Prova 1 — PMR2300 / PMR3201 1o. semestre 2015

Prof. Thiago Martins

Instruções: Escreva o nome e o número USP na folha de papel almaço. Indique na linha seguinte *quatro* das cinco questões abaixo que devem ser corrigidas.

1. (2,5 pontos) O produto de duas matrizes $\mathbf{A} = \mathbf{X} \cdot \mathbf{Y}$ quadradas $n \times n$ é definido como:

$$A_{i,j} = \sum_{k=1}^{n} X_{i,k} Y_{k,j}$$

Em Java, matrizes podem ser representadas por vetores de linhas, as linhas por sua vez, vetores de pontos flutuantes. Assim, por exemplo, se a variável `a` do tipo `double [][]` é uma referência a uma matriz $\mathbf{A}$, então o elemento $\mathbf{A}_{i,j}$ pode ser acessado por `a[i][j]`.

(a) (1.5 pontos) Com a representação acima, escreva uma função em Java que recebe 3 matrizes quadradas, `x`, `y` e `a` *de tamanhos idênticos*. A função deve escrever na matriz `a` o resultado do produto matricial de `x` e `y`. Use a seguinte assinatura:

```
public static void matrixmult(double [][] x, double [][] y, double[][] a)
```

Resposta: Embora a matriz `a` esteja pré-iniciada, *seu conteúdo é indeterminado.*

```
public static void matrixmult(double [][] x, double [][] y, double[][] a) {
    for(int i=0;i<a.length;i++) {
        for(int j=0;j<a[i].length;j++) {
            double total = 0;
            for(int k=0;k<y.length;k++) {
                total += x[i][k] + y[k][j];
            }
            a[i][j] = total;
        }
    }
}
```

(b) (1.0 pontos) Escreva em função de N, onde N é o número de linhas/colunas das matrizes quadradas, a ordem de complexidade da função escrita.

Resposta:

$$\mathcal{O}(N^3)$$

2. (2,5 pontos) A listagem a seguir mostra uma implementação do algoritmo de transformada rápida de Fourier segundo o algoritmo de Cooley-Tuckey:

```
import org.apache.commons.math3.complex.Complex;
import org.apache.commons.math3.complex.ComplexUtils;
import org.apache.commons.math3.util.FastMath;

1    public static void fft(Complex x[]) {
2        // Aloca espaco
```

```
 3        Complex[] t = new Complex[x.length];
 4        for(int i=0;i<x.length;i++)
 5            t[i] = new Complex(x[i].getReal(), x[i].getImaginary());
 6        fft_r(x,t,0,x.length,1);
 7    }
 8
 9    private static void fft_r(Complex x[], Complex y[], int start, int n, int s) {
10        if(n==1) return;
11
12        fft_r(y, x, start, n/2, s*2);
13        fft_r(y, x, start+s, n/2, s*2);
14
16        for(int k=0; k<n/2; k++)  {
16            double arg = -2*Math.PI*((double)k)/((double)n);
17            Complex t = y[start+s*(2*k+1)].
                  multiply(ComplexUtils.polar2Complex(1.0, arg));
18            x[start+k*s] = y[start+k*2*s].add(t);
19            x[start+(n/2+k)*s] = y[start+k*2*s].subtract(t);
20        }
21    }
```

Este algoritmo trabalha *exclusivamente* com vetores cujo tamanho é uma potência de 2. O seu ponto de entrada é a função `fft`, que por sua vez chama a função recursiva `fft_r`.

(a) (1.0 pontos) Mostre que o algoritmo *sempre* termina para um vetor com tamanho igual a uma potência de 2.

> **Resposta:** O algoritmo é um algoritmo recursivo cujo caso base é o parâmetro `n` igual a 1. Caso contrário, o algoritmo chama a si mesmo duas vezes, passando `n/2` como novo parâmetro `n`. Se o valor inicial de `n`, o tamanho inicial do vetor, for exatamente uma potência de 2, as sucessivas divisões por 2 farão com que o caso base seja atendido.

(b) (1.0 pontos) Escreva a equação de recorrência que dá a ordem, em notação *big Oh* do tempo de execução deste algoritmo para uma entrada do tamanho N. Considere que todas as operações com números complexos têm complexidade constante ($\mathcal{O}(1)$).

> **Resposta:** Seja $T(N)$ a ordem de complexidade do algoritmo. O algoritmo chama a si mesmo duas vezes com $N/2$ como parâmetro e realiza $N \cdot \mathcal{O}(1)$ operações sobre os vetores. A equação é:
>
> $$T(N) = 2T\left(\frac{N}{2}\right) + \mathcal{O}(N)$$

(c) (0.5 pontos) Resolva a equação e obtenha a ordem em notação *big Oh* do tempo de execução do algoritmo.

> **Resposta:** Na notação do *Master Theorem*, temos $A =$, $B = 2$ e $L = 1$. Neste caso, tem-se exatamente $A = B^L$ donde a solução é:
>
> $$T(N) = \mathcal{O}(N \log N)$$

3. (2,5 pontos) O fator h é uma métrica que mede a produção de um pesquisador, que combina quantidade de publicações com frequência na qual estas são citadas. O fator h de um pesquisador é o número

máximo n de publicações que ele produziu que foram citadas ao menos n vezes cada uma. Por exemplo, um pesquisador que possui 5 publicações que foram citadas respectivamente $\{5, 4, 3, 2, 1\}$ vezes possui um fator h de 3, visto que as 3 publicações mais citadas foram citadas ao menos 3 vezes cada e nenhuma das subsequentes foi citada 4 vezes. Já um pesquisador com 10 publicações que foram citadas $\{11, 10, 6, 5, 4, 3, 2, 1, 1, 1\}$ vezes cada uma possui um fator h de 4 (verifique!).

(a) (2,0 pontos) Escreva uma função em Java que, dado um vetor com a quantidade de citações que cada uma de suas publicações recebeu *ordenado em ordem decrescente*, calcula o fator h. A função deve possuir a seguinte assinatura:

```
public static int fatorh(int[] citacoes)
```

Uma função com complexidade $\mathcal{O}(N)$ vale 1 ponto, uma função com complexidade $\mathcal{O}(\log N)$ vale 2 pontos, onde N é o tamanho da lista de citações.

Resposta:

Função com complexidade $\mathcal{O}(N)$ (valendo 1 ponto):

```
1    public static int fatorh(int[] citacoes)
2        int i=0;
3        while(i<citacoes.length && citacoes[i]<=(i+1)) i++;
4        return i;
5    }
```

A função com complexidade $\mathcal{O}(N \log N)$ (valendo 2 pontos) é uma variante da busca binária:

```
1    public static int fatorh(int[] citacoes) {
2        int a = 0;
3        int b = citacoes.length;
4        int best = 0;
5        while(a<=b) {
6            int ref = (a+b)/2;
7            if(citacoes[ref]>=ref+1) { // Procura na metade superior
8                best = ref;
9                a = ref+1;
10           } else { // Procura na metade inferior
11               b = ref-1;
12           }
13       }
14      return best+1;
15   }
```

É possível omitir completamente a variável `best`:

```
1    public static int fatorh(int[] citacoes) {
2        int a = 0;
3        int b = citacoes.length;
4        while(a<=b) {
5            int ref = (a+b)/2;
6            if(citacoes[ref]>=ref+1) { // Procura na metade superior
7                a = ref+1;
8            } else { // Procura na metade inferior
9                b = ref-1;
10           }
11       }
12      return a;
```

```
13    }
```

Finalmente, há uma versão recursiva bastante compacta:

```
 1    public static int fatorh(int[] citacoes) {
 2        return fatorh(citacoes, 0, citacoes.length-1);
 3    }
 4    public static int fatorh(int[] citacoes, int a, int b) {
 5        if(a>b) return a;
 6        int ref = (a+b)/2;
 7        if(citacoes[ref]>=ref+1) return fatorh(citacoes, ref+1, b);
 8        return fatorh(citacoes, a, ref-1);
 9    }
```

(b) (0,5 pontos) Explique como, dado o seu conhecimento de algoritmos, é possível calcular o fator h a partir de uma lista *desordenada* com complexidade $\mathcal{O}(N \log N)$ (não é necessário mostrar o código).

Resposta: O algoritmo de ordenação *mergesort* ordena um vetor com complexidade $\mathcal{O}(N \log N)$. Assim, a complexidade das operações conjuntas ordenar por *mergesort* e calcular o fator h é de $\mathcal{O}(N \log N + N) = \mathcal{O}(N \log N)$.

4. (2,5 pontos) Considere o código abaixo para calcular a exponenciação x^a:

```
 1    public static int exp(int x, int a) {
 2        int r = 1;
 3        while(a!=0) {
 4            if(a%2!=0) { // Verdadeiro se a ímpar
 5                r *= x;
 6                a--;
 7            }
 8            a /= 2;
 9            x = x*x;
10        }
11        return r;
12    }
```

Para este algoritmo são relevantes as seguintes propriedades da exponenciação:

- Se a é par então $x^a = \left(x^2\right)^{\lfloor \frac{a}{2} \rfloor}$
- Para a positivo vale $x^a = x \cdot x^{a-1}$.

(a) (1,0 pontos) Mostre que o algoritmo acima termina em tempo finito para a não-negativo.

Resposta: seja i o número de iterações realizadas pelo laço da linha 3, a_i, r_i e x_i o valor das variáveis `a`, `r` e `x` respectivamente *após* a execução da linha 9 na i-gésima iteração. Assim, pelas linhas 6 e 8, $a_{i+1} < a_i$. No entanto, $a_i \geq 0$ pois a variável `a` só é submetida a operações de divisão por 2 e decremento *quando é par não-nulo*. Conclui-se daí que a_i é decrescente mas limitado inferiormente. Como a_i é inteiro, há um número finito possível de iterações.

(b) (1,0 pontos) Mostre que o algoritmo acima efetivamente retorna x^a (sugestão: Considere como varia a cada iteração o valor de $\mathtt{r} \cdot \mathtt{x}^{\mathtt{a}}$ após a execução da linha 9).

Resposta: Seja a'_i e r'_i o valor das variáveis `a` e `r` e `x` respectivamente após a execução (eventual) da condição da linha 4 na i-gésima iteração. Há duas possibilidades:

- a_{i-1} é par. Neste caso o código condicional não é executado e $a'_i = a_{i-1}$, $r'_i = r_{i-1}$.
- a_{i-1} é par. Neste caso o código condicional é executado e $a'_i = a_{i-1} - 1$, $r'_i = r_{i-1} \cdot x_{i-1}$.

Em *ambos os casos* vale $r'_i \cdot {x_{i-1}}^{a'_i} = r_{i-1} \cdot {x_{i-1}}^{a_{i-1}}$. Além disso, em ambos os casos, a'_i é par. Então, vale $r_i \cdot {x_i}^{a_i} = r_i \cdot \left({x_{i-1}}^2\right)^{\lfloor \frac{a'_i}{2} \rfloor} = r_i \cdot {x_{i-1}}^{a'_i}$. Compondo-se as duas relações, vê-se que $r_i \cdot {x_i}^{a_i} = r_{i-1} \cdot {x_{i-1}}^{a_{i-1}}$, ou, em geral, $r_i \cdot {x_i}^{a_i} = {x_0}^{a_0}$. Mas na n-ésima iteração, quando o algoritmo termina, $a_n = 0$ (condição de término. Assim, $r_n = {x_0}^{a_0}$, ou seja, o valor final da variável `r` é x^a.

(c) (0,5 pontos) Determine, em termos do *valor* de a, a ordem do número de iterações em notação *big Oh*.

Resposta: Se $a_i \leq \frac{a_{i-1}}{2}$ então $2^i a_i \leq a_0$. Mas na última iteração n, $a_n = 0$ e consequentemente $a_{n-1} = 1$. Deste modo, $2^{n-1} \leq a_0 \Rightarrow n \leq \log_2 a_0 + 1$. Assim, o número de iterações é $\mathcal{O}(\log a)$.

5. (3,5 pontos) O código a seguir mostra uma implementação em java do algoritmo de ordenação Quicksort para vetores de inteiros entre os índices a e b:

```
 1   public static void quicksort(int s[], int a, int b) {
 2        if (a>=b) return;
 3        int p=s[b];     // pivot
 4        int l=a;
 5        int r=b-1;
 6        while (l<=r) {
 7             while ((l<=r)&&(s[l]<=p)) l++;
 8             while ((l<=r)&&(s[r]>=p)) r--;
 9                  if (l<r) {
10                        int temp=s[l];
11                        s[l]=s[r];
12                        s[r]=temp;
13                  }
14        }
15        s[b]=s[l];
16        s[l]=p;
17        quicksort(s, a, (l-1));
18        quicksort(s, (l+1), b);
19   }
```

Esta implementação escolhe como pivô o *último* elemento do vetor. O vetor é dividido em dois subvetores, um de a a $l-1$ e outro de $l+1$ a b de modo que no primeiro há somente elementos menores ou iguais ao pivô e no segundo somente elementos maiores ou iguais ao pivô.

(a) (1,0 pontos) Qual a ordem em notação *Big Oh* do nível *máximo* de profundidade da árvore de chamadas do algoritmo (e consequentemente uso de memória de pilha) para uma entrada de tamanho N, considerando o *pior* caso?

Resposta: A quantidade de chamadas subsequente é limitada pelo tamanho do vetor de entrada. Assim, naturalmente, a maior quantidade de chamadas ocorrerá quando um dos sub-vetores pós-divisão for o maior possível. Isso ocorre por exemplo em um vetor pré-ordenado, quando o primeiro sub-vetor tem tamanho $N-1$. Neste caso a profundidade máxima é $\mathcal{O}(N)$.

(b) (0,5 pontos) Em sua tese de doutorado em 1975, Robert Sedgewick propôs uma alteração no algoritmo: A *primeira* chamada recursiva do algoritmo deve ser feita sobre o *menor* dos sub-vetores. Reescreva o algoritmo acima de modo a comparar o tamanho relativo dos subvetores após a divisão e fazer as chamadas recursivas de acordo com o proposto por Sedgewick.

Resposta: O resultado é idêntico à listagem do enunciado até a linha 17:

```
 1    public static void quicksort(int s[], int a, int b) {
 2         if (a>=b) return;
 3         int p=s[b];     // pivot
 4         int l=a;
 5         int r=b-1;
 6         while (l<=r) {
 7              while ((l<=r)&&(s[l]<=p)) l++;
 8              while ((l<=r)&&(s[r]>=p)) r--;
 9                   if (l<r) {
10                          int temp=s[l];
11                          s[l]=s[r];
12                          s[r]=temp;
13                   }
14         }
15         s[b]=s[l];
16         s[l]=p;
17         if(l-1-a > b-l-1) {
18              quicksort(s, a, (l-1));
19              quicksort(s, (l+1), b);
20         } else {
21              quicksort(s, (l+1), b);
22              quicksort(s, a, (l-1));
23         }
24    }
```

Naturalmente, há outros testes possíveis na linha 17, como `l-a > b-l` ou mesmo `a+b < 2*l`.

(c) (1,0 pontos) Reescreva o algoritmo do item (b) de modo a transformar a chamada de cauda (ou seja, a *última* chamada da função) em um laço.

Resposta: Nota-se que é impossível retirar *ambas* as chamadas recursivas (ao menos sem uma estrutura de dados auxiliar).

```
 1    public static void quicksort(int s[], int a, int b) {
 2         while(a<b) {
 3            int p=s[b];     // pivot
 4            int l=a;
 5            int r=b-1;
 6            while (l<=r) {
 7                 while ((l<=r)&&(s[l]<=p)) l++;
```

```
 8                while ((l<=r)&&(s[r]>=p)) r--;
 9                     if (l<r) {
10                            int temp=s[l];
11                            s[l]=s[r];
12                            s[r]=temp;
13                     }
14           }
15           s[b]=s[l];
16           s[l]=p;
17           if(l-a > b-l) {
18              quicksort(s, a, (l-1));
19              a = l+1;
20           } else {
21              quicksort(s, (l+1), b);
22              b = l-1;
23           }
24       }
25   }
```

(d) (1,0 pontos) O objetivo da modificação de Sedgewick é reduzir a profundidade máxima da árvore de chamadas e consequentemente o uso de memória de pilha pelo algoritmo. Qual a ordem em notação *big Oh* da profundidade máxima da árvore de chamadas do algoritmo produzido no item (c) para um vetor de N posições?

Resposta: A profundidade máxima ocorre quando o sub-vetor passado para a chamada recursiva tem o tamanho máximo a cada iteração. Ora, este sub-vetor tem no máximo o tamanho da *metade* do vetor original (visto que ele é menor ou igual ao outro sub-vetor). Finalmente, é possível dividir um vetor em 2 no máximo $log_2 N$ vezes, de modo que a profundidade máxima é de ordem $\mathcal{O}(N)$.

Formulário

Somas de sequências:

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

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

Solução de equações de recorrência pelo *Master Theorem*:

A equação:

$$T(N) = A \cdot T\left(\frac{N}{B}\right) + cN^L,$$

com $T(1) = \mathcal{O}(1)$, tem solução

$$T(N) = \begin{cases} \mathcal{O}\left(N^{\log_B A}\right) & \text{se } A > B^L \\ \mathcal{O}\left(N^L \log N\right) & \text{se } A = B^L \\ \mathcal{O}\left(N^L\right) & \text{se } A < B^L \end{cases}$$

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