Exercício 1 (c)

Para encontrar uma fórmula fechada (não recursiva) para a recorrência $T(n) := 7T(n/3) + Cn^2$, consideramos as condições $T(1) = 1$ e $n \in S = \{3^k : k \in \mathbb{N}\}$.

Por definição, a recorrência é dada por:

$$T(n) = 7T(n/3) + Cn^2 \tag{1}$$
$$T(1) = 1 \tag{2}$$
$$n \in S = \{3^k : k \in \mathbb{N}\} \tag{3}$$

Como o domínio é composto por potências de 3, fazemos a substituição $n = 3^k$, o que implica $k = \log_3 n$.

Primeiro, vamos expandir a recorrência para encontrar um padrão:

$$\begin{aligned}
T(n) &= 7T(n/3) + Cn^2 \\
&= 7\left(7T(n/9) + C(n/3)^2\right) + Cn^2 \\
&= 7^2T(n/3^2) + \frac{7}{9}Cn^2 + Cn^2 \\
&= 7^2T(n/3^2) + Cn^2\left(1 + \frac{7}{9}\right) \\
&= 7^2\left(7T(n/27) + C(n/9)^2\right) + Cn^2\left(1 + \frac{7}{9}\right) \\
&= 7^3T(n/3^3) + \frac{7^2}{9^2}Cn^2 + Cn^2\left(1 + \frac{7}{9}\right) \\
&= 7^3T(n/3^3) + Cn^2\left(1 + \frac{7}{9} + \left(\frac{7}{9}\right)^2\right)
\end{aligned}$$

Após i iterações, a fórmula geral é:

$$T(n) = 7^iT(n/3^i) + Cn^2\sum_{j=0}^{i-1}\left(\frac{7}{9}\right)^j \tag{4}$$

A recursão termina quando atingimos o caso base $T(1)$, o que ocorre quando $n/3^i = 1$, ou seja, quando $n = 3^i$. Como $n = 3^k$, temos que a parada ocorre quando $i = k$.

Agora, substituindo $i = k$ e $T(1) = 1$ na equação geral:

$$T(n) = 7^kT(1) + Cn^2\sum_{j=0}^{k-1}\left(\frac{7}{9}\right)^j = 7^k + Cn^2\sum_{j=0}^{k-1}\left(\frac{7}{9}\right)^j \tag{5}$$

Para expressar os termos em função de n, consideramos primeiro o termo 7^k. Como $k = \log_3 n$, temos $7^k = 7^{\log_3 n}$. Usando a propriedade de mudança de base de logaritmos ($a^{\log_b c} = c^{\log_b a}$), obtemos:

$$7^k = n^{\log_3 7} \tag{6}$$

Para o segundo termo, temos uma série geométrica finita com razão $r = 7/9$:

$$\sum_{j=0}^{k-1} \left(\frac{7}{9}\right)^j = \frac{1-(7/9)^k}{1-7/9} = \frac{1-(7/9)^k}{2/9} = \frac{9}{2}\left(1-\left(\frac{7}{9}\right)^k\right) \tag{7}$$

Substituindo as partes simplificadas de volta na equação de $T(n)$:

$$T(n) = n^{\log_3 7} + Cn^2 \cdot \frac{9}{2}\left(1 - \frac{7^k}{9^k}\right) \tag{8}$$

Agora, substituímos $7^k = n^{\log_3 7}$ e $9^k = (3^2)^k = (3^k)^2 = n^2$:

$$\begin{aligned} T(n) &= n^{\log_3 7} + \frac{9C}{2}n^2\left(1 - \frac{n^{\log_3 7}}{n^2}\right) \\ &= n^{\log_3 7} + \frac{9C}{2}n^2 - \frac{9C}{2}n^2 \cdot \frac{n^{\log_3 7}}{n^2} \\ &= n^{\log_3 7} + \frac{9C}{2}n^2 - \frac{9C}{2}n^{\log_3 7} \end{aligned}$$

Portanto, agrupando os termos semelhantes, obtemos a fórmula fechada:

$$\boxed{T(n) = \frac{9C}{2}n^2 + \left(1 - \frac{9C}{2}\right)n^{\log_3 7}} \tag{9}$$

Exercício 5

Para resolver o problema de intercalar k listas ordenadas contendo um total de n elementos em uma única lista ordenada, utilizamos o algoritmo conhecido como *k-way merge*. Este algoritmo emprega uma estrutura de dados *min-heap* para atingir a complexidade de tempo pedida $O(n \log k)$.

Estratégia do Algoritmo

A estratégia fundamental consiste em manter um heap com o menor elemento ainda não processado de cada uma das k listas. A cada iteração, extraímos o menor elemento do heap e o inserimos na lista resultado, substituindo-o pelo próximo elemento da lista correspondente.

Algoritmo

Algoritmo 1: IntercalarKListas

Data: k listas ordenadas $L_1, L_2, \ldots, L_k$ com total de n elementos
Result: Lista ordenada *resultado* contendo todos os n elementos

```
resultado ← lista vazia;
heap ← MinHeap vazio // armazena (valor, índice_da_lista)
for i ← 1 to k do
    if listas[i] não está vazia then
        heap.inserir((listas[i][0], i));
    end
end
while heap não está vazio do
    (valor_min, i) ← heap.extrair_min();
    resultado.adicionar(valor_min);
    listas[i].remover_primeiro();
    if listas[i] não está vazia then
        heap.inserir((listas[i][0], i));
    end
end
return resultado;
```

Funcionamento do Algoritmo

O algoritmo utiliza um min-heap para determinar eficientemente o próximo menor elemento entre todas as k-listas:

1. **Inicialização:** O heap é populado com o primeiro elemento de cada uma das k listas. Por definição do min-heap, o menor elemento global estará na raiz.
2. **Processo Iterativo:** Em cada uma das n iterações, extraímos o elemento mínimo do heap, que é garantidamente o menor elemento disponível entre todas as listas.
3. **Manutenção:** O elemento extraído é substituído pelo próximo elemento da mesma lista, garantindo que cada lista mantenha um representante no heap.
4. **Término:** O processo continua até que todas as listas sejam esvaziadas e o heap se torne vazio.

Prova de Corretude

Para provar a corretude do algoritmo, utilizamos uma invariante de laço:

Invariante: No início de cada iteração do laço principal, a lista `resultado` contém os m menores elementos do conjunto total em ordem crescente, e o min-heap contém o $(m+1)$-ésimo menor elemento na raiz.

- **Inicialização:** Antes da primeira iteração, `resultado` está vazia ($m = 0$) e o heap contém o menor elemento de cada lista. O menor elemento global está necessariamente na raiz do heap.
- **Manutenção:** Assumindo que a invariante é válida no início de uma iteração, o algoritmo extrai o $(m+1)$-ésimo menor elemento da raiz e o adiciona a `resultado`. O próximo elemento da lista correspondente (maior ou igual ao removido) é inserido no heap, mantendo a propriedade de que a raiz contém o $(m+2)$-ésimo menor elemento.
- **Término:** Após n extrações, temos $m = n$ e `resultado` contém todos os elementos em ordem crescente.

Portanto, o algoritmo está correto.

Análise de Complexidade

Para analisar a complexidade de tempo, consideramos n como o número total de elementos e k como o número de listas.

1. **Inicialização do heap:** São realizadas k inserções no heap. Como cada inserção custa $O(\log k)$, o custo total é:

$$T_{\text{init}} = k \times O(\log k) = O(k \log k) \tag{10}$$

2. **Laço principal:** Executamos n iterações, uma para cada elemento. Em cada iteração:

   - Extrair o mínimo: $O(\log k)$
   - Inserir próximo elemento: $O(\log k)$

   O custo total do laço é:

$$T_{\text{loop}} = n \times O(\log k) = O(n \log k) \tag{11}$$

3. **Complexidade total:**

$$T(n, k) = T_{\text{init}} + T_{\text{loop}} = O(k \log k) + O(n \log k) \tag{12}$$

   Como geralmente $n \geq k$, temos que $O(n \log k)$ domina.

Casos especiais:

- Para $k = n$ (cada lista tem 1 elemento): $O(n \log n)$ (equivalente ao heapsort)
- Para $k = 2$ (duas listas): $O(n \log 2) = O(n)$ (intercalação padrão)

Portanto, a complexidade de tempo final do algoritmo é $\boxed{O(n \log k)}$.