IMD0029 - Estrutura de Dados Básicas 1 – 2018.1 – Prova 01
Prof. Eiji Adachi M. Barbosa

Nome:_____

Matrícula: _____

ANTES DE COMEÇAR A PROVA, leia atentamente as seguintes instruções:

- Esta é uma prova escrita de caráter individual e sem consultas a pessoas ou material (impresso ou eletrônico).
- A prova vale 5,0 pontos na Unidade I e o valor de cada questão é informado no seu enunciado.
- Preze por respostas legíveis, bem organizadas e simples.
- As respostas devem ser fornecidas preferencialmente em caneta. Respostas fornecidas a lápis serão aceitas, mas eventuais questionamentos sobre a correção não serão aceitos.
- Celulares e outros dispositivos eletrônicos devem permanecer desligados durante toda a prova.
- Desvios éticos ou de honestidade levarão à anulação da prova do candidato (nota igual a zero).

Questão 1: (1,5 ponto) Sequências bitônicas inversas são aquelas que possuem duas sequências, sendo uma sequência inicial decrescente, seguida de uma sequência crescente. Ou seja, os elementos de uma sequência bitônica obedecem a seguinte relação:

$$A_0 > A_1 > ... > A_{i-1} > A_i < A_{i+1} < ... < A_n$$

Considere que um array bitônico-inverso é um array de inteiros sem repetições cujos elementos representam uma sequência bitônica inversa. Neste contexto, implemente uma função que recebe como entrada um array bitônico-inverso e retorna o índice do elemento da "base". O elemento da "base" é o último elemento da sequência inicial decrescente e o primeiro elemento da sequência final crescente, ou seja, é o elemento Ai da relação acima. Sua função deverá obrigatoriamente ser **recursiva**, ter complexidade **$\Theta(\lg(n))$** e seguir a assinatura:

```
int findBase(int a[], int arraySize)
```

Obs.: Nesta questão, não podem ser usadas instruções para realizar repetição, como for, while e do-while. Ou seja, você não poderá usar instruções de repetição; você deverá construir sua solução apenas com chamadas recursivas.

Questão 2: (1,0 ponto) O algoritmo de ordenação por inserção funciona mantendo num array A de tamanho N duas regiões distintas: uma região que fica entre [0...i], onde ficam os elementos já ordenados, e uma outra região entre [i+1...N-1], onde ficam os elementos ainda não ordenados. A cada iteração, o algoritmo pega o primeiro elemento na região não ordenada e insere-o na região ordenada. Para isto, é necessário encontrar a posição correta de onde inserir o elemento. Considere uma versão do algoritmo de ordenação por inserção que usa a busca binária para encontrar a posição correta de onde o primeiro elemento da região não ordenada deve ser inserido na região ordenada. O uso da busca binária melhora a eficiência do algoritmo de ordenação por inserção? Justifique.

Questão 3: (1,0 ponto) Considere nesta questão que estamos trabalhando apenas com arrays não ordenados, que possuem N elementos e apenas 3 valores distintos para cada elemento (ex.: apenas os valores 1, 2, 3, podendo ser um destes arrays igual a [3, 1, 2, 3, 4, 1, 2, 3, 1]). Ou seja, os arrays possuem diversas repetições de apenas 3 elementos. Neste contexto, crie uma função capaz de ordenar este array sem utilizar outro array auxiliar. Sua função deverá obrigatoriamente ser **iterativa**, ter complexidade **$\Theta(n)$** e seguir a assinatura:

```
void sort(int a[], int arraySize)
```

Questão 4: (1,5 ponto) Para cada uma das afirmações a seguir, marque V (verdadeiro) ou F (falso), **justificando sucintamente** sua resposta. Marcações de V ou F **sem justificativas não serão aceitas.**

1 – ( ) Os algoritmos *Quick Sort*, e *Merge Sort* possuem a mesma complexidade assintótica para o melhor e pior caso.

2 – ( ) Os algoritmos *Insertion Sort*, *Bubble Sort* e *Selection Sort* possuem a mesma complexidade assintótica para o melhor caso.

3 – ( ) O algoritmo *Quick Sort* possui complexidade assintótica $\Theta(n^2)$ para o pior caso, mas se selecionarmos o pivô aleatoriamente garantimos que não se cai no pior caso.

4 – ( ) O algoritmo *Merge Sort* possui complexidade assintótica $\Theta(n*\log_2(n))$ para o melhor caso e $\Theta(n^2)$ para o pior caso.

5 – ( ) O pior caso do *Quick Sort* é quando nas sucessivas chamadas recursivas se divide um problema de tamanho N em dois sub-problemas de tamanhos bastante similares.

6 – ( ) No *Merge Sort,* é impossível dividir um problema de tamanho N em dois sub-problemas de tamanhos 1 e N-1.

7 – ( ) Os algoritmos de busca sequencial e de busca binária podem ser empregados nos mesmos tipos de array.

8 – ( ) Os algoritmos de busca sequencial e de busca binária possuem mesma complexidade assintótica para o pior caso.