

# Atividade Avaliativa 2 - AED2

Cauã Borges Faria (834437)

Maio 2025

## 1 Questão 1

### Código

<https://onlinegdb.com/7MKRJ6trM>

### Complexidade

Seguindo a análise apresentada na apostila, a complexidade do algoritmo CountingSort é determinada pela soma das complexidades de seus passos constituintes. Seja  $n$  o número de elementos na lista de entrada  $L$  e  $m$  o valor máximo em  $L$ .

Os passos principais são:

- Encontrar o maior elemento ( $m$ ) em  $L$ :  $\mathcal{O}(n)$ .
- Inicializar o vetor de contagem  $C$  de tamanho  $m + 1$ :  $\mathcal{O}(m)$ . (A apostila não lista este passo explicitamente na soma final, mas inclui na descrição).
- Inicializar o vetor de saída  $S$  de tamanho  $n$ :  $\mathcal{O}(n)$ . (A apostila não lista este passo explicitamente na soma final).
- Contar ocorrências (primeiro loop FOR): Percorre  $L$  ( $n$  elementos),  $\sum_{i=0}^{n-1} 1 = n$ . Complexidade  $\mathcal{O}(n)$ .
- Calcular soma cumulativa (segundo loop FOR): Percorre  $C$  ( $m$  elementos),  $\sum_{i=1}^m 1 = m$ . Complexidade  $\mathcal{O}(m)$ .
- Construir o vetor de saída  $S$  (terceiro loop FOR): Percorre  $L$  ( $n$  elementos), realizando duas operações por iteração (decremento em  $C$  e atribuição em  $S$ ). A apostila agrupa como  $2 \sum_{i=0}^{n-1} 1 = 2n$ . Complexidade  $\mathcal{O}(n)$ .

A apostila apresenta a função  $T(n)$  (que deveria ser  $T(n, m)$ ) como:

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

Onde o primeiro  $\mathcal{O}(n)$  representa o custo de encontrar o máximo. Avaliando as somas:

$$T(n, m) = \mathcal{O}(n) + n + m + 2n$$

Agrupando os termos em notação Big-O:

$$T(n, m) = \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(m) + \mathcal{O}(n)$$

$$T(n, m) = \mathcal{O}(n + m)$$

Esta é a complexidade geral do CountingSort, conforme derivado na apostila.

Para mostrar que a complexidade é  $\mathcal{O}(n)$  no contexto da Questão 1, observamos que  $n = 30000$  e o valor máximo  $m$  também é 30000. Portanto,  $m = n$ . Substituindo  $m = \mathcal{O}(n)$  na complexidade geral:

$$T(n) = \mathcal{O}(n + \mathcal{O}(n))$$

$$T(n) = \mathcal{O}(n + n)$$

$$T(n) = \mathcal{O}(2n)$$

$$T(n) = \mathcal{O}(n)$$

Concluimos, que quando  $m = \mathcal{O}(n)$ , a complexidade do CountingSort é linear,  $\mathcal{O}(n)$ .

## 2 Questão 2

### Código

<https://onlinegdb.com/LAO254xVU>

### Complexidade do Countingsort\_R

É possível notar que temos 5 estruturas de repetição (FOR) sequenciais para Countingsort\_R:

1. Encontrar o maior dígito  $m$  (na faixa 0-9):  $\sum_{i=1}^{n-1} 1 = n - 1 = \mathcal{O}(n)$ .
2. Inicializar  $C$ :  $\mathcal{O}(m + 1)$ . Como  $m = 9$  para dígitos,  $\mathcal{O}(10) = \mathcal{O}(1)$ .
3. Contar ocorrências dos dígitos:  $\sum_{i=0}^{n-1} 1 = n = \mathcal{O}(n)$ .
4. Calcular soma cumulativa:  $\sum_{i=1}^m 1 = m = \mathcal{O}(m)$ . Como  $m = 9$ ,  $\mathcal{O}(9) = \mathcal{O}(1)$ .
5. Montar o vetor de saída  $S$ :  $2 \sum_{i=0}^{n-1} 1 = 2n = \mathcal{O}(n)$ . (A apostila agrupa as duas operações dentro do loop).
6. Copiar  $S$  de volta para  $L$ :  $\sum_{i=0}^{n-1} 1 = n = \mathcal{O}(n)$ .

A apostila apresenta a função  $T(n)$  referente ao Countingsort\_R como:

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

Avaliando as somas e usando a notação Big-O:

$$T(n) = \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(m) + \mathcal{O}(n) + \mathcal{O}(n)$$

Como  $m$  é o maior dígito (0 a 9),  $m = 9$ , que é uma constante  $\mathcal{O}(1)$ .

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

A apostila conclui: "Porém, neste caso, o valor de  $m$  é no máximo 9, o que faz com que  $n$  seja dominante, levando à complexidade  $\mathcal{O}(n)$ ."

### Complexidade do RadixSort

Prosseguindo para a função RadixSort, seja  $k$  o número de dígitos do maior elemento da lista  $L$ . A função  $T(n)$  é dada por:

- Encontrar o maior elemento  $m$  em  $L$ :  $\mathcal{O}(n)$ .
- Loop 'while' que executa  $k$  vezes (uma para cada dígito).
- Dentro do loop: chamada a Countingsort\_R ( $\mathcal{O}(n)$ ) e uma multiplicação ( $\mathcal{O}(1)$ ).

A apostila escreve a função  $T(n)$  como:

$$T(n) = \mathcal{O}(n) + \sum_{i=1}^k [\mathcal{O}(n) + 1]$$

(Onde  $\mathcal{O}(n)$  é o custo de Countingsort\_R e 1 é o custo da multiplicação  $d = d * 10$ ).

$$T(n, k) = \mathcal{O}(n) + k \times [\mathcal{O}(n) + \mathcal{O}(1)]$$

$$T(n, k) = \mathcal{O}(n) + k \times \mathcal{O}(n)$$

$$T(n, k) = \mathcal{O}(n + k \cdot n)$$

$$T(n, k) = \mathcal{O}(k \cdot n)$$

A apostila conclui: "onde  $k$  denota o número de dígitos do maior inteiro em  $L$ . Como em geral  $k$  é uma constante muito menor que  $n$ , temos que a complexidade resultante é  $\mathcal{O}(n)$ ."

No contexto da Questão 2,  $n = 30000$  e  $k = 6$ . Como  $k$  é uma constante pequena em relação a  $n$ , a complexidade efetiva é  $\mathcal{O}(6n) = \mathcal{O}(n)$ .

### 3 Questão 3

#### Código

<https://onlinegdb.com/xyWaM8uhU>

#### Complexidade

Seguindo a análise apresentada na apostila, utilizamos 3 primitivas básicas no algoritmo Bucketsort:

- **insert(p, x)**: insere  $x$  na lista apontada pela referência  $p$ . Possui complexidade  $\mathcal{O}(1)$ .
- **sort(p)**: ordena a lista apontada pela referência  $p$ . Se utilizarmos um algoritmo baseado em comparações, a complexidade seria entre  $\mathcal{O}(n \log n)$  e  $\mathcal{O}(n^2)$ .
- **concatenate(B, n)**: retorna a lista obtida pela concatenação das listas  $B[0], B[1], \dots, B[n-1]$ . Possui complexidade  $\mathcal{O}(kn)$ , onde  $k$  é uma constante que representa o tamanho médio dos baldes, o que resulta em  $\mathcal{O}(n)$ .

O ponto crítico é a análise da primitiva **sort()**, sendo que devemos realizar uma análise probabilística. Seja  $X_i$  o número de elementos na lista  $B[i]$ . Seja ainda a variável binária (indicadora):

$$X_{ij} = \begin{cases} 1, & \text{se o elemento } j \text{ de } L \text{ foi para a lista } B[i] \\ 0, & \text{se o elemento } j \text{ de } L \text{ não foi para a lista } B[i] \end{cases}$$

Note que  $X_i = \sum_j X_{ij}$ .

Iremos denotar por  $Y_i$  o número de comparações necessárias para ordenar a lista  $B[i]$ . Observe que  $Y_i \leq X_i^2$ , pois no pior caso a ordenação será  $\mathcal{O}(n^2)$  (assumindo, por exemplo, Insertion Sort). Logo, como desejamos analisar o caso médio, podemos escrever:

$$E[Y_i] \leq E[X_i^2] = E \left[ \left( \sum_j X_{ij} \right)^2 \right]$$

Mas podemos desenvolver o valor esperado como (a apostila expande o quadrado e depois aplica a expectativa):

$$E \left[ \left( \sum_j X_{ij} \right)^2 \right] = E \left[ \left( \sum_j X_{ij} \right) \left( \sum_k X_{ik} \right) \right] = E \left[ \sum_j \sum_k X_{ij} X_{ik} \right]$$

Separando os casos  $k = j$  e  $k \neq j$ :

$$E \left[ \sum_j \sum_k X_{ij} X_{ik} \right] = E \left[ \sum_j X_{ij}^2 + \sum_j \sum_{k \neq j} X_{ij} X_{ik} \right]$$

Como  $X_{ij}$  é uma variável indicadora,  $X_{ij}^2 = X_{ij}$ .

$$E \left[ \sum_j X_{ij} + \sum_j \sum_{k \neq j} X_{ij} X_{ik} \right]$$

Pela linearidade da expectativa:

$$E[Y_i] \leq \sum_j E[X_{ij}] + \sum_j \sum_{k \neq j} E[X_{ij} X_{ik}]$$

Como  $X_{ij}$  é uma variável aleatória binária,  $E[X_{ij}] = P(X_{ij} = 1)$ . Assumindo que a probabilidade de um elemento cair em um dos  $n$  baldes é uniforme,  $P(X_{ij} = 1) = 1/n$ . Portanto,  $E[X_{ij}] = 1/n$ .

Para calcular o valor esperado do produto, note que para  $j \neq k$ , as duas variáveis aleatórias  $X_{ij}$  e  $X_{ik}$  são independentes (a decisão de onde o elemento  $j$  cai não afeta onde o elemento  $k$  cai). Assim:

$$E[X_{ij} X_{ik}] = E[X_{ij}] E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$$

Substituindo na desigualdade para  $E[Y_i]$ :

$$E[Y_i] \leq \sum_{j=1}^n \frac{1}{n} + \sum_{j=1}^n \sum_{k \neq j} \frac{1}{n^2}$$

O primeiro somatório é  $n \times (1/n) = 1$ . O segundo somatório tem  $n$  termos para  $j$ . Para cada  $j$ , o somatório interno sobre  $k \neq j$  tem  $n - 1$  termos. Assim:

$$E[Y_i] \leq 1 + \sum_{j=1}^n (n-1) \frac{1}{n^2}$$

$$E[Y_i] \leq 1 + n(n-1) \frac{1}{n^2} = 1 + \frac{n-1}{n} = 1 + 1 - \frac{1}{n} = 2 - \frac{1}{n}$$

Esse é o número médio de comparações esperado para ordenar o balde  $B[i]$ . Como temos  $n$  baldes, a complexidade total esperada para ordenar todos os baldes é:

$$E[Y] = E \left[ \sum_{i=0}^{n-1} Y_i \right] = \sum_{i=0}^{n-1} E[Y_i]$$

$$E[Y] \leq \sum_{i=0}^{n-1} \left( 2 - \frac{1}{n} \right) = n \left( 2 - \frac{1}{n} \right) = 2n - 1$$

Isso indica que o custo esperado total para ordenar todos os baldes é  $\mathcal{O}(n)$ .

A complexidade total do Bucketsort é a soma das complexidades dos passos:

$$\begin{aligned} T(n) &= \underbrace{\mathcal{O}(n)}_{\text{Criar baldes}} + \underbrace{\mathcal{O}(n)}_{\text{Distribuir}} + \underbrace{E[Y]}_{\text{Ordenar baldes}} + \underbrace{\mathcal{O}(n)}_{\text{Concatenar}} \\ T(n) &= \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n) \end{aligned}$$

A apostila conclui: "Por essa razão, a complexidade total do Buckesort é:  $n * \mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n)$  o que finalmente resulta em  $\mathcal{O}(n)$ ."

Assim, o BucketSort tem complexidade de tempo linear  $\mathcal{O}(n)$  no caso médio, sob a hipótese de distribuição uniforme dos dados de entrada.

## 4 Questão 4

Esta questão deve ser resolvida manualmente, mostrando o passo a passo da construção e modificação de uma Árvore AVL.

### Parte A: Plantando a Floresta

Inserção da sequência: 50, 30, 20, 60, 70, 10, 25, 40, 45, 35, 80

### 1. Inserir 50

50(0)

Árvore balanceada.

### 2. Inserir 30

50(1)  
/  
30(0)

Árvore balanceada.

### 3. Inserir 20

50(2)  
/  
30(1)  
/  
20(0)

Nó 50 desbalanceado (fator 2). Nó 30 desbalanceado (fator 1). O nó desbalanceado mais próximo da inserção com fator  $> 1$  é 50. O nó inserido (20) está na subárvore esquerda do filho esquerdo (30) de 50. Caso Esquerda-Esquerda. **Rotação Simples à Direita (RSD) em 50:**

30(0)  
/ \  
20(0) 50(0)

Árvore balanceada.

### 4. Inserir 60

30(-1)  
/ \  
20(0) 50(-1)  
      \  
      60(0)

Árvore balanceada.

### 5. Inserir 70

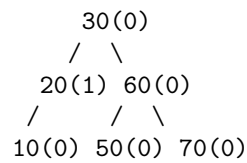
30(-2)  
/ \  
20(0) 50(-1)  
      \  
      60(-1)  
      \  
      70(0)

Nó 50 ficou com fator -2 após a inserção de 70 em 60. A inserção (70) foi na subárvore direita do filho direito (60) de 50. Caso Direita-Direita. **RSE em 50.**

30(-1)  
/ \  
20(0) 60(0)  
      /  
      50(0) 70(0)

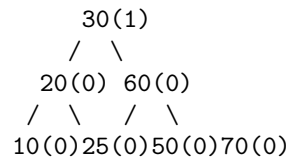
Árvore balanceada.

### 6. Inserir 10



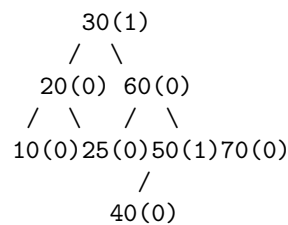
Árvore balanceada.

### 7. Inserir 25



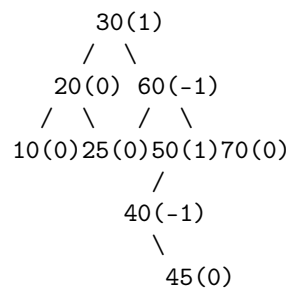
Árvore balanceada.

### 8. Inserir 40



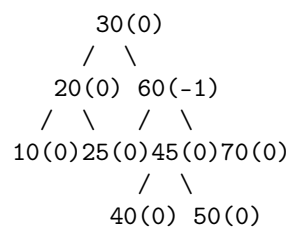
Nó 50 com fator 1. Árvore balanceada.

### 9. Inserir 45



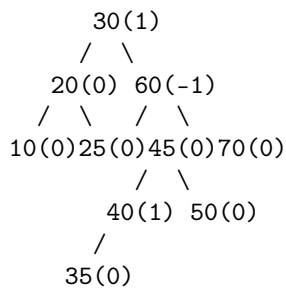
Nó 50 desbalanceado (fator 2). A inserção (45) foi na subárvore esquerda (40) → direita (45) de 50. Caso Esquerda-Direita. **Rotação Dupla Esquerda-Direita (RED) em 50:**

- Passo 1: RSE em 40.
- Passo 2: RSD em 50.



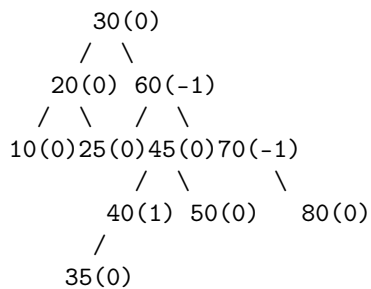
Árvore balanceada.

#### 10. Inserir 35



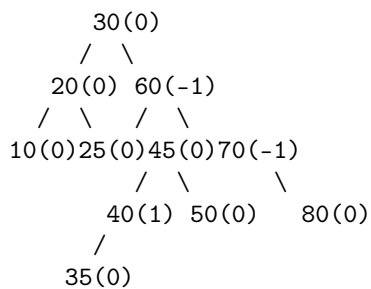
Árvore balanceada.

#### 11. Inserir 80



Árvore balanceada.

#### Árvore AVL final após todas as inserções (Parte A)



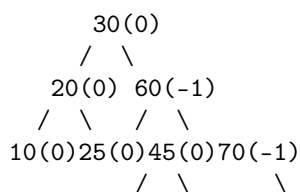
#### Sequência de Rotações Necessárias (Parte A)

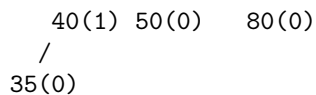
1. Após inserir 20: Rotação Simples à Direita (RSD) em 50.
2. Após inserir 70: Rotação Simples à Esquerda (RSE) em 50.
3. Após inserir 45: Rotação Dupla Esquerda-Direita (RED) em 50 (RSE em 40, seguida de RSD em 50).

#### Parte B: A Tempestade do Caos

Remoção do nó 45 da árvore final da Parte A.

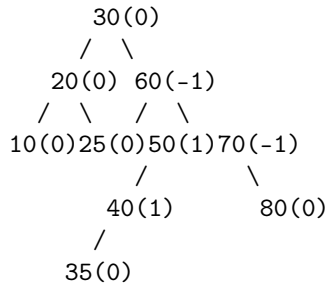
#### Árvore antes da remoção



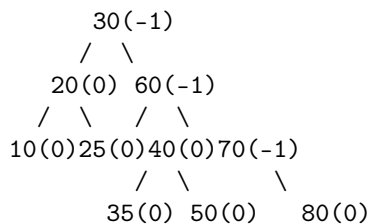


O nó 45 tem dois filhos (40 e 50). Encontramos o sucessor in-order de 45, que é o mínimo da subárvore direita, ou seja, 50. Substituímos a chave 45 pela chave 50 e removemos o nó original 50 (que é uma folha).

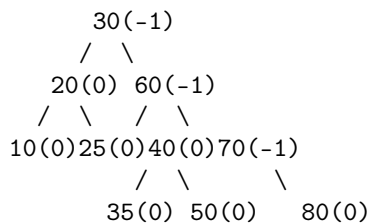
#### Árvore após substituir 45 por 50 e remover o nó 50 original



Verificamos o balanceamento subindo a partir do pai do nó removido (que era 45, agora 50). O pai é 60. Recalculando fatores de balanço: Nó 50 (antigo 45):  $h_{esq}(40) = 2$ ,  $h_{dir} = 0 \Rightarrow fb = 2$ . Desbalanceado! O nó desbalanceado é 50 (fb=2). O filho esquerdo é 40 (fb=1). Caso Esquerda-Esquerda. **Rotação Simples à Direita (RSD) em 50:**



#### Árvore AVL final após a remoção e rebalanceamento (Parte B)



#### Rotações Necessárias na Remoção (Parte B)

1. Após remover 45 (substituído por 50, nó 50 original removido): Rotação Simples à Direita (RSD) em 50 (o nó que continha 45).

### Parte C: O Caminho da Energia Mágica

#### a) Percurso in-order da árvore final (Parte B)

Percorrendo a árvore final em ordem (Esquerda, Raiz, Direita): **10, 20, 25, 30, 35, 40, 50, 60, 70, 80**

#### b) Altura da árvore final e eficiência AVL

- A altura da árvore é o número de arestas no caminho mais longo da raiz até uma folha.
- Raiz: 30



- Caminhos mais longos:  $30 \rightarrow 60 \rightarrow 40 \rightarrow 35$  (3 arestas),  $30 \rightarrow 60 \rightarrow 40 \rightarrow 50$  (3 arestas),  $30 \rightarrow 60 \rightarrow 70 \rightarrow 80$  (3 arestas).
- **Altura da árvore final = 3.**

#### **Explicação da eficiência AVL:**

A propriedade AVL garante que, para qualquer nó, a diferença de altura entre suas subárvores esquerda e direita é no máximo 1 (fator de balanço em  $\{-1, 0, 1\}$ ). Isso impede que a árvore se degenera em uma lista encadeada (pior caso para busca, inserção e remoção em árvores binárias de busca,  $\mathcal{O}(n)$ ). Ao manter a árvore balanceada, a altura da árvore AVL é sempre  $\mathcal{O}(\log n)$ , onde  $n$  é o número de nós. Como as operações básicas (busca, inserção, remoção) dependem da altura da árvore, a propriedade AVL garante que essas operações tenham uma complexidade de tempo logarítmica ( $\mathcal{O}(\log n)$ ) no pior caso, o que é muito mais eficiente do que a complexidade linear ( $\mathcal{O}(n)$ ) de uma árvore desbalanceada.

## **5 Questão 5**

### **Código**

<https://onlinegdb.com/nu5wYWJa2>