Programa de Pós Graduação em Ciência da Computação - UFOP
PCC 104 - Projeto e Análise de Algoritmos
Trabalho Final.
Aluno(a): Mariana de Oliveira Lopes
Matrícula: 141.669.586 – 90

Questões

1. Conjunto 1 - Dividir e Conquistar - **Merge Sort**

   - **Operação básica:** A operação básica do Merge Sort ocorre na parte de mesclagem e ordenação durante a função merge. Essa operação é representada pela comparação entre elementos de B e C, que determina qual elemento será inserido no array A.
   - **Análise de Custo:**
     A recorrência básica para o número de comparações C(n) no Merge Sort é dada por:

     $$C(n) = 2C\left(\frac{n}{2}\right) + C_{\textbf{merge}}(n) \quad \textbf{para} \;\; n > 1, \;\; C(1) = 0. \tag{1}$$

     Onde:
     - $2C\left(\frac{n}{2}\right)$: O array é dividido em duas partes (metades) e o Merge Sort é aplicado recursivamente em cada uma. Como são duas metades, o número de comparações será o dobro do que acontece em uma única metade.
     - $C_{\textbf{merge}}$: Representa o número de comparações realizadas durante a etapa de mesclagem das duas metades ordenadas.
     - $C(1) = 0$: Quando o array tem apenas um elemento ($n = 1$), não há comparações, porque o array já está ordenado.

     Durante a etapa de mesclagem (merge), os elementos dos dois arrays resultantes da divisão são comparados e combinados para formar o array final.

     O comportamento do número de comparações no **pior, melhor** e no **caso médio** é o mesmo, pois $C_{\textbf{merge}}$ não depende da ordem inicial dos elementos no array, ou seja, o processo de mesclagem sempre compara e combina todos os elementos das duas sublistas, independentemente de como estão organizados. Portanto, o número de comparações é dado por:

     $$C_{\text{merge}}(n) = n - 1.$$

     Substituíndo em (3), temos que a recorrência e dada por:

     $$C(n) = 2C\left(\frac{n}{2}\right) + n - 1 \quad \textbf{para} \;\; n > 1, \;\; C(1) = 0. \tag{2}$$

   - **Complexidade**
     Aplicando o **Teorema Mestre**, temos que:
     - $a = 2$, O array é dividido em 2 partes.
     - $b = 2$, Cada divisão reduz o tamanho do array pela metade $(\frac{n}{2})$.
     - $d = 1$, pois $n^1 = n^d \Rightarrow d = 1$.

     Como $2 = 2^1$, temos que $C(n) \in \Theta(n\ log\ n)$.
   - **Classificação: P, NP, NP-Completo**: Como o Merge Sort tem complexidade $\Theta(n\ log\ n)$, ele pode ser resolvido em tempo polinomial. Além disso ele realiza operações definidas, sem dependência de sorteios, probabilidades ou decisões não determinísticas.
     Portanto, ele está na classe **P**.

2. Conjunto 2 - Programação Dinâmica - **Knapsack Problem**

   - **Operação básica:** Comparação entre dois valores numéricos para determinar o máximo entre eles (Função max).
   - **Análise de Custo:** A recorrência que representa o número de operações, definida por $F(n, W)$, é dada por:

$$F(n,W) = \begin{cases} 0 & \text{se } n = 0 \text{ ou } W = 0, \\ F(n-1,W) & \text{se peso do item} > W, \\ \max\left(F(n-1,W), \text{Valor}(n-1) + F(n-1, W - \text{Peso}(n-1))\right) & \text{caso contrário.} \end{cases}$$

Nesse caso, podemos observar que, para cada item n, estamos fazendo duas chamadas recursivas: uma para $F(n-1,W)$ (sem incluir o item) e uma para $F(n-1,W-Peso(n-1))$ (incluindo o item). Esse comportamento leva a uma relação de recorrência do tipo:

$$T(n,W) = 2T(n-1,W) + O(1),$$

em que $O(1)$ representa o tempo constante gasto para comparar e fazer a escolha entre incluir ou não o item.

- **Complexidade**: A complexidade assintótica do algoritmo de programação dinâmica para o problema da mochila é $\Theta(nW)$, onde n é o número de itens e W é a capacidade da mochila. Essa complexidade resulta da necessidade de preencher uma tabela de tamanho $n \times W$, e cada célula da tabela é preenchida em tempo constante $O(1)$.
- **Classificação: P, NP, NP-Completo**: A programação dinâmica resolve o problema da mochila em tempo polinomial em relação ao número de itens n e à capacidade W da mochila. Portanto, é um problema da classe **P**.

3. Conjunto 3 - Algoritmos Gulosos - **Algoritmo de Prim**

- **Operação básica:** Comparação do peso da aresta no loop interno (if not in_tree[v] and weight < min_weight and weight > 0).
- **Análise de Custo:**
  - O loop externo roda até que todos os vértices sejam incluídos na árvore gerador mínima, fazendo $n-1$ iterações.
  - No loop interno que percorre os vértices, em cada iteração o código percorre todos eles, fazendo n iterações.
  - No loop interno que verifica as arestas adjacentes, assumindo o pior caso, ele percorre n arestas, fazendo n iterações.
  - Cada iteração do loop interno faz uma comparação de peso para verificar a menor aresta válida. Logo, a operação de custo é dada por:

  $$C(n) = \sum_{i=1}^{n-1} \sum_{j=1}^{n} \sum_{k=1}^{n} 1.$$

  Simplificando, temos:

  $$C(n) = (n-1) \cdot n \cdot n.$$

  Ou seja:

  $$C(n) = n^2 \cdot (n-1).$$

- **Complexidade:** Portanto, $C(n) \in \Theta(n^3)$.
- **Classificação: P, NP, NP-Completo:** Como o Algoritmo de Prim tem complexidade $\Theta(n^3)$, ele pode ser resolvido em tempo polinomial, independente do tamanho da entrada. Logo, ele está na classe **P**.

4. Conjunto 4 - Backtracking - **Subset-Sum Problem**

- **Operação básica:** Decisão de incluir ou excluir um elemento no subconjunto e a verificação da soma.
- **Análise de Custo:** Sendo $T(n)$ o tempo necessário para resolver o problema para um conjunto de n elementos, a recorrência é dada por:

$$T(n) = 2T(n-1) + 1, \quad T(0) = 1. \tag{3}$$

Onde:

- $2T(n-1)$: É o número de chamadas recursivas feitas, uma para incluir o elemento atual e outra para excluí-lo.
- 1: A verificação se a soma do subconjunto é igual ao alvo é feita em tempo constante para cada chamada recursiva
- $C(0) = 1$: Se $n = 0$, então o tempo necessário para verificar se a soma do subconjunto é igual ao alvo é 1, porque apenas a verificação final precisa ser feita.

  Vamos expandir a relação de recorrência definida como:

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

  Substituímos $T(n-1)$ na fórmula inicial:

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

  Substituindo na equação inicial:

  $$T(n) = 2\big[2T(n-2) + 1)\big] + 1$$

  $$T(n) = 4T(n-2) + 2 + 1$$

  $$T(n) = 4T(n-2) + 3$$

  Continuamos expandindo a recorrência, seguindo o padrão:

  $$T(n) = 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i$$

  Sabemos que $T(n-k)$ atinge o caso base $T(0) = 1$ quando $k = n$. Assim, substituímos $k = n$:

  $$T(n) = 2^n T(0) + \sum_{i=0}^{n-1} 2^i$$

  Substituímos $T(0) = 1$:

  $$T(n) = 2^n \cdot 1 + \sum_{i=0}^{n-1} 2^i$$

  A soma dos primeiros n termos de uma progressão geométrica com razão 2 é dada por:

  $$\sum_{i=0}^{n-1} 2^i = \frac{2^n - 1}{2 - 1} = 2^n - 1$$

  Substituímos isso na expressão de $T(n)$:

  $$T(n) = 2^n + 2^n - 1 = 2^{n+1} - 1$$

- **Complexidade:**
  Assim, a complexidade final é:

  $$T(n) = \Theta(2^n)$$

- **Classificação: P, NP, NP-Completo:** Para o Subset-Sum, se considerarmos a busca exaustiva, ele não pode ser resolvido em tempo polinomial, portanto, ele **não é da classe P**. Por outro lado, neste caso, dado um subconjunto A e um valor d, podemos facilmente verificar se a soma dos elementos S é igual a d em tempo $O(n)$. Portanto, o Subset-Sum pertence à classe **NP**.