Universidade Estadual de Maringá
Departamento de Informática – Ciência da Computação
6889 – Projeto e Análise de Algoritmos / Prof. Daniel Kikuti

Aluno(a): ______________________________

Primeira avaliação (Valor: 10,0)

1. [Valor: 2,0] Considere o algoritmo a seguir. Assuma que cada linha está associada a um custo fixo c_i $(i = 1, \ldots, 4)$ e que $T(n)$ é o tempo de execução do algoritmo em função de n.

```
Faz-Algo(n)
1 for i ← 1 to n do
2     for j ← 1 to n step j ← j * 2 do
3         for k ← 1024 to 1 step k ← k/2 do
4             print "Easy"
```

(a) [Valor: 1,0] É correto afirmar que $T(n) \in O(n^2)$? Justifique.

(b) [Valor: 1,0] É correto afirmar que $T(n) \in \Omega(n^2)$? Justifique.

Solução:

Nível da questão: médio.

O que avalia:

- Análise de complexidade de algoritmo iterativo.
- Notação assintótica.

Primeiramente precisamos determinar a complexidade do algoritmo.

Linha	Número de vezes	Custo
1	$n+1$	$c_1 * (n+1)$
2	$n * (\lfloor \lg n \rfloor + 2)$	$c_2 * n * (\lfloor \lg n \rfloor + 2)$
3	$12 * n \lfloor \lg n \rfloor$	$c_3 * 12 * n \lfloor \lg n \rfloor$
4	11	$c_4 * 11 * n \lfloor \lg n \rfloor$

Somando-se o custo de cada linha temos:

$$\begin{aligned} T(n) &= c_1 n + c_1 + c_2 n \lfloor \lg n \rfloor + 2c_2 n + 12 c_3 n \lfloor \lg n \rfloor + 11 c_4 n \lfloor \lg n \rfloor \\ &= (c_2 + 12c_3 + 11c_4) n \lfloor \lg n \rfloor + (c_1 + 2c_2) n + c_1 \\ &= \Theta(n \lg n). \end{aligned}$$

Para responder as questões usaremos as definições das notações O e Ω:

- $f(n) = O(g(n)) \iff 0 \leq \lim_{n \to \infty} \dfrac{f(n)}{g(n)} < \infty.$
- $f(n) = \Omega(g(n)) \iff 0 < \lim_{n \to \infty} \dfrac{f(n)}{g(n)} \leq \infty.$

Calculando o limite temos:

$$\begin{aligned} L = \lim_{n \to \infty} \frac{n \lg n}{n^2} &\overset{\mathrm{H}}{=} \lim_{n \to \infty} \frac{\frac{(\ln n)+1}{\ln 2}}{2n} = \lim_{n \to \infty} \frac{\ln n}{2n \ln 2} - \frac{1}{2n \ln 2} \\ &= \lim_{n \to \infty} \frac{\ln n}{2n \ln 2} - \lim_{n \to \infty} \frac{1}{2n \ln 2} \overset{\mathrm{H}}{=} \lim_{n \to \infty} \frac{1}{2n \ln 2} - 0 = 0 - 0 = 0. \end{aligned}$$

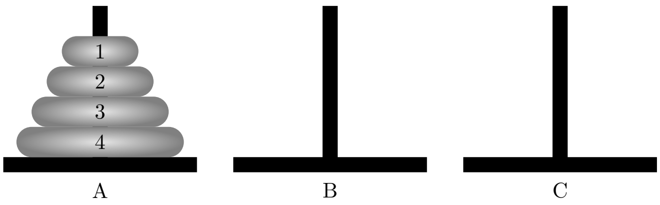
Portanto, concluimos que a questão a) é verdadeira e a questão b) é falsa.

2. [Valor: 2,0] Torres de Hanói amaldiçoadas. No problema original as regras eram simples: apenas um disco poderia ser movido por vez e nunca um disco maior deveria ficar por cima de um disco menor. Segundo a lenda, quando

todos os discos fossem transferidos de um pino para outro, o mundo desapareceria. Na versão amaldiçoada, além das restrições do problema original, acrescenta-se a restrição que o movimento só pode ser feito de um pino para outro adjacente (de A para B, de B para A ou C, de C para B – não é possível mover diretamente de A para C e vice-versa). O objetivo é mover todas os n discos de A para C. Diz a lenda que quem resolver esta questão receberá 2,0 pontos.

(a) [Valor: 1,0] Seja $T(n)$ o número de movimentos necessários para mover n discos do pino origem para o pino destino. Formule recursivamente o problema e resolva a recorrência por Árvore de Recursão.

(b) [Valor: 1,0] Use o Método da Substituição para provar que a fórmula fechada para a recorrência está correta.



Solução:

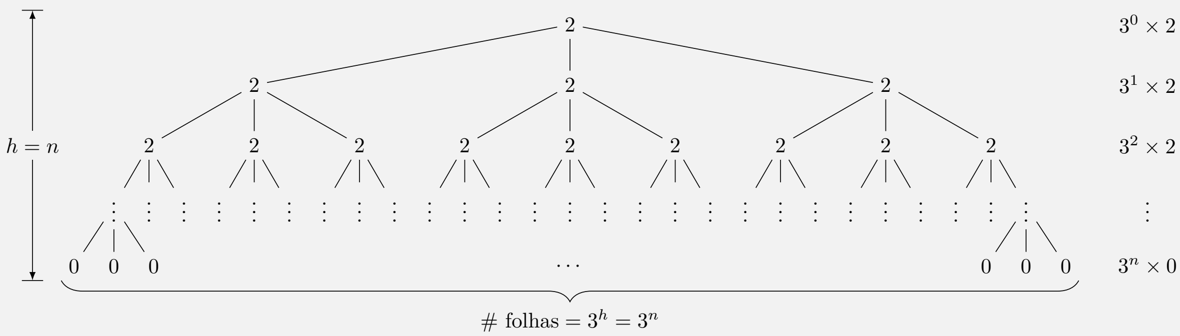
Nível da questão: difícil.

O que avalia:

- Formulação recursiva da solução do problema.
- Definição de equação de recorrência.
- Resolução de recorrência.
- Aplicação dos métodos: Árvore de Recursão e Substituição.

Observe que, para mover nenhum disco de A para C o custo seria zero, para mover um disco de A para C precisamos de dois movimentos (A →B, B →C). Para mover dois discos de A para C precisamos de oito movimentos (A →B, B →C, A →B, C →B, B →A, B →C, A →B, B →A). No caso geral, para mover n discos de A para C, precisamos primeiramente mover os $n-1$ discos de A para C, movemos o n-ésimo disco de A para B, movemos os $n-1$ discos de C para A, movemos o n-ésimo disco de B para C, finalmente, movemos os $n-1$ discos de A para C. Assim, teremos a seguinte recorrência:

$$T(n) = 3T(n-1) + 2$$



Como nas folhas não fazemos nenhum movimento, faremos a soma desconsiderando o último nível.

$$\begin{aligned} T(n) &= 2 \times \sum_{i=0}^{n-1} 3^i \\ &= 2 \times \frac{(3^{n-1+1} - 1)}{3 - 1} \\ &= 3^n - 1. \end{aligned}$$

Seja T(n) o número de movimentos necessários. Vamos demonstrar pelo método da substituição que $3^n - 1$ é o número de movimentos necessários para uma entrada $n \geq 0$.

- **Base:** para $n = 0$, $3^0 - 1 = 0$. Ou seja, para $n = 0$ nenhum movimento é necessário e verifica-se a base.
- **Hipótese:** $T(n) = 3^n - 1$.
- **Passo:** Mostrar que $T(n+1) = 3^{n+1} - 1$.

$$\begin{aligned} T(n+1) &= 3T(n+1-1) + 2 \\ &= 3(3^n - 1) + 2 \\ &= 3^{n+1} - 3 + 2 \\ &= 3^{n+1} - 1. \end{aligned}$$

3. [Valor: 2,0] Dadas as recorrências dos algoritmos A, B e C, determine a complexidade de cada um deles (usando a notação $\Theta(\cdot)$) e ordene-os do mais eficiente para o menos eficiente considerando a classe de complexidade a qual pertencem (use $X \prec Y$ para indicar que o Algoritmo X é mais eficiente que o Algoritmo Y).

   - $T_A(n) = 3T_A(n/9) + \sqrt{n}$
   - $T_B(n) = T_B(n/2) + n$
   - $T_C(n) = 3T_B(n/2) + n$

Solução:

Nível da questão: fácil.

O que avalia:

- Resolução de recorrência (uso do Método Mestre).
- Comparação assintótica entre funções.

Usaremos o Método Mestre para definir o consumo de tempo de cada algoritmo.

- **Algoritmo A:** $T_A(n) = 3T_A(n/9) + \sqrt{n}$. $a = 3$, $b = 9$, $f(n) = \sqrt{n}$. Comparando $f(n)$ com $n^{\log_b a} = n^{\log_9 3} = n^{(1/2)}$, temos o Caso 2, pois $f(n) = \Theta(n^{(1/2)})$. Portanto, a complexidade de tempo do Algoritmo A é $\Theta(\sqrt{n} \lg n)$.
- **Algoritmo B:** $T_B(n) = T_B(n/2) + n$. $a = 1$, $b = 2$, $f(n) = n$. Comparando $f(n)$ com $n^{\log_b a} = n^{\log_2 1} = n^0$, temos o Caso 3, pois $f(n) = \Omega(n^{0+\varepsilon})$ para $\varepsilon = 1$. A condição de regularidade se verifica, pois:

$$af(n/b) \leq cf(n) \iff 1(n/2)^1 \leq cn \iff n/2 \leq cn \iff 1/2 \leq c$$

  e a condição se verifica para $1/2 \leq c < 1$. A complexidade de tempo do Algoritmo B é $\Theta(n)$.
- **Algoritmo C:** $T_C(n) = 3T_C(n/2) + n$. $a = 3$, $b = 2$, $f(n) = n$. Comparando $f(n)$ com $n^{\log_b a} = n^{\log_2 3} \approx n^{1.58}$, temos o Caso 1, pois $f(n) = O(n^{1.58-\varepsilon})$ para $\varepsilon = 0.58$. Portanto, a complexidade de tempo do Algoritmo C é $\Theta(n^{1.58})$.

Assim, os algoritmos ordenados por classe de complexidade são: $A \prec B \prec C$.

4. [Valor: 2,0] O algoritmo a seguir recebe como entrada dois vetores A e B, representando dois conjuntos contendo n elementos cada, e devolve a interseção destes dois conjuntos ($C = A \cap B$).

```
Interseção(A, B, n)
C ← ∅
for i ← 1 to n do
    j ← 1
    while A[i] ≠ B[j] e j ≤ n do
        j ← j + 1
    if j ≤ n then
        C ← C ∪ A[i]
return C
```

   (a) [Valor: 1,0] Apresente um invariante para o laço externo e use-o para mostrar que o algoritmo está correto.

   (b) [Valor: 1,0] Se A e B estão ordenados, informe como obter a interseção em tempo linear no pior caso.

Solução:

Nível da questão: fácil.

O que avalia:

- Correção de algoritmos iterativos.
- Adaptação de algoritmo na resolução de problema semelhante (procedimento merge).
- Análise de complexidade.

a) **Invariante para o laço externo:** $C = A[1 \dots i{-}1] \cap B$.

- **Inicialização:** $i = 1$ e $A[1 \dots 1{-}1] = A[1 \dots 0] = \emptyset$. Na linha 1, $C \leftarrow \emptyset = \emptyset \cap B$.
- **Manutenção:** No começo de uma iteração i qualquer, assuma que o invariante é válido, ou seja, $C = A[1 \dots i{-}1] \cap B$. Precisamos mostrar que ao executar o corpo do laço, o invariante permanecerá válido para a próxima iteração. O corpo do laço **for** basicamente procura o elemento $A[i]$ em B. Caso $A[i] \not\subset B$ $(j > n)$, $C = A[1 \dots i{-}1] \cap B = A[1 \dots i] \cap B$, não alteramos o conjunto C e o invariante se mantém para a próxima iteração. Caso $A[i] \subset B$, na linha 7 acrescentamos o elemento A[i] a C, portanto, $C = (A[1 \dots i{-}1] \cup \{A[i]\}) \cap B = A[1 \dots i] \cap B$, e o invariante se mantém para a próxima iteração.
- **Término:** Quando $i = n + 1$, temos o invariante $C = A[1 \dots n{+}1{-}1] \cap B = A[1 \dots n] \cap B = A \cap B$, ou seja, o algoritmo devolve corretamente a interseçao entre dois conjuntos.

b) Interseção2(A, B, n)

1 $C \leftarrow \emptyset$, $i \leftarrow 1$, $j \leftarrow 1$
2 **while** $i \leq n$ *e* $j \leq n$ **do**
3   **if** $A[i] < B[j]$ **then**
4         $i \leftarrow i + 1$
5   **else if** $A[i] > B[j]$ **then**
6         $j \leftarrow j + 1$
7   **else**
8         $C \leftarrow C \cup A[i]$
9         $i \leftarrow i + 1$, $j \leftarrow j + 1$
10 **return** C

O algoritmo executa em tempo linear porque dentro do laço **while** ou i é incrementado ou j é incrementado ou ambos são incrementados. Quando i ou j alcançam o valor $n + 1$, o condicional do laço se torna falso. Assim, seriam executadas no máximo $2n$ iterações com custo constante no corpo do laço. Portanto, o algoritmo executa em tempo $\Theta(n)$.

5. [Valor: 1,0] Dados um número x e um conjunto S contendo n números inteiros distintos, determine se existem dois elementos em S cuja soma é exatamente x. Exemplo: para $S = \{9, 5, 12, 1, 6\}$ e $x = 11$ a resposta é sim (6 e 5); para o mesmo conjunto S e $x = 16$ a resposta seria não. Descreva um algoritmo que resolve este problema com complexidade de tempo $\Theta(n \lg n)$ no pior caso. Explique a ideia de funcionamento de seu algoritmo e detalhe a análise da complexidade.

Solução:

Nível da questão: médio.

O que avalia:

- Se o aluno está resolvendo exercícios sugeridos e busca tirar dúvidas (problema da lista de exercícios).
- Uso de algoritmos vistos em aula (ordenação e busca binária).
- Análise de complexidade do algoritmo desenvolvido.

Somam-x(S, x)

1 Ordene o conjunto S usando o algoritmo Mergesort
2 **for** $i \leftarrow 1$ **to** $|S|$ **do**
3     $pos \leftarrow$ Busca-Binária$(S, 1, n, x - S[i])$
4   **if** $pos \neq -1$ *e* $i \neq pos$ **then**
5   **return** true
6 **return** false

Precisamos que os elementos em S estejam ordenados para usarmos busca binária. O algoritmo considera então cada elemento i de $|S|$ e busca um elemento y tal que $S[i]+y=x$, ou seja, procura por um elemento $y=x-S[i]$. Se este elemento estiver em S (e for diferente do próprio $S[i]$, pois o enunciado informa que todos os elementos são distintos), então $\textsc{Busca-Binária}(S,1,n,x-S[i])$ irá devolver a posição em que o elemento se encontra. Neste caso, encontrou-se o par que soma x e o algoritmo devolve $true$. Caso contrário, $\textsc{Busca-Binária}(S,1,n,x-S[i])$ irá devolver -1, indicando que o elemento i não faz parte da solução (e o algoritmo deverá testar outro elemento). Se nenhum i faz parte da solução, então o algoritmo devolve $false$.

Em relação ao consumo de tempo, a linha 1 custa $\Theta(n \lg n)$, a linha 2 custa $O(n)$, na linha 3 são feitas no pior caso n chamadas de busca binária, custando portanto $O(n \lg n)$, as linhas 4 e 5 custam $O(n)$, a linha 6 custa tempo $\Theta(1)$. Portanto, o consumo de tempo total do algoritmo no pior caso será $\Theta(n \lg n)$.

6. [Valor: 1,0] Suponha que iremos gerar n números aleatórios no intervalo $[0 \ldots n^3]$. Informe como podemos ordená-los em tempo linear no pior caso. Explique detalhadamente por que seu algoritmo seria linear.

Solução:

Nível da questão: difícil.

O que avalia:

- Se o aluno está resolvendo exercícios sugeridos e busca tirar dúvidas (problema da lista de exercícios).
- Entendimento de métodos de ordenação lineares.
- Análise de complexidade.
- Desenvolvimento de algoritmos.

Uma idéia aqui seria usar o algoritmo Radix-sort. Sabe-se que a complexidade do Radix-sort é $\Theta(d(n+k))$, onde d é a quantidade de dígitos de um número (assumindo que todos os números possuem a mesma quantidade de dígitos) e $n+k$ representa o custo para aplicar o Counting-sort num conjunto de n números.

Para responder a questão, precisamos considerar o formato dos números da entrada. Note que precisamos saber a quantidade de dígitos que um número possui, pois ela depende do valor de n. Por exemplo, se $n = 10$, poderíamos ter um número com 4 dígitos; se $n = 1000$, poderíamos ter um número com 10 dígitos. De forma geral, um número n na base 10 possui $d = \lfloor \log_{10} n \rfloor + 1$ dígitos. Assim, considerando base 10 o Radix-sort teria complexidade $\Theta(n \log_{10} n)$, que não é linear.

Para conseguir fazer ordenação em tempo linear, a ideia seria usar o Radix sort, mas considerando os números na base n, ou seja, inicialmente precisamos mudar a base de todos os números. Por que isto? Porque $d = \lfloor \log_n n^3 \rfloor + 1 = 3 + 1$, ou seja, a complexidade do Radix sort seria $O(4(n+n)) = O(8n) = O(n)$.

Para que nosso raciocínio sobre ordenação em tempo linear esteja correto e completo, precisamos mostrar que é possível converter todos os números para base n em tempo linear. Um número $A = (a_k a_{k-1} \ldots a_0)_B$ na base B com $0 \leq a_i < B$ é representado como:

$$\sum_{i=0}^{k} a_k B^i = a_k B^k + \ldots + a_0 B^0.$$

Para converter de decimal para a base n, devemos começar dividindo o número por n. O resto desta divisão corresponde ao a_0. Dividimos o resultado da primeira divisão novamente por n (o resto desta segunda divisão corresponde ao a_1), e assim sucessivamente até que a base seja maior do que o resultado (resultado da divisão seja igual a zero).

Como notado anteriormente, o número de divisões necessárias para converter um número no intervalo $[0 \ldots n^3]$ para a base n será igual a 4. Portanto, converter n números pode ser feito em tempo $O(n)$.

Boa Prova!!!