Lista de Exercícios Avaliativa 3

MC458 - 2s2020 - Tiago de Paula Alves - 187679

1.

```
MEDIANA(X, Y, n)
if n == 1
    return X[1]
elseif n == 2
    return máx(X[1], Y[1])
else
    mx = ⌊(n+1)/2⌋
    my = ⌈(n+1)/2⌉
    if X[mx] < Y[my]
        return MEDIANA(X[mx..n], Y[1..my], my)
    else
        return MEDIANA(X[1..mx], Y[my..n], mx)
```

Corretude. Suponha um inteiro positivo n tal que, para todo $1 \le k < n$ e quaisquer vetores X e Y ordenados e com k elementos distintos, podemos encontrar um elemento mediano de $X \cup Y$. Suponha ainda dois vetores ordenados X e Y com n elementos distintos cada, em que $X \cap Y = \varnothing$. Seja $U = X \cup Y$.

Caso 1: $n = 1$. Então, $c = X_1$ é elemento mediano de U.

Caso 2: $n = 2$. Se $X_1 < Y_1$, como $Y_1 < Y_2$, então $c = Y_1$ deve ser um dos elementos medianos. Por outro lado, se $X_1 > Y_1$, então $Y_1 < X_1 < X_2$, ou seja, $c = X_1$ deve ser mediano. Como $X_1 = Y_1$ é impossível, temos em ambos os casos um elemento mediano c de U.

Caso 3: $n \ge 3$. Considere as posições intermediárias $m_1 = \lfloor (n+1)/2 \rfloor$ e $m_2 = \lceil (n+1)/2 \rceil$. Considere também os subvetores $X^{(1)} = [X_1, \ldots, X_{m_1-1}]$, $X^{(2)} = [X_{m_1+1}, \ldots, X_n]$, $Y^{(1)} = [Y_1, \ldots, Y_{m_2-1}]$ e $Y^{(2)} = [Y_{m_2+1}, \ldots, Y_n]$, de forma que $\#(X^{(1)}) = \#(Y^{(2)}) = m_1 - 1$ e $\#(X^{(2)}) = \#(Y^{(1)}) = m_2 - 1$.

Caso 3a: $X_{m_1} < Y_{m_2}$. Note que $\{X_{m_1}\} \cup X^{(2)}$ e $Y^{(1)} \cup \{Y_{m_2}\}$ estão ordenados e têm $1 < m_2 < n$ elementos distintos. Logo, pela hipótese indutiva, temos um elemento mediano c' de $U' = \{X_{m_1}\} \cup X^{(2)} \cup Y^{(1)} \cup \{Y_{m_2}\}$.

Como $X_{m_1} \le c'$ e X está ordenado, então $X_i^{(1)} < X_{m_1} \le c'$ para todo $1 \le i < m_1$. Logo, $\#(U_<) = \#(U'_<) + \#(X^{(1)}) = \#(U'_<) + m_1 - 1$. Da mesma forma, $c' \le Y_{m_2}$, então $\#(U_>) = \#(U'_>) + \#(Y^{(2)}) = \#(U'_>) + m_1 - 1$.

Portanto, temos que $|\#(U_<) - \#(U_>)| = |\#(U'_<) + m_1 - 1 - \#(U'_>) - m_1 + 1| = |\#(U'_<) - \#(U'_>)|$. Então, $c = c'$ também é um elemento mediano de U.

Caso 3b: $X_{m_1} > Y_{m_2}$. Então teremos os vetores $X^{(1)} \cup \{X_{m_1}\}$ e $\{Y_{m_2}\} \cup Y^{(2)}$ ordenados e com $1 < m_1 < n$ elementos cada. Pela hipótese indutiva, temos então um elemento mediano c' de $U' = X^{(1)} \cup \{X_{m_1}\} \cup \{Y_{m_2}\} \cup Y^{(2)}$.

Agora teremos que $Y_i^{(1)} < Y_{m_2} \leq c' \leq X_{m_1} < X_j^{(2)}$, para todos $1 \leq i, j < m_2$. Então, $\#(U_<) = \#(U'_<) + m_2 - 1$ e $\#(U_>) = \#(U'_>) + m_2 - 1$, ou seja, $|\#(U_<) - \#(U_>)| = |\#(U'_<) - \#(U'_>)|$.

Portanto, $c = c'$ também é mediano de U.

Por fim, podemos, em todos os casos possíveis, encontrar um elemento mediano c de U. ■

Complexidade. Para este algoritmo temos que o tempo de execução é:

$$T(1) = T(2) = \Theta(1)$$

$$T(n) = \begin{cases} T\left(\left\lceil \frac{n+1}{2} \right\rceil\right) + \Theta(1) & \text{se } X[\lfloor (n+1)/2 \rfloor] < Y[\lceil (n+1)/2 \rceil] \\ T\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + \Theta(1) & \text{caso contrário} \end{cases}$$

Em uma análise de pior caso, a relação de recorrência seria $T(n) = T\left(\left\lceil \frac{n+1}{2} \right\rceil\right) + \Theta(1)$. Logo, pelo Teorema Master, o algoritmo tem complexidade $T(n) \in \Theta\left(n^{\log_2 1} \lg n\right) = \Theta(\lg n) \subset o(n)$. ■

---

2. Suponha um algoritmo que dado dois vetores ordenados X e Y com n elementos dististintos, sendo $X \cap Y = \varnothing$, retorna a posição de algum elemento mediano de $X \cup Y$ na forma de um par (V,i), onde V indica qual vetor, X ou Y, contém o elemento e i é sua posição no vetor.

Note que se X e Y têm distribuições de valores semelhantes, então os elementos medianos estarão no entorno da posições $n/2$ de X ou Y. Por outro lado, se $X_n < Y_1$, como eles estão ordenados, as medianas serão esses dois elementos. O mesmo vale quando $Y_n < X_1$. Portanto, dadas distribuições específicas, quaisquer posições de X ou Y podem ser soluções do problema. Ou seja, existem $2n$ soluções possíveis das quais o algoritmo deve decidir entre uma das 2 corretas.

Para um modelo baseado em comparações, podemos imaginar o algoritmo como uma árvore de decisão binária, sendo as folhas as soluções do problema. Para que o algoritmo seja ótimo no pior caso, a árvore deve estar balanceada e com o menor número de folhas possíveis, mantendo 2^h nós à uma altura h da raiz, ou seja, após h comparações. Assim, teremos no útltimo nível, com $2n$ folhas, que $2n \leq 2^h$, ou seja, $h \geq \lg n + 1$.

Além disso, como existem duas soluções corretas, um algoritmo ótimo conseguiria decidir uma das soluções sem comparação entre elas, reduzindo a altura ótima para $h^* \geq \lg n$, mas sem alterar a complexidade.

Por fim, temos então que para todo algoritmo A baseado em comparações, $T_A(n) \geq ch \geq c \lg n$ para $c > 0$ e n suficientemente grande, ou seja, $T_A(n) \in \Omega(\lg n)$. Portanto, a cota inferior do problema neste modelo é $\Omega(\lg n)$.

3.

a) Como o algoritmo HeapSort tem complexidade $\Theta(n \lg n)$, a solução de Xitoró terá tempo de execução total de

$$T(n) = \sum_{i=1}^{k} \Theta(n_i \lg n_i) + \Theta(1) = \Theta\left(\sum_{i=1}^{k} n_i \lg n_i\right)$$

Logo, dependendo do comportamento de n_i, é possível que $T(n) \in \omega(n)$, ou seja, $T(n) \notin O(n)$. Portanto, essa solução não terá necessariamente o comportamento assintótico desejado. Os requisitos de espaço, no entanto, são alcaçados.

b) Como o CountingSort tem uso de espaço $O(n)$, o espaço total seria $O(n^2)$, no entanto, o compartilhamento de buffer de contagem permite espaço em ordem $O(n)$. Apesar disso, a complexidade da solução de Chorãozinho terá, para este problema:

$$\begin{aligned}
T(n) &= \sum_{i=1}^{k} O(n_i + n) + O(1) \\
&= O\left(\sum_{i=1}^{k} n_i\right) + n \sum_{i=1}^{k} O(1) \\
&= O(n) + nO(k) = nO(n) \\
&= O(n^2)
\end{aligned}$$

Portanto, também não terá o comportamento assintótico desejado.

c) A solução desse problema, apresentada abaixo, se baseia na ideia de que um número inteiro pode ser visto como um vetor de dígitos em uma base b qualquer. Assim, o algoritmo funciona como o CountingSort, contando as repetições de um número, sendo que cada vetor V_i tem sua contagem em um dígito $i-1$ do contador. Para que o algoritmo seja linear, vamos assumir um modelo computacional similar ao RAM, mas com um operação extra: PRÓXIMODIGITO(n, b), que descobre o próximo dígito não-nulo de n na base b. Além disso, a operação de potência também é assumida constante.

Isso não é muito longe de arquiteturas atuais, como x86 e ARM, com as operações *find first set* ou *count leading zeroes*. Uma implementação real, em uma máquina dessas, deveria aproximar b para a potência de 2 mais próxima, onde PRÓXIMODIGITO e a potência b^k podem ser realizadas em tempo constante. No entanto, uma implementação real teria muitos limites quanto aos tamanhos n e k do problema se quisesse manter o comportamento assintótico do algoritmo.

COUNTSORTVARIOS(V, k, n)

```
1   // Contador de repetições de cada elemento.
2   Seja C[1..n] = [0..0] um novo vetor.
3
4   // Maior quantidade possível em um dígito.
5   b = máx (V[i].tamanho for i = 1 to k) + 1
6
7   // Contagem das repetições.
8   for i = 1 to k
9       for j = 1 to V[i].tamanho
10          // Cada vetor V_i é contado no dígito i − 1 base b.
11          C[V[i][j]] = C[V[i][j]] + 1 · b^(i−1)
12      // Parte já ordenada do vetor V_i.
13      V[i].tamanho = 0
14
15  // Reposição dos elementos nos vetores.
16  for i = 1 to n
17      while C[i] > 0
18          // Próximo vetor com contagens em C_i.
19          d = PRÓXIMODIGITO(C[i], b)
20          // Quantidade de repetições de i em V_{d+1}.
21          cnt = C[i]/b^d mod b
22          C[i] = C[i] − cnt · b^d
23          for j = 1 to cnt
24              V[d+1].tamanho = V[d+1].tamanho + 1
25              V[d+1][V[d+1].tamanho] = i
```

Corretude. Nas linhas 9 a 11 é feita a contagem das repetições dos elementos do vetor V_i. A contagem acontece apenas no i-ésimo dígito do número no vetor C, que é alcançado com o fator b^{i-1}. A invariante aqui é que C contém nos dígitos i todas as repetições de $V_{i,1}$ até $V_{i,j}$

Para que seja correta a invariante, é necessário que as repetições não extrapolem o tamanho de um dígito, por isso a base foi escolhida como $b = 1 + \max_{1 \leq i \leq k}(n_i)$, na linha 5. Além disso, a linha 13 recomeça o marcador de tamanho do vetor para 0, que agora deverá marcar a quantidade de posições já ordenadas no vetor. Com isso, temos que as linhas 8 a 13 garantem que C tem todas as contagens do vetores $V_1, \dots, V_i$, nos dígitos correspondentes.

A partir disso, o laço nas linhas 16 até 25 é responsável por repopular os vetores de forma ordenada, mantendo a invariante de que todos os números de 1 a i já foram inseridos nos vetores $V_1, \dots, V_k$ nas ordens e quantidades corretas. Para alcaçar isso, o laço interno 17-25 percorre o contador de repetições C_i, recuperando o próximo dígito não nulo d (19 e 21), que corresponde à quantidade de repetições de i em V_{d+1}. Esse valor então é removido de C_i, no dígito correspondente, para manter a invariante de que C_i contém apenas as repetições de i ainda não inseridas em $V_1, \dots, V_k$.

Por fim, mantidas as invariantes, o laço 16-25 encerra com todos os números de 1 até n inseridos em $V_1, \ldots, V_k$ de forma ordenada. Portanto, $V_1, \ldots, V_k$ estão ordenados. ■

Complexidade. Podemos ver que a linha 2 executa em $O(n)$ e a linha 5 executa em $O(k) = O(n)$. Os laços em 8-13 executam em:

$$T_{\{L8-L13\}}(n) = \sum_{i=1}^{k} n_i = O(n)$$

Por fim, temos ainda os laços de 16 até 25. Assumindo PRÓXIMODIGITO em tempo constante, como discutido anteriormente, podemos afirmar que o laço tem mesma complexidade que o número de todas as repetições de 1 a n em $V_1, \ldots, V_k$. Pela invariante de 8-13, isso equivale à quantidade de elementos em todos os vetores. Portanto:

$$T_{\{L16-L25\}}(n) = \sum_{i=1}^{n} \sum_{(d \,\in\, \text{dígitos não-nulos de } C_i)} \text{dígito } d \text{ de } C_i = \sum_{j=1}^{k} n_j = O(n)$$

Logo, a complexidade do algoritmo como um todo é:

$$T(n) = O(n) + O(n) + O(n) + O(n) = O(n)$$

Para os requisitos de espaço, assumindo que C é capaz de armazenar qualquer inteiro em $[0, b^k)$, o algoritmo usa $O(n)$ de armazenamento na sua execução.

Para uma implementação em uma máquina binária, o espaço para o número deveria crescer com $O(k \lg b)$, mas para pequenos valores de k e b, isso pode ser desconsiderado.

■