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] Para cada item a seguir, assinale **V**erdadeiro ou **F**also. **Justifique** sua resposta usando as definições de notação assintótica. [Respostas sem justificativa não serão consideradas.]

   (a) ■ **V** □ F Se $f(n) = \log_{16} n$ então $f(n) = \Theta(\lg n)$?

   Solução:
   Verdadeiro, pois $\log_{16} n = \frac{\lg n}{\lg 16} = \frac{\lg n}{4}$. Temos que mostrar que: $\frac{\lg n}{4}$ pertence a $O(\lg n)$ e a $\Omega(\lg n)$. No primeiro caso, $\frac{1}{4}\lg n \leq c_2 \lg n$. Portanto, para $c_2 = 1$ e $n_0 = 1$ (qualquer valor de c_2 maior que 1/4 serve) temos $f(n) = O(\lg n)$. No segundo caso, $\frac{1}{4}\lg n \geq c_1 \lg n$ (c_1 deve ser menor ou igual a 1/4). Fixando $c_1 = 1/4$ e usando o mesmo valor para n_0, temos que $f(n) = \Omega(\lg n)$. Logo, $f(n) = \Theta(\lg n)$.

   (b) □ V ■ **F** $2^{n+a} = \Theta(2^{2n})$? Onde $a \in \mathbb{N}$ (conjunto dos números naturais) é uma constante.

   Solução:
   Falso. Basta observar que $2^{n+a} \notin \Omega(2^{2n})$. Suponha (com a intenção de chegar a uma contradição) que $2^{n+a} \geq c2^{2n}$. Desenvolvendo os dois lados temos: $2^n 2^a \geq c2^n 2^n$. Isolando o c, para que esta desigualdade seja verdadeira, $c \leq \frac{2^a}{2^n} \Leftrightarrow c \leq 2^{a-n}$ (ou seja, c não é uma constante, pois depende do valor de n). Uma outra forma de chegar em uma contradição é observar que $2^n \leq \frac{2^a}{c}$, ou seja, n estaria limitado por um valor constante, o que contradiz a definição.

   (c) ■ **V** □ F $\frac{n^2}{4} - 3n - 16 = \Omega(n^2)$?

   Solução:
   Verdadeiro. Precisamos mostrar que existem constantes positivas c e n_0 tal que: $n^2/4 - 3n - 16 \geq cn^2$, $\forall n \geq n_0$. Dividindo os dois lados da desigualdade $n^2/4 - 3n - 16 \geq cn^2$ por n^2 temos: $c \leq 1/4 - 3/n - 16/n^2$. Observando que $\lim_{n\to\infty}(1/4 - 3/n - 16/n^2) = 1/4$, sabemos que c deve ser menor ou igual a 1/4. Para que c seja positivo, defina $n_0 = 20$ (por exemplo). Assim, $(1/4 - 3/20 - 16/400) = 0.06$ Ou seja, $n_0 = 20$ e $c = 0.06$.

   (d) ■ **V** □ F $7n^2 + 13n = O(n^2)$

   Solução:
   Verdadeiro. Precisamos mostrar que existem constantes positivas c e n_0 tal que: $7n^2 + 13n \leq cn^2$. Observe que $7n^2 + 13n \leq 7n^2 + 13n^2 = 20n^2$, portanto, para $c = 20$ e $n_0 = 1$, temos $7n^2 + 13n \leq cn^2$ $\forall n > n_0$.

2. [Valor: 2,0] Suponha que, para entradas de tamanho n, você tenha que escolher um dentre os três algoritmos A, B e C, descritos a seguir.

   (a) Algoritmo A resolve problemas dividindo-os em cinco subproblemas de tamanho $n/2$, recursivamente resolve cada subproblema, e então combina suas soluções em tempo linear para obter uma solução do problema original.

   (b) Algoritmo B resolve problemas dividindo-os em dois problemas de tamanho $n-1$, recursivamente resolve cada um dos subproblemas e então combina as soluções em tempo constante para obter a solução do problema original.

(c) Algoritmo C resolve problemas dividindo-os em nove subproblemas de tamanho $n/3$, recursivamente resolve cada subproblema e então combina suas soluções em tempo $O(n^2)$ para obter uma solução do problema original.

Qual é o consumo de tempo de cada um destes algoritmos (em notação assintótica)? Qual algoritmo é assintóticamente mais eficiente?

Solução:

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

(a) A recorrência do Algoritmo A é $T(n) = 5T(n/2) + \Theta(n)$. Usando o Teorema Mestre para identificar o custo, temos que $a = 5$, $b = 2$, $f(n) = n$. Comparamos $f(n)$ com $n^{\log_b a} = n^{\log_2 5} = n^{2,32}$. Caso 1 do Método Mestre, pois $f(n) = O(n^2)$ para $\epsilon = 0,32$. Portanto a complexidade de tempo do Algoritmo A é $\Theta(n^{2,32})$.

(b) A recorrência do Algoritmo B é $T(n) = 2T(n-1) + \Theta(1)$. Não podemos usar o Método Mestre. Usaremos o método iterativo (poderíamos usar árvore de recursão também).

$$
\begin{aligned}
T(n) &= 2.T(n-1) + c \\
&= 2.(2.T(n-2) + c) + c \\
&= 2.2.T(n-2) + 2c + c \\
&= 2.2.(2.T(n-3) + c) + 2c + c \\
&= 2.2.2.T(n-3) + 2.2c + 2c + c \\
&= \vdots \\
&= 2^k.T(n-k) + \textstyle\sum_{i=0}^{k-1} 2^i c & (T(n-k) = T(1) = \Theta(1) \text{ quando } k = n-1) \\
&= 2^{n-1}.c + \textstyle\sum_{i=0}^{n-2} 2^i c & (\textstyle\sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1) \\
&= 2^{n-1}.c + c(2^{n-1} - 1) \\
&= 2^{n-1}.c + c2^{n-1} - c \\
&= 2.2^{n-1}.c - c \\
&= c.2^n - c.
\end{aligned}
$$

Portanto, a complexidade do Algoritmo B é $\Theta(2^n)$.

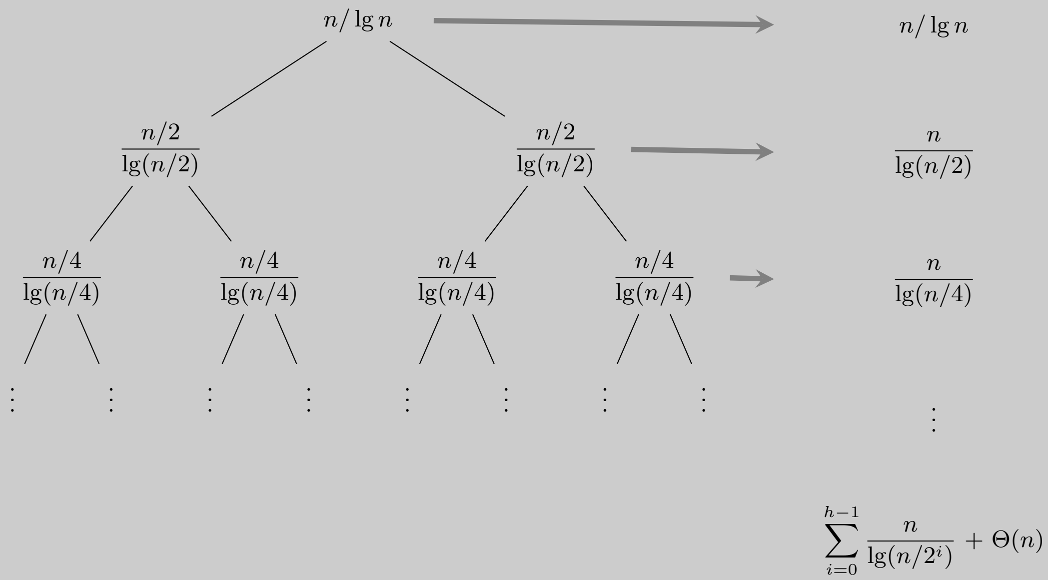
(c) A recorrência do Algoritmo C é $T(n) = 9T(n/3) + \Theta(n^2)$. Usando o Teorema Mestre para identificar o custo, temos que $a = 9$, $b = 3$, $f(n) = n^2$. Comparamos $f(n)$ com $n^{\log_b a} = n^{\log_3 9} = n^2$. Caso 2 do Método Mestre, pois $f(n) = \Theta(n^2)$. Portanto a complexidade de tempo do Algoritmo C é $\Theta(n^2 \lg n)$.

O algoritmo assintoticamente mais eficiente é o Algoritmo C.

3. [Valor: 2,0] Utilize o método de árvore de recursão ou o método iterativo para supor um limite assintótico superior restrito para a recorrência $T(n) = 2T(n/2) + n/\lg n$. Depois verifique pelo método de substituição que este limite está correto.

Solução:

Construindo a árvore de recursão (ver figura a seguir), observamos que cada nível i possui 2^i nós. O tamanho de um problema em cada nó no nível i é $n/2^i$. O custo de cada nó é $\frac{n/2^i}{lg(n/2^i)}$. A altura (h) da árvore é $h = \lg n$. A quantidade de nós folhas é dada por $2^h = 2^{\lg n} = n^{\lg 2} = n$. Assim, sabemos que no último nível temos n nós com custo $\Theta(1)$, ou seja, o custo somado de todas as folhas é $\Theta(n)$.



Somando o custo de todos os níveis obtemos:

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{h-1} \frac{n}{\lg(n/2^i)} + \Theta(n) \\
&= n \sum_{i=0}^{h-1} \frac{1}{\lg(n/2^i)} + \Theta(n) \\
&= n \sum_{i=0}^{h-1} \frac{1}{\lg n - \lg 2^i} + \Theta(n) \\
&= n \sum_{i=0}^{h-1} \frac{1}{\lg n - i} + \Theta(n) \\
&= n\Big(\frac{1}{\lg n - 0} + \frac{1}{\lg n - 1} + \ldots + \frac{1}{\lg n - (\lg n - 1)} \Big) + \Theta(n) \\
&= n \sum_{i=1}^{\lg n} \frac{1}{i} + \Theta(n) \qquad \text{(série harmônica} \sum_{k=1}^{n} \frac{1}{k} = \ln n + \Theta(1)) \\
&= n.\ln(\lg n) + \Theta(1) + \Theta(n) \\
&= O(n \lg(\lg n)).
\end{aligned}
$$

Vamos mostrar por substituição que a recorrência $T(n) = 2T(n/2) + n/\lg n$ é $O(n \lg(\lg n))$, porém, não vamos usar diretamente substituição (como vimos em aula). Vamos mostrar que $T(n) \leq n.(1 + H_{\lfloor \lg n \rfloor})$ onde H_k é o k-ésimo número harmônico, isto é $H_k = \sum_{i=1}^{k} \frac{1}{i}$. Definimos $H_0 = 0$.

Caso base: Para $n = 1$ e $T(1) = 1$, $\lg n = \lfloor \lg n \rfloor = 0$. Como $H_0 = 0$, $T(1) = 1 \leq 1(1 + H_0)$.

Hipótese: $T(k) \leq k.(1 + H_{\lfloor \lg k \rfloor})$ para $k < n$ (em particular para $k = n/2$).

$$
\begin{aligned}
T(n) &\leq 2.(n/2).(1 + H_{\lfloor \lg n/2 \rfloor}) + n/\lg n \\
&= n.(1 + H_{\lfloor \lg n - \lg 2 \rfloor}) + n/\lg n \\
&= n.(1 + H_{\lfloor \lg n - 1 \rfloor}) + n/\lg n \\
&= n.(1 + H_{\lfloor \lg n - 1 \rfloor} + 1/\lg n) \qquad \text{(n em evidência)} \\
&\leq n.(1 + H_{\lfloor \lg n - 1 \rfloor} + 1/\lfloor \lg n \rfloor) \\
&= n.(1 + H_{\lfloor \lg n \rfloor}).
\end{aligned}
$$

4. [Valor: 2,0] O i-ésimo menor elemento de um conjunto de n elementos é chamado de *i-ésima estatística de ordem*. Por exemplo, o mínimo de um conjunto de elementos é a primeira estatística de ordem ($i = 1$), e o máximo é a n-ésima estatística de ordem ($i = n$). Dado um conjunto A de n números (distintos) e um número i, com $1 \leq i \leq n$, definimos o *problema de seleção* como sendo o problema de encontrar o elemento $x \in A$ que é maior que exatamente $i - 1$ outros elementos de A. Este problema pode ser resolvido no tempo $O(n \lg n)$, pois podemos ordenar os números usando o Mergesort (por exemplo) e então indexar o i-ésimo elemento no vetor de saída. Outra forma de fazer isto é usando o algoritmo descrito a seguir, sendo `RANDOMIZED-PARTITION` o mesmo algoritmo usado no Quicksort aleatório. Argumente (informalmente, mas de maneira precisa) por que o algoritmo funciona. Faça a análise de complexidade do algoritmo para o melhor caso e para o pior caso (use a notação assintótica).

```
RANDOMIZED-SELECT(A,p,r,i)
if p == r
   return A[p]
q = RANDOMIZED-PARTITION(A,p,r)
k = q - p + 1
if i == k      //O valor pivô é a resposta
   return A[q]
else if i < k
   return RANDOMIZED-SELECT(A,p,q-1,i)
else return RANDOMIZED-SELECT(A,q+1,r,i-k)
```

Solução:

Para entender por que o algoritmo funciona, é preciso entender o que o procedimento `RANDOMIZED-PARTITION` faz. Este algoritmo de partição do Quicksort pega um elemento do vetor A como pivô e divide-o em duas partes, tal que todos os elementos à esquerda do pivo são menores e todos os elementos à direita do pivô são maiores. A função `RANDOMIZED-PARTITION` devolve o índice do pivô em A.

Com esta informação sobre o índice do pivô, existem três casos possíveis.

- O i-ésimo elemento que estou procurando é igual ao índice do pivô. Neste caso basta devolver o elemento que está neste índice.
- O i-ésimo elemento que estou procurando é maior que o índice do pivô. Neste caso não precisamos procurar nos elementos com índice menor que o pivô, pois são todos menores. Então temos que tentar encontrar este elemento na parte direita.
- O i-ésimo elemento que estou procurando é menor que o índice do pivô. De maneira análoga ao caso anterior, descartamos a parte a parte superior (ficamos apenas com a parte da esquerda).

A partir destas observações, fica fácil observar que o algoritmo encontra corretamente o i-ésimo menor elemento.

Quanto à complexidade, o melhor caso ocorre quando ao sortear o pivô, coincidentemente é o i-ésimo elemento que estou procurando é próprio pivô. Neste caso o custo total é $\Theta(n)$.

O pior caso ocorre quando um dos lados do pivô está vazio (mesmo caso ruim para o Quicksort). Neste caso, temos a recorrência $T(n) = T(n-1) + n$ que se desenvolvermos iremos obter o custo $\Theta(n^2)$.

5. [Valor: 2,0] Considere o algoritmo `heapsort` descrito a seguir.

   (a) Argumente 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.

(b) É verdade que para qualquer entrada o algoritmo `heapsort` tem comportamento $O(n \lg n)$? Justifique.

```
heapsort(A)
1 build-max-heap(A)
2 for i = A.comprimento downto 2
3    troca(A[1], A[i])
4    A.tamanho-do-heap = A.tamanho-do-heap - 1
5    max-heapify(A, 1)
```

Solução:

(a) Para mostrar que o algoritmo está correto, precisamos mostrar que o invariante é válido na inicialização, manutenção e término.

- **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 $A[1..n]$.

(b) Sim. Considere o pior caso. A construção do heap (linha 1) pode ser feita em $\Theta(n)$. As linhas 3 e 4 consomem tempo constante e são executadas $n-1$. A linha 5 no pior caso consome $\Theta(\lg n)$ e como é executada $n-1$, então no pior caso ela consome tempo $\Theta(n \lg n)$. Portanto, o consumo total é $O(n \lg n)$.

Boa Prova!!!

Aluno(a): ____