. Quel est le type de complexité associé aux opérations suivantes ?
   (a). Additionner deux entiers.
   (b). Additionner les éléments d'un tableau de longueur 'n'.
   (c). Effectuer un tri á bulle sur un tableau de longueur 'n'..
   (d). Effectuer un tri par insertion sur un tableau de longueur 'n'.