

Lista de exercícios

1 Introdução

Exercício 1.1

([Solução 1.1](#))

Assista ao vídeo “*What’s an algorithm?*” (David J. Malan).

⇒ <https://youtu.be/6hfOvs8pY1k>

Exercício 1.2

([Solução 1.2](#))

Assista ao vídeo “*Top 7 Data Structures for Interviews*”.

⇒ <https://youtu.be/cQWr9DFE1ww>

2 Complexidade de algoritmos

Exercício 2.1

([Solução 2.1](#))

Desenhe o gráfico das funções $8n$, $4n \log n$, $2n^2$, n^3 e 2^n usando uma escala logarítmica para os eixos x e y , isto é, se o valor da função $f(x)$ é y , desenhe esse ponto com a coordenada x em $\log x$ e a coordenada y em $\log y$.

Exercício 2.2

([Solução 2.2](#))

O número de operações executadas por dois algoritmos A e B é $40n^2$ e $2n^3$, respectivamente. Determine n_0 tal que A seja melhor que B para $n \geq n_0$.

Exercício 2.3

([Solução 2.3](#))

O número de operações executadas por dois algoritmos A e B é $8n \log n$ e $2n^2$, respectivamente. Determine n_0 tal que A seja melhor que B para $n \geq n_0$.

Exercício 2.4

([Solução 2.4](#))

Ordene as funções a seguir por sua taxa assintótica de crescimento.

- $4n \log n + 2n$
- 2^{10}
- $3n + 100 \log n$
- $4n$
- 2^n
- $n^2 + 10n$
- n^3
- $n \log n$

Exercício 2.5

(Solução 2.5)

Qual a complexidade assintótica no pior caso (em termos de \mathcal{O}) do algoritmo abaixo?

```
1  /** Returns the sum of the integers in given array. */
2  public static int alg1(int[] arr) {
3      int n = arr.length, total = 0;
4      for (int j=0; j < n; j++)
5          total += arr[j];
6      return total;
7  }
```

Exercício 2.6

(Solução 2.6)

Qual a complexidade assintótica no pior caso (em termos de \mathcal{O}) do algoritmo abaixo?

```
1  /** Returns the sum of the integers with even index in given array. */
2  public static int alg2(int[] arr) {
3      int n = arr.length, total = 0;
4      for (int j=0; j < n; j += 2)
5          total += arr[j];
6      return total;
7  }
```

Exercício 2.7

(Solução 2.7)

Qual a complexidade assintótica no pior caso (em termos de \mathcal{O}) do algoritmo abaixo?

```
1  /** Returns the sum of the prefix sums of given array. */
2  public static int alg3(int[] arr) {
3      int n = arr.length, total = 0;
4      for (int j=0; j < n; j++)
5          for (int k=0; k <= j; k++)
6              total += arr[k];
7      return total;
8  }
```

Exercício 2.8

(Solução 2.8)

Qual a complexidade assintótica no pior caso (em termos de \mathcal{O}) do algoritmo abaixo?

```
1  /** Returns the sum of the prefix sums of given array. */
2  public static int alg4(int[] arr) {
3      int n = arr.length, prefix = 0, total = 0;
4      for (int j=0; j < n; j++) {
5          prefix += arr[j];
6          total += prefix;
7      }
8      return total;
9  }
```

Exercício 2.9

(Solução 2.9)

Qual a complexidade assintótica no pior caso (em termos de \mathcal{O}) do algoritmo abaixo?

```
1  /** Returns the number of times second array stores sum of prefix sums from first. */
2  public static int alg5(int[] first, int[] second) {
3      int n = first.length, count = 0;
4      for (int i=0; i < n; i++) {
5          int total = 0;
6          for (int j=0; j < n; j++)
7              for (int k=0; k <= j; k++)
8                  total += first[k];
9          if (second[i] == total) count++;
10     }
11     return count;
12 }
```

Exercício 2.10

(Solução 2.10)

O algoritmo A executa uma computação em tempo $\mathcal{O}(\log n)$ para cada entrada de um arranjo de n elementos. Qual o pior caso em relação ao tempo de execução de A?

Exercício 2.11

(Solução 2.11)

Dado um arranjo X de n elementos, o algoritmo B escolhe $\log n$ elementos de X, aleatoriamente, e executa um cálculo em tempo $\mathcal{O}(n)$ para cada um. Qual o pior caso em relação ao tempo de execução de B?

Exercício 2.12

(Solução 2.12)

Dado um arranjo X de n elementos inteiros, o algoritmo C executa uma computação em tempo $\mathcal{O}(n)$ para cada número par de X e uma computação em tempo $\mathcal{O}(\log n)$ para cada elemento ímpar de X. Qual o melhor caso e o pior caso em relação ao tempo de execução de C?

Exercício 2.13

(Solução 2.13)

Dado um arranjo X de n elementos, o algoritmo D chama o algoritmo E para cada elemento X[i]. O algoritmo E executa em tempo $\mathcal{O}(i)$ quando é chamado sobre um elemento X[i]. Qual o pior caso em relação ao tempo de execução do algoritmo D?

Exercício 2.14

(Solução 2.14)

Implemente os algoritmos `disjoint1` e `disjoint2` (apresentados nos materiais de aula), e execute uma análise experimental dos seus tempos de execução. Visualize seus tempos de execução como uma função do tamanho da entrada usando um gráfico *di-log*.

Exercício 2.15

(Solução 2.15)

Execute uma análise experimental para testar a hipótese de que o método da biblioteca Java, `java.util.Arrays.sort` executa em um tempo médio $\mathcal{O}(n \log n)$.

Exercício 2.16

(Solução 2.16)

Execute uma análise experimental para determinar o maior valor de n para os algoritmos `unique1` e `unique2` (apresentados nos materiais de aula), de modo que o algoritmo execute em um minuto ou menos.

3 Estruturas de dados fundamentais

Exercício 3.1

(Solução 3.1)

O método `removeFirst` da classe `SinglyLinkedList` inclui um caso especial para redefinir o campo `tail` para `null` na remoção do último elemento da lista. Quais são as consequências de remover essas linhas de código? Explique por que a classe não funcionaria com essa modificação.

Exercício 3.2

(Solução 3.2)

Forneça uma implementação para o método `size()` da classe `SinglyLinkedList`, considerando que a mesma não mantenha o tamanho armazenado em uma variável. Qual a implicação dessa modificação na complexidade assintótica do método?

Exercício 3.3

(Solução 3.3)

Proponha um algoritmo para encontrar o penúltimo nodo em uma lista simplesmente encadeada na qual o último nodo possui uma referência nula no campo `next`.

Exercício 3.4

(Solução 3.4)

Forneça a implementação do método `removeLast` para a classe `SinglyLinkedList`, para remover o último elemento da lista.

Exercício 3.5

(Solução 3.5)

Considere o método `addFirst` da classe `CircularlyLinkedList`. O corpo do `else` depende de uma variável local `newest`. Projete um novo código para esta cláusula sem o uso de nenhuma variável local.

Exercício 3.6

(Solução 3.6)

Descreva um método para encontrar o nodo central de uma lista duplamente encadeada com nodos sentinelas, sem o uso de informações sobre o tamanho da lista. No caso de um número par de nodos, o método deve devolver o nodo à esquerda do ponto central. Qual a complexidade deste algoritmo?

Exercício 3.7

(Solução 3.7)

Forneça uma implementação para o método `size()` da classe `CircularlyLinkedList`, considerando que a mesma não mantenha o tamanho armazenado em uma variável. Qual a implicação dessa modificação na complexidade assintótica do método?

Exercício 3.8

(Solução 3.8)

Forneça uma implementação para o método `size()` da classe `DoublyLinkedList`, considerando que a mesma não mantenha o tamanho armazenado em uma variável. Qual a implicação dessa modificação na complexidade assintótica do método?

Exercício 3.9

(Solução 3.9)

Implemente o método `equals()` para a classe `CircularlyLinkedList`, assumindo que duas listas são iguais se elas possuem a mesma sequência de elementos, com os elementos correspondentes no início da lista.

Exercício 3.10

(Solução 3.10)

Descreva um algoritmo para concatenar duas listas simplesmente encadeadas L e M , em uma lista única L' que contém todos os nodos de L seguido de todos os nodos de M .

Exercício 3.11

(Solução 3.11)

Descreva um algoritmo para concatenar duas listas duplamente encadeadas L e M com sentinelas, em uma lista única L' .

Exercício 3.12

(Solução 3.12)

Descreva em detalhes como trocar dois nodos x e y de posição (não apenas seu conteúdo) em uma lista simplesmente encadeada L , dadas as referências para x e y somente. Repita este exercício para o caso em que L é uma lista duplamente encadeada. Qual algoritmo possui maior complexidade?

Exercício 3.13

(Solução 3.13)

Descreva em detalhes um algoritmo para reverter uma lista simplesmente encadeada L usando somente uma quantidade constante de espaço adicional.

4 Pilhas, Filas e Deques

Exercício 4.1 (teste)

([Solução 4.1](#))

Suponha que inicialmente uma pilha vazia *S* tenha realizado um total de 25 operações *push*, 12 operações *top* e 10 operações *pop*, 3 das quais retornaram *null*, indicando uma pilha vazia. Qual é o tamanho atual de *S*?

Exercício 4.2 (teste)

([Solução 4.2](#))

Sendo a pilha do exercício anterior uma instância da classe `ArrayStack`, qual o valor final da variável *t*?

Exercício 4.3 (teste)

([Solução 4.3](#))

Quais valores são retornados durante as seguintes operações, se executadas em uma pilha inicialmente vazia? *push*(5), *push*(3), *pop*(), *push*(2), *push*(8), *pop*(), *pop*(), *push*(9), *push*(1), *pop*(), *push*(7), *push*(6), *pop*(), *pop*(), *push*(4), *pop*(), *pop*().

Exercício 4.4 (teste)

([Solução 4.4](#))

Implemente um método com a assinatura `transfer(S, T)` que transfere todos os elementos da pilha *S* para a pilha *T*, de modo que o elemento que iniciou no topo de *S* é o primeiro elemento a ser inserido em *T*, e o último elemento de *S* termina no topo de *T*.

Exercício 4.5 (teste)

([Solução 4.5](#))

Apresente um método recursivo que remove todos os elementos de uma pilha.

Exercício 4.6 (teste)

([Solução 4.6](#))

Suponha que uma fila vazia *Q* realizou um total de 32 operações de *enqueue*, 10 operações *first* e 15 operações *dequeue*, 5 das quais retornaram *null*, indicando uma fila vazia. Qual é o tamanho atual de *Q*?

Exercício 4.7 (teste)

([Solução 4.7](#))

Sendo a fila do exercício anterior uma instância da classe `ArrayQueue` com uma capacidade de 30 elementos nunca excedida, qual o valor final da variável *f*?

Exercício 4.8 (teste)

([Solução 4.8](#))

Quais são os valores retornados após as seguintes operações, se executadas em uma fila inicialmente vazia? *enqueue*(5), *enqueue*(3), *dequeue*(), *enqueue*(2), *enqueue*(8), *dequeue*(), *dequeue*(), *enqueue*(9), *enqueue*(1), *dequeue*(), *enqueue*(7), *enqueue*(6), *dequeue*(), *dequeue*(), *enqueue*(4), *dequeue*(), *dequeue*().

Exercício 4.9 (teste)

([Solução 4.9](#))

Faça um adaptador simples (classe) que implemente uma pilha usando uma instância de deque para armazenamento.

Exercício 4.10 (teste)

([Solução 4.10](#))

Faça um adaptador simples (classe) que implemente uma fila usando uma instância de deque para armazenamento.

Exercício 4.11 (teste)

[\(Solução 4.11\)](#)

Quais são os valores retornados após as seguintes operações, se executadas em um deque inicialmente vazio? `addFirst(3)`, `addLast(8)`, `addLast(9)`, `addFirst(1)`, `last()`, `isEmpty()`, `addFirst(2)`, `removeLast()`, `addLast(7)`, `first()`, `last()`, `addLast(4)`, `size()`, `removeFirst()`, `removeFirst()`.

Exercício 4.12 (teste)

[\(Solução 4.12\)](#)

Suponha que você tenha um deque `D` contendo os números (1, 2, 3, 4, 5, 6, 7, 8), nessa ordem. Supondo que além disso você tenha uma fila `Q` inicialmente vazia. Apresente um pseudocódigo que utilize somente `D` e `Q` (mais nenhuma variável) e resulte em `D` armazenando os elementos na ordem (1, 2, 3, 5, 4, 6, 7, 8).

Exercício 4.13 (teste)

[\(Solução 4.13\)](#)

Repita o exercício anterior usando o deque `D` e uma pilha `S`, inicialmente vazia.

Soluções dos exercícios

Importante: as informações apresentadas neste documento visam conduzir o estudante na resolução dos exercícios da disciplina. Na maioria dos casos, são apresentados apenas os direcionamentos da solução esperada. Cabe ao estudante o desenvolvimento completo da solução.

1 Introdução

Solução 1.1

([Exercício 1.1](#))

Assista ao vídeo sugerido.

Solução 1.2

([Exercício 1.2](#))

Assista ao vídeo sugerido.

2 Complexidade de algoritmos

Solução 2.1

([Exercício 2.1](#))

Gráfico disponível [aqui](#).

Solução 2.2

([Exercício 2.2](#))

Igualando as equações, temos que $40n^2 = 2n^3$ para $n' = 0$ e $n'' = 20$. Logo, $40n^2 \leq 2n^3$ para $n \geq 20$. Veja a representação gráfica em [aqui](#).

Solução 2.3

([Exercício 2.3](#))

Por inspeção, assumindo $n_0 = 10$, temos que $8n \log n \leq 2n^2$ para todo $n \geq n_0$. Veja a representação gráfica em [aqui](#).

Solução 2.4

([Exercício 2.4](#))

$2^{10} \ll 3n + 100 \log n = 4n \ll n \log n = 4n \log n + 2n \ll n^2 + 10n \ll n^3 \ll 2^n$.

Solução 2.5

([Exercício 2.5](#))

$\mathcal{O}(n)$.

Solução 2.6

([Exercício 2.6](#))

$\mathcal{O}(n)$.

Solução 2.7

([Exercício 2.7](#))

$\mathcal{O}(n^2)$.

Solução 2.8

([Exercício 2.8](#))

$\mathcal{O}(n)$.

Solução 2.9

(Exercício 2.9)

$\mathcal{O}(n^3)$.

Solução 2.10

(Exercício 2.10)

$\mathcal{O}(n \log n)$.

Solução 2.11

(Exercício 2.11)

$\mathcal{O}(n \log n)$.

Solução 2.12

(Exercício 2.12)

Melhor caso: $\mathcal{O}(n \log n)$, quando todos os elementos são ímpares. Pior caso: $\mathcal{O}(n^2)$, quando todos os elementos são pares.

Solução 2.13

(Exercício 2.13)

O tempo de execução é proporcional a $\sum_{i=1}^n i = n(n-1)/2 = \mathcal{O}(n^2)$.

Solução 2.14

(Exercício 2.14)

Exercício de análise experimental.

Solução 2.15

(Exercício 2.15)

Exercício de análise experimental.

Solução 2.16

(Exercício 2.16)

Exercício de análise experimental.

3 Estruturas de dados fundamentais

Solução 3.1

(Exercício 3.1)

Modifique o código e teste as alterações. Não há consequências negativas em remover essas linhas (na verdade, isso deve produzir uma leve melhoria na eficiência). Enquanto o estado interno está inconsistente com o `tail` referenciando um nó “inexistente”, a referência `tail` nunca é acessada em uma lista vazia, então a inconsistência não tem efeito.

Solução 3.2

(Exercício 3.2)

Uma possível solução é percorrer a estrutura, contando a quantidade de nodos. Isso implica em um aumento na complexidade assintótica de constante para linear, i.e. $\mathcal{O}(1)$ para $\mathcal{O}(n)$.

```
1 public int size(){
2     int count = 0;
3     Node<E> walk = head;
4     while(walk != null){
5         count++;
6         walk = walk.getNext();
7     }
8     return count;
9 }
```

Solução 3.3

(Exercício 3.3)

Uma possível solução consiste em um algoritmo linear simples.


```
1 private Node<E> penultimate(){
2     if(size<2) throw new IllegalStateException("list must have 2 or more entries");
3     Node<E> walk = head;
4     while(walk->next->next != null)
5         walk = walk->next;
6     return walk;
7 }
```

Solução 3.4

(Exercício 3.4)

Dica: use o método `penultimate` do exercício anterior para atualizar a referência do atributo `tail`.

Solução 3.5

(Exercício 3.5)

Existe uma solução de linha única.

```
1 tail.setNext(new Node<>(e, tail.getNext()));
```

Solução 3.6

(Exercício 3.6)

Considere uma busca combinada de ambas extremidades. Lembre-se que um *link hop* é uma atribuição do formato `p = p.getNext()`; ou `p = p.getPrev()`; . O método a seguir executa em tempo $O(n)$.

```
1 private Node<E> middle(){
2     if(size == 0) throw new IllegalStateException("list must be nonempty");
3     Node<E> moddle = header->next
4     Node<E> partner = trailer->prev;
5     while(middle != partner && middle->next != partner){
6         middle = middle.getNext();
7         partner = partner.getPrev();
8     }
9     return middle;
10 }
```

Solução 3.7

(Exercício 3.7)

Uma possível solução é percorrer a estrutura, contando a quantidade de nodos. Isso implica em um aumento na complexidade assintótica de constante para linear, i.e. $\mathcal{O}(1)$ para $\mathcal{O}(n)$.

```
1 public int size(){
2     if(tail == null)
3         return 0;
4     Node<E> wal = tail->next;
5     int count = 1;
6     while(walk != tail){
7         count++;
8         walk = walk.getNext();
9     }
10    return count;
11 }
```

Solução 3.8

(Exercício 3.8)

Uma possível solução é percorrer a estrutura, contando a quantidade de nodos. Isso implica em um aumento na complexidade assintótica de constante para linear, i.e. $\mathcal{O}(1)$ para $\mathcal{O}(n)$.

```
1 public int size(){
2     int count = 0;
3     Node<E> walk = header->next;
4     while(walk != trailer){
5         count++;
6         walk = walk.getNext();
7     }
8     return count;
9 }
```

```
9 }
}
```

Solução 3.9

(Exercício 3.9)

Dica: você pode contar com a variável **size** para definir o número de passos corretos ao percorrer a estrutura.

Solução 3.10

(Exercício 3.10)

A operação de concatenação não precisa procurar tudo em L e M . Use um nodo temporário para caminhar até o final da lista L . Então, faça o último elemento de L apontar para o primeiro elemento de M como seu próximo nodo. O número de passos é proporcional ao tamanho de L .

```
1 Concatenate(L,M):
2   Create a node v
3   v = L.getHead()
4   while v.getNext() != null do
5       v = v.getNext()
6   v.setNext(M.getHead())
7   return L
```

Solução 3.11

(Exercício 3.11)

Junte o final de L no começo de M . Use dois nodos temporários, **temp1** e **temp2**. Inicialize **temp1** como o **trailer** de L e **temp2** como o **header** de M . Atribua **temp2** como o próximo elemento de **temp1** e **temp1** como o elemento anterior de **temp2**. Faça $L' \leftarrow L$ e atribua o **trailer** de M como **trailer** de L' .

Solução 3.12

(Exercício 3.12)

Realizar uma troca (*swap*) em uma lista simplesmente encadeada levará mais tempo do que em uma lista duplamente encadeada. Essa implementação requer muito cuidado, especialmente quando x e y são vizinhos um do outro. A dificuldade na eficiência ocorre porque para trocar x e y em uma lista simplesmente encadeada devemos localizar os nodos imediatamente anteriores a x e y percorrendo a estrutura, e não tem uma maneira rápida de fazer isso.

Solução 3.13

(Exercício 3.13)

Dica: considere mudar a orientação das ligações enquanto faz uma única passagem pela lista. Tal método da classe **SinglyLinkedList** poderá ser implementado da seguinte forma (observe que esta implementação funciona mesmo para listas triviais).

```
1 public void reverse(){
2     Node<E> prev = null;
3     Node<E> walk = head;
4     while(walk != null){
5         Node<E> adv = walk.getNext();
6         walk.setNext(prev);
7         prev = walk;
8         walk = adv;
9     }
10    head = prev;
11 }
```