Kajy University: Informatique

Algorithmes et structures de données

Author: Dimby Rabearivony

Date: 25 novembre 2024

Version: 1.1

"Ny fahalalana no valamparihiko."

Table des matières

Chapitre 2

Algorithmes de tri et de recherche

2.1 Qu’est-ce qu’un algorithme ?

Un *algorithme* est une série d’étapes visant à résoudre un problème ou à accomplir une tâche spécifique. En informatique, les algorithmes permettent de manipuler, trier, rechercher, et transformer des données. Pour qu’un algorithme soit efficace, il doit être :

1. **Correct** : Donner le bon résultat pour tous les cas.
2. **Efficace** : Rapide et utilisant des ressources raisonnables.
3. **Simple** : Compréhensible et facile à mettre en œuvre.
4. **Flexible** : Adaptable à différentes situations.

Un exemple simple d’algorithme qui trouve le maximum de deux nombres :

Algorithm 1 Trouver le maximum de deux nombres

```
Si a > b Alors
    retour a
Sinon
    retour b
Fin Si
```

2.2 Récursion

La récursion est un concept fondamental en informatique où une fonction s’appelle elle-même. C’est souvent utilisé pour résoudre des problèmes qui peuvent être décomposés en sous-problèmes similaires. Les fonctions récursives nécessitent une condition de terminaison pour éviter des appels infinis, ce qui pourrait entraîner des débordements de pile et des erreurs critiques.

Un exemple classique de récursion est le calcul de la factorielle d’un nombre. Voici un algorithme qui montre comment la récursion fonctionne pour la factorielle :

Dans cet exemple, la fonction ‘factorielle‘ utilise la récursion pour multiplier un nombre par la factorielle du nombre précédent, jusqu’à ce qu’elle atteigne le cas de base (n == 0). La condition de terminaison empêche la récursion infinie et garantit que la fonction finit par retourner une valeur.

Les fonctions récursives peuvent entraîner des complexités spatiales élevées en raison de l’utilisation de la pile pour stocker les appels récursifs. Chaque appel de fonction récursive crée

Algorithm 2 Calcul de la factorielle d'un nombre

```
function FACTORIELLE(n)
    Si n == 0 Alors
        Retour 1
    Sinon
        Retour n × FACTORIELLE(n - 1)
    Fin Si
Fin function
```

un nouveau contexte d'exécution dans la pile, ce qui peut augmenter l'utilisation de la mémoire. Pour éviter des débordements de pile, il est essentiel d'avoir des conditions de terminaison robustes et de gérer la profondeur de récursion.

Exemples d'autres problèmes résolus par la récursion

Calculer la suite de Fibonacci : La suite de Fibonacci peut être calculée récursivement en additionnant les deux termes précédents, avec des cas de base pour les premiers termes. Rapellons la définition de la suite de Fibonacci :

$$\begin{cases} U_0 = U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \quad \text{pour } n \geq 0. \end{cases}$$

La récursion est un outil puissant pour résoudre des problèmes de manière élégante, mais elle doit être utilisée avec précaution pour éviter des complications liées à la mémoire et à la performance.

2.3 Algorithmes de tri

Le *tri* consiste à organiser les données dans un ordre particulier. Les algorithmes de tri couramment utilisés comprennent :

1. **Tri à bulles (Bubble Sort)** : Cet algorithme compare des éléments adjacents (côte à côte) et les échange s'ils ne sont pas dans le bon ordre. On répète le processus jusqu'à ce que toute la liste soit triée. C'est le plus simple des algorithmes de tri, mais il n'est pas très efficace pour les grandes listes, car il doit comparer de nombreux éléments.
2. **Tri par insertion (Insertion Sort)** : Cet algorithme place chaque élément à la bonne position dans la liste, un par un. C'est efficace pour les petites listes ou les listes presque triées, car il insère chaque nouvel élément à sa position correcte sans avoir besoin de beaucoup de déplacements.
3. **Tri rapide (QuickSort)** : Cet algorithme utilise la technique de division et conquête. Il choisit un pivot (un élément central), divise la liste en deux parties (éléments plus petits d'un côté, plus grands de l'autre), puis trie chaque partie séparément. Cela en fait un algorithme rapide et efficace pour les grandes listes.

4. **Tri fusion (MergeSort)** : Cet algorithme divise la liste en deux moitiés, trie chaque moitié séparément, puis les fusionne pour créer une liste triée. C'est un tri stable, ce qui signifie qu'il maintient l'ordre relatif des éléments égaux. C'est souvent utilisé pour les grandes listes car il offre de bonnes performances.

Algorithm 3 Tri à bulles

```
Tant que non triée Faire
    Pour i de 0 à n - 2 Faire
        Pour j de 0 à n - i - 2 Faire
            Si tableau[j] > tableau[j + 1] Alors
                échanger tableau[j] et tableau[j + 1]
            Fin Si
        Fin Pour
    Fin Pour
Fin Tant que
```

Algorithm 4 Tri par insertion

```
Pour i de 1 à n - 1 Faire
    clé := tableau[i]
    j := i - 1
    Tant que j >= 0 et tableau[j] > clé Faire
        tableau[j + 1] = tableau[j]
        j = j - 1
    Fin Tant que
    tableau[j + 1] = clé
Fin Pour
```

Algorithm 5 Tri rapide (QuickSort)

```
function QuickSort(tableau, bas, haut)
    Si bas < haut Alors
        pivot := Partition(tableau, bas, haut)
        QuickSort(tableau, bas, pivot - 1)
        QuickSort(tableau, pivot + 1, haut)
    Fin Si
Fin function
```

Les algorithmes de tri sont essentiels pour organiser les données efficacement. Chacun de ces algorithmes a ses avantages et inconvénients, avec des applications spécifiques en fonction des contraintes de temps et d'espace. Le choix de l'algorithme de tri dépendra de nombreux facteurs, y compris la taille de la liste, les caractéristiques des données, et les exigences de performance.

Tâche : Montrer que chacune des algorithmes de tri ci-dessus se termine.

Algorithm 6 Tri fusion (MergeSort)

```
function MergeSort(tableau, gauche, droite)
    Si gauche < droite Alors
        milieu := (gauche + droite) / 2
        MergeSort(tableau, gauche, milieu)
        MergeSort(tableau, milieu + 1, droite)
        Fusion(tableau, gauche, milieu, droite)
    Fin Si
Fin function
```

2.4 Algorithmes de recherche

Les algorithmes de recherche permettent de trouver des éléments spécifiques dans un ensemble de données. Voici quelques-uns des types courants d'algorithmes de recherche, avec des explications détaillées et des exemples :

1. **Recherche linéaire (Linear Search)** : Ce type de recherche parcourt une liste d'éléments un par un, de manière séquentielle, jusqu'à ce qu'il trouve l'élément recherché ou atteigne la fin de la liste. C'est le plus simple des algorithmes de recherche, mais il peut être lent pour les grandes listes.
2. **Recherche binaire (Binary Search)** : La recherche binaire utilise une technique de division pour trouver rapidement un élément dans une liste triée. Elle divise la liste en moitiés, recherche le milieu, puis réduit le domaine de recherche en fonction de la valeur recherchée. Cela la rend plus rapide que la recherche linéaire, mais elle nécessite que la liste soit triée.
3. **Recherche interpolée (Interpolation Search)** : Cette recherche est une version améliorée de la recherche binaire pour les listes avec des valeurs bien réparties. Elle utilise une formule pour estimer la position de la valeur recherchée, ce qui la rend plus précise.
4. **Recherche par saut (Jump Search)** : C'est une recherche linéaire optimisée pour les grandes listes. Elle saute de blocs en blocs pour trouver la plage dans laquelle l'élément pourrait se trouver, puis fait une recherche linéaire à l'intérieur de cette plage.

Algorithm 7 Recherche linéaire

```
Pour i de 0 à n - 1 Faire
    Si tableau[i] == cible Alors
        Retourner i
    Fin Si
Fin Pour
Retourner -1
```

2.5 Exercices

1. Pour chacun des algorithmes de tri cités dans ce chapitre :
   (a). Implémentez l'algorithme en C.

Algorithm 8 Recherche binaire

```
gauche := 0
droite := n - 1
Tant que gauche <= droite Faire
    milieu := gauche + (droite - gauche) / 2
    Si tableau[milieu] == cible Alors
        Retourner milieu
    Sinon Si tableau[milieu] < cible Alors
        gauche := milieu + 1
    Sinon
        droite := milieu - 1
    Fin Si
Fin Tant que
Retourner -1
```

Algorithm 9 Recherche interpolée

```
bas := 0
haut := n - 1
Tant que bas <= haut et cible >= tableau[bas] et cible <= tableau[haut] Faire
    pos := bas + ((cible - tableau[bas]) ÷ (tableau[haut] - tableau[bas])) × (haut - bas)
    Si tableau[pos] == cible Alors
        Retourner pos
    Sinon Si tableau[pos] < cible Alors
        bas := pos + 1
    Sinon
        haut := pos - 1
    Fin Si
Fin Tant que
Retourner -1
```

Algorithm 10 Recherche par saut

```
tailleBloc := Racine n
i := 0
Tant que i < n et tableau[i] < cible Faire
    i := i + tailleBloc
Fin Tant que
Pour j de i - tailleBloc à Min(i, n) Faire
    Si tableau[j] == cible Alors
        Retourner j
    Fin Si
Fin Pour
Retourner -1
```

   (b). Testez votre code avec les cas suivants :
      I. Un tableau vide.
      II. Un tableau contenant un seul élément.
      III. Un tableau déjà trié.
      IV. Le tableau [9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7].

2. Pour chacun des algorithmes de recherche cités dans ce chapitre :
   (a). Implémentez l'algorithme en C.
   (b). Testez votre code en cherchant les éléments 8, 1, et 0 dans le tableau [9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7].
   (c). Lequel de ces algorithmes n'est pas applicable au tableau précédent ? Pourquoi ?

3. Testez votre implémentation de la recherche binaire en cherchant les éléments 0, 2, 7 et 9 dans le tableau [0, 1, 4, 4, 4, 5, 6, 7, 9, 9].