Algoritmos e Estruturas de Dados II

Exercícios de Revisão para a 1ª Prova

Questões adaptadas do material da Prof.ª Josiane Rezende
Resoluções elaboradas pelo aluno Luca Ferrari Azalim

---

1. (Algoritmos e Estruturas de Dados, Complexidade de Algoritmos, Analista de Sistemas Pleno, Processos, Petrobrás, CESGRANRIO). A respeito de funções e algoritmos, assinale a afirmativa correta.

a) O limite inferior de um algoritmo (O) é utilizado para a análise do pior caso de sua execução.

b) Uma função f(n) domina assintoticamente g(n), se existem duas constantes positivas c e n0, tais que, para n = n0, temos que |g(n)| = c|f(n)|.

c) A função f(5 * $\log_2 n$) é Θ(n).

d) A função f($5n^3 + 2n^2$) é O(n).

e) Se uma função g(n) é o limite superior justo de outra função f(n), então f(n) é O(g(n)) e g(n) é O(f(n)).

---

2. (Sistemas de Informação, Complexidade de algoritmos, Analista de Sistemas, TJ SP, VUNESP). Considerando o conceito de Complexidade de Algoritmos, representado por O(função), assinale a alternativa que apresenta, de forma crescente, as complexidades de algoritmos.

a) $O(2^n)$; $O(n^3)$; $O(n^2)$; $O(\log_2 n)$; $O(n*\log_2 n)$.
b) $O(n^2)$; $O(n^3)$; $O(2^n)$; $O(\log_2 n)$; $O(n*\log_2 n)$.
c) $O(n^3)$; $O(n^2)$; $O(2^n)$; $O(n*\log_2 n)$; $O(\log_2 n)$.
d) $O(\log_2 n)$; $O(n*\log_2 n)$; $O(n^2)$; $O(n^3)$; $O(2^n)$.
e) $O(n*\log_2 n)$; $O(\log_2 n)$; $O(2^n)$; $O(n^3)$; $O(n^2)$.

---

3. (Algoritmos e Estrutura de Dados, Complexidade de algoritmos, Técnico Judiciário em Tecnologia da Informação, TRT 19ª Região, FCC). Considere os seguintes algoritmos e suas complexidades na notação Big O:

Algoritmo A: O(log n)
Algoritmo B: $O(n^2)$
Algoritmo C: O(n * log n)

Considerando-se o pior caso de execução desses algoritmos, é correto afirmar que o algoritmo

a) A é o menos eficiente.
b) C é o menos eficiente.
c) A não é o mais eficiente nem o menos eficiente.
d) B é o menos eficiente.
e) C é o mais eficiente.

---

4. Dois vetores ordenados, contendo, cada um deles, N números inteiros, precisam ser unidos em outro vetor maior, que conterá os 2N números, que também serão armazenados de forma ordenada. A complexidade de tempo de melhor caso desse processo será, então,

a) O(1), pois é necessário fazer apenas uma cópia simples de cada um dos elementos originais.
b) O(log N), pois usa-se a busca binária para determinar qual será o próximo elemento copiado para o vetor de destino.
c) O(N), pois precisa-se fazer uma cópia de cada um dos elementos originais, o que implica uma varredura completa de cada vetor de origem.
d) O(N * log N), pois é necessário fazer uma busca de cada elemento para depois inseri-lo no vetor de destino.
e) O(N^2), pois, como há dois vetores, precisa-se fazer dois laços de forma aninhada (um dentro do outro), gerando uma multiplicação das quantidades de elementos.

---

5. Depois de pensar sobre determinado problema, João fez um rascunho de uma função, produzindo o algoritmo em pseudocódigo abaixo:

```
função cálculo(n)
    K=0
    se (n > 1000) então
        para i de 1 até n faça
            para j de 1 até n faça
                k = (k * k) + j
            fim para
        fim para
    senão
        para j de 1 até n faça
            k = (k*j) +2
```

```
        fim para
    fim se
    retorne k
```

É correto afirmar que o algoritmo é (assinale todas as opções corretas):

a) $O(n)$
b) $O(n^2)$
c) $O(n^3)$
d) $O(\log n)$
e) $O(n * \log n)$

---

6. (CESPE / CEBRASPE - 2022 - DPE-RO - Analista da Defensoria Pública - Programação). A complexidade do algoritmo abaixo é (assinale todas as alternativas verdadeiras):

```
funcao algoritmo(n)
início
    se n = 0 então
        retorne 0
    senão
        se n = 1 então
            retorne 1
        senão
            penultimo = 0
            ultimo = 1
            para i = 2 até n faça
                atual = penultimo + ultimo
                último = atual
            fim para
            retorne atual
        fim se
    fim se
fim
```

a) $O(2^n)$
b) $O(n^2)$
c) $O(n)$
d) $O(\log(n))$
e) $O(n*\log(n))$

Resolução:

A ordem de complexidade do algoritmo é $O(n)$ e, consequentemente, O de qualquer outra função maior do que $f(n) = n$.

---

7. (COMPERVE - 2016 - UFRN - Analista de Tecnologia da Informação) Analise o algoritmo a seguir:

```
função algo(n)
    i <- 1
    j <- 0
    para k de 1 até n faça
        x <- i + j
        i <- j
        j <- x
    retorne j
```

Em relação ao algoritmo exposto, é correto afirmar que:

a) o algoritmo é $\Theta(2^n)$.
b) o algoritmo é $\Theta(n)$.
c) o algoritmo é $\Theta(n^2)$.
d) o algoritmo é $\Theta(n^3)$.

---

8. (ACEP - 2019 - Prefeitura de Aracati - CE - Analista de Sistemas) Considere o trecho de pseudocódigo abaixo:

```
para i <- 0 até n passo 1 faça
    para j <- 0 até n passo 1 faça
    início
        p[i][j] <- 0;
        k <- 0;
        enquanto k < n faça
        início
            p[i][j] <- p[i][j] + a[i][k] * b[k][j];
            k <- k + 1;
        fim
    fim
```

A ordem de complexidade do trecho em questão é:

a) O(n)
b) $O(n^2)$
c) O(n*log n)

d) $O(n^3)$

---

9. (IF-SP - 2019 – Informática) A notação O é amplamente utilizada como ferramenta de análise para calcular a complexidade computacional de um algoritmo caracterizando seu tempo de execução e limites espaciais em função de um parâmetro n. Considere o código de um método em Java contendo o algoritmo a seguir:

```
public static boolean saoDisjuntos(int[] a, int[] b) {
    for(int i = 0; i < a.length; i++)
        for(int j = 0; j < b.length; j++)
            if(a[i] == b[j]) return false;
    return true;
}
```

Se cada um dos arranjos a e b do algoritmo acima tem tamanho n, então, o pior caso para o tempo de execução desse método é (assinale todas as alternativas corretas):

a) $O(2^n)$
b) $O(n)$
c) $O(n^2)$
d) $O(\log n)$

---

10. (CCV-UFC - 2013 - UFC - Analista de Tecnologia da Informação - Arquitetura e Desenvolvimento de Software) O algoritmo a seguir, descrito em pseudocódigo, pode ser utilizado para ordenar um vetor A[0..n].

```
Algoritmo(A[], n)

    var i, j, elemento;

    PARA j <- 1 ATÉ n FAÇA
        elemento <- A[j]
    i <- j - 1

        ENQUANTO ((i >= 0) E (A[i] > elemento)) FAÇA
            A[i+1] <- A[i]
            A[i] <- elemento
            i <- i - 1
        FIM_ENQUANTO
```

```
    FIM_PARA

FIM
```

No pior caso, a complexidade deste algoritmo é:

a) O(n²)
b) O(1)
c) O(n)
d) O(log n)
e) O(n*log n)

---

11. Dada a sequência de números: 3 4 9 2 5 8 2 1 7 4 6 2 9 8 5 1, ordene-a em ordem não decrescente segundo os seguintes algoritmos, apresentando a sequência obtida após cada passo do algoritmo:

a) MergeSort
b) QuickSort
c) HeapSort

Resolução:

a) Mergesort

Vetor original:

[3, 4, 9, 2, 5, 8, 2, 1, 7, 4, 6, 2, 9, 8, 5, 1]

Passo 1: dividir o vetor ao meio.

[3, 4, 9, 2, 5, 8, 2, 1] [7, 4, 6, 2, 9, 8, 5, 1]

Passo 2: dividir cada metade ao meio novamente.

[3, 4, 9, 2] [5, 8, 2, 1] [7, 4, 6, 2] [9, 8, 5, 1]

Passo 3: continuar dividindo até que todos os vetores possuam apenas um elemento.

[3, 4] [9, 2] [5, 8] [2, 1] [7, 4] [6, 2] [9, 8] [5, 1]

[3] [4] [9] [2] [5] [8] [2] [1] [7] [4] [6] [2] [9] [8] [5] [1]

Passo 4: mesclar os vetores em novos vetores ordenados.

[3, 4] [2, 9] [5, 8] [1, 2] [4, 7] [2, 6] [8, 9] [1, 5]

Passo 5: continuar mesclando os vetores ordenados em novos vetores ordenados.

[2, 3, 4, 9] [1, 2, 5, 8] [2, 4, 6, 7] [1, 5, 8, 9]

[1, 2, 2, 3, 4, 5, 8, 9] [1, 2, 4, 5, 6, 7, 8, 9]

Vetor ordenado:

[1, 1, 2, 2, 2, 3, 4, 4, 5, 5, 6, 7, 8, 8, 9, 9]

b) Heapsort

c) Quicksort

---

12. Dados três vetores ordenados, implemente uma função que intercale e retorne o vetor resultante ordenado (sugestão: baseie-se no algoritmo do mergesort original). Qual a complexidade desse algoritmo?

Resolução:

Complexidade: O(n)

```java
public static int[] combineSortedVectors(int[] v1, int[] v2, int[] v3) {

    int[] v = new int[v1.length + v2.length + v3.length];
    int p = 0, p1 = 0, p2 = 0, p3 = 0;

    while (p1 < v1.length && p2 < v2.length && p3 < v3.length) {

        if (v1[p1] <= v2[p2] && v1[p1] <= v3[p3]) {
            v[p++] = v1[p1++];
        } else if (v2[p2] <= v3[p3] && v2[p2] <= v3[p3]) {
            v[p++] = v2[p2++];
        } else {
            v[p++] = v3[p3++];
        }

    }

    while (p1 < v1.length) {
        v[p++] = v1[p1++];
```

```
    }

    while (p2 < v2.length) {
        v[p++] = v2[p2++];
    }

    while (p3 < v3.length) {
        v[p++] = v3[p3++];
    }

    return v;

}
```

---

13. Faça uma tabela comparativa dos algoritmos vistos em aula considerando os seguintes aspectos: número de comparações (melhor, pior e caso médio), número de movimentações, facilidade de implementação e uso de memória auxiliar.

Resolução:

Método	**Comparações**			**Trocas**			**Estável**
	Melhor	**Médio**	**Pior**	**Melhor**	**Médio**	**Pior**	
Selection	O(nˆ2)	O(nˆ2)	O(nˆ2)	O(1)	?	O(n)	Não
Insertion	O(nˆ2)	O(nˆ2)	O(n)	O(1)	O(nˆ2)	O(nˆ2)	Sim
Bubble	O(nˆ2)	O(nˆ2)	O(n)	O(1)	?	?	Sim
Merge	O(n log n)	Θ(n log n)	Ω(n log n)	?	?	?	Sim
Heap	Θ(n log n)	Θ(n log n)	Θ(n log n)	?	?	?	Não
Quick	O(nˆ2)	O(n log n)	O(n log n)	?	?	?	Não

---

14. Faça um programa que leia n nomes e ordene-os utilizando o algoritmo heapsort. No final, o algoritmo deve mostrar todos os nomes ordenados.

Resolução:

---

15. Considere o código abaixo, que corresponde a uma possível implementação do método de ordenação Bubble Sort, e escreva, utilizando a notação Σ, o somatório do número de comparações em função do tamanho n do arranjo de entrada (n = array.length). Em seguida, encontre a fórmula fechada do somatório em função apenas de n.

```
void bubblesort(int[] array) {
    for (int i = (array.length - 1); i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
            }
        }
    }
}
```

Resolução:

Sabemos que:

- $\sum_{0}^{n} i = \frac{n(n+1)}{2}$ é a fórmula a ser utilizada;
- i vai de $n - 1$ até 1, portanto, o primeiro loop executa $n - 1$ vezes;
- j vai de 0 à $i - 1$, portanto, o segundo loop executa i vezes.

Portanto, a função de complexidade é:

$f(n) = (n - 1) \times i \times 1 = (n - 1) \times i$

Com isso, basta aplicar a fórmula ao somatório abaixo:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n-1+1)}{2} = \frac{(n-1)(n)}{2} = \frac{n^2-n}{2}$$