Résolution de tp d'algorithme et programmation

1. C'est une procédure de calcul qui prend en entrée une valeur ou un ensemble de valeur, et qui donne en sortie une ou un ensemble de valeur. C'est donc une séquence d'étapes de calcul qui transforme une entrée en une sortie.

C'est aussi un outil permettant de résoudre un problème de calcul bien spécifié.

C'est aussi une procédure, étape par étape, permettant de résoudre un problème dans un intervalle de temps fini.

2. Un algorithme est efficace lorsqu'il prend moins d'espace et moins de temps d'exécution.
3. L'efficacité d'un algorithme est synonyme de temps d'exécution minimal.
4. Technique de conception d'un algorithme :
   Voici quelques techniques de conception d'algorithmes couramment utilisées :

- ✓ Diviser pour mieux régner: Cette technique consiste à diviser un problème complexe en sous-problèmes plus simples qui peuvent être résolus indépendamment.

- ✓ Greedy algorithms: Les algorithmes gloutons sont basés sur l'idée de faire le choix optimal à chaque étape sans se soucier de ce qui se passera plus tard.

- ✓ Algorithmes de tri: Les algorithmes de tri sont utilisés pour classer les données en ordre croissant ou décroissant. Les algorithmes de tri couramment utilisés comprennent le tri par insertion, le tri par sélection et le tri rapide.

- ✓ Recherche dichotomique: La recherche dichotomique est une technique de recherche efficace utilisée pour trouver un élément spécifique dans une liste ordonnée de données.

- ✓ Algorithmes de parcours de graphe: Les algorithmes de parcours de graphe sont utilisés pour parcourir les noeuds d'un graphe et pour effectuer des opérations telles que la recherche de chemins, la reconnaissance de composantes connexes, etc.

- ✓ Algorithmes de programmation dynamique: Les algorithmes de programmation dynamique utilisent une approche bottom-up pour résoudre des problèmes complexes en les divisant en sous-problèmes plus simples.

- ✓ Algorithmes de force brute: Les algorithmes de force brute sont des algorithmes simples qui cherchent à résoudre un problème en examinant toutes les solutions possibles jusqu'à ce qu'une solution valide soit trouvée.

5. Un algorithme glouton, étapes par étapes, fait le d'un optimum local.

-Diviser pour mieux régner :le problème est divisé en sous problèmes sous problème semblable au problème initial, mais de taille moindre, on résout reconsidérant les sous problèmes et on combine ses solutions pour avoir la solution du problème original .

On peut le résumer en : diviser-Reigner-combiner.

-La méthode probabiliste fait appel aux nombres aléatoires ,un algorithme pareil fais des choix aléatoires au cours de son exécution . Il fait appel à plusieurs générateurs des nombres aléatoire.

-La programmation dynamique : la solution optimale est trouvée en combinant des solutions optimales d'une séries de sous problèmes qui se chevauches.

6. *Pseudo-code ou LDA est une façon de décrire un algorithme sans référence à un langage de programmation particulier
   - ✓ Pseudo-code: Le pseudo-code est une forme de notation qui ressemble à du code informatique, mais qui n'est pas destiné à être exécuté. Il a pour but de décrire un algorithme de manière claire et compréhensible, sans se soucier des détails de la programmation. Le pseudo-code peut être utilisé pour planifier et développer des algorithmes avant de les implémenter en code.

   - ✓ Organigramme: Un organigramme est une représentation visuelle d'un algorithme qui utilise des formes géométriques, telles que des boîtes et des flèches, pour décrire les différentes étapes d'un processus. Les organigrammes peuvent aider à visualiser les relations entre les différentes étapes d'un algorithme et à clarifier les idées.

   une implémentation directe en langage de programmation
7. Ils choisissent selon pour résoudre contexte, objectif de la démarche compétence de l'équipe et les attentes du public cible.

   Les développeurs de logiciels choisissent souvent de représenter les algorithmes sous différentes formes telles que du pseudo-code, des organigrammes ou du code pour les raisons suivantes:

   Compréhension du concept: Le pseudo-code et les organigrammes fournissent une description haut niveau de l'algorithme qui peut être facilement compris par les personnes qui ne sont pas familières avec la programmation. Cela peut aider à clarifier les idées et les concepts avant de les implémenter en code.

   Communication: Les représentations visuelles telles que les organigrammes peuvent aider les équipes de développeurs à communiquer sur les algorithmes et à s'assurer que tout le monde comprend les étapes impliquées.

   Debugging: Le code peut aider à identifier les erreurs et à trouver des bugs plus facilement en fournissant une représentation concrète de l'algorithme.

Implémentation: Le code peut être directement utilisé pour implémenter l'algorithme dans un programme informatique.

En conclusion, les représentations différentes des algorithmes ont chacune leurs avantages et peuvent être utilisées en fonction des besoins spécifiques de l'équipe de développement.

8. Il existe plusieurs critères pour évaluer et identifier le meilleur algorithme pour un problème donné. Ceux-ci incluent :

   Complexité temporelle: la vitesse à laquelle l'algorithme s'exécute, mesurée en fonction du nombre d'opérations requises.

   Complexité mémoire: la quantité de mémoire nécessaire pour exécuter l'algorithme.

   Précision: la précision des résultats produits par l'algorithme.

   Facilité de mise en œuvre: la simplicité avec laquelle l'algorithme peut être compris et implémenté.

   Stabilité: la capacité de l'algorithme à fournir des résultats fiables même avec des entrées ou des données incohérentes ou corrompues.

   Il est important de prendre en compte ces critères lors de la comparaison et de la sélection des algorithmes pour un problème donné. En général, il est souhaitable de choisir un algorithme qui soit à la fois rapide, mémoire efficace, précis, facile à mettre en œuvre et stable.
9. Il existe deux méthodes couramment utilisées pour l'analyse d'un algorithme: la méthode théorique et la méthode expérimentale.

   La méthode théorique consiste à utiliser des expressions mathématiques pour déterminer la complexité de l'algorithme en termes de temps et d'espace. Cette méthode comprend souvent la méthode des opérations primitives, qui mesure le nombre d'opérations élémentaires effectuées par l'algorithme en fonction de la taille de l'entrée.

   La méthode expérimentale consiste à implémenter l'algorithme et à le mesurer en exécutant de nombreux cas de test avec différentes entrées. Les résultats sont ensuite analysés pour déterminer la complexité de l'algorithme en termes de temps et d'espace. Cette méthode fournit une évaluation concrète et précise de la performance de l'algorithme, mais peut être coûteuse en termes de temps et de ressources.
   en faisant l'analyse théorique ou l'analyse expérimental de chaque algorithme et voir celui qui a le temps d'exécution minimal.

10. L'analyse d'un algorithme consiste à analyse de son pseudo-code

11. Il y a plusieurs inconvénients à la méthode expérimentale pour évaluer les algorithmes :

Temps et coût : Effectuer des expériences pour mesurer les performances d'un algorithme peut être coûteux en termes de temps et de ressources. Il faut écrire le code de l'algorithme, le tester avec différentes entrées et mesurer les résultats.

Interférences environnementales : Les facteurs environnementaux tels que la mémoire disponible, la charge du système et les autres applications en cours d'exécution peuvent affecter les résultats de l'expérience.

Variabilité des résultats : Les mesures de performance peuvent varier d'une exécution à l'autre pour les mêmes entrées, ce qui peut rendre difficile l'obtention de résultats précis.

Biais : Les méthodes de mesure utilisées pour évaluer les algorithmes peuvent influencer les résultats, ajoutant un biais aux mesures.

Non-représentativité des entrées : Les entrées utilisées pour tester un algorithme peuvent ne pas être représentatives de l'ensemble des entrées auxquelles l'algorithme sera confronté dans la pratique.

En général, la méthode expérimentale peut être utile pour évaluer les performances d'un algorithme, mais elle doit être utilisée en combinaison avec d'autres méthodes, telles que l'analyse théorique, pour obtenir une évaluation complète de l'algorithme.

12. La méthode des opérations primitives consiste à décomposer un algorithme en opérations élémentaires, appelées opérations primitives, et à mesurer le nombre de ces opérations en fonction de la taille de l'entrée. Les opérations primitives peuvent être des opérations telles que des additions, des multiplications, des comparaisons, des accès à la mémoire, etc.

L'objectif est d'obtenir une expression mathématique qui représente le nombre d'opérations primitives en fonction de la taille de l'entrée, appelée fonction de complexité. Cette fonction permet d'évaluer la complexité du temps et de l'espace de l'algorithme, ce qui aide à évaluer sa performance et sa scalabilité.

La méthode des opérations primitives est souvent utilisée pour évaluer la complexité d'un algorithme de manière théorique, sans la nécessité de réellement implémenter l'algorithme et de le mesurer avec des entrées réelles. Cela permet d'obtenir une évaluation préliminaire de la performance de l'algorithme avant de le tester plus en détail par la suite.12. La complexité d'un algorithme décrit la quantité de ressources, telles que la mémoire ou le temps, nécessaires pour exécuter l'algorithme en fonction de la taille de l'entrée. En termes plus simples, elle mesure la performance de l'algorithme pour différentes tailles d'entrée.

13. La complexité peut être exprimée de différentes manières, mais la plus courante est la notation de la complexité en temps, qui décrit le nombre d'opérations effectuées par l'algorithme en fonction de la taille de l'entrée. Les algorithmes peuvent avoir une complexité en temps constante, logarithmique, linéaire, exponentielle ou polynomiale, selon le nombre d'opérations effectuées en fonction de la taille de l'entrée.

Il est important de comprendre la complexité d'un algorithme pour choisir celui qui convient le mieux pour une tâche donnée, en fonction des contraintes en termes de temps et de mémoire. Les algorithmes plus rapides et plus efficaces en termes de mémoire peuvent être préférables pour les grands ensembles de données, tandis que les algorithmes plus lents peuvent être préférables pour les petits ensembles de données. en gros c'est l'analyse théorique

14. La notation asymptotique est un moyen de mesurer les performances d'un algorithme en termes de complexité en temps d'exécution ou d'utilisation de la mémoire. Au lieu de se concentrer sur les valeurs exactes de temps d'exécution ou de mémoire nécessaires pour un nombre donné d'entrées, la notation asymptotique se concentre sur la façon dont ces valeurs augmentent en fonction de la taille de l'entrée.

    La notation asymptotique la plus couramment utilisée est la notation de la complexité en temps, qui décrit la façon dont le temps d'exécution de l'algorithme augmente en fonction de la taille de l'entrée. Par exemple, un algorithme peut avoir une complexité en temps O(n), ce qui signifie que le temps d'exécution augmente proportionnellement à la taille de l'entrée (n).

    La notation asymptotique est utile car elle permet de comparer les performances d'algorithmes de manière objective, en négligeant les constantes et les facteurs de petits ordres pour se concentrer sur les tendances à long terme. Cela permet également de prédire les performances des algorithmes pour des entrées de taille très grande, ce qui est souvent important pour les applications complexes.)

15. C'est une écriture qui nous permet de décrire les temps d'exécution des algorithmes
16. fonction constante, fonction logarithme fonction linéaire.
17. La taille d'une entrée désigne la quantité de données ou la complexité d'un problème qui doit être résolu par un algorithme. C'est un facteur clé pour évaluer les performances d'un algorithme car plus la taille de l'entrée est grande, plus le temps d'exécution sera long.

    Exemples de taille d'entrée pour des problèmes simples :

    Pour un algorithme de tri, la taille d'entrée peut être définie par le nombre d'éléments dans la liste à trier.

    Pour un algorithme de recherche dans un tableau, la taille d'entrée peut être définie par la taille du tableau.

    Pour un algorithme de calcul de la somme des n premiers nombres, la taille d'entrée peut être définie par la valeur de n.

    Pour un algorithme de résolution d'un labyrinthe, la taille d'entrée peut être définie par la taille du labyrinthe ou le nombre de cases à parcourir.

18. L'analyse de l'algorithme consiste à évaluer ses performances en termes de temps d'exécution et d'utilisation de la mémoire.

Le cas le plus défavorable décrit la situation où l'algorithme prend le plus de temps pour terminer, ou utilise le plus de mémoire. Il est important car il donne une idée de la limite supérieure des performances de l'algorithme, ce qui peut être important pour les applications critiques en termes de temps ou de mémoire.

Le cas le plus favorable décrit la situation où l'algorithme prend le moins de temps pour terminer, ou utilise le moins de mémoire. Il donne une idée de la limite inférieure des performances de l'algorithme, mais n'est pas toujours représentatif de la plupart des situations réelles.

Le cas moyen (probabiliste) décrit la situation où l'algorithme a des performances moyennes en termes de temps d'exécution et d'utilisation de la mémoire, en prenant en compte la probabilité de différents scénarios. Il donne une idée plus réaliste des performances de l'algorithme dans la plupart des situations, mais peut être difficile à calculer pour des algorithmes complexes.

En général, le cas le plus défavorable a une importance particulière car il indique la limite supérieure des performances de l'algorithme et permet de déterminer si les ressources informatiques disponibles sont suffisantes pour résoudre le problème en un temps raisonnable. Cependant, il est également important de prendre en compte les autres cas pour obtenir une image complète des performances de l'algorithme.

Autre

Lors de l'analyse d'un algorithme, on peut distinguer trois cas d'utilisation différents : le cas le plus défavorable, le cas le plus favorable et le cas moyen (probabiliste).

Le cas le plus défavorable correspond à la situation où les hypothèses les plus pessimistes sont considérées quant aux entrées de l'algorithme. Par exemple, le pire cas d'une recherche binnaire est celui où la valeur recherchée ne se trouve pas dans le tableau.

Le cas le plus favorable, d'un autre côté, correspond à la situation où les hypothèses les plus optimistes sont considérées, comme par exemple, le meilleur cas d'une recherche binnaire est celui où la valeur recherchée se trouve dans la première case du tableau.

Le cas moyen (probabiliste) est une analyse qui prend en compte la probabilité statistique des différentes situations. Par exemple, la moyenne arithmétique des temps d'exécution de l'algorithme pour un grand nombre de données aléatoires.

Le cas le plus défavorable a une importance particulière en raison de la nécessité de garantir la performance minimale de l'algorithme dans toutes les situations possibles. C'est pourquoi, les algorithmes sont souvent conçus et optimisés en fonction du pire cas.

19. La récursivité est un concept en programmation où une fonction s'appelle elle-même pour résoudre une tâche complexe en la décomposant en sous-tâches plus simples, jusqu'à ce qu'une condition de base soit atteinte. Cette technique de programmation permet de résoudre des problèmes en utilisant une approche diviser pour régner et peut être utilisée pour implémenter des algorithmes tels que les tri par fusion et la recherche dans un arbre.
20. La récursivité linéaire consiste en une seule appel récursif à la fonction pour résoudre un sous-problème. Cette technique est souvent utilisée pour décomposer un problème en sous-tâches plus petites, chacune étant résolue par une seule récursion.

La récursivité binaire implique deux appels récursifs à la fonction pour résoudre deux sous-problèmes distincts. Cette technique est souvent utilisée pour diviser un problème en deux parties égales et traiter chacune d'entre elles séparément, en utilisant des récursions supplémentaires pour les résoudre.

La récursivité multiple implique plusieurs appels récursifs à la fonction pour résoudre plusieurs sous-problèmes différents. Cette technique est souvent utilisée pour décomposer un problème complexe en plusieurs tâches plus petites, chacune étant résolue par une récursion distincte.

Ces trois types de récursivité peuvent être combinés pour résoudre des problèmes complexes, en fonction de la nature du problème et de la façon dont il peut être décomposé en sous-problèmes plus petits.

21. La récurrence est la notion selon laquelle une fonction peut appeler elle-même pour effectuer une tâche. Cela permet à une fonction de se répéter jusqu'à ce que la tâche soit terminée. Par exemple, une fonction récursive peut être utilisée pour calculer la factorielle d'un nombre. La factorielle est définie comme le produit des nombres entiers allant de 1 à n, où n est le nombre spécifié. En utilisant une fonction récursive, le code peut être écrit comme suit:
    Python

```
def factorial(n):
  if n == 0:
            return 1
   else:
      return n * factorial(n-1)
```

Dans cet exemple, la fonction récursive factorial() est définie avec un seul paramètre, n. Si le nombre passé en argument vaut 0, la fonction retourne 1. Sinon, elle fait appel à elle-même en passant (n-1) en argument. Ces appels se répètent jusqu'à ce que n vaut 0, ce qui est alors retourné jusqu'à ce que la fonction principale retourne le produit des arguments précédents.