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[j];
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 (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 (Solução 4.2)
Sendo a pilha do exercício anterior uma instância da classe `ArrayStack`, qual o valor final da variável `t`?

Exercício 4.3 (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 (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 (Solução 4.5)
Apresente um método recursivo que remove todos os elementos de uma pilha.

Exercício 4.6 (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 (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 (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 (Solução 4.9)
Faça um adaptador simples (classe) que implemente uma pilha usando uma instância de deque para armazenamento.

Exercício 4.10 (Solução 4.10)
Faça um adaptador simples (classe) que implemente uma fila usando uma instância de deque para armazenamento.

Exercício 4.11 (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 (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` incialmente 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 (Solução 4.13)
Repita o exercício anterior usando o deque `D` e uma pilha `S`, inicialmente vazia.

5 Listas dinâmicas

Exercício 5.1 (Solução 5.1)
A complexidade dos métodos utilizados pela `LinkedList` se origina no procedimento `searchNode`, responsável por buscar o nodo da posição que se deseja acessar. O desempenho dessa estrutura de dados pode ser melhorado inibindo essa busca quando se tratar de acesso ao início ou fim da lista. Implemente essa estratégia.

Exercício 5.2 (Solução 5.2)
Outra forma de melhorar o desempenho de uma `LinkedList` é fazer com que a busca implementada no procedimento `searchNode` seja feita "de trás para frente", quando conveniente. Implemente essa estratégia. Qual o impacto na complexidade do procedimento de busca?

Exercício 5.3 (Solução 5.3)
Crie um método `toArray` na classe `LinkedList` que retorne um *array* com os elementos da lista encadeada. Implemente a operação inversa na classe `ArrayList`.

Exercício 5.4 (Solução 5.4)
Forneça uma representação de uma lista `L`, inicialmente vazia, após realizar as seguintes operações: `add(0,4)`, `add(0,3)`, `add(0,2)`, `add(2,1)`, `add(1,5)`, `add(1,6)`, `add(3,7)`, `add(0,8)`.

Exercício 5.5 (Solução 5.5)
Implemente uma pilha usando um `ArrayList` para armazenamento dos dados.

Exercício 5.6 (Solução 5.6)
O `java.util.ArrayList` possui um método `trimToSize` que substitui o *array* de dados por um com capacidade equivalente ao número de elementos atuais da lista. Implemente tal método na classe `ArrayList`.

Exercício 5.7 (Solução 5.7)
Considere uma implementação de um `ArrayList` usando um *array* dinâmico, mas em vez de copiar os elementos para um *array* com o dobro do tamanho (isto é, de N para $2N$) quando sua capacidade é atingida, copiamos os elementos para um *array* com $\lceil N/4 \rceil$ células adicionais, indo da capacidade N para

$N + \lceil N/4 \rceil$. Mostre experimentalmente que ao realizar uma sequência de n operações `add` (i.e., inserindo no fim), a estrutura ainda opera em tempo $\mathcal{O}(n)$.

Exercício 5.8 (Solução 5.8)
Supondo que estamos mantendo uma coleção `C` de elementos de tal modo que, cada vez que adicionamos um novo elemento na coleção, copiamos o conteúdo de `C` em um novo `ArrayList` do tamanho exato ao necessário. Qual o tempo de processamento de adição de n elementos em uma coleção `C` inicialmente vazia?

Exercício 5.9 (Solução 5.9)
O método `add` para um *array* dinâmico tem a seguinte ineficiência: no caso em que um redimensionamento ocorre, a operação correspondente leva tempo para copiar todos os elementos do antigo *array* para o novo, e então o laço subsequente muda alguns deles para dar espaço para o novo elemento. Modifique o método `add` para, no caso de redimensionamento, os elementos copiados ficarem na sua posição final do novo *array* (ou seja, nenhuma realocação deve ser feita).

Exercício 5.10 (Solução 5.10)
Reimplemente a classe `ArrayStack` usando *arrays* dinâmicos para suportar uma capacidade ilimitada.

Exercício 5.11 (Solução 5.11)
A interface `java.util.Collection` inclui um método `contains(o)`, que retorna `true` se a coleção possui um objeto que é igual a `Object o`. Implemente tal método na classe `ArrayList`.

Exercício 5.12 (Solução 5.12)
A interface `java.util.Collection` inclui um método `clear()`, que remove todos os elementos de uma coleção. Implemente tal método na classe `ArrayList`.

Exercício 5.13 (Solução 5.13)
Desenvolva um experimento para testar a eficiência de n chamadas sucessivas ao método `add` de um `ArrayList` para vários n diferentes, e analise os resultados empíricos sob os seguintes cenários:

a. Cada `add` acontece no índice 0.

b. Cada `add` acontece no índice `size()/2`.

c. Cada `add` acontece no índice `size()`.

6 Filas de prioridade

Exercício 6.1 (Solução 6.1)
O que cada uma das chamadas `removeMin` retorna, dentre a seguinte sequências de operações em uma fila de prioridade: `insert(5,A)`, `insert(4,B)`, `insert(7,F)`, `insert(1,D)`, `removeMin()`, `insert(3,J)`, `insert(6,L)`, `removeMin()`, `removeMin()`, `insert(8,G)`, `removeMin()`, `insert(2,H)`, `removeMin()`, `removeMin()`?

Exercício 6.2 (Solução 6.2)
Um aeroporto está desenvolvendo uma simulação computacional de controle de tráfego aéreo para lidar com eventos como aterrissagens e decolagens. Cada evento tem um *timestamp* que simboliza o tempo em que o evento ocorrerá. A simulação necessita realizar eficientemente duas operações fundamentais:

- Inserir um evento com um *timestamp* (isto é, adicionar um evento futuro).
- Retornar o evento com o *timestamp* mais próximo (i.e., determinar o próximo evento para processar).
- Qual estrutura de dados deverá ser usada para realizar essas operações? Por quê?

Exercício 6.3 (Solução 6.3)
O método `min` de uma fila de prioridade não-ordenada executa em tempo $\mathcal{O}(n)$. Proponha uma alteração nessa estrutura para que o método `min` execute em tempo $\mathcal{O}(1)$. Explique qualquer modificação necessária em outros métodos da estrutura.

Exercício 6.4 (Solução 6.4)
Você pode adaptar sua solução do exercício anterior para fazer o método `removeMin` de uma fila de prioridade não-ordenada executar em tempo $\mathcal{O}(1)$? Justifique sua resposta.

Exercício 6.5 (Solução 6.5)
Mostre como implementar uma pilha (LIFO) usando apenas uma fila de prioridade e uma variável inteira adicional.

Exercício 6.6 (Solução 6.6)
Mostre como implementar uma fila (FIFO) usando apenas uma fila de prioridade e uma variável inteira adicional.

Exercício 6.7 (Solução 6.7)
O Professor Idle sugere a seguinte solução para o exercício anterior. Sempre que uma entrada é inserida na fila, é dada uma chave que é igual ao tamanho atual da fila. Esta estratégia resulta em uma semântica FIFO? Prove que é verdadeiro ou dê um contra exemplo.

Exercício 6.8 (Solução 6.8)
Considere uma fila de prioridade que armazena valores inteiros e usa uma string como chave. Essa estrutura usa a classe abaixo como comparador específico das suas chaves.

```
1  public class MyStringComparator implements Comparator<String> {
2    public int compare(String a, String b) {
3      if(a.length() < b.length())
4        return -1;
5      else if(a.length() > b.length())
6        return 1;
7      else
8        return a.compareTo(b);
9      }
10 }
```

Informe a sequência de entradas retornadas e a configuração final de uma fila de prioridade ordenada ao executar esta sequência de comandos: `insert("Phoebe", 15)`, `insert("Joey", 20)`, `insert("Ross", 10)`, `min()`, `insert("Rachel", 23)`, `removeMin()`, `removeMin()`, `min()`, `insert("Monica", 28)`, `removeMin()`.

7 Buscas em estruturas lineares

Exercício 7.1 (Solução 7.1)
Implemente uma versão recursiva do algoritmo de busca binária, conforme as ideias do Capítulo 5 (Recursão) de Goodrich et al. (2014) – *Data Structures and Algorithms in Java.*

Exercício 7.2 (Solução 7.2)
Simule o algoritmo de busca binária para os seguintes casos:

a) `x = 15`, `v = {15, 27, 33, 46, 51, 63, 71, 82, 90}`.

b) `x = 33`, `v = {15, 27, 33, 46, 51, 63, 71, 82, 90}`.

c) `x = 63`, `v = {15, 27, 33, 46, 51, 63, 71, 82, 90}`.

d) `x = 81`, `v = {15, 27, 33, 46, 51, 63, 71, 82, 90}`.

e) `x = 22`, `v = {15, 27, 33, 46, 51, 63, 71, 82, 90}`.

Compare o número de avaliações realizadas pelas buscas binária e sequencial.

Exercício 7.3 (Solução 7.3)
Quando o vetor está ordenado, a busca sequencial não precisa percorrer toda a lista para saber que o elemento buscado não existe. Ela pode parar quando o elemento analisado for maior que o buscado. Implemente as modificações necessárias para essa estratégia. Qual o impacto na complexidade assintótica do novo algoritmo?

Exercício 7.4 (Solução 7.4)
Implemente os algoritmos de busca sequencial e binária para a lista dinâmica implementada pela classe `ArrayList`. Os métodos devem retornar a posição onde o elemento foi encontrado, ou -1 caso ele não seja encontrado.

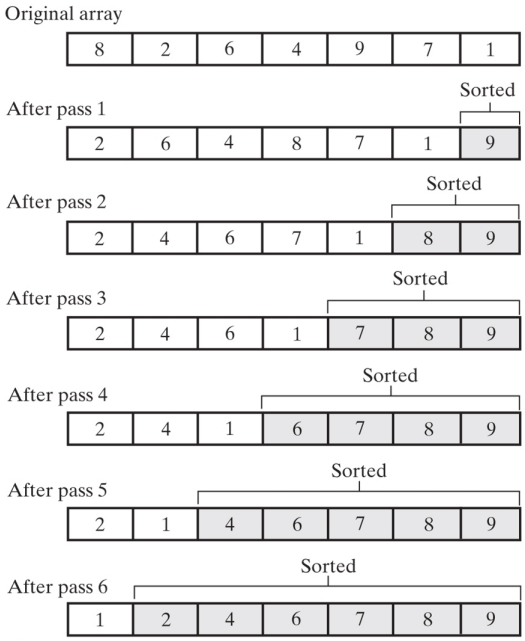
Exercício 7.5 (Solução 7.5)
Veja as demonstrações das buscas sequencial e binária disponíveis em https://www.cs.usfca.edu/~galles/visualization/Search.html.

8 Ordenação de estruturas lineares

Exercício 8.1 (Solução 8.1)
Mostre o conteúdo do *array* de inteiros $(5, 7, 4, 9, 8, 5, 6, 3)$ para cada vez que o que o algoritmo insertion sort o modifica durante o processo de ordenação.

Exercício 8.2 (Solução 8.2)
Qual modificação é necessária para que o algoritmo insertion sort ordene os elementos de forma decrescente?

Exercício 8.3 (Solução 8.3)
O algoritmo bubble sort ordena um *array* de n elementos em ordem crescente, executando $n-1$ passagens pelo *array*. Em cada passagem, ele compara elementos adjacentes e os troca se estiverem fora de ordem. Por exemplo, na primeira passagem ele compara o primeiro e o segundo elementos, depois o segundo e o terceiro elementos, e assim por diante. No final da primeira passagem, o maior elemento está em sua posição adequada no final do *array*. Cada passagem subsequente ignora os elementos no final do *array*, pois eles estão ordenados e são maiores que qualquer um dos elementos restantes. Assim, cada passagem faz uma comparação a menos que a passagem anterior. A figura abaixo ilustra o funcionamento do bubble sort.



Implemente o bubble sort para ordenar um *array* genérico.

Exercício 8.4 (**Solução 8.4**)
Qual a complexidade assintótica de tempo do bubble sort?

Exercício 8.5 (**Solução 8.5**)
Implemente um algoritmo para verificar se um *array* está em ordem não-decrescente. Você pode usar esse método para verificar se um algoritmo de ordenação executou corretamente.

Exercício 8.6 (**Solução 8.6**)
No algoritmo insertion sort, podemos usar uma busca binária para encontrar a posição de inserção de cada elemento. Qual o impacto na complexidade assintótica do algoritmo?

Exercício 8.7 (**Solução 8.7**)
Estude os seguintes algoritmos de ordenação:

- Merge sort;
- Quick sort;
- Radix sort.

Exercício 8.8 (**Solução 8.8**)
Veja as demonstrações dos diferentes algoritmos de ordenação estudados em https://www.cs.usfca.edu/~galles/visualization/Algorithms.html e https://visualgo.net/en/sorting.

Soluções dos exercícios

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 $\sum_{i=1}^{n} i = n(n-1)/2 = O(n^2)$.

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 IllegalStateExeception("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   }
```

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.

```
Concatenate(L,M):
  Create a node v
  v = L.getHead()
  while v.getNext() != null do
    v = v.getNext()
  v.setNext(M.getHead())
  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).

```
public void reverse(){
  Node<E> prev = null;
  Node<E> walk = head;
  while(walk != null){
    Node<E> adv = walk.getNext();
    walk.setNext(prev);
    prev = walk;
    walk = adv;
  }
  head = prev;
}
```

4 Pilhas, filas e deques

Solução 4.1 (Exercício 4.1)
Se a pilha está vazia quando `pop` é chamado, seu tamanho não muda. Logo, o tamanho da pilha é $25 - 10 + 3 = 18$.

Solução 4.2 (Exercício 4.2)
É uma posição menor que o tamanho. Logo, $t = 17$.

Solução 4.3 (**Exercício 4.3**)
Desenhe a estrutura para simular as operações e mudanças realizadas. Resultado: $3, 8, 2, 1, 6, 7, 4, 9$.

Solução 4.4 (**Exercício 4.4**)
Você deve transferir um item de cada vez.

```
1 static <E> void transfer(Stack<E> S, Stack<E> T) {
2   while (!S.isEmpty( )) {
3     T.push(S.pop( ));
4   }
5 }
```

Solução 4.5 (**Exercício 4.5**)
Se a pilha está vazia, retorne "pilha vazia". Caso contrário, remova o elemento do topo da pilha e chame a operação recursivamente com a pilha atualizada.

Solução 4.6 (**Exercício 4.6**)
Se a pilha está vazia quando `dequeue` é chamado, seu tamanho não é modificado. Logo, o tamanho da fila é $32 - 15 + 5 = 22$.

Solução 4.7 (**Exercício 4.7**)
Cada operação `dequeue` de sucesso implica em mover o índice para a direita de maneira circular. Logo, $f = 10$.

Solução 4.8 (**Exercício 4.8**)
Desenhe a estrutura para simular as operações e mudanças realizadas. Resultado: $5, 3, 2, 8, 9, 1, 7, 6$.

Solução 4.9 (**Exercício 4.9**)
Dica: basta usar as operações apropriadas nas extremidades do deque.

Solução 4.10 (**Exercício 4.10**)
Dica: basta usar as operações apropriadas nas extremidades do deque.

Solução 4.11 (**Exercício 4.11**)
Desenhe a estrutura para simular as operações e mudanças realizadas. Resultado: 9, `false`, $9, 2, 7, 6, 2, 1$.

Solução 4.12 (**Exercício 4.12**)
A solução consiste em usar o resultado dos métodos de remoção como argumentos para os métodos de inserção. Solução:

```
D.addLast(D.removeFirst())
D.addLast(D.removeFirst())
D.addLast(D.removeFirst())
Q.enqueue(D.removeFirst())
Q.enqueue(D.removeFirst())
D.addFirst(Q.dequeue())
D.addFirst(Q.dequeue())
D.addFirst(D.removeLast())
```

```
D.addFirst(D.removeLast())
D.addFirst(D.removeLast())
```

Solução 4.13 (**Exercício 4.13**)
A solução consiste em usar o resultado dos métodos de remoção como argumentos para os métodos de inserção. Adicionalmente, você precisará usar mais de uma pilha para armazenamento temporário. Solução:

```
D.addLast(D.removeFirst())
D.addLast(D.removeFirst())
D.addLast(D.removeFirst())
S.push(D.removeFirst())
D.addLast(D.removeFirst())
D.addFirst(S.pop())
D.addFirst(D.removeLast())
D.addFirst(D.removeLast())
D.addFirst(D.removeLast())
D.addFirst(D.removeLast())
```

5 Listas dinâmicas

Solução 5.1 (**Exercício 5.1**)
Caso a posição buscada seja `0` ou `size - 1`, retorna o elemento correspondente (associados ao primeiro ou último nodos, respectivamente). É necessário verificar se a lista está vazia.

Solução 5.2 (**Exercício 5.2**)
Basta verificar se o índice buscado é menor ou maior que `size/2`, para saber em qual metade da estrutura o índice se encontra. Caso se trate da primeira metade, a busca deve ser realizada a partir do primeiro elemento. Caso contrário, a busca inicia pelo último elemento. Com isso, apenas metade dos elementos precisará ser varrido no pior caso. Logo, a complexidade cai para $n/2$, o que é mais eficiente na prática, mas não altera a complexidade assintótica $\mathcal{O}(n)$.

Solução 5.3 (**Exercício 5.3**)
Os métodos devem criar as listas alternativas, atribuir os elementos a elas e retornar a estrutura criada. Diferentes implementações podem devolver uma lista com as mesmas referências ou com cópias dos elementos.

Solução 5.4 (**Exercício 5.4**)
Desenhe a lista, mostrando os estados antes e depois de cada operação. A configuração final da lista deve ser $(8, 2, 6, 5, 7, 3, 1, 4)$.

Solução 5.5 (**Exercício 5.5**)
Dica: use o método `size` para ajudar a manter o controle do topo da pilha.

Solução 5.6 (**Exercício 5.6**)
O método `resize` pode ser usado para diminuir o vetor.

```java
public void trimToSize() {
  if (data.length != size)
    resize(size);
}
```

Solução 5.7 **(Exercício 5.7)**
Implemente a estratégia proposta e compare o tempo de execução da versão original e da nova versão. Plote os tempos para verificar o comportamento da curva de resposta (i.e. seu crescimento linear).

Solução 5.8 **(Exercício 5.8)**
O tempo de execução para inserir um novo elemento é $\mathcal{O}(n)$. Como n elementos são incluídos, o tempo de execução total é $\mathcal{O}(n^2)$.

Solução 5.9 **(Exercício 5.9)**
O método `resize` não pode ser utilizado.

```java
public void add(int i, E e) {
  checkIndex(i, size + 1);
  if (size == data.length) {
    E[] temp = (E[]) new Object[2*data.length];
    for (int k = 0; k < i; k++)
      temp[k] = data[k];
    temp[i] = e;
    for (int k = i + 1; k < size + 1; k++)
      temp[k] = data[k + 1];
    data=temp;
  } else {
    for (int k = size - 1; k >= i; k--)
      data[k + 1] = data[k];
    data[i] = e;
  }
  size++;
}
```

Solução 5.10 **(Exercício 5.10)**
Dica: modifique o método `push` de tal forma que ele modifique o tamanho quando necessário.

Solução 5.11 **(Exercício 5.11)**
Lembre-se de usar o método `equals` para testar a igualdade. A classe do elemento deve implementar esse método.

```java
public boolean contains(Object o) {
  for (int k = 0; k < size; k++)
    if (data[k].equals(o))
      return true;
  return false;
}
```

Solução 5.12 **(Exercício 5.12)**
Uma boa estratégia consiste em atualizar todas as referências para `null`.

```java
public void clear() {
  for (int k = 0; k < size; k++)
    data[k] = null;
  size = 0;
}
```

Solução 5.13 (Exercício 5.13)
Exercício de análise experimental.

6 Filas de prioridade

Solução 6.1 (Exercício 6.1)
$(1, D)$, $(3, J)$, $(4, B)$, $(5, A)$, $(2, H)$, $(6, L)$.

Solução 6.2 (Exercício 6.2)
A melhor estrutura de dados para uma simulação de controle de tráfego aéreo é uma fila de prioridade. Essa estrutura permite manipular os *timestamps* e manter os eventos em ordem, de tal forma que o evento com menor instante de tempo seja facilmente extraído.

Solução 6.3 (Exercício 6.3)
Mantenha uma variável adicional que referencie a entrada mínima atual. Isso permite executar a operação `min` em tempo constante $\mathcal{O}(1)$. Para que isso funcione, o método `insert` deve ser alterado, atualizando a variável adicional sempre que o novo elemento sendo inserido seja menor que `min`, bem como ao inserir quando a estrutura está vazia. O método `removeMin` também deve ser alterado, pois ele será responsável por identificar o novo elemento mínimo e atualizar a referência da variável adicional, para então remover o `min`.

Solução 6.4 (Exercício 6.4)
Não. A operação `removeMin` continua necessitando tempo linear $\mathcal{O}(n)$. Apesar do `min` atual ser facilmente encontrado e removido, tal método precisa percorrer todos os elementos restantes para identificar o novo mínimo.

Solução 6.5 (Exercício 6.5)
A solução é manter os *timestamps* nas entradas da fila de prioridades. Mantenha uma variável `minKey` inicializada com 0. Quando executar a operação `push` com um elemento e, chame `insert(minKey, e)` e decremente `minKey`. Na operação `pop`, chame `remove` e incremente `minKey`.

Solução 6.6 (Exercício 6.6)
A estratégia é similar ao exercício anterior. Mantenha uma variável `maxKey` inicializada com 0. Na operação de enfileirar um elemento e, chame `insert(maxKey, e)` e incremente `maxKey`. Na operação de desenfileirar, chame `removeMin`.

Solução 6.7 (Exercício 6.7)
Não. Contra exemplo:

Dada a sequência de operações `enqueue(A)`, `enqueue(B)`, `enqueue(C)`, `dequeue()`, `dequeue()`, `enqueue(D)`. A fila contém os elementos `C` e `D`, sendo `C` o mais antigo na estrutura. No entanto, a operação `dequeue()` removerá o elemento `D`, se usarmos a implementação proposta. ■

Solução 6.8 (Exercício 6.8)
`("Joey", 20)`, `("Joey", 20)`, `("Ross", 10)`, `("Phoebe", 15)`, `("Monica", 28)`.

7 Buscas em estruturas lineares

Solução 7.1 **(Exercício 7.1)**
Uma possível implementação é apresentada na Seção 5.1.3 (Busca Binária) de Goodrich et al. (2014) – *Data Structures and Algorithms in Java*.

Solução 7.2 **(Exercício 7.2)**
Com uma caneta, marque as referências para início e fim da sub-estrutura considerada em cada iteração da busca. Caso essas referências se cruzem, o elemento não foi encontrado. Caso a referência para o meio da lista aponte para o elemento buscado, seu índice é encontrado.

a) Busca binária: 3 avaliações (51, 27, 15). Busca sequencial: 1 avaliação (15).

b) Busca binária: 3 avaliações (51, 27, 33). Busca sequencial: 3 avaliações (15, 27, 33).

c) Busca binária: 3 avaliações (51, 71, 63). Busca sequencial: 6 avaliações (15, 27, 33, 46, 51, 63).

d) Busca binária: 3 avaliações (51, 71, 82). Busca sequencial: 9 avaliações (todos os elementos).

e) Busca binária: 3 avaliações (51, 27, 15). Busca sequencial: 9 avaliações (todos os elementos).

Solução 7.3 **(Exercício 7.3)**
Para implementar essa modificação, você deve parar a busca com sucesso quando o elemento buscado é igual ao elemento analisado. A busca continua se o elemento buscado for menor que o elemento analisado, e pára sem sucesso, caso contrário. Na prática, essa modificação torna o algoritmo mais eficiente nos casos em que ele pára antes de percorrer toda a lista. Porém, no pior caso o elemento buscado não está na lista e é maior que qualquer elemento dela, implicando na necessidade aa lista ser totalmente percorrida. Log, a complexidade continua sendo $\mathcal{O}(n)$ no pior caso.

Busca sequencial modificada simples em um *array* de inteiros:

```java
public int search(int[] array, int value) {
  for(int i = 0; i < array.length; i++) {
    if(array[i] == value)
      return i;
    if(array[i] < value)
      return -1;
    }
  return -1;
}
```

Busca sequencial modificada genérica em uma lista:

```java
public int indexOf_sorted(List<E> list, E value) {
  for(int i = 0; i < list.size(); i++) {
    if(comp.compare(list.get(i), value) == 0)
      return i;
    if(comp.compare(list.get(i), value) < 0)
      return -1;
  }

  return -1;
}
```

Solução 7.4 **(Exercício 7.4)**
Dica: use o código das buscas sequencial e binária em *arrays* genéricos para buscar o elemento desejado no *array* `data`. Para isso, deverá ser criado um atributo para armazenar o comparador, o qual deve ser usado para ambas as buscas.

Solução 7.5 (Exercício 7.5)
Analise as demostrações para entender os dois tipos de busca.

8 Ordenação de estruturas lineares

Solução 8.1 (Exercício 8.1)
Array inicial:

(5, 7, 4, 9, 8, 5, 6, 3)

Array após cada inserção:

(**5**, 7, 4, 9, 8, 5, 6, 3)
(**5**, **7**, 4, 9, 8, 5, 6, 3)
(**4**, **5**, **7**, 9, 8, 5, 6, 3)
(**4**, **5**, **7**, **9**, 8, 5, 6, 3)
(**4**, **5**, **7**, **8**, **9**, 5, 6, 3)
(**4**, **5**, **5**, **7**, **8**, **9**, 6, 3)
(**4**, **5**, **5**, **6**, **7**, **8**, **9**, 3)
(**3**, **4**, **5**, **5**, **6**, **7**, **8**, **9**)

Solução 8.2 (Exercício 8.2)
Basta inverter o operador relacional na comparação dos elementos. Ou seja, em vez de usar `comp.compare(element, array[index]) < 0`, usamos `comp.compare(element, array[index]) > 0`. Com isso, é fácil implementar um método de ordenação que recebe a ordem desejada ("normal" ou "inversa") como um parâmetro e executa a ordenação correspondente.

Solução 8.3 (Exercício 8.3)
Algoritmo bubble sort:

```
1  public void bubbleSort(E[] array) {
2    for (int lastIndex = array.length - 1; lastIndex > 0; lastIndex--) {
3      for (int index = 0; index < lastIndex; index++) {
4        if (comp.compare(array[index], array[index + 1]) > 0) {
5          E temp = array[index];
6          array[index] = array[index + 1];
7          array[index + 1] = temp;
8        }
9      }
10   }
11 }
```

Solução 8.4 (Exercício 8.4)
O bubble sort tem complexidade assintótica de tempo $\mathcal{O}(n^2)$ no pior, médio e melhor casos. É possível interromper o algoritmo quando uma passagem não faz nenhuma modificação, indicando que o *array* já está ordenado. Neste caso, a complexidade assintótica no melhor caso é reduzida para $\mathcal{O}(n)$.

Solução 8.5 (Exercício 8.5)

```
1  public boolean isSorted(E[] array) {
2    boolean sorted = true;
3    for (int index = 0; sorted && (index < array.length - 1); index++) {
```

```
4       if (comp.compare(array[index], array[index + 1]) > 0)
5         sorted = false;
6     }
7     return sorted;
8   }
```

Solução 8.6 **(Exercício 8.6)**
Ao usar uma busca binária, reduzimos a complexidade assintótica da busca pela posição de inserção do elemento de $\mathcal{O}(n)$ para $\mathcal{O}(\log n)$ no pior caso. Apesar dessa redução, o algoritmo ainda precisa deslocar elementos pela estrutura para efetivar a inserção, o que faz com que sua complexidade assintótica se mantenha em $\mathcal{O}(n^2)$ no pior caso. Além disso, a busca binária ainda executaria em $\mathcal{O}(\log n)$ no melhor caso (quando a estrutura já está ordenada), enquanto a busca sequencial executa em $\mathcal{O}(1)$. Portanto, a modificação proposta aumenta a complexidade do algoritmo de $\mathcal{O}(n)$ para $\mathcal{O}(n \log n)$ no melhor caso.

Solução 8.7 **(Exercício 8.7)**
As obras listadas na bibliografia da disciplina apresentam esses algoritmos.

Solução 8.8 **(Exercício 8.8)**
Analise as demostrações para entender os diferentes algoritmos.