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: 1,0] É verdade que $\frac{n}{2} \lg \frac{n}{2} \in \Omega(n \lg n)$? Justifique.

Solução:

Sim, é verdadeira.

(a) Usando a definição da notação assintótica, precisamos encontrar constantes $c > 0$ e $n_0 > 0$ tal que, $\frac{n}{2} \lg \frac{n}{2} \geq cn \lg n$ para todo $n \geq n_0$.

$$\begin{aligned}
\frac{n}{2} \lg \frac{n}{2} &\geq cn \lg n && \text{(reescrevendo divisão em logaritmo)} \\
\frac{n}{2} (\lg n - \lg 2) &\geq cn \lg n && \text{(distributiva)} \\
\frac{n}{2} \lg n - \frac{n}{2} &\geq cn \lg n && \text{(dividindo ambos os lados por } n \lg n\text{)} \\
\frac{1}{2} - \frac{1}{2 \lg n} &\geq c
\end{aligned}$$

Analisando o desenvolvimento, observamos que quanto maior o valor de n, mais próximo de $1/2$ é o valor de c, ou seja, c deve estar no intervalo $0 < c \leq 1/2$. Consideramos então um valor inicial para $n_0 = 4$ (por quê?). Assim, Para $c = 1/4$ temos que $\frac{n}{2} \lg \frac{n}{2} \in \Omega(n \lg n)$.

(b) Usando limite, devemos mostrar que: $L = \lim_{n\to\infty} \frac{\frac{n}{2} \lg \frac{n}{2}}{n \lg n}$, $0 < L \leq \infty$.

$$\begin{aligned}
L &= \lim_{n\to\infty} \frac{\frac{n}{2} \lg \frac{n}{2}}{n \lg n} && \text{(multiplicando por 2/2 e reescrevendo divisão de logaritmo)} \\
&= \lim_{n\to\infty} \frac{n(\lg n - \lg 2)}{2n \lg n} && \text{(distributiva)} \\
&= \lim_{n\to\infty} \frac{n \lg n - n}{2n \lg n} && \text{(simplificando)} \\
&= \lim_{n\to\infty} \frac{1}{2} - \frac{1}{2 \lg n} && \text{(reescrevendo limite)} \\
&= \lim_{n\to\infty} \frac{1}{2} - \lim_{n\to\infty} \frac{1}{2 \lg n} = \frac{1}{2} - 0 = \frac{1}{2}.
\end{aligned}$$

2. [Valor: 1,0] Seja $p(n) = \sum_{i=0}^{k} a_i n^i$ (polinômio em n de grau k), onde k é um inteiro não-negativo, a_i é uma constante e $a_k > 0$, mostre que $p(n) = \Theta(n^k)$.

Solução:

Usando limite, devemos mostrar que: $L = \lim_{n\to\infty} \frac{\sum_{i=0}^{k} a_i n^i}{n^k}$, $0 < L < \infty$.

$$\begin{aligned}
L &= \lim_{n\to\infty} \frac{\sum_{i=0}^{k} a_i n^i}{n^k} && \text{(reescrevendo somatório)} \\
&= \lim_{n\to\infty} \frac{a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n^1 + a_0}{n^k} && \text{(reescrevendo limite)} \\
&= \lim_{n\to\infty} \frac{a_k n^k}{n^k} + \lim_{n\to\infty} \frac{a_{k-1} n^{k-1}}{n^k} + \ldots + \lim_{n\to\infty} \frac{a_1 n^1}{n^k} + \lim_{n\to\infty} \frac{a_0}{n^k} \\
&= \lim_{n\to\infty} a_k + \lim_{n\to\infty} \frac{a_{k-1}}{n^1} + \ldots + \lim_{n\to\infty} \frac{a_1}{n^{k-1}} + \lim_{n\to\infty} \frac{a_0}{n^k} \\
&= a_k + 0 + \ldots + 0 + 0 \\
&= a_k.
\end{aligned}$$

3. [Valor: 2,0] Dados três algoritmos (A, B e C) e suas respectivas recorrências, analise a complexidade de cada um deles e classifique-os em ordem crescente de complexidade (use a notação $X \prec Y$ para indicar que X é mais eficiente que Y).

- $T_A(n) = 2T_A(n/2) + 1$
- $T_B(n) = 2T_B(n/4) + \sqrt{n}$
- $T_C(n) = 2T_C(n/2) + n \lg n$

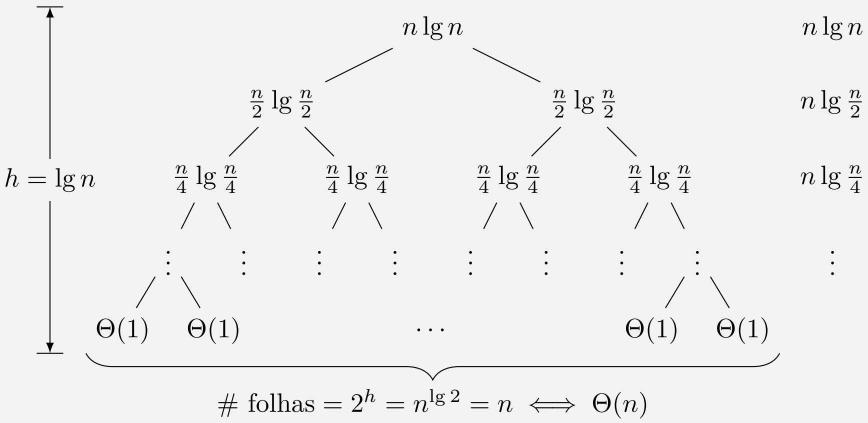
Solução:

Precisamos definir as recorrências e o consumo de tempo de cada algoritmo.

- **Algoritmo** A**:** $T_A(n) = 2T_A(n/2) + 1$. Usando o Teorema Mestre, temos: $a = 2$, $b = 2$, $f(n) = 1$. Comparando $f(n)$ com $n^{\log_b a} = n^{\log_2 2} = n^1$ temos o caso 1 do Método Mestre, pois $f(n) = O(n^1)$ para $\epsilon = 1$. Portanto, a complexidade de tempo do Algoritmo A é $\Theta(n)$.

- **Algoritmo** B**:** $T_B(n) = 2T_B(n/4) + \sqrt{n}$. Usando o Teorema Mestre, temos: $a = 2$, $b = 4$, $f(n) = \sqrt{n}$. Comparando $f(n)$ com $n^{\log_b a} = n^{\log_4 2} = n^{0.5}$ temos o caso 2 do Método Mestre, pois $f(n) = \Theta(n^{0.5})$. Portanto, a complexidade de tempo do Algoritmo B é $\Theta(\sqrt{n} \lg n)$.

- **Algoritmo** C**:** $T_C(n) = 2T_C(n/2) + n \lg n$.



$$
\begin{aligned}
T(n) &= \sum_{i=0}^{h-1} n \lg \frac{n}{2^i} + \Theta(n) \\
&= n \sum_{i=0}^{h-1} (\lg n - \lg 2^i) + \Theta(n) \\
&= n \sum_{i=0}^{h-1} \lg n - n \sum_{i=0}^{h-1} i + \Theta(n) \\
&= n \lg n \lg n - n \frac{\lg n(\lg n - 1)}{2} + \Theta(n) \\
&= O(n \lg n \lg n).
\end{aligned}
$$

A complexidade do Algoritmo C é $O(n \lg n \lg n)$.

Alternativamente, poderíamos demonstrar usando o método da substituição. Vamos assumir por hipótese que $T(k) \leq ck \lg k \lg k$ para $k < n$, em particular para $k = n/2$. Assim:

$$
\begin{aligned}
T(n) &\leq 2c\frac{n}{2} \lg \frac{n}{2} \lg \frac{n}{2} + n \lg n \\
&= cn(\lg n - 1)(\lg n - 1) + n \lg n \\
&= cn(\lg n \lg n - \lg n - \lg n + 1) + n \lg n \\
&= cn \lg n \lg n - 2cn \lg n + cn + n \lg n
\end{aligned}
$$

Note que a parte destacada em vermelho (residual) é menor ou igual a zero para $c = 1$ e $n_0 = 2$.

Ordem de complexidade: $B \prec A \prec C$

4. [Valor: 1,5] Considere o algoritmo Heapsort descrito a seguir. Mostre que o algoritmo está correto usando o seguinte invariante de laço: "No começo de cada iteração do laço **for** das linhas 2–5, o subvetor $A[1..i]$ contém os i menores elementos de $A[1..n]$, e o subvetor $A[i+1..n]$ contém os $n-i$ maiores elementos de $A[1..n]$ em ordem.

```
Heapsort(A)
1 Build-Max-Heap(A)
2 for i ← A.length to 2 do
3 |  swap(A[1], A[i])
4 |  A.heap-size ← A.heap-size − 1
5 |  Max-Heapify(A, 1)
```

Solução:

- **Inicialização:** para $i = n$, o subvetor $A[1..n]$ contém os n menores elementos do subvetor A, e o subvetor $A[n+1..n]$ não contém elementos, portanto o invariante é válido na inicialização.
- **Manutenção:** considerando que no início de uma iteração i qualquer o invariante seja válido, precisamos mostrar que a execução do corpo do laço (linhas 3–5) faz com que o invariante permaneça válido para a próxima iteração. A execução da linha 3 faz com que o maior elemento do heap seja colocado na posição $A[i]$. Este elemento, pelo invariante, é menor que qualquer elemento de $A[i+1..n]$. Portanto, se $A[i+1..n]$ continha os $n-i$ maiores elementos de $A[1..n]$ em ordem, colocando este elemento na i-ésima posição faz com que o vetor $A[i..n]$ contenha os $n-(i+1)$ maiores elementos em ordem (a segunda afirmação do invariante fica válida para a próxima iteração). A execução das linhas 4 e 5 reestabelece o heap (removendo a última folha e fazendo com que o elemento que foi colocado na raiz desça para a sua posição correta no heap) de modo que os elementos do heap em $A[1..i-1]$ são os menores elementos de $A[1..n]$, portanto, a primeira afirmação do invariante fica válida para a próxima iteração.
- **Término:** O laço termina quando $i = 1$. Neste caso, $A[1..1]$ contém apenas o menor elementos de $A[1..n]$ e $A[2..n]$ contém os maiores elementos de $A[1..n]$.

Portanto, após o término, os elementos de A estarão ordenados.

5. [Valor: 1,5] Reações durante as provas da disciplina de PAA são sempre diversas. Algumas vezes ocorrem uma sequência de "Ha"s, outras "Buá"s e em alguns casos extremos uns surtam exclamando "Fora Temer!"s. Sabendo que as expressões são sempre repetitivas, o professor resolveu escrever um algoritmo em que são passados como parâmetros um n (assuma que $n = 2^k$, tal que k é um número inteiro positivo), e *expr* que corresponde a expressão que o aluno deseja manifestar. Porém, o professor restringe o uso do algoritmo àqueles que são capazes de informar, por meio de notação assintótica, o número de mensagens expressas em função de n. Mostre que você é capaz de usar o algoritmo.

```
Prog(n, expr)
1 while n ≥ 1 do
2 |  for j ← 1 to n do
3 |  |  print expr
4 |  n ← n/2
```

Solução:

Desejamos saber quantas vezes a linha 3 será executada. Note que a variável n sempre é dividida por 2, ou seja, a quantidade de vezes que o laço **while** será executado será: $n/2^k \iff k = \lfloor \lg n \rfloor$. O laço interno sempre executa a mesma quantidade de vezes que o valor de n. Assim, o total de expressões impressas será dado pelo somatório:

$$1 + 2 + 4 + \ldots + \frac{n}{2} + n = \sum_{i=0}^{\lg n} 2^i = \frac{2^{\lg n+1} - 1}{2-1} = 2n^{\lg 2} - 1 = \Theta(n).$$

6. [Valor: 2,0] Um vetor de inteiros distintos $A = \{a_1, \ldots, a_n\}$ é *unimodal* se existe um elemento máximo a_k tal que a sequência de valores de a_1 até a_k é crescente $(a_i > a_{i-1},$ para todo $1 < i \leq k)$ e, a sequência de valores de a_k até a_n é decrescente $(a_i < a_{i-1},$ para todo $k < i \leq n)$. Exemplo: $\{1, 3, 5, 9, 6, 4\}$ é unimodal, mas $\{1, 3, 9, 4, 2, 5\}$ não é unimodal.

   (a) [Valor: 1,0] Descreva um algoritmo que, dado um vetor unimodal A, determina k em tempo $O(\lg n)$. Argumente que seu algoritmo está correto.

(b) [Valor: 1,0] Explique por que sua solução leva tempo $O(\lg n)$.

Solução:

a) Usar uma ideia parecida com busca binária, consultando o elemento anterior e o posterior.

```
UNIMODAL(A[], left, right)
if right − left <= 2 then
|  return MAXIMO(A, left, right)
k ← left + ((right − left)/2)
if A[k] > A[k − 1] and A[k] > A[k + 1] then
|  return A[k]
else if A[k] > A[k − 1] and A[k] < A[k + 1] then
|  return UNIMODAL(A, k + 1, right)
else
|  return UNIMODAL(A, left, k − 1)
```

b) Recorrência $T(n) = T(n/2) + \Theta(1)$. Caso 2 do método mestre $T(n) = \Theta(\lg n)$.

7. [Valor: 1,0] Considere o algoritmo de ordenação RADIX-SORT, aplicado a um conjunto de n inteiros representados na base decimal, em que cada número é maior ou igual a 0 e menor ou igual a $n^2 - 1$. Sabendo que um número x possui $\lfloor \log_b x \rfloor + 1$ dígitos na base b, é correto afirmar que nestas condições o algoritmo tem complexidade $O(n)$? Explique.

Solução:

Não é correto afirmar que nestas condições o `Radix sort` tem complexidade $O(n)$. Vimos que o `Radix sort` tem complexidade $\Theta(d(n + k))$, onde d é a quantidade de dígitos dos números, n é a quantidade de números, e k a quantidade de possíveis valores para cada dígito. Se usarmos o `Radix` diretamente como vimos em aula (base 10), a complexidade do algoritmo ficaria $O((\lfloor \log_{10} n^2 \rfloor + 1)(n + 10)) = O(n \lg n)$.

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 $d = \lfloor \log_n n^2 \rfloor + 1 = 2 + 1$, ou seja, a complexidade do `Radix sort` seria $O(3(n + n)) = O(6n) = O(n)$.

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 $[0 \ldots n^2 - 1]$ para a base n será igual a 3. Portanto, converter n números pode ser feito em tempo $O(n)$.

Boa Prova!!!