AV1 – Atividade de Laboratório

T198 – Construção e Análise de Algoritmos
Prof. Napoleão Nepomuceno
Equipe: Caio Ribeiro e Carlos Huan

1. A atividade deve ser realizada em dupla.
2. Resoluções iguais estão sujeitas à anulação definitiva.
3. O prazo de entrega é até 19:00 da segunda-feira, dia 17 de março de 2025.
4. Cada aluno deve obrigatoriamente saber explicar cada exercício entregue.

Exercício 1

Passo 1: Implementar o seguinte código em Java ou equivalente em outra linguagem de programação. (0%)

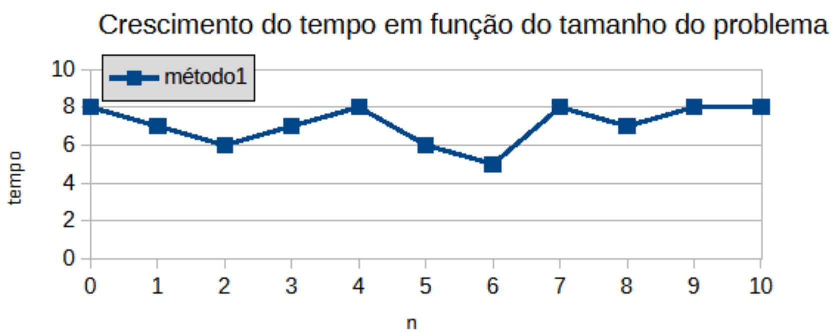
```
import java.util.concurrent.TimeUnit;

public class exercicio1 {
   public static void main(String[] args) {
      System.out.printf("metodo1\n");
      System.out.printf("%10s%20s%10s\n","n", "solucao", "tempo");
      for (int n = 0; n <= 10; n+=1) {
         metodo1(n);
      }
      System.out.printf("metodo2\n");
      System.out.printf("%10s%20s%10s\n","n", "solucao", "tempo");
      for (int n = 0; n <= 10; n+=1) {
         metodo2(n);
      }
      System.out.printf("metodo3\n");
      System.out.printf("%10s%20s%10s\n","n", "solucao", "tempo");
      for (int n = 0; n <= 10; n+=1) {
         metodo3(n);
      }
      System.out.printf("metodo4\n");
      System.out.printf("%10s%20s%10s\n","n", "solucao", "tempo");
      for (int n = 0; n <= 10; n+=1) {
         metodo4(n);
      }
      System.out.printf("metodo5\n");
      System.out.printf("%10s%20s%10s\n","n", "solucao", "tempo");
      for (int n = 0; n <= 10; n+=1) {
         metodo5(n);
      }

   }
```

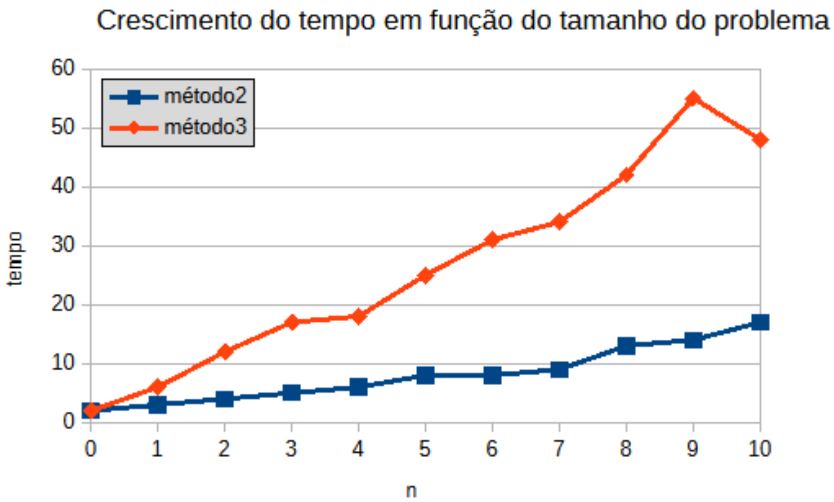
```
static void metodo1 (long n) {
    double inicio = System.currentTimeMillis();
    long valor = 0;
    long termo = n * n * n * n;
    try {
        TimeUnit.MILLISECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    for (long i = 1; i <= 4; i++) {
        valor += termo;
        try {
            TimeUnit.MILLISECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    double fim = System.currentTimeMillis();
    double tempo = fim - inicio;
    System.out.printf("%10d%20d%10.0f\n", n, valor, tempo);
}

static void metodo2 (long n) {
    double inicio = System.currentTimeMillis();
    long valor = 0;
    long termo = 4 * n * n * n;
    try {
        TimeUnit.MILLISECONDS.sleep(10);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    for (long i = 1; i <= n; i++) {
        valor += termo;
        try {
            TimeUnit.MILLISECONDS.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    double fim = System.currentTimeMillis();
    double tempo = fim - inicio;
    System.out.printf("%10d%20d%10.0f\n", n, valor, tempo);
}

static void metodo3 (long n) {
    double inicio = System.currentTimeMillis();
    long valor = 0;
    long termo = n * n * n;
    try {
        TimeUnit.MILLISECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
```

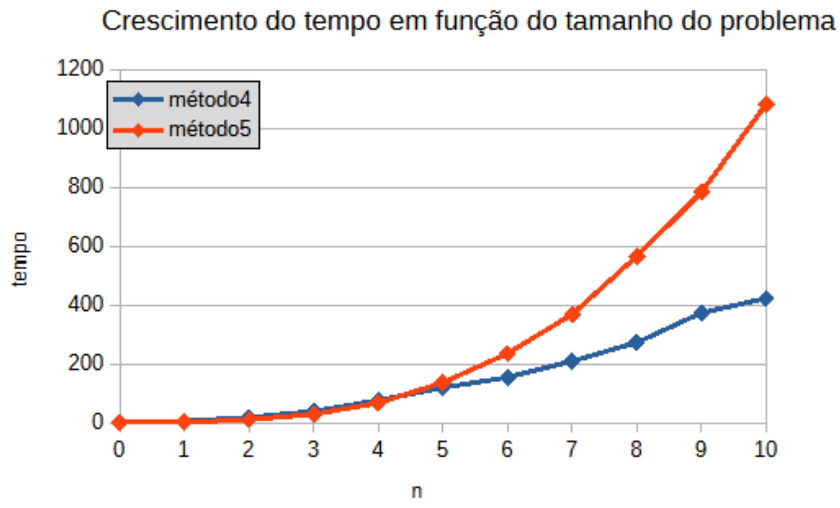
```
    }
    for (long i = 1; i <= 4; i++) {
        for (long j = 1; j <= n; j++) {
            valor += termo;
            try {
                TimeUnit.MILLISECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    double fim = System.currentTimeMillis();
    double tempo = fim - inicio;
    System.out.printf("%10d%20d%10.0f\n", n, valor, tempo);
}

static void metodo4 (long n) {
    double inicio = System.currentTimeMillis();
    long valor = 0;
    long termo = n * n;
    try {
        TimeUnit.MILLISECONDS.sleep(10);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    for (long i = 1; i <= 2 * n; i++) {
        for (long j = 1; j <= 2 * n; j++) {
            valor += termo;
            try {
                TimeUnit.MILLISECONDS.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    double fim = System.currentTimeMillis();
    double tempo = fim - inicio;
    System.out.printf("%10d%20d%10.0f\n", n, valor, tempo);
}

static void metodo5 (long n) {
    double inicio = System.currentTimeMillis();
    long valor = 0;
    long termo = 4 * n;
    try {
        TimeUnit.MILLISECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    for (long i = 1; i <= n; i++) {
        for (long j = 1; j <= n; j++) {
            for (long k = 1; k <= n; k++) {
```

```
                valor += termo;
                try {
                    TimeUnit.MILLISECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    double fim = System.currentTimeMillis();
    double tempo = fim - inicio;
    System.out.printf("%10d%20d%10.0f\n", n, valor, tempo);
  }
}
```

Passo 2: Os diferentes métodos computam o valor de 4n4. Executar o código e preencher o resultado na planilha disponibilizada (aba Exercicio1). Copiar os gráficos neste documento. (0%)







Passo 3: Realizar a análise de complexidade para cada um dos métodos. Desconsiderar na análise as instruções do try catch (utilizadas apenas para simular uma máquina mais lenta) e as de rastreamento do tempo de execução. (20%)

metodo1	**#vezes**
valor = 0	1
termo = n * n * n * n	1
for (i = 1; i ≤ 4; i++)	5
valor += termo	4
print(valor)	1

$T(n) = \Theta(1)$

metodo2	**#vezes**
valor = 0	1
termo = 4 * n * n * n	1
for (i = 1; i ≤ n; i++)	n+1
valor += termo	n
print(valor)	1

$T(n) = \Theta(n)$

metodo3	**#vezes**
valor = 0	1
termo = n * n * n	1
for (i = 1; i ≤ 4; i++)	5
for (j = 1; j ≤ n; j++)	n+1 + n+1 + n+1 + n+1 = 4 * (n+1)
valor += termo	n + n + n + n = 4 * n
print(valor)	1

$T(n) = \Theta(n)$

metodo4	#vezes
valor = 0	1
termo = n * n	1
for (i = 1; i ≤ 2 * n; i++)	2n+1
for (j = 1; j ≤ 2 * n; j++)	(2n + 1) + (2n + 1) + ... + (2n + 1) = $\Theta(n^2)$
valor += termo	2n + 2n + ... + 2n = $\Theta(n^2)$
print(valor)	1

$T(n) = \Theta(n^2)$

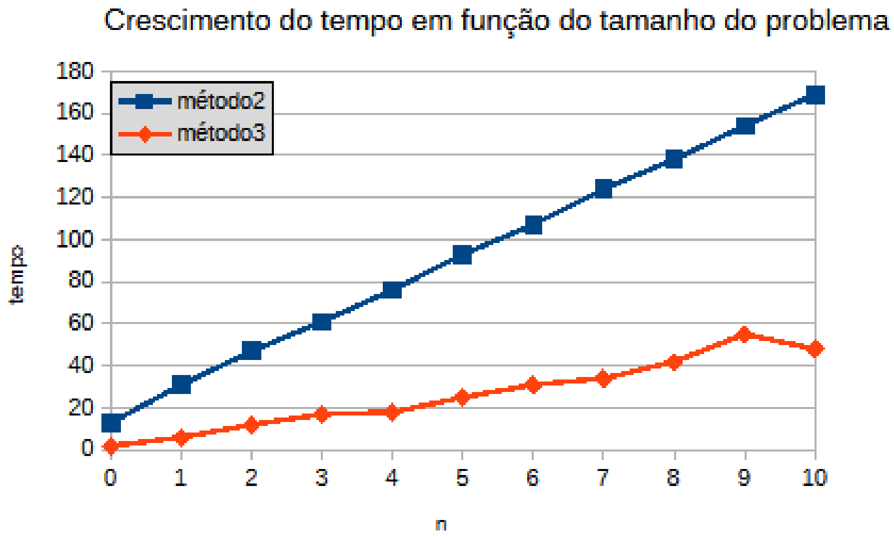
metodo5	#vezes
valor = 0	1
termo = 4 * n	1
for (i = 1; i ≤ n; i++)	n + 1
for (j = 1; j ≤ n; j++)	(n + 1) + (n + 1) + ... + (n + 1) = $n(n+1)$
for (k = 1; k ≤ n; k++)	$(n+1)(n(n+1)-n) = \Theta(n^3)$
valor += termo	$n(n(n+1)-n) = \Theta(n^3)$
print(valor)	1

$T(n) = \Theta(n^3)$

Passo 4: Em seu experimento, qual método tem melhor tempo de execução: metodo2 ou metodo3? Para simular a execução do metodo2 em uma máquina 10 vezes mais lenta, modificar a instrução de sleep para TimeUnit.MILLISECONDS.sleep(10) apenas para este método, executar novamente o programa, alterar a planilha e copiar o gráfico respectivo neste documento. Neste novo experimento, qual método tem o melhor tempo de execução para n suficientemente grande: metodo2 ou metodo3? Explicar a que se deve este comportamento. (2%)

metodo2 foi mais rápido no primeiro teste.

Após modificar a instrução sleep, obtivemos o seguinte gráfico:

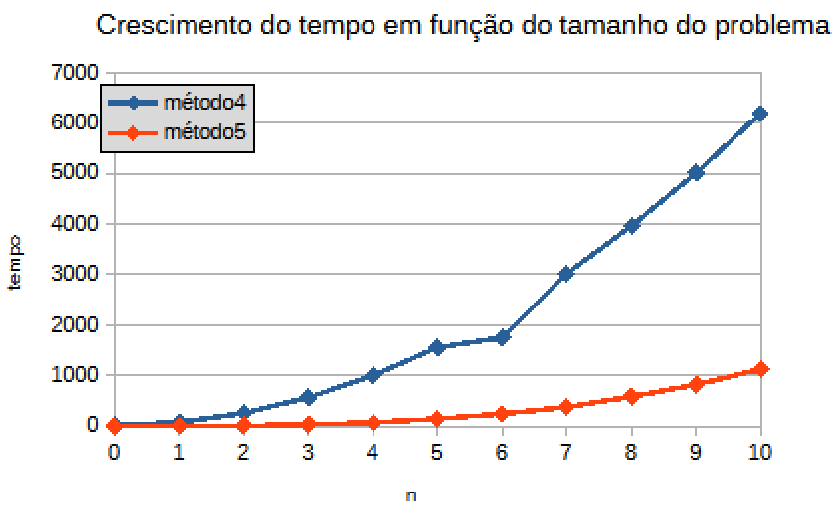


Nesse caso, **metodo3** foi bem mais rápido que o **metodo2**.
Apesar dos dois algoritmos serem de classe de complexidade linear
$\Theta(n)$, o tempo real de execução foi afetado pela simulação de um estado de maquina mais lento.

Passo 5: Em seu experimento, qual método tem melhor tempo de execução: metodo4 ou metodo5? Para simular a execução do metodo4 em uma máquina 10 vezes mais lenta, modificar a instrução de sleep para TimeUnit.MILLISECONDS.sleep(10) apenas para este método, executar novamente o programa, alterar a planilha e copiar o gráfico respectivo neste documento. Neste novo experimento, qual método tem o melhor tempo de execução para n suficientemente grande: metodo4 ou metodo5? Explicar este comportamento. (3%)

Os dois métodos tiveram tempo de execução parecido ate n=5, quando **metodo4** passa a ter tempo melhor.

Simulando uma máquina mais lenta para **metodo4**, obtivemos o seguinte gráfico:



Nesta simulação, **metodo5** foi bem mais rápido.

Semelhante ao passo 4, a simulação de uma máquina 10 vezes mais lenta altera consideravelmente o tempo de execução do método. A complexidade não muda: o tempo continua "crescendo" no método 4 de forma quadrática e de forma cúbica no método 5. Mas naturalmente o estado da máquina influencia no tempo real de execução de cada um.

Passo 6: Indicar o nome dos integrantes da equipe que participaram efetivamente na resolução deste exercício. (0%)

Caio Ribeiro; Carlos Huan

Exercício 2

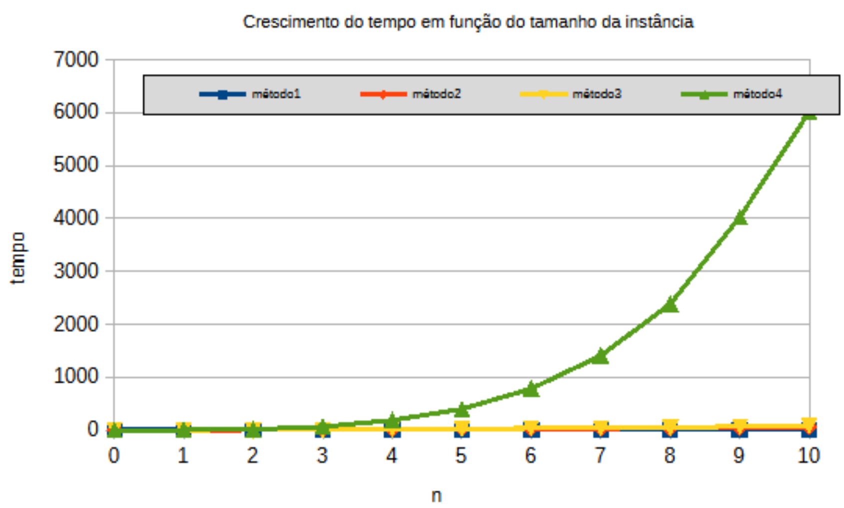
Passo 1: Implementar o seguinte código em Java ou equivalente em outra linguagem de programação. (0%)

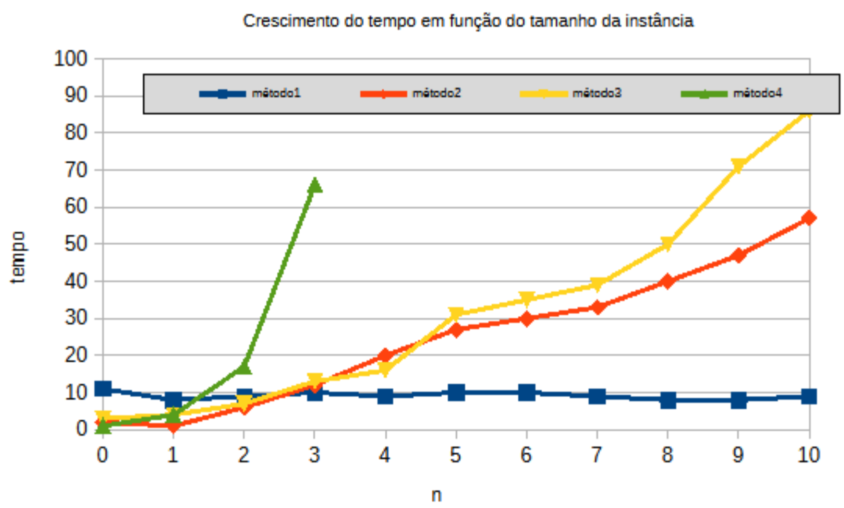
```
import java.util.concurrent.TimeUnit;

public class Exercicio3 {
    public static void main(String[] args) {
        double inicio1, fim1, tempo1;
        double inicio2, fim2, tempo2;
        double inicio3, fim3, tempo3;
        double inicio4, fim4, tempo4;

        System.out.printf("%5s%10s%10s%10s%10s\n","n", "tempo1", "tempo2", "tempo3", "tempo4");
        System.out.println("--------------------------------------------");
        for (int n = 0; n <= 10; n+=1) {
            inicio1 = System.currentTimeMillis();
            metodo1(n);
            fim1 = System.currentTimeMillis();
            tempo1 = fim1 - inicio1;
            inicio2 = System.currentTimeMillis();
            metodo2(n);
            fim2 = System.currentTimeMillis();
            tempo2 = fim2 - inicio2;
            inicio3 = System.currentTimeMillis();
            metodo3(n);
            fim3 = System.currentTimeMillis();
            tempo3 = fim3 - inicio3;
            inicio4 = System.currentTimeMillis();
            metodo4(n);
            fim4 = System.currentTimeMillis();
            tempo4 = fim4 - inicio4;
            System.out.printf("%5d%10.0f%10.0f%10.0f%10.0f\n", n, tempo1, tempo2, tempo3, tempo4);
        }

    }

    static void metodo1 (long n) {
        long valor = 0;
        try {
```

```
            TimeUnit.MILLISECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (long i = 10; i < 12; i++) {
            for (long j = 4; j <= 6; j++) {
                valor += 1;
                try {
                    TimeUnit.MILLISECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    static void metodo2 (long n) {
        long valor = 0;
        try {
            TimeUnit.MILLISECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (long i = 1; i < n; i++) {
            for (long j = 1; j <= 5; j++) {
                valor += 1;
                try {
                    TimeUnit.MILLISECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    static void metodo3 (long n) {
        long valor = 0;
        try {
            TimeUnit.MILLISECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (long i = 0; i <= n; i++) {
            for (long j = 1; j <= n – i + 1; j++) {
                valor += 1;
                try {
                    TimeUnit.MILLISECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
```

```
    }

    static void metodo4 (long n) {
        long valor = 0;
        try {
            TimeUnit.MILLISECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (long i = 1; i <= n * n; i++) {
            for (long j = 0; j <= i; j++) {
                valor += 1;
                try {
                    TimeUnit.MILLISECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

}
```

Passo 2: Executar o código e preencher o resultado na planilha disponibilizada (aba Exercicio2). Copiar os gráficos neste documento. (0%)





Passo 3: Realizar a análise de complexidade para cada um dos métodos. Desconsiderar na análise as instruções do try catch (utilizadas apenas para simular uma máquina mais lenta) e as de rastreamento do tempo de execução. (16%)

metodo1	**#vezes**
valor = 0	1
for (i = 10; i < 12; i++)	3
for (j = 4; j ≤ 6; j++)	4 + 4 + 4 = 12
valor += 1	3 + 3 + 3 = 9

$T(n) = \Theta(1)$

metodo2	**#vezes**
valor = 0	1
for (i = 1; i < n; i++)	n
for (j = 1; j ≤ 5; j++)	6 + 6 + ... + 6 = $6(n+1) = \Theta(n)$
valor += 1	5 + 5 + ... + 5 = $5(n+1) = \Theta(n)$

$T(n) = \Theta(n)$

metodo3	**#vezes**
valor = 0	1
for (i = 0; i ≤ n; i++)	n + 2
for (j = 1; j ≤ n - 1 + 1; j++)	(n+2) + (n+1) + ... + 2 = $[(n+4)(n+1)] \div 2$
valor += 1	(n+1) + n + ... + 1 = $[(n+2)n] \div 2$

$T(n) = \Theta(n^2)$

metodo4	**#vezes**
valor = 0	1
for (i = 1; i ≤ n * n; i++)	$n^2 + 1$

for (j = 0; j ≤ i; j++)

valor += 1

3 + 4 + ... + $n^2 + 2 = \Theta(n^4)$

$\Theta(n^4)$

$T(n) = \Theta(n^4)$

Passo 4: Observando os gráficos obtidos e considerando as análises de complexidade assintótica: (1) indicar qual é o método assintoticamente mais eficiente; (2) indicar qual é o método assintoticamente menos eficiente; e (3) indicar a partir de que ponto o método assintoticamente mais eficiente passou a ser efetivamente mais rápido que os demais métodos no experimento realizado. (4%)

O método assintoticamente **mais** eficiente é **metodo1** pois tem complexidade assintótica constante.
O método assintoticamente
menos eficiente é **metodo4** pois tem complexidade assintótica $\Theta(n^4)$

Passo 5: Indicar o nome dos integrantes da equipe que participaram efetivamente na resolução deste exercício. (0%)

Caio Ribeiro; Carlos Huan

Exercício 3

Passo 1: Implementar o seguinte código em Java ou equivalente em outra linguagem de programação. (0%)

```
import java.util.Random;
import java.util.concurrent.TimeUnit;

public class Exercicio4{
  public static void main(String[] args) {
    int n = 1000;
    int[] A;
    A = criaVetorAleatorio(n);
    double inicio, fim, tempo;
    inicio = System.currentTimeMillis();
    metodo(A, n);
    fim = System.currentTimeMillis();
    tempo = fim - inicio;
    System.out.printf("Tempo: %1.0f", tempo);
  }

  static double metodo (int[] vetor, int n) {
    double v = 1;
    for (int i = 0; i < n; i++) {
      try {
        TimeUnit.MILLISECONDS.sleep(1);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
```

```
            v = v * vetor[i];
            if (v == 0) {
                return 0;
            }
        }
        return v;
    }

    static int[] criaVetorAleatorio (int n) {
        Random randomGenerator = new Random();
        int[] A = new int[n];
        for (int i = 0; i < n; i++) {
            A[i] = randomGenerator.nextInt(100);
        }
        return A;
    }
}
```

Passo 2: Dado um vetor, o que exatamente a função metodo está computando matematicamente? (2%)

O produto entre todos os elementos do vetor $(v[0] * v[1] * \ldots * v[n-1])$.

Passo 3: Executar o código 10 vezes e copiar a saída de cada execução do programa aqui abaixo. Visto que o tamanho da instância não se modifica, o que justifica a grande variação do tempo de uma execução para outra? (2%)

Execução	Tempo
#1	66
#2	38
#3	66
#4	49
#5	5
#6	54
#7	172
#8	229
#9	229
#10	89

A variação no tempo de execução da função *metodo* ocorre devido à sua condição de parada, que depende da presença de um ou mais valores "0" no vetor. Assim que um "0" é encontrado, a operação $v = v * v[i]$ resulta em $v = 0$, fazendo com que o método retorne imediatamente, encerrando sua execução.

Dessa forma, a posição do primeiro "0" no vetor influencia diretamente o tempo de execução. Se o primeiro elemento $v[0]$ for "0" , a execução será interrompida já na primeira iteração do *for*. Por outro lado, se não houver nenhum "0" no vetor ou se o único "0" estiver na última posição, o laço percorrerá todos os elementos, resultando em um tempo de execução significativamente maior.

Passo 4: Realizar a análise de complexidade de melhor e pior casos para o método. Obs.: Desconsiderar na análise as instruções do try catch (utilizadas apenas para simular uma máquina mais lenta). (4%)

Melhor caso **#vezes**

v = 1	1
for (i = 0; i < n; i++)	1
v = v * vetor[i]	1
if (v == 0)	1
return 0	1
return v	0

$T(n) = \Theta(1)$

Pior caso	**#vezes**
v = 1	1
for (i = 0; i < n; i++)	n + 1
v = v * vetor[i]	n
if (v == 0)	0
return 0	0
return v	1

$T(n) = \Theta(n)$

Passo 5: Se o vetor A, em vez de 1000 elementos, tivesse 1.000.000 elementos, a complexidade do algoritmo aumentaria? Justificar. (2%)

A complexidade do algoritmo não muda com o aumento do tamanho do vetor, pois sua complexidade depende da estrutura do código, e não do tamanho da instancia.

Se o vetor passar de 1.000 para 1.000.000 elementos, a execução levará mais tempo, mas a ordem de crescimento continua a mesma. A complexidade não se altera porque o número de operações ainda cresce de forma constante em situações de melhor caso, e de forma linear em situações de pior caso.

Passo 6: Indicar o nome dos integrantes da equipe que participaram efetivamente na resolução deste exercício. (0%)

Caio Ribeiro; Carlos Huan

Exercício 4

Passo 1: Considerar o seguinte código em Java ou equivalente em outra linguagem de programação. (0%)

```
import java.util.Random;
import java.util.concurrent.TimeUnit;

public class Exercicio5 {
   public static void main(String[] args) {

      int[] A;
```

```
        double inicio1, fim1, tempo1;
        double inicio2, fim2, tempo2;

        System.out.printf("%5s%10s%10s%10s%10s\n","n", "soma1", "tempo1", "soma2", "tempo2");
        System.out.println("---------------------------------------------");
        for (int n = 1; n <= 50; n++) {
            A = criaVetorAleatorio(n);
            inicio1 = System.currentTimeMillis();
            int soma1 = soma1(A, n);
            fim1 = System.currentTimeMillis();
            tempo1 = fim1 - inicio1;
            inicio2 = System.currentTimeMillis();
            int soma2 = soma2(A, 0, n-1);
            fim2 = System.currentTimeMillis();
            tempo2 = fim2 - inicio2;
            System.out.printf("%5d%10d%10.0f%10d%10.0f\n", n, soma1, tempo1, soma2, tempo2);
        }
    }

    static int soma1 (int[] vetor, int n) {
        int total = 0;
        for (int i = 0; i < n; i++) {
            try {
                TimeUnit.MILLISECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            total = total + vetor[i];
        }
        return total;
    }

    static int soma2 (int[] vetor, int i, int f) {
        if (i == f) {
            return vetor[i];
        } else {
            int m = (i+f) / 2;
            try {
                TimeUnit.MILLISECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return soma2(vetor, i, m) + soma2(vetor, m+1, f);
        }
    }
    static int[] criaVetorAleatorio (int n) {
        Random randomGenerator = new Random();
        int[] A = new int[n];
        for (int i = 0; i < n; i++) {
            A[i] = randomGenerator.nextInt(100*n);
        }
        return A;
```

```
    }

}
```

Passo 2: Realizar a análise de complexidade da função soma1. (1%)

soma1	#vezes
total = 0	1
for (i = 0; i < n; i++)	n + 1
total = total + vetor[i]	n
return total	1

$T(n) = \Theta(n)$

Passo 3: Montar a equação de recorrência para a função soma2. (2%)

soma2	#vezes
if (i == f)	
return vetor[i]	
else	
int m = (i+f) / 2	$\Theta(1)$
return soma2(vetor, i, m) + soma2(vetor, m+1, f)	$T(n/2) + T(n/2)$

$T(n) = 2T(n/2) + \Theta(1)$

Passo 4: Resolver a equação de recorrência pelo teorema mestre. (2%)

Pelo teorema mestre, comparamos $n^{log_b a}$ vs $f(n)$ dada uma equação de recorrência do tipo $T(n) = aT(n/b) + f(n)$.

Para $T(n) = 2T(n/2) + \Theta(1)$ temos $a = 2$ e $b = 2$.

Portanto,

(1) $\quad n^{log_2 2} = n^1 = \Theta(n)$.

(2) $\quad f(n) = \Theta(1)$

Temos que $\Theta(n) > \Theta(1)$,

$\therefore \quad T(n) = \Theta(n)$

Passo 5: Indicar o nome dos integrantes da equipe que participaram efetivamente na resolução deste exercício. (0%)

Caio Ribeiro; Carlos Huan

Exercício 5

Passo 1: Considerar o seguinte código em Java ou equivalente em outra linguagem de programação. (0%)

```
import java.util.concurrent.TimeUnit;

public class Exercicio6 {
   public static void main(String[] args) {
      double inicio1, fim1, tempo1;
      double inicio2, fim2, tempo2;
      System.out.printf("%5s%20s%10s%20s%10s\n","n", "pot1", "tempo1", "pot2", "tempo2");
      System.out.println("-----------------------------------------------------------------");
      for (int n = 1; n <= 30; n++) {
         inicio1 = System.currentTimeMillis();
         int pot1 = potencia1(2, n);
         fim1 = System.currentTimeMillis();
         tempo1 = fim1 - inicio1;
         inicio2 = System.currentTimeMillis();
         int pot2 = potencia3(2, n);
         fim2 = System.currentTimeMillis();
         tempo2 = fim2 - inicio2;
         System.out.printf("%5d%20d%10.0f%20d%10.0f\n", n, pot1, tempo1, pot2, tempo2);
      }
   }

   static int potencia1 (int a, int n) {
      int total = 1;
      for (int i = 1; i <= n; i++) {
         try {
            TimeUnit.MILLISECONDS.sleep(1);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
         total = total * a;
      }
      return total;
   }

   static int potencia2 (int a, int n) {
      if (n == 0) {
         return 1;
      } else {
         try {
            TimeUnit.MILLISECONDS.sleep(1);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
         int aux = potencia2 (a, n/2);
         if (n % 2 == 0) {
```

```
            return aux * aux;
        } else {
            return aux * aux * a;
        }
    }
  }
}
```

Passo 2: Realizar a análise de complexidade da função potencia1. (1%)

potencia1	#vezes
total = 1	1
for (i = 1; i ≤ n; i++)	n + 1
total = total * a	n
return total	1

$T(n) = \Theta(n)$

Passo 3: Montar a equação de recorrência para a função potencia2. (2%)

Temos que $T(n) = aT(n/b) + f(n)$.

$f(n) = \Theta(1)$

Sendo $a = 1$ e $b = 2$,

$T(n) = T(n/2) + \Theta(1)$.

Passo 4: Resolver a equação de recorrência pelo teorema mestre. (2%)

$n^{log_b a} = n^{log_2 1} = n^0 = 1 = \Theta(1)$

$f(n) = \Theta(1)$

Caímos no caso 2 do Teorema Mestre, em que $n^{log_b a} = f(n)$.

Nesse caso,

$T(n) = \Theta(n^{log_b a} \times lg\ n)$.

$\therefore \quad T(n) = \Theta(lg\ n)$.

Passo 5: Indicar o nome dos integrantes da equipe que participaram efetivamente na resolução deste exercício. (0%)

Caio Ribeiro; Carlos Huan

Exercício 6 - Exercícios para prova!

Q1. Suponha que dois algoritmos, A e B, resolvem um mesmo problema. Assuma ainda que o tamanho das instâncias do problema é dado por um parâmetro n. Para cada item abaixo, assumindo-se n suficientemente grande, indique se A é mais rápido que B para toda e qualquer instância, se B é mais rápido que A para toda e qualquer instância, ou se não podemos inferir qual dos dois algoritmos é mais rápido. Só serão pontuados os itens devidamente justificados. (10%)

(a) O algoritmo A consome tempo $O(n^2)$ e o B consome tempo $\Omega(n^3)$.

R: A é mais rápido que B para toda e qualquer instância.

Justificativa: $O(n^2)$ indica que A tem complexidade assintótica não mais que quadrática. $\Omega(n^3)$ indica que B tem complexidade assintótica não menos que cubica.

(b) O algoritmo A consome tempo $\Theta(n^2)$ e o B consome tempo $O(n^3)$.

R: Não podemos inferir qual dos dois algoritmos é mais rápido.

Justificativa: $O(n^3)$ indica que B tem complexidade não maior que cubica. $\Theta(n^2)$ indica que A tem complexidade assintótica quadrática. Não podemos inferir qual algoritmo é mais rápido pois temos apenas um limite assintótico superior de B. Caso B fosse, por exemplo, $\Theta(1)$, $O(n^3)$ ainda seria verdade, mas nesse caso B seria mais rápido que A.

(c) O algoritmo A em instâncias de pior caso consome tempo $O(n^3)$ e o B em instâncias de pior caso consome tempo $O(n^2)$.

R: Não podemos inferir qual dos dois algoritmos é mais rápido.

Justificativa: Como estamos comparando os piores casos dos dois algoritmos, não podemos afirmar que $O(n^3)$ é um limite justo para A, portanto não podemos inferir que B é mais rápido.

(d) O algoritmo A em instâncias de pior caso consome tempo $O(n^2)$ e o B em instâncias de pior caso consome tempo $\Omega(n)$.

R: Não podemos inferir qual dos dois algoritmos é mais rápido.

Justificativa: Não sabemos se os limites assintóticos de pior caso são justos para A e B e também desconhecemos os limites assintóticos em situações de melhor caso.

(e) O algoritmo A em instâncias de pior caso consome tempo $O(n^2)$ e o B em instâncias de melhor caso consome tempo $\Omega(n^3)$.

R: A é mais rápido que B para toda e qualquer instância.

Justificativa: Sabemos que B tem complexidade não menor que cubica em instâncias de melhor caso. No pior caso, A não passa de quadrática.

Q2. Aplique o método mestre para resolver as seguintes recorrências. (10%)

(a) $T(n) = 9T(n/3) + n$

$a = 9;\, b = 3;\, f(n) = n = \Theta(n)$

$n^{log_3 9} = n^2 = \Theta(n^2)$

$\Theta(n^2) > \Theta(n)$

$T(n) = \Theta(n^2)$

(b) $T(n) = T(n/8) + 1$

$a = 1;\, b = 8;\, f(n) = 1$

$n^{log_8 1} = n^0 = 1$

$T(n) = \Theta(lg\ n)$

(c) $T(n) = 2T(n/2) + n^2$

$a = 2;\ b = 2;\ f(n) = n^2 = \Theta(n^2)$

$n^{log_2 2} = n^1 = n = \Theta(n)$

$\Theta(n) < \Theta(n^2)$

$T(n) = \Theta(n^2)$

(d) $T(n) = 8T(n/2) + n^2$

$a = 8;\ b = 2;\ f(n) = \Theta(n^2)$

$n^{log_2 8} = n^3 = \Theta(n^3)$

$\Theta(n^3) > \Theta(n^2)$

$T(n) = \Theta(n^3)$

(e) $T(n) = 16T(n/4) + n^2$

$a = 16;\ b = 4;\ f(n) = n^2$

$n^{log_4 16} = n^2$

$T(n) = \Theta(n^2\ lg\ n)$

Q3. Dada o método abaixo, encontre um limite assintótico, utilizando notação Θ, para determinar sua complexidade. (5%)

funcao	#vezes
sum = 0	1
for (i = 0; i ≤ n; i++)	n + 2
for (j = i; j < n*n; j++)	(n2 + 1) + n2 + ... + [(n2 + 1 - n)] = $\Theta(n^3)$
sum = sum + 1;	$\Theta(n^3)$
return sum	1

$T(n) = \Theta(n^3)$

Q4. Seja um vetor A de n elementos inteiros. É possível determinar o produto dos elementos do vetor em Θ(n) percorrendo-se os elementos do vetor de forma iterativa. Alternativamente, pode-se utilizar um método de divisão-e-conquista. Faça uma função recursiva para determinar o produto dos elementos do vetor. O algoritmo deve recursivamente dividir o vetor ao meio até se chegar a um caso trivial. Determine e resolva a equação de recorrência para o seu algoritmo. O algoritmo recursivo é mais eficiente do que o algoritmo iterativo? (10%)

```
static int produtoVetor(int[] vetor, int l, int r) {
  if (l == r)
    return vetor[l];

  int m = (l + r) / 2;
  int pl = produtoVetor(vetor, l, m);
  int pr = produtoVetor(vetor, m + 1, r);
```

```
    return pl * pr;
  }
```

Equação de recorrência: $T(n) = 2T(n/2) + \Theta(1)$

Resolvendo via teorema mestre:

$a = 2; b = 2; f(n) = \Theta(1)$

$n^{log_2 2} = n^0 = 1 = \Theta(1)$

Caímos no caso 2,

$\therefore \quad T(n) = \Theta(lg\ n)$

Indicar o nome dos integrantes da equipe que participaram efetivamente na resolução deste exercício. (0%)

Caio Ribeiro; Carlos Huan