**Prova de Recuperação — PMR3201/PMR2300 1o. semestre 2015**

Prof. Thiago Martins

Instruções: Escreva o nome e o número USP na folha de papel almaço.

1. (2,0 pontos) Uma pilha é uma estrutura de dados na qual o elemento a ser retirado é o *último* a ser inserido. A classe `PilhaAr` implementa uma pilha empregando um arranjo como base. Os métodos públicos da classe são:

   - `push(Object x)`: Adiciona o objeto `x` à pilha.
   - `pop()`: Remove e retorna o objeto mais recente a ser inserido, ou lança a exceção `ErroPilhaVazia` se não há mais objetos empilhados.
   - `tamanho()`: Retorna a quantidade de objetos empilhados.

   Escreva em java uma função que recupera o valor no *topo* de uma pilha, *sem alterar o seu conteúdo*. Use a seguinte assinatura:

   ```
   static Object topo(PilhaAr p)
   ```

   Onde `p` é a pilha cujo topo deve ser recuperado. A função deve lançar a exceção `PilhaAr.ErroPilhaVazia` caso `p` esteja vazia (lembre-se de usar apenas métodos públicos de `PilhaAr`).

   **Resposta:** O *único* método públic que retorna uma referência ao objeto no topo da pilha é `pop()`. Este método, no entanto, modifica o estado da pilha, removendo o objeto da mesma. A solução, naturalmente, é adicionar o objeto de volta:

   ,
   ```
   static Object topo(PilhaAr p) {
       Object x = pilha.pop();
       pilha.push(x);
       return x;
   }
   ```

   O comportamento de lançar a exceção caso a pilha esteja vazia é naturalmente fornecido pelo método `pop()`.

2. (2,0 pontos) Os resultados das segundas provas das turmas de PMR2300 e de PMR3201 podem ser representados por conceitos com a distribuição como indicado nas tabelas abaixo:

PMR2300:

| Conceito | Faixa | Fração |
|---|---|---|
| A+ | $\geq 10$ | 0 |
| A | $> 8,33$ e $< 10$ | 11% |
| B | $> 6,67$ e $< 8,33$ | 42% |
| C | $> 5$ e $< 6,67$ | 29% |
| D | $> 3,33$ e $< 5$ | 15% |
| E | $> 1,67$ e $< 3.33$ | 3% |
| F | $> 1,67$ | 0 |

Média: 6,45

PMR3201:

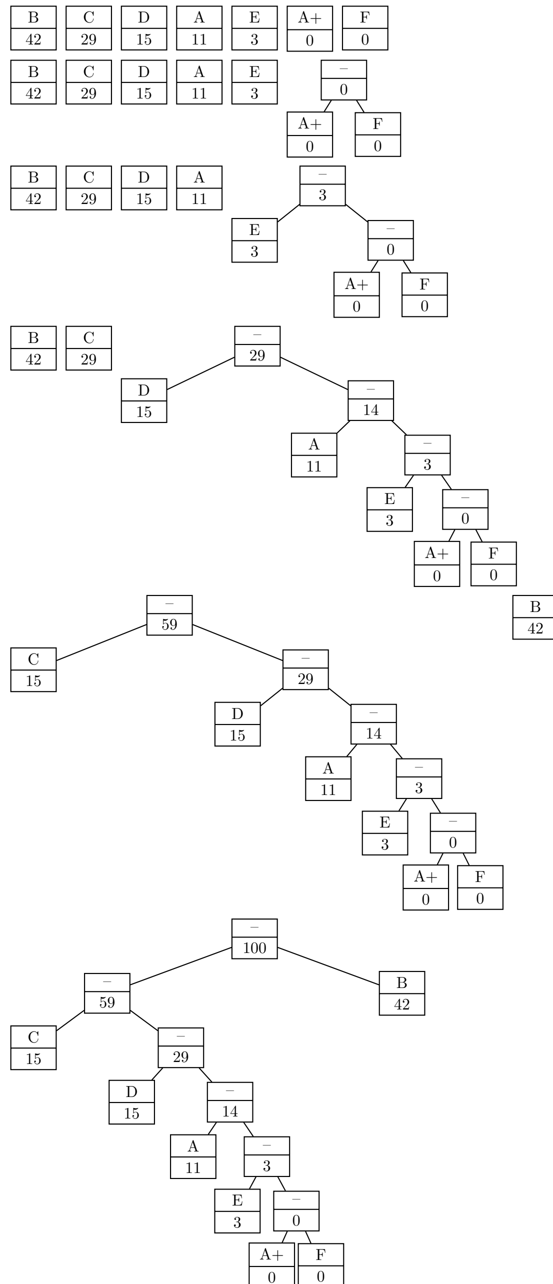
| Conceito | Faixa de Nota | Fração |
|---|---|---|
| A+ | $\geq 10$ | 4% |
| A | $> 8,33$ e $< 10$ | 22% |
| B | $> 6,67$ e $< 8,33$ | 18% |
| C | $> 5$ e $< 6,67$ | 26% |
| D | $> 3,33$ e $< 5$ | 26% |
| E | $> 1,67$ e $< 3.33$ | 4% |
| F | $> 1,67$ | 0 |

Média: 6,4

Normalmente seriam necessários 3 bits para codificar um conceito entre os 7 definidos (teoricamente o limite inferior é $\log_2 7 \approx 2,807$). Uma tabela de símbolos específica, no entanto, pode-se fazer melhor do que isso.

(a) (1,5 pontos) Crie um dicionário de codificação de Huffman ajustado para a distribuição de conceitos em PMR2300. Quantos bits em média são necessários para codificar a lista de conceitos da turma de PMR2300? Compare com o limite teórico $\log_2 7$.

**Resposta:** Construção da árvore de código de Huffman:




A partir desta árvore é possível construir o dicionário de Huffman para estes símbolos:

| Símbolo | Codificação | bits |
|---|---|---|
| A+ | `011110` | 6 |
| A | `0110` | 4 |
| B | `1` | 1 |
| C | `00` | 2 |
| D | `010` | 3 |
| E | `01110` | 5 |
| F | `0111111` | 6 |

Usando este dicionário, a codificação das notas de PMR2300 usaria em média 2,04 bits por símbolo, uma melhoria de 33% sobre os 3 bits e de 27% sobre $\log_2 7$.

(b) (0,5 pontos) Com o dicionário produzido no item (a), quantos bits em média são necessários para codificar a lista de conceitos da turma de PMR3201? Compare com o limite teórico $\log_2 7$.

**Resposta:** Com o mesmo dicionário obtido no item (a), a codificação das notas de PMR3201 usaria em média 2,8 bits. Este é praticamente o valor de $\log_2 7$, o que mostra uma diferença expressiva entre as distribuições, embora as médias sejam muito próximas.

3. (2,0 pontos) O código a seguir mostra uma implementação em java do Bubblesort para ordenação em ordem crescente de vetores de inteiros:

```
1    static void bubblesort(int[] x) {
2        int ultimo_alterado;
3        int ultimo = x.length-1;
4        do {
5            ultimo_alterado = 0;
6            for(int j=1;j<=ultimo;j++) {
7                if(x[j]<x[j-1]) {
8                    int temp = x[j];
9                    x[j] = x[j-1];
10                   x[j-1] = temp;
11                   ultimo_alterado = j;
12               }
13           }
14           ultimo = ultimo_alterado - 1;
15       } while(ultimo>0);
16   }
```

O princípio do algoritmo é comparar a ordem relativa de elementos contíguos em um vetor e realizar a troca de posição sempre que necessário. Note que ao final do laço interno, todos os elementos entre as posições apontadas por `ultimo` e `ultimo_alterado` estão em ordem crescente e são maiores ou iguais a todos os elementos que os precedem. Como `ultimo` é inicializado com a última posição do vetor, e a cada iteração do laço externo é modificado para a posição anterior a `ultimo_alterado`, o algoritmo termina em tempo finito e ordena efetivamente o vetor.

(a) (1,0 pontos) Calcule em notação *Big Oh* a ordem de complexidade do algoritmo Bubblesort no seu *pior caso*.

**Resposta:** Seja $u_i$ o valor da variável `ultimo` ao final da execução da linha 14, na $i$-gésima iteração do laço externo (iniciado na linha 6). O laço na linha 6 vai executar um total de

$u_i$ passos. Seja $a_i$ o valor da variável `ultimo_alterado` ao final da execução da linha 14, na $i$-gésima iteração do laço interno. O valor *máximo* de $a_i$ é $u_{i-1}$, quando o último elemento a ser inspecionado pelo laço interno é modificado (isso ocorre por exemplo com uma entrada em ordem inversa, na qual o laço interno transporta o primeiro elemento até a posição $u_i$ e mantém a ordem relativa dos demais elementos inalterada). Assim, o valor máximo que $u_i$ pode assumir é $u_{i-1} - 1$. Ou, seja, no pior caso, $u_i$ segue uma sequência aritmética decrescente de razão -1. O número total de iterações do laço interno no pior caso para uma entrada com $n$ elementos é dado por:

$$\sum_{i=n}^{1} i = \frac{n^2+n}{2} = \mathcal{O}\left(n^2\right)$$

(b) (1,0 pontos) Calcule em notação *Big Oh* a ordem de complexidade do algoritmo Bubblesort no seu *melhor caso.*

**Resposta:** Quando o algoritmo recebe um vetor previamente ordenado, a comparação na linha 7 é *sempre* negativa, de modo que a alteração na variável `ultimo_alterado` na linha 11 *nunca* ocorre. Ora, neste caso, o valor de `ultimo_alterado` na linha 14 permanece 0, a variável `ultimo` recebe -1 e o laço externo sai após uma única iteração. Assim a complexidade é unicamente a de uma execução do laço interno, que para uma entrada de $n$ posições é $\mathcal{O}(n)$.

4. (2,0 pontos) Considere o código abaixo para converter um inteiro *maior* do que zero em uma String com sua representação decimal:

```
1   public static String convert(int x) {
2       String r = "";
3       while(x!=0) {
4           int d = x % 10;
5           // Adiciona o digito de d
6           //  a esquerda de r
7            r = (char)('0' + d) + r;
8            x /= 10;
9       }
10      return r;
11  }
```

(a) (1,0 pontos) Mostre que o algoritmo acima termina em tempo finito para $x$ maior do que zero.

**Resposta:** Seja $x_i$ o valor da variável `x` ao fina da execução da linha 8, ao final da $i$-gésima iteração do laço iniciado na linha 3. Pela linha 8, $x_i = \lfloor x_{i-1}/10 \rfloor$ e assim $x_i < x_{i-1}$. Por outro lado, $x_{i-1} > 0 \Rightarrow x_i \geq 0$. Assim, $x_i$ é inteiro, estritamente decrescente e limitado inferiormente por 0. Existe portanto um valor finito $n$ para o qual $x_n = 0$, condição de encerramento do laço.

(b) (1,0 pontos) Mostre que o algoritmo acima está correto, ou seja, ele efetivamente retorna a representação decimal de $x$ (sugestão: seja $i$ o número da iteração do loop interno. Considere ao final de cada iteração do loop os valores do número representado pela String armazenada em `r` e de $\mathtt{x} \cdot 10^i$).

**Resposta:** Seja $r_i$ o inteiro representado pela cadeia armazenada na variável `r` ao final da $i$-gésima iteração do laço (é trivial mostrar que `r` *sempre* contém uma representação válida de um inteiro). Seja $d_i$ o valor da variável `d` ao final da $i$-gésima iteração do laço. Vale $9 \geq d_i \geq 0$. Pela linha 7, $r_i = r_{i-1} + d_i \cdot 10^{i-1}i$. Isso procede, pois ao final da $i$-gésima iteração, a representação decimal de $r_i$ tem exatos $i$ dígitos. Assim, o novo dígito entra na casa com valor $10^{i-1}$. Por outro lado, $d_i = 10x_i - x_{i-1}$. Decorre de ambas as relações que $10^i x_i + r_i = 10^{i-1}x_{i-1} + r_{i-1}$, ou seja, o valor $10^i x_i + r_i$ é *invariante*. Antes da entrada do laço, $10^0 x_0 + r_0$ é o valor inicial da variável $x$ (a cadeia vazia representa 0). Ao final do laço, na $n$-ésima iteração quando $x_n = 0$, $10^n x_n + r_n$ é igual a $r_n$, o valor final representado pela cadeia. Assim, $r_n = x_0$ e a cadeia contém efetivamente a representação decimal do valor da variável `x`.

5. (2,0 pontos) Uma árvore de busca binária é uma árvore binária na qual cada nó é *estritamente maior* que *todos* os elementos da sua sub-árvore esquerda e *estritamente menor* que *todos* os elementos da sua sub-árvore direita (vide exemplos a seguir). Esta propriedade permite a rápida localização de elementos nela contidos. A classe `NoBinarioFloat` contém nós de uma árvore binária que armazena números em ponto flutuante. O campo `val` contém o número armazenado no nó, o campo `esquerda` contém uma referência para a raiz da sub-árvore esquerda e o campo `direita` contém uma referência para a raiz da sub-árvore direita.

   (a) (1,5 pontos) Escreva de forma *não*-recursiva um código em Java que adiciona um novo valor a uma árvore pré-existente. Use a seguinte assinatura:

   ```
   static void insereNovoNo(NoBinarioFloat raiz, double n)
   ```

   Onde `raiz` é o nó raiz da árvore à qual deve ser adicionado o novo valor (considere sempre não-nulo) e `n` é o novo valor a ser inserido.

**Resposta:** O novo elemento deve ser sempre inserido como uma folha (mantendo os já existentes). O grande desafio deste problema para quem é familiar com a versão recursiva da inserção é que esta possui 5 comportamentos distintos:

1. Insere uma nova folha esquerda e termina.
2. Insere uma nova folha direita e termina.
3. Prossegue recursivamente na sub-árvore esquerda.
4. Prossegue recursivamente na sub-árvore direita.
5. Termina sem fazer nada.

A codificação da inserção na forma de um laço iterativo deve considerar estes 5 comportamentos distintos. Esta é uma situação em que um laço "perpétuo", quebrado por instruções `break` leva a uma codificação mais compacta. No entanto, um código com uma *flag* booleana que indica condição de continuidade para o laço é menos sujeito a erros:

```
static void insereNovoNo(NoBinarioFloat raiz, double n) {
    boolean continua = true;
    while(continua) { // Itera até inserir o no
        if(n<raiz.val) { // Insere na sub-arvore esquerda
            if(raiz.esquerda!=null)
                raiz = raiz.esquerda; // Continua na esquerda
            else {
                // Arvore vazia, cria novo no
```

```
                raiz.esquerda = new NoBinarioFloat(n, null, null);
                continua = false; // Encerra laco
            }
        } else if(n>raiz.val) { // Insere na sub-arvore direita
            if(raiz.direita!=null)
                raiz = raiz.direita; // Continua na direita
            else {
                // Arvore vazia, cria novo no
                raiz.direita = new NoBinarioFloat(n, null, null);
                continua = false; // Encerra laço
            }
        } else continua = false; // n não é menor nem maior, é igual,
                                 //  o elemento ja existe
    }
}
```

(b) (0,5 pontos) Qual a diferença em termos de consumo de recursos computacionais da versão recursiva para a não-recursiva?

**Resposta:** A versão recursiva usa memória de pilha em linguagens que não suportam *tail call optimization* (como é o caso da máquina virtual Java da Oracle).

---

# Formulário

## Somas de sequências:

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}, \quad \sum_{i=0}^{n-1} i^2 = \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}, \quad \sum_{i=0}^{n-1} i^3 = \frac{n^4}{4} - \frac{n^3}{2} + \frac{n^2}{4},$$

$$\sum_{i=0(a\neq 1)}^{n-1} a^i = \frac{1-a^n}{1-a}, \quad \sum_{i=0(a\neq 1)}^{n-1} ia^i = \frac{a - na^n + (n-1)a^{n+1}}{(1-a)^2}.$$

## Equivalência de recursão por *tail call* a laço:

**function** $F(\boldsymbol{X})$
    **if** $C(\boldsymbol{X})$ **then**
        **return** $E(\boldsymbol{X})$
    **else**
        **return** $F(G(\boldsymbol{X}))$
    **end if**
**end function**

Versão recursiva

**function** $F(\boldsymbol{X})$
    **while NOT** $C(\boldsymbol{X})$ **do**
        $\boldsymbol{X} \leftarrow G(\boldsymbol{X})$
    **end while**
    **return** $E(\boldsymbol{X})$
**end function**

Versão iterativa com laço

# Códigos-fonte de apoio

## Classe PilhaAr

```
public class PilhaAr {
   public class PilhaVazia extends java.lang.RuntimeException {}
   private Object arranjo[];
   private int topo;

   public PilhaAr() {
      arranjo = new Object[16];
      topo = -1;
   }

   public void push(Object x) {
      if (++topo == arranjo.length) dupliqueArranjo();
      arranjo[topo] = x;
   }

   public Object pop() {
      if (topo < 0) throw new PilhaVazia();
      return(arranjo[topo--]);
   }

   public boolean pilhaVazia() {
       return topo<0;
   }

   private void dupliqueArranjo()  {
      Object na[] =
              new Object[2 * arranjo.length];
      for (int i=0; i<arranjo.length; i++) na[i] = arranjo[i];
      arranjo = na;
   }
}
```

## Classe NoBinarioFloat

```
public class NoBinarioFloat {
    double val;
    NoBinarioFloat esquerda;
    NoBinarioFloat direita;


    NoBinarioFloat(double val, NoBinarioFloat e, NoBinarioFloat d) {
        this.val = val;
        esquerda = e;
        direita = d;
    }
}
```