

Instruções: Escreva o nome e o número USP na folha de papel almaço. Numere cada página. Indique o total de páginas na primeira página. Os códigos fornecidos na seção “Códigos-fonte de apoio” podem ser referenciados em qualquer resposta sem necessidade de reprodução.

1. (2,0 pontos) A codificação de Hoffman codifica símbolos em cadeias de *bits* organizando-os em árvores binárias a partir de sua distribuição relativa de modo a minimizar a quantidade de *bits* necessária para codificar uma mensagem com estes símbolos.

- (a) (1,0 pontos) A tabela 1 mostra a quantidade de vitórias por país na natação masculino dos jogos olímpicos de 2008. Construa o código de Hoffman que codifica o país vitorioso usando esta distribuição. Quantos bits seriam necessários no total para codificar os vitoriosos de cada prova?

Tabela 1:

País	Vitórias
USA	14
AUS	5
NED	5
ITA	3
UKR	2
ROM	2
SWE	1
HUN	1

**Resposta:** A árvore de código de Horner é a seguinte:

País	código	tam.
USA	1	1
AUS	000	3
NED	001	3
ITA	0100	4
UKR	0101	4
ROM	0110	4
SWE	01110	5
HUN	01111	5

E o código equivalente é:

Com esta codificação a quantidade total de bits para codificar os vencedores de 2008 é  $14 \times 1 + 5 \times 3 + 5 \times 3 + 3 \times 4 + 2 \times 4 + 2 \times 4 + 1 \times 5 + 1 \times 5 = 82$  bits ou aproximadamente 2,5 bits por prova.

- (b) (1,0 pontos) As tabelas abaixo indicam propostas de codificação para os símbolos 'A', 'B', 'C', 'D', e 'E'.

Quais delas seriam possíveis codificações de Hoffman para estes símbolos?  
Quais não poderiam sê-lo? Justifique a sua resposta.

Código 1	Código 2	Código 3	Código 4
A 001	A 0	A 01	A 0
B 010	B 01	B 10	B 10
C 011	C 001	C 10	C 110
D 100	D 0001	D 11	D 1110
E 101	E 00001	E 011	E 11110

**Resposta:** Códigos de Hoffman podem ser interpretados como caminhos em uma árvore binária na qual os símbolos codificados estão em *folhas*. Disso segue que nenhum código para um símbolo pode ser prefixo para outro. Deste modo o código 2 não pode ser um código de Hoffman, visto que o código do símbolo 'A' é prefixo para os símbolos seguintes. Do mesmo modo, o código 3 não pode ser código de Hoffman, visto que o código do símbolo 'A' é prefixo para o símbolo 'E'. Os códigos 1 e 4 podem ser códigos de Hoffman.

2. (2,0 pontos) O código abaixo apresenta uma implementação do algoritmo de ordenação mergesort:

```
1 static void mergesort(int a[]) {
2     int temp[]=new int[a.length];
3     mergesort(a, temp, 0, a.length-1);
4 }
5
6 static void mergesort(int a[], int temp[], int left, int right) {
7     if (left<right) {
8         int center=(left+right)/2;
9         mergesort(a, temp, left, center);
10        mergesort(a, temp, (center+1), right);
11        merge(a, temp, left, center, (center+1), right);
12    }
13 }
14
15 static void merge(int a[], int temp[], int leftStart, int leftEnd, int rightStart, int rightEnd) {
16     int start=leftStart;
17     int aux=start;
18     while((leftStart<=leftEnd) &&(rightStart<=rightEnd)) {
19         if(a[leftStart]<a[rightStart])
20             temp[aux++]=a[leftStart++];
21         else
22             temp[aux++]=a[rightStart++];
23     }
24     while (leftStart<=leftEnd) temp[aux++]=a[leftStart++];
25     while (rightStart<=rightEnd) temp[aux++]=a[rightStart++];
26     for (int i=start; i<=rightEnd; i++) a[i]=temp[i];
27 }
```

A função `mergesort(int a[])` ordena em ordem crescente o vetor de inteiros `a` dividindo-o recursivamente em sub-vetores e mesclando-os dois a dois de forma a garantir a ordenação. Mostre o funcionamento da função `mergesort` sobre o vetor `{6, 5, 4, 3, 2, 1}` indicando o conteúdo do vetor `a` a cada vez que uma chamada da função auxiliar `mergesort` da linha 6 *retorna* (ou seja, mostre o estado do vetor ao *final* da chamada da função). Note que a função auxiliar é invocada *várias* vezes durante a ordenação do vetor. A ordem correta em que os laços na linha 26 são executados é *fundamental* na sua resposta!

**Resposta:** 654321  
654321  
564321  
564321  
456321  
456321  
456231  
456231  
456123  
123456

3. (2,0 pontos) A listagem a seguir mostra uma implementação do método de Horner para avaliação de polinômios

```

1     static float horner(float x, float[] a) {
2         float res = 0;
3         for(int i=a.length-1;i>=0;i--) {
4             res += x*res + a[i];
5         }
6         return res;
7     }

```

Onde  $a$  é o vetor  $\{a_0, \dots, a_n\}$  de coeficientes do polinômio (com  $n = a.length - 1$ ) e  $x$  é o valor  $x$  sobre o qual o polinômio é calculado. O valor do polinômio é

$$p(x) = \sum_{i=0}^n a_i x^i$$

Mostre que o algoritmo está correto, ou seja, a função efetivamente retorna o valor  $p(x)$ .

**Resposta:** No laço da linha 3 faz a  $i$  decrescer unitariamente de  $n = a.length - 1$  a 0. Seja  $i$  o valor da variável  $i$  a cada iteração. Do mesmo modo, seja  $p_i$  o valor da variável  $res$  ao final da iteração na qual a variável  $i$  vale  $i$ . A regra de recorrência para  $p_i$  é:

$$p_{i-1} = x \cdot p_i + a_{i-1}$$

Mostra-se que

$$p_i = \frac{\sum_{j=i}^n a_j x^j}{x^i}$$

De fato, ao final da primeira iteração, com  $i = n$ ,  $p_n = a_n = \frac{\sum_{j=n}^n a_j x^j}{x^n}$ .

Por outro lado, se existe  $i$  para o qual a relação é verdadeira, então, pela lei de iteração,

$$p_{i-1} = x \frac{\sum_{j=i}^n a_j x^j}{x^i} + a_{i-1} = \frac{\sum_{j=i}^n a_j x^j}{x^{(i-1)}} + a_{i-1} = \frac{\sum_{j=(i-1)}^n a_j x^j}{x^{(i-1)}}$$

Ou seja, se ela é válida para  $i$ , também é válida para  $i - 1$ . Por outro lado, ao final da última iteração,  $i = 0$  e  $p_0 = \sum_{i=0}^n a_i x^i$ , que é o valor procurado.

4. (2,0 pontos) Considere o código abaixo:

```

1     static void mostraNos(NoArvoreBinaria raiz) {
2         Pilha p = new Pilha();
3         NoArvoreBinaria no;
4         p.push(raiz);
5         p.push('D');
6         do {
7             char c = (char)p.pop();
8             if(c=='D') {
9                 no = (NoArvoreBinaria)p.top();
10                p.push('E');
11                while(no!=null) {
12                    no = no.esquerda;
13                    p.push(no);
14                    p.push('E');
15                }
16            } else {
17                no = (NoArvoreBinaria)p.pop();
18                if(no!=null) {

```

```

19         System.out.println(no.valor);
20         p.push(no.direita);
21         p.push('D');
22     }
23 }
24 } while(!p.PilhaVazia());
25 }

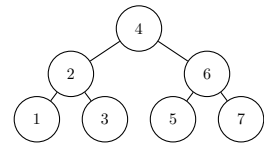
```

Este código usa as classes `Pilha` e `NoArvoreBinaria`.

A classe `NoArvoreBinaria` representa um nó de árvore binária. O campo `valor` contém o valor armazenado no nó. Os campos `esquerda` e `direita` são referências para as sub-árvores esquerda e direita respectivamente.

A classe `Pilha` representa uma pilha que armazena dados *genéricos* (como referências para o tipo de dado `Object`). Seu construtor cria uma nova pilha vazia. O método `push(Object x)` empilha um novo objeto na pilha. O método `pop()` retira e retorna o objeto no topo da pilha. O método `top()` retorna o objeto no topo da pilha *sem modificá-la*. O método `PilhaVazia()` retorna verdadeiro caso a pilha esteja vazia, falso caso contrário.

- (a) (1,0 pontos) Mostre *na sequência correta* o estado da pilha `p` a cada vez que as instruções da linha 5, 14 e 21 são executadas (imediatamente após a execução das mesmas) quando o parâmetro `raiz` aponta para o nó raiz da árvore ilustrada à direita:



	4	D						
	4	E						
	4	E	2	E				
	4	E	2	E	1	E		
	4	E	2	E	1	E	NULL	E
	4	E	2	E	NULL	D		
	4	E	2	E	NULL	E		
	4	E	3	D				
	4	E	3	E				
	4	E	3	E				
	4	E	3	E	NULL	E		
<b>Resposta:</b>	4	E	NULL	D				
	4	E	NULL	E				
	6	D						
	6	E						
	6	E	5	E				
	6	E	5	E	NULL	E		
	6	E	NULL	D				
	6	E	NULL	E				
	7	D						
	7	E						
	7	E	NULL	E				
	NULL	D						
	NULL	E						

- (b) (1,0 pontos) Em que ordem os nós da árvore são apresentados?

**Resposta:** Na ordem *interior* (primeiro o nó à esquerda é visitado, depois o nó raiz, depois o nó à direita).

5. (2,0 pontos) Escreva uma função que determina se em um vetor  $X$  ordenado em ordem crescente há 2 inteiros que somam *exatamente*  $y$ . Use a seguinte assinatura:

```
boolean existeSoma(int y, int[] x);
```

onde  $x$  é o vetor  $X$  de inteiros ordenado em ordem crescente e  $y$  é o número  $y$  que seve se igualar exatamente à soma de dois componentes de  $X$ . A função deve retornar verdadeiro se existem tais componentes ou falso caso contrário. Um algoritmo com complexidade *melhor* do que  $\mathcal{O}(N^2)$  vale 2 pontos, complexidades iguais ou piores valem 1 ponto.

**Resposta:** A solução envolve varrer o vetor elemento a elemento e verificar se ele somado a algum dos subsequentes dá exatamente  $y$ . Como os elementos subsequentes estão ordenados em ordem crescente, esta busca pode ser feita com complexidade  $\mathcal{O}(\log N)$ . A complexidade total é  $\mathcal{O}(N \log N)$

```
static boolean existeSoma(int y, int[] x) {  
    for(int i=0;i<x.length;i++) {  
        int esquerda = i+1;  
        int direita = x.length-1;  
        while(esquerda<=direita) {  
            int meio=(esquerda+direita)/2;  
            if(y==x[i]+x[meio]) return true;  
            else if(y<x[i]+x[meio]) direita=meio-1;  
            else esquerda = meio+1;  
        }  
    }  
    return false;  
}
```

## Códigos-fonte de apoio

### Classe Pilha

```
public class Pilha {
    public class ErroPilhaVazia extends java.lang.RuntimeException {}

    private NoListaLigada topo;

    public Pilha() {
    }

    public void push(Object x) {
        this.topo = new NoListaLigada(x, this.topo);
    }

    public Object pop() {
        if (topo == null) throw new ErroPilhaVazia();
        Object x = topo.value;
        topo = topo.next;
        return x;
    }

    public Object top() {
        if (topo==null) throw new ErroPilhaVazia();
        return topo.value;
    }

    public boolean PilhaVazia() {
        return topo==null;
    }
}
```

### Classe NoListaLigada:

```
public class NoListaLigada {
    public Object value;
    public NoListaLigada next;

    NoListaLigada(Object value, NoListaLigada next) {
        this.value = value; this.next = next;
    }
}
```

### Classe NoArvoreBinaria

```
public class NoArvoreBinaria {
    public int valor;
    public NoArvoreBinaria esquerda, direita;
}
```