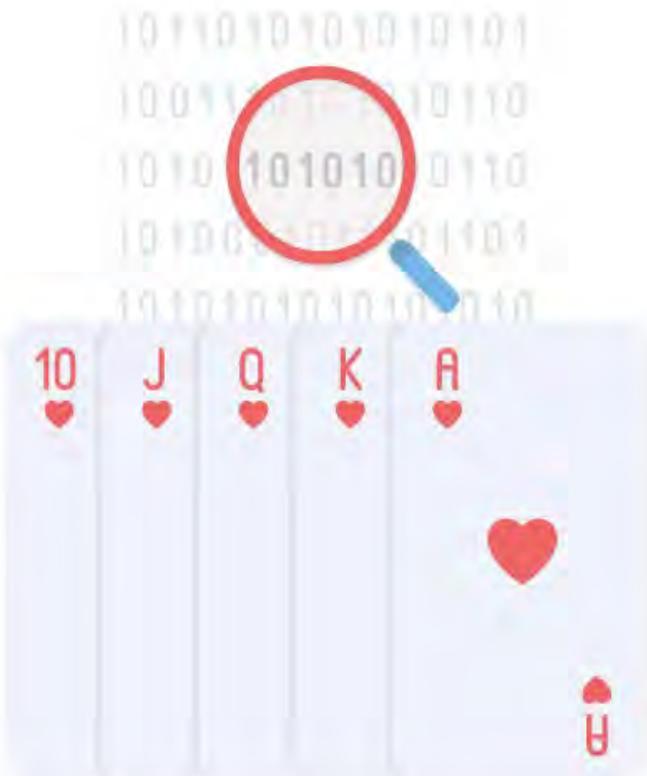


Algoritmos em Java

Busca, ordenação e análise



Casa do
Código

— ■ —
SÉRIE CAELUM

GUILHERME SILVEIRA

ISBN

Impresso e PDF: 978-85-5519-243-2

EPUB: 978-85-5519-244-9

MOBI: 978-85-5519-245-6

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br>.

SOBRE O AUTOR

Meu nome é Guilherme Silveira, e aprendi a programar aos 9 anos. Meu irmão foi meu professor particular, que me instigou a criar pequenos jogos em Basic. Trabalho com desenvolvimento de software desde meu primeiro estágio em 1996.

Apesar de ter trabalhado por um bom tempo com Visual Basic, C e C++, cresci como profissional utilizando a linguagem Java, que até hoje é a minha principal ferramenta. Entretanto, nunca me mantive restrito a ela — apesar do gosto pessoal.

Por gostar de ensinar computação desde criança, acabei por fundar junto ao Paulo Silveira a Caelum em 2004. Hoje sou responsável pelos cursos da Alura e escrevo livros para a Casa do Código.

Participei de diversas maratonas de programação, inclusive fazendo parte da equipe da USP ganhadora de medalhas de ouro nas competições sul americanas, e participando nas finais mundiais de programação no Japão e Canadá. Como a capacidade de pensamento lógico aplicada ao desenvolver algoritmos utilizados nas maratonas me ajudaram muito no dia a dia de desenvolvedor, aproveitei a oportunidade para compartilhar um conhecimento base de algoritmos de ordenação e busca para quem está começando na área de desenvolvimento de software, ou gostaria de entender como as coisas funcionam por trás.

Quem nunca jogou baralho, acompanhou os jogos da Copa do Mundo, precisou pesquisar preço para comprar um produto, tentou descobrir quem era o melhor dentro de um grupo de pessoas em uma competição, ou ainda usou uma agenda de contatos em papel ou digital? Todos esses são exemplos reais de ordenação e busca que

usamos no dia a dia, sem perceber!

Baseado nas maratonas de programação, trago exemplos reais de uso de ordenação e busca que temos diariamente. Também implemento algoritmos de uma maneira que tenta simular o que nós, seres humanos, já fazemos no nosso cotidiano.

PREFÁCIO

Minha história no ensino

Foi nas aulas da Massako, na sexta e sétima série, que tomei gosto por lógica e algoritmos. Como encontrar uma solução para uma equação parecida, mas não exatamente igual a que vimos antes? Como resolver um problema financeiro de uma família? Como descobrir qual empresa será mais importante após alguns anos? Para um garoto de 10 a 11 anos, o mundo real e a teoria matemática se encontravam ao tentar solucionar problemas do dia a dia de "gente grande".

No ano seguinte, tive minha primeira oportunidade de ensino de computação: viria a dar aula de Word para ela. Era o primeiro passo na tradição de meus pais, Carlos e Lúcia, e avó Ceres, de lecionar. Algumas famílias são de médicos, outras de policiais, outras de advogados. A minha é de educadores.

Trabalhando na Alemanha aos 18, tive a oportunidade de juntar o gosto que tinha por lógica e desenvolvimento de software com o ensino. Com um alemão quebrado, dei minha primeira aula, de multithreading em brokers de mensagens com Java, para alemães que nunca haviam programado. Quando cheguei à empresa após a aula, minha chefe Miriam me chamou e disse: "o que você fez?". Achei que vinha bronca... Mas eles adoraram.

Voltando ao Brasil, representando minha faculdade, quis seguir os passos de meu irmão, que participaria da maratona de programação. Então, me juntei ao time dele.

Também precisava de um emprego, novamente seguindo o trabalho que meu irmão fazia, queria dar aulas de Java. Foi assim

que criamos a Caelum.

De lá para cá, participei de diversos campeonatos. Minha equipe trouxe medalhas de ouro, prata e bronze para nossa faculdade e nosso país em campeonatos sul-americanos. Desenvolvi minha habilidade de raciocínio lógico e resolução de problemas.

Em paralelo, criamos a Alura, a Casa do Código, o Galandra e o Music Dot.

Durante esses anos, aprendi que minhas aulas podem ter algo a ser aproveitado pelos alunos. E descobri que fazer parte do desenvolvimento intelectual de um ser humano é algo muito gratificante.

Mas esses caminhos sempre andaram em paralelo:

1. a lógica através da capacidade de descrever e resolver um problema; e
2. o gosto pela educação.

Finalmente tenho uma oportunidade de juntar minha herança familiar com o gosto pessoal por algoritmos.

Este livro foi baseado em meu curso da Alura, que foi então transscrito pela Patrícia Alcântara, com revisão da Bianca Hubert e Vivian Matsui.

O caminho com algoritmos

Este livro foca em algoritmos de busca e ordenação, que aparecem em nosso dia a dia desde crianças. Se você já jogou buraco, truco ou rouba-monte, já ordenou as cartas na sua mão.

Se já torceu para um time de futebol durante a Copa do Mundo, já viu a tabela ordenada dos times por grupo, durante a primeira

fase.

Se você já usou uma lista telefônica ou um dicionário em papel, já fez uma busca "binária". Se você não é da geração do papel, ao procurar um contato na sua agenda do celular, você já deve ter feito a mesma busca.

Se você já teve, pelo menos, três notas de dinheiro em uma carteira, já as ordenou da maior para a menor, ou da menor para a maior, para ficar mais fácil encontrá-las.

Tudo isso sem perceber, sem piscar, sem pensar. Ordenação e buscas são tão naturais no nosso dia a dia que não percebemos sua presença ubíqua, e como o nosso cotidiano é mais fácil com elas.

Na computação, a realidade é similar: quer resolver problemas complexos de lógica? Muitos deles passam por conceitos básicos de ordenação e busca. O problema é tão famoso que matemáticos (cientistas da computação?) famosos estudaram e criaram maneiras geniais de como fazer tais tarefas consumindo menos tempo — ou menos memória, dependendo do caso.

Neste livro, veremos como funcionam, como implementar e como já usamos diversos desses algoritmos na nossa rotina.

Na prática, raramente um desenvolvedor implementa buscas básicas e ordenação em uma empresa; esse código já está feito para nós. Mas entendê-los e ser capaz de recriá-los nos leva a um conhecimento que é utilizado no dia a dia de todo programador: o pensamento lógico e a quebra de problemas em partes menores que podem ser resolvidas com algoritmos que conhecemos.

Passei 7 anos desenvolvendo algoritmos para os mais diversos problemas na minha equipe de maratona. Este livro é uma introdução para os problemas que atacávamos. Espero que eles sirvam para incentivá-lo a desenvolver seu lado de raciocínio lógico

e solucionador de problemas.

Público-alvo e pré-requisitos

Quem está estudando algoritmos de ordenação e busca na faculdade e gostaria de entendê-los de uma forma mais natural, como eles são usados no nosso cotidiano inconscientemente, poderá se beneficiar com o livro. Desenvolvedores que já estão acostumados a usar esses algoritmos, mas que queiram conhecer como eles funcionam por trás, e alunos que querem participar de maratonas de programação poderão tirar proveito do que compartilhamos aqui também.

Professores que ensinam introdução a algoritmos através de ordenação e busca poderão utilizar o livro como material didático para seus alunos, uma vez que cobrimos os principais algoritmos de ordenação, busca linear e binária.

Para acompanhar o conteúdo deste livro, basta conhecer a linguagem Java. Não há necessidade de conhecer Orientação a Objetos a fundo. Quem já domina outras linguagens não terá dificuldades em aprender o pouco que usamos de Java à medida que estuda o livro. Para quem está começando com programação e não conhece Java, será um pouco mais puxado. Dê uma folheada e veja se a maneira como a linguagem é mostrada o deixa confortável em seus estudos.

Sumário

1 Já conheço os problemas, mas quem são as soluções? Os algoritmos.	1
1.1 Encontrando o menor valor	1
1.2 O produto mais barato	3
1.3 Encontrando o mais barato na minha cabeça	4
1.4 O algoritmo do mais barato	8
1.5 Escrevendo o algoritmo	12
1.6 Criando o projeto	17
1.7 Transformando pseudocódigo em Java	21
1.8 Simulando encontrar o menor valor	24
1.9 Trabalhando com objetos	28
1.10 Detalhes de implementação na linguagem	33
1.11 Refatoração: extraindo uma função	35
1.12 Buscando do início ao fim de um array	37
1.13 Buscando somente em um trecho específico de um array	39
1.14 O próximo desafio: os mais baratos	44
2 Como a ordenação pode nos ajudar	46
2.1 Os n mais baratos	46
2.2 Algoritmo básico (e ruim?) para encontrar o n mais barato	47
2.3 A importância da ordenação	49

2.4 O próximo desafio: ordenação	51
3 Selecionando e ordenando	52
3.1 Como ordenar?	52
3.2 Ordenando ao selecionar o mais barato	52
3.3 Analisando cada passo	54
3.4 Implementando em Java a ordenação pela seleção do menor valor	55
3.5 Algoritmos e o menos 1	63
3.6 Extraindo a ordenação por seleção de elementos	64
3.7 Visualizando a troca de posições dos elementos durante a seleção	67
3.8 Simulando no papel o algoritmo de seleção de menores elementos para ordenação	70
3.9 Selection Sort	74
3.10 O próximo desafio: questionando a velocidade de um algoritmo	75
4 Inserindo e ordenando: o Insertion Sort	77
4.1 Ordenando cartas de baralho	77
4.2 Simulando passo a passo a ordenação de cartas de baralho	80
4.3 Pensando no algoritmo de ordenação	82
4.4 Implementando a ordenação por inserção	92
4.5 Logando as informações da ordenação por inserção	96
4.6 Pequenas refatorações e melhoria do código	103
4.7 Simulando no papel com o nosso código	109
4.8 Insertion Sort	117
4.9 O próximo desafio: assentando o conhecimento	118
5 Alguns componentes de um algoritmo	119
5.1 Algoritmos: entrada e saída	119
5.2 Reduzindo um problema a outro	120

5.3 O próximo desafio: comparar nossos algoritmos	123
6 Comparando algoritmos e a análise assintótica	124
6.1 Como analisar o desempenho de algoritmos?	124
6.2 Os algoritmos	125
6.3 Analisando o buscaMenor	126
6.4 Criando uma tabela de operações por desempenho de algoritmo	128
6.5 Gráfico de um algoritmo linear	131
6.6 Analisando o Selection Sort	134
6.7 Criando a tabela de operações de um algoritmo quadrático	136
6.8 O gráfico de um algoritmo quadrático	138
6.9 Comparando o desempenho dos algoritmos	140
6.10 Como comparar o desempenho do algoritmo	144
6.11 Comparando o desempenho do algoritmo em um computador	145
6.12 A análise assintótica	149
6.13 Analisando o Insertion Sort	152
6.14 Algoritmos que rodam com o tempo constante	157
6.15 Algoritmos com desempenho baseado em log	160
6.16 Algoritmos $n \log n$	162
6.17 Algoritmos cúbicos	164
6.18 Algoritmo exponencial	165
6.19 Análise assintótica de um algoritmo	166
6.20 O próximo desafio: novos algoritmos	168
7 Intercalando arrays pré-ordenados	169
7.1 Dividindo o trabalho	170
7.2 Conquistando o resultado	171
7.3 Como juntar ou intercalar duas listas ordenadas	172
7.4 Simulando com as variáveis	179

7.5 Criando o projeto e preparando para juntar ordenado	183
7.6 Implementando o junta/intercala	188
7.7 Procurando um erro em um algoritmo	194
7.8 Intercalando os elementos que sobraram	197
7.9 Intercalando os elementos que sobraram, independente do lado	199
7.10 Pequenas refatorações possíveis	200
7.11 O próximo desafio: intercalando em um único array	206
8 Intercalandoo baseado em um único array	208
8.1 Implementando o único array	211
8.2 Simulando o método intercala em um único array	217
8.3 O problema de intercalar parte de um array	224
8.4 Copiando parte do array em Java	226
8.5 O próximo desafio: outras invocações ao intercala()	232
9 Diversas simulações do intercala()	234
9.1 Tamanhos válidos para o intercala	237
9.2 Rodando as variações do intercala	242
9.3 Intercalando um trecho pequeno	245
9.4 O próximo desafio: intercalando diversas vezes seguidas	254
10 O algoritmo Merge Sort	255
10.1 Intercalando até o fim	286
10.2 Intercalando valores inválidos em Java	292
10.3 Intercalando diversas vezes	296
10.4 Intercalando passo a passo	300
10.5 O merge sort	302
10.6 O próximo desafio: outra maneira de entender o que é ordem	307
11 Encontrando a posição relativa de um elemento	308
11.1 Simulando quantos são menores	309

11.2 Implementando o encontra menores	316
11.3 O próximo desafio: colocando um elemento em seu lugar	320
12 Colocando um elemento no seu lugar: o pivô	322
12.1 O pivô da separação	325
12.2 Variáveis para particionar	326
12.3 Colocando um elemento no seu lugar do array	330
12.4 Implementando o código de posicionamento	337
12.5 O próximo desafio: e os outros elementos?	342
13 Pivotando um array por completo	343
13.1 Verificando a mudança de posição	343
13.2 Simulando a partição	345
13.3 Pivota coloca na posição	353
13.4 O próximo desafio: pivotando mais vezes	354
14 O quick sort	355
14.1 Particionar após particionar, e depois particionar novamente	
14.2 Ordenando através das partições	355
14.3 Quem é este algoritmo esperto?	366
14.4 O próximo desafio: a busca	366
15 A busca linear	368
15.1 A ideia da busca linear	369
15.2 Implementação da busca linear	371
15.3 O próximo desafio: uma busca mais rápida	373
16 A busca binária	375
16.1 Buscando em um array ordenado	376
16.2 Dividindo o problema em dois para depois buscar	377
16.3 Dividindo, dividindo novamente e dividindo mais uma vez	
16.4 Implementando a busca pela metade	380

16.5 Desempenho ao dividir e buscar	386
16.6 Quando não encontramos um elemento	388
16.7 Definindo a busca binária	392
16.8 O próximo desafio: comparando as buscas	393
17 Análise assintótica das buscas	394
17.1 O desempenho de busca binária	396
17.2 Analisando a busca binária	401
17.3 O próximo desafio: comparar as ordenações	405
18 Análise assintótica das ordenações	406
18.1 O desempenho de merge sort	408
18.2 Comparando o Merge Sort com outros sorts	409
18.3 Analisando o Merge Sort	414
18.4 Analisando o particiona	416
18.5 Desempenho do Quick Sort	418
18.6 Comparando o Quick Sort com o Merge Sort	419
18.7 Quicksort	421
19 Conclusão	422
19.1 Como continuar os estudos	424

CAPÍTULO 1

JÁ CONHEÇO OS PROBLEMAS, MAS QUEM SÃO AS SOLUÇÕES? OS ALGORITMOS.

1.1 ENCONTRANDO O MENOR VALOR

Aqui nós vamos falar sobre soluções para diversos problemas. Eu sempre quero resolver os meus problemas, mas que tipo de problemas quero resolver aqui?

Vou resolver problemas com ordem, quem é mais caro, quem é mais barato. Qual o menor caminho? Qual é o melhor produto?

Todos esses problemas envolvem dizer "que coisa é maior do que outra coisa". Qual é o hotel mais próximo? Qual é o cinema mais próximo? Qual o horário do próximo trem que sairá? Tudo isto envolve uma pergunta do tipo "**quem**" ou "**algo**", que é "**mais**" ou "**menos**". Envolve **ordenação**.

- Quem ganhou o jogo? Quem fez mais gols.
- Quem ganhou o campeonato? Quem fez mais pontos.
- Quem perdeu o campeonato e foi rebaixado? Quem fez menos pontos.

Tudo isso envolve um conjunto de itens, um monte de coisa, um

monte de times de futebol, um monte de produtos que estão à venda, vários corredores em uma maratona. Envolve pegar todos esses itens e ordenar. Definir quem ficou em primeiro, quem ficou em segundo, quem ficou em terceiro, quem ficou em último.

Quem são as pessoas que passaram e não passaram no vestibular? Quem foi reprovado? Quem passou e não passou na prova? Tudo isso envolve dizer **algo que é maior*** e algo que é menor**. Envolve ordenar as pessoas, os produtos, as coisas. Dar uma ordem para tudo.

Muitas perguntas que fazemos estão ligadas a uma **ordem**. Se você precisa buscar um hotel, quer encontrar um hotel que seja próximo de uma determinada região da cidade. Você vai buscar primeiro os que são mais baratos, depois os que são mais caros. Ou você começa buscando por aqueles que têm uma determinada característica e, em seguida, procura os que são mais baratos.

Pare para pensar: ao nosso redor, tudo tem uma questão de **ordem**. Se você quer ir para o trabalho, você quer o ônibus que vai chegar mais rápido, em menos tempo. Ou se você está em casa, quer pegar o primeiro ônibus, que está mais próximo da saída. Tudo isso envolve ordenar um monte de coisas.

Isso é o que faremos aqui: aprender a ordenar os maiores, os menores, os melhores. Não importa. Vamos aprender a encontrar aquilo que procuramos, com uma ordem. O mais rápido, o melhor, o com maior nota, o com menor e maior tempo. Tudo isso conseguimos fazer com o que vamos aprender aqui no livro.

Os **algoritmos** são soluções, maneiras de resolver esses problemas.

É isso que veremos em seguida.

1.2 O PRODUTO MAIS BARATO

Primeiro problema do meu dia a dia: eu quero comprar um carro. Vou entrar em um site para pesquisar preços de carros usados e ele me trouxe esses preços:

- Uma Lamborghini por R\$ 1.000.000
- Um Jipe por R\$ 46.000
- Uma Brasília por R\$ 16.000
- Um Smart por R\$ 46.000
- Um Fusca por R\$ 17.000



Figura 1.1: Lista de carros

Imagine que está difícil procurar um carro nesse site, pois tenho muitas opções, com diferentes faixas de preço. No meu caso, estou procurando o carro mais barato, o que tem o **menor número possível**.

Em diversas situações, eu vou procurar este número. Por exemplo, quando eu quero encontrar outro produto **mais barato**, como um livro. Três livrarias diferentes vendem este livro e eu quero comprar o mais barato das três.

Outros exemplos:

1. Eu quero comprar um brinquedo e entro em um site que compara os preços do produto em diversas lojas. Eu quero saber: qual é o mais barato deles?
2. Quantos erros cada aluno cometeu? Quem cometeu menos

erros?

3. Qual ônibus chegará mais rápido no meu destino?

Nestes casos, eu quero saber qual é o **menor número possível**.

No exemplo dos carros, a primeira coisa que quero saber é "qual é o mais barato de todos?". Não adianta sugerir uma **Lamborghini**, se o carro custa R\$ 1 milhão e eu quero o mais barato.

Nós temos os dados dos carros: cada um tem *nome* e *preço*. E temos cinco opções de carros. Qual tem o menor preço? Dê uma olhada e me responda.

Você já viu os preços e sabe qual é o carro mais barato. A Brasília tem o menor preço:



Figura 1.2: O menor preço

É bem provável que você tenha respondido rápido. Agora, eu quero saber: o que você pensou para conseguir responder bem rápido? Pense como foi e escreva (também, vale dizer em voz alta). Responda e eu já vou lhe contar o que fiz para concluir qual era o mais barato dos cinco carros na lista.

1.3 ENCONTRANDO O MAIS BARATO NA MINHA CABEÇA

Como você resolveu o problema "*Qual é o carro mais barato?*"? Quer saber como eu resolvi?

Eu olhei todos os carros e vi qual era o mais barato? Não foi assim. Meu pensamento passou por etapas antes disso. Houve um processo, um algoritmo que rodamos na nossa cabeça. Por exemplo, eu olhei para todos os carros (sem isto, eu não descubro qual é o mais barato):



Figura 1.3: Lista de carros

Depois, olhei um de cada vez.

- Primeiro carro: a Lamborghini custa R\$ 1.000.000. É o carro mais barato que eu conheço até este momento.



Lamborghini
R\$ 1.000.000

Figura 1.4: Lamborghini baratinha

- Segundo carro: o Jipe custa R\$ 46.000. É mais barato do que a opção anterior.



Jipe
R\$ 46.000

Figura 1.5: Mas o Jipe é mais barato

- Terceiro carro: a Brasília custa R\$ 16.000. Ela é mais barata? É mais barata do que o carro de R\$ 46.000.



Brasília
R\$ 16.000

Figura 1.6: Mas a Brasília é mais barata

- Quarto carro: o Smart custa R\$ 46.000. Este carro é mais caro do que a Brasília, de R\$ 16.000, o carro mais barato até agora.



Smart
R\$ 46.000

Figura 1.7: O Smart não é mais barato que a Brasília

- Quinto carro: o Fusca custa R\$ 17.000. Também é mais caro do que a Brasília.



Fusca
R\$ 17.000

Figura 1.8: O Fusca também não é

Então, eu olhei todos os carros, sempre comparando o atual com o mais barato até o momento.

Por exemplo, quando analiso o Jipe, que custa R\$ 46.000, e comparo com a Lamborghini, que custa R\$ 1.000.000, o Jipe será o carro mais barato. Quando eu faço a comparação com outro carro *mais caro*, eu ignoro e escolho o carro *mais barato*. O processo é feito rapidamente. Eliminei as opções mais caras e fiquei apenas com a Brasília, de R\$ 16.000.

As etapas do processo ficam ainda mais claras quando aumentamos o número de opções. Se compararmos 50 ou 100 carros, teremos maior dificuldade em memorizar qual é o mais barato até agora.

Se você precisar descobrir qual é o produto mais barato entre 100 produtos, é provável que queira fazer anotações. Isto porque é difícil processar rapidamente quando o número de dados é muito grande.

Então, o que podemos fazer? Podemos anotar quais são as nossas opções e escrever qual carro é o mais barato até o momento.



Figura 1.9: O carro mais barato até agora, o atual

Eu anotarei qual é a melhor opção **atual** (entre os carros 0, 1, 2, 3 ou 4) e qual é o carro **mais barato** até o momento. Se o número de itens que vou comparar é pequeno, eu não preciso anotar. Mas se o número for grande, eu escolho fazer anotações.

Eu denominei os carros como "0, 1, 2, 3, 4", porque, em programação, nós sempre começamos uma coleção (um *array* de elementos) com a **posição 0**.

Experimente fazer esse processo visualmente. Escreva as cinco opções de carro no papel. Compare os elementos entre si e anote qual possui o menor preço. Por exemplo: "entre o carro 1 e 2, o mais barato é...", e assim por diante. Faça o exercício.

1.4 O ALGORITMO DO MAIS BARATO

Quando executamos o processo com calma, no papel, percebemos que nosso pensamento é bem rápido. Quando nós executamos o processo, como por exemplo, soma, multiplicação, divisão simples ou qualquer tipo de algoritmo simples, não notamos que, mentalmente, estamos fazendo um monte de coisas malucas.

Sem perceber, fazemos várias contas. Por exemplo:

- O número 17 é par ou ímpar?
- O número 30 é divisível por 10?

Nós respondemos automaticamente, mas, na nossa cabeça, nossos pensamentos estão passando por vários processos. A mesma coisa acontece no exemplo dos carros.

Quando quero encontrar o maior e o menor preço (um **algoritmo**), faço várias contas, mentalmente. Quando transcrevo um programa para o algoritmo, eu dito o processo passo a passo para o programa. É o que faremos agora: vamos simular com imagens o processo e depois faremos a transcrição para o código.

Vamos começar com o primeiro carro da lista: o **carro 0**. Como ele é o primeiro, o **carro atual** será também o **mais barato**.



Figura 1.10: Iniciando o algoritmo com atual sendo o mais barato: o carro inicial

O próximo elemento da lista é o **carro 1**. O Jipe (que custa R\$ 46.000) é mais barato do que o **carro 0** (que custa R\$ 1.000.000)? Sim, é mais barato. Então, vamos anotar que o **carro 1** é o mais barato.



Figura 1.11: O carro 1 é o mais barato

Vamos analisar a Brasília, que custa R\$16.000. O carro 2 é o atual. Ele é mais barato que o anterior (que custa R\$46.000)? Sim. Vamos anotar que o carro 2 é o mais barato.



Figura 1.12: O carro 2 é o mais barato

Agora, vamos para o carro 3 . O "carro atual" (que custa R\$46.000) é mais barato do que o carro 2 ? Não. Então, não faremos alterações.



Figura 1.13: O carro 3 é mais caro

O próximo elemento é o carro 4 . Ele é mais barato do que o carro 2 ? Não. Então, não faremos alterações.



Figura 1.14: O carro 4 é mais caro

Vamos para o carro 5 . Não temos mais carros para analisar! Logo, acabou.



Figura 1.15: O carro 5 não existe

Temos, então, que o **carro 2** é o mais barato!

O que nós fizemos no exemplo? Nós começamos com as variáveis **atual** e **mais barato**, sendo **0** (a posição do primeiro carro). Em seguida, analisamos cada elemento, ou seja, fomos com o contador atual de **0** até o número total de carros. Verificamos: "Esse carro é mais barato, sim ou não?". Nos casos em que era, anotamos qual era o mais barato.

Este é o processo que fizemos graficamente. Agora, vamos transformá-lo em código? Será o nosso próximo passo.

1.5 ESCREVENDO O ALGORITMO

Vamos passar a limpo o processo que executamos na nossa mente e que criamos graficamente. Como faremos isto?

Primeiro, você lembra do que definimos? Nós definimos dois valores: o do produto **atual** que estamos analisando no nosso array (que iniciamos com **0**) e do **mais barato** até agora (que também é o produto na posição **0**). Isto é, tanto **atual** como **maisBarato** são

iguais a 0 .

```
maisBarato = 0  
atual = 0
```

Na nossa memória, nós analisamos exatamente assim: `atual` é igual a 0 e `maisBarato` é igual a 0 . Este foi o início do nosso processo (o nosso **algoritmo**).

Depois, o que fizemos? Nós começamos a analisar cada produto dentro do array. Isto significa que, se eu tenho cinco produtos, eu vou do 0 até 4 inclusive ou do 0 até 5 exclusive. Então, para `atual`, vou do 0 até 4 inclusive.

```
maisBarato = 0  
atual = 0  
  
para atual = 0 até 4 inclusive {  
  
}
```

Depois disso, se o preço do `atual` for menor do que o `maisBarato` , significa que ele é o produto mais barato do que encontrei até agora. Então, nós trocaremos o `maisBarato` e especificaremos qual elemento vai substituí-lo. O produto **mais barato** é o elemento `atual` , que tem o menor preço. Esse é o código que terei:

```
maisBarato=0  
atual = 0  
  
para atual = 0 até inclusive {  
    se precos[atual] < precos[maisBarato] {  
        maisBarato = atual  
    }  
}
```

Agora, vamos comparar o preço do produto 0 (que custa R\$1.000.000) com ele mesmo. Logo, não faremos alterações.



Figura 1.16: Passo 1



Figura 1.17: Variável zerada

O próximo é o produto 1. O **atual** será igual a **1**, que custa R\$46.000. Este preço é menor do que o **maisBarato**, que agora é igual a **0**? O valor R\$ 46.000 é menor do que R\$ 1.000.000? Sim, entra no **maisBarato=atual**. Eu vou atribuir o valor **1** no **maisBarato**.



Figura 1.18: Passo 2



Figura 1.19: Variável 1

Seguimos para o produto **2**. O preço do carro 2 (que custa R\$ 16.000) é menor do que o preço do carro 1 (que custa R\$ 46.000)? Sim, então, vamos substituir o `maisBarato` pelo `atual`.



Figura 1.20: Passo 3



Figura 1.21: Variável 2

O próximo elemento é o carro 3. Vamos alterar o `atual` para **3** e comparar o valor R\$ 46.000 com o valor do `maisBarato`, que custa R\$ 16.000. O preço do produto **3** é menor do que **2**? Não. Vamos para outro elemento.



Figura 1.22: Passo 4



Figura 1.23: Variável 3

Agora, o `atual` é igual a `4`. O preço do produto é R\$46.000. Este preço é menor do que o valor do `maisBarato` até o momento? Não. Seguimos.



Figura 1.24: Passo 5



Figura 1.25: Variável 4

E com `atual` igual a 5? Ele não está no intervalo em que estamos interessados. Nós vamos parar por aqui.

Já encontramos o produto mais barato da lista. É o elemento que está na posição 2, a `Brasília`, que custa R\$16.000.



Figura 1.26: Passo 6



Figura 1.27: Variável 5

```
maisBarato=0
atual = 0

para atual = 0 até inclusive {
    se precos[atual] < precos[maisBarato] {
        maisBarato = atual
    }
}
```

Este é o nosso **pseudoalgoritmo** para encontrar o valor mais barato e descobrir o menor valor de um array.

1.6 CRIANDO O PROJETO

Nós já vimos como executar o processo, o **algoritmo**, que é uma sequência de tarefas para detectar o elemento mais barato dentro de um array de produtos.

Meu array tem cinco posições, indo do *0* até *4*, criando uma variável **atual** e atualizando-a (sempre verificando se o preço do produto **atual** é menor do que o **mais barato**). Agora, vamos escrever isto em *Java*?

Criarei um projeto chamado "produtos". Dentro do meu projeto, vou criar a Classe, ou *class TestaMenorPreco*. O pacote que vou usar é *br.com.alura.algoritmos*.

Agora, eu tenho a minha classe:

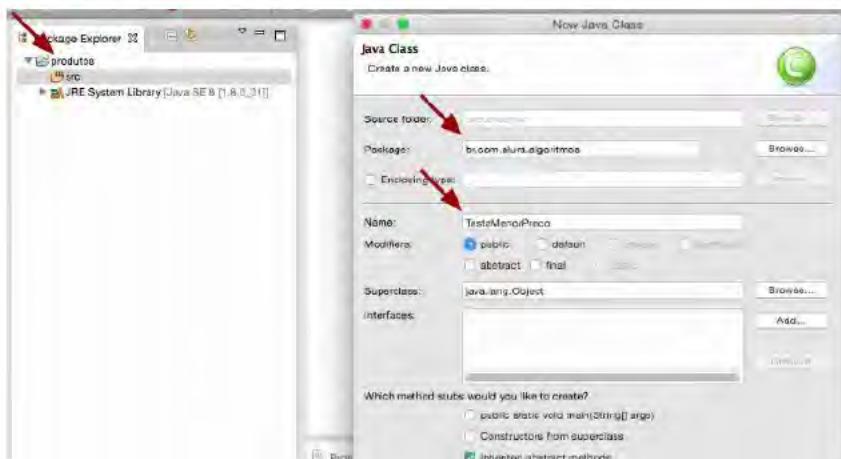


Figura 1.28: Java class Produto

Vou colocar o método `main()` de execução do nosso programa. Inicialmente, o que precisamos inserir dentro dele são os preços dos carros. Para isso, vou criar um array de preços, com cinco posições. Vou defini-las passo a passo.

Os preços dos carros são:

- carro 0 custa R\$ 1000000;
- carro 1 custa R\$ 46000;
- carro 2 custa R\$ 16000;
- carro 3 custa R\$ 46000;

- carro 4 custa R\$ 17000.

Então meu código fica assim:

```
package br.com.alura.algoritmos;
public class TestaMenorPreco {

    public static void main(String[] args) {

        double precos[] = new double[5];
        precos[0] = 1000000;
        precos[1] = 46000;
        precos[2] = 16000;
        precos[3] = 46000;
        precos[4] = 17000;
    }
}
```

O que eu desejo saber? Qual elemento tem o menor preço? Para fazer isso, nós criamos uma variável que armazena aquele que é mais barato. Tanto `maisBarato` como `atual` são variáveis inteiras que começam pelo `0`. Como esta é uma variável inteira, eu continuo com os produtos 1, 2, 3 e 4.

Em seguida, vou executar uma tarefa para os diferentes valores da variável `atual`, de `0` até `4`. O `4` será *inclusive*, porque eu quero incluir o preço do Fusca (o carro 4).

Do `0` até `4`, o que nós queremos fazer? Nós queremos criar a condição: "se o preço do `atual` for menor do que o do `maisBarato`, o preço do `maisBarato` será igual ao preço do produto `atual`".

```
double precos[] = new double[5];
precos[0] = 1000000;
precos[1] = 46000;
precos[2] = 16000;
precos[3] = 46000;
precos[4] = 17000;

int maisBarato = 0;
int atual = 0;
executo do 0 ate 4 inclusive {
```

```
        se preco do atual < preco do mais barato
            mais barato = atual
    }
```

Toda vez que nós executarmos o *loop*, precisamos atualizar o *atual* e, só então, seguimos para o próximo produto. Teremos de somar 1 no *atual*, ou seja, *atual +1*.

```
double precos[] = new double[5];
precos[0] = 1000000;
precos[1] = 46000;
precos[2] = 16000;
precos[3] = 46000;
precos[4] = 17000;

int maisBarato = 0;
int atual = 0;
executo do 0 ate 4 inclusive {
    se preco do atual < preco do mais barato {
        mais barato = atual
    }
    atual = atual + 1
}
```

Continuamos com o próximo produto, independentemente se ele é ou não o mais caro. O que fizemos até agora: definimos o *maisBarato* e o *atual* igual a 0. Executamos o código do 0 até 4, *inclusive*.

Se o preço do *atual* for menor do que o preço do *maisBarato*, nós definiremos qual é o novo elemento *maisBarato*. Caso contrário, não modificamos. E seguimos para o próximo. No fim, pedimos para o programa: *imprime o maisBarato e o preço do mais barato*. Nossa código ficará assim:

```
package br.com.alura.algoritmos;

public class TestaMenorPreco {

    public static void main(String[] args) {

        double precos[] = new double[5];
        precos[0] = 1000000;
        precos[1] = 46000;
```

```

    precos[2] = 16000;
    precos[3] = 46000;
    precos[4] = 17000;
    int maisBarato = 0;
    int atual = 0;
    executo do 0 ate 4 inclusive {
        se preco do atual < preco do mais barato {
            mais barato = atual
        }
        atual = atual + 1
    }
    imprime o mais barato
    imprime o preco do mais barato
}
}

```

Vamos traduzir este código para Java? Será o nosso próximo desafio.

1.7 TRANSFORMANDO PSEUDOCÓDIGO EM JAVA

Nós já escrevemos um código que é quase *Java*. Na verdade, o que fizemos é um pseudoJava. Ele ainda não funciona, mas já nos dá uma ideia do que está acontecendo.

```

int maisBarato = 0;
int atual = 0;
executo do 0 ate 4 inclusive {
    se preco do atual < preco do mais barato {
        mais barato = atual
    }
    atual = atual + 1
}
imprime o maisbarato
imprime o preco do mais barato

```

O que estou querendo fazer aqui? Quero construir um "laço" de *0 até 4*. Comumente, nós vamos usar o laço `for`, desde `atual = 0` (e por isso, não vou precisar da definição do `atual`) até que ele seja menor ou igual a 4. Somo `+1` ou digito `atual ++`, e removo a linha: `atual = atual + 1`.

```
int maisBarato = 0;
for(int atual = 0; atual <= 4; atual++) {
    se preco da atual < preco do mais barato {
        mais barato = atual
    }
}
imprime o maisbarato
imprime o preco do mais barato
```

Agora, o que eu tenho que fazer? Aqui dentro estou verificando se o preço do `atual` é menor do que o preço do `mais barato`:

```
if(precos[atual] < precos[maisBarato])`
```

Isso significa que o **mais barato** é o elemento **atual**:

```
maisBarato = atual
```

Então, nosso código fica:

```
int maisBarato = 0;
for(int atual = 0; atual <= 4; atual++){
    if(precos[atual] < precos[maisBarato]) {
        maisBarato = atual;
    }
}
imprime o maisbarato
imprime o preco do mais barato
```

Depois, eu imprimo "O carro mais barato custa" + `precos[maisBarato]`, e o nosso código ficará assim:

```
int maisBarato = 0;
for(int atual = 0; atual <= 4; atual++){
    if(precos[atual] < precos[maisBarato]) {
        maisBarato = atual;
    }
}
System.out.println(maisBarato);
System.out.println("O carro mais barato custa" + precos[maisBarato]);
```

Vamos testar nosso código? Clique no botão direito do mouse, e depois em *Run As* e *Java Application*:

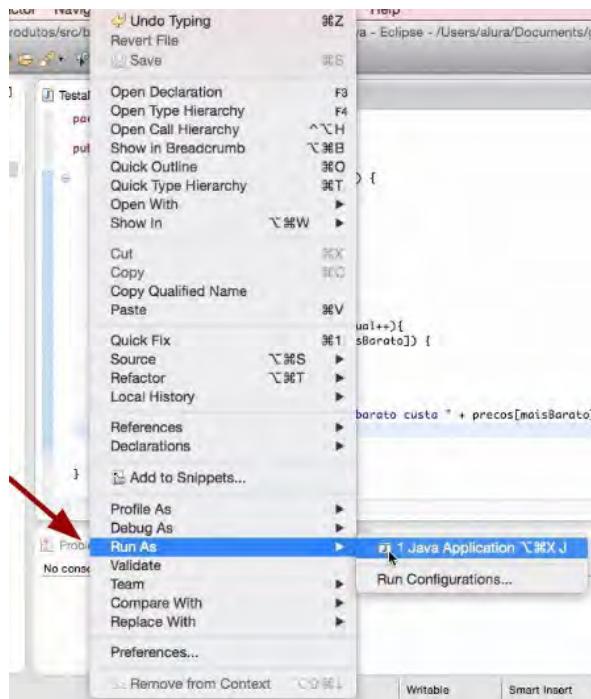


Figura 1.29: Testando o código

Na tela, aparecerá que o carro 2 é o produto mais barato:

```
2
0 carro mais barato custa 16000.0
```

O computador consegue executar o algoritmo que passa em nossa mente e do qual somos inconscientes. Observe que, quando nós executamos o algoritmo mentalmente, nós olhamos os cinco produtos e, de imediato, identificamos qual era o carro mais barato. Como? Nós comparamos cada produto e apontamos qual era o mais barato.

O computador faz a mesma coisa: ele anota na memória qual é o produto mais barato e qual elemento foi analisado. Depois, ele armazena o produto **atual** e o **mais barato**, e segue verificando qual tem o menor preço.

Nós fizemos a mesma coisa de forma instantânea, porque eram poucos produtos. Caso trabalhássemos com 500 produtos, provavelmente nós faríamos anotações com as variáveis `atual` e `maisBarato`.

Adiante, vamos simular esse algoritmo graficamente junto com o código.

1.8 SIMULANDO ENCONTRAR O MENOR VALOR

Agora que nós temos o nosso código Java, vamos simular o algoritmo com o programa.

```
int maisBarato = 0;
for(int atual = 0; atual <= 4; atual++) {
    if(precos[atual] < precos[maisBarato]) {
        maisBarato = atual;
    }
}
```

Começaremos com as nossas variáveis `atual` e `maisBarato` iguais a **0**. De acordo com o nosso laço: `atual <= 4`.

O produto **0** é `<=4`? Sim. Então, ele entra no nosso laço.

```
for(int atual = 0; atual <= 4; atual++) {
    if(precos[atual] < precos[maisBarato]) {
```

Ao analisarmos o primeiro carro, `atual` é igual a **0** e o preço do produto é R\$1.000.000. O carro **mais barato** também será o `atual`, então, o preço mais barato será R\$ 1.000.000. Logo, não vamos modificar o `maisBarato`.



Figura 1.30: Mantendo o mais barato

Seguimos para o próximo carro.

Quando verificamos o carro 1, o valor de `atual` será menor do que 4 e continuamos dentro do nosso laço. O preço do produto (que custa R\$ 46.000) é menor do que o do mais barato até o momento (R\$ 1.000.000)? Sim, ele é menor. Podemos entrar dentro do `if`. Vamos trocar o valor de `maisBarato` por 1, que é o `atual`.



Figura 1.31: Trocando o mais barato

No carro 2, o valor de `atual` é ≤ 4 ? Sim, continua dentro do laço. O preço do carro 2 (R\$ 16.000) é menor do que o `maisBarato` (R\$ 46.000)? Sim, então:

```
maisBarato = atual`
```

Seguimos para o carro 3. Ainda `atual` ≤ 4 , logo:

```
if(precos[atual] < precos[maisBarato])
```



Figura 1.32: É mais caro

O preço do `atual` (R\$ 46.000) é menor do que o `maisBarato` (R\$ 16.000)? Não, então não faremos modificações.



Figura 1.33: Posso continuar?

Próximo carro. Atual é ≤ 4 ? Sim. O preço atual (que custa R\$ 17.000) é menor do que o maisBarato (que custa R\$ 16.000)? Não, então, não modificamos o maisBarato .



Figura 1.34: É mais caro

Assim, atual++ e passamos para o carro 5 . atual ≤ 4 ? Não. Logo, acabou o laço.



Figura 1.35: Encontramos o mais barato

O que o nosso **algoritmo** fez? Ele executou o mesmo processo que fizemos ao observar os cinco elementos e armazenou na memória "quem" estamos analisando e "qual" é o valor menor até agora.

Fizemos isto na nossa mente. Enquanto estávamos observando os produtos, nosso **olho** foi a variável `atual` e o nosso **dedo** foi a variável `maisBarato`, que indicava o produto com o menor preço. Porém, como temos poucos produtos, o processo é feito rapidamente, de forma inconsciente. Apenas observamos e, logo, identificamos o carro mais barato.

Agora o computador também identifica imediatamente qual é o mais barato, pois ele já sabe qual é o processo, isto é, qual é o **algoritmo de detecção** dentro de um array.

1.9 TRABALHANDO COM OBJETOS

Nós sabemos que, quando trabalhamos com Java, não precisamos necessariamente utilizar tipos primitivos. Temos a opção de criar novas classes e objetos. Como nossos elementos têm

nome e preço, não faz sentido que um produto seja um *double*. Por exemplo, a Lamborghini, o Smart, o Fusca, todos têm um nome.

Desejamos obter um produto com nome e com preço. Então, em vez de ter um array de *double*, eu vou ter um array de produtos. Vamos, então, criar uma classe de **Produtos**? A classe **Produto** terá um nome e um preço:

```
public class Produto {  
  
    private String nome;  
    private double preco;  
  
}
```

Quando construiremos um produto, vamos passar tanto o nome quanto o seu preço para a nossa classe. Isto significa que, em vez de ter um array de *double*, teremos um array de cinco produtos.

```
Produto produtos[] = new Produto[5];
```

O primeiro item do array será uma Lamborghini, que custa R\$ 1.000.000:

```
Produto produtos[] = new Produto[5];  
produtos[0] = new Produto("Lamborghini", 1000000);
```

Ainda não existe o construtor que recebe o **nome** e o **preço**. Clico em **Ctrl + 1** e depois, em *create constructor*, e o programa criará o construtor para mim:

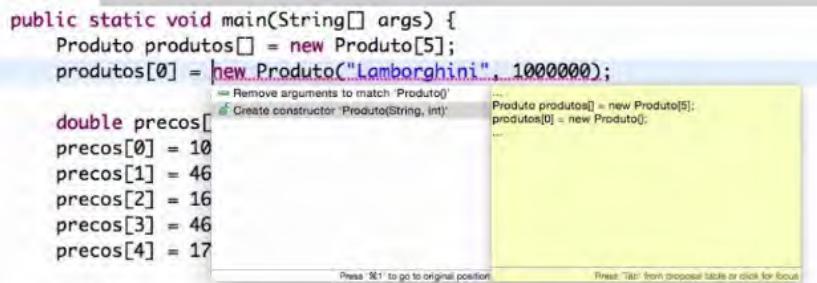


Figura 1.36: Construtor

```
public class Produto {  
  
    private String nome;  
    private double preco;  
  
    public Produto(String nome, double preco) {  
        this.nome = nome;  
        this.preco = preco;  
    }  
}
```

A Lamborghini ocupará a posição 0 na lista. Nas posições seguintes, teremos:

- Posição 1 : Jipe que custa R\$ 46.000;
- Posição 2 : Brasilia que custa R\$ 16.000;
- Posição 3 : Smart que custa R\$ 46.000;
- Posição 4 : Fusca que custa R\$ 17.000.

E também podemos remover o array de *doubles*:

```
public static void main(String[] args) {  
    Produto produtos[] = new Produto[5];  
    produtos[0] = new Produto("Lamborghini", 1000000);  
    produtos[1] = new Produto("Jipe", 46000);  
    produtos[2] = new Produto("Brasilia", 16000);  
    produtos[3] = new Produto("Smart", 46000);  
    produtos[4] = new Produto("Fusca", 17000);  
  
    int maisBarato = 0;  
    for(int atual = 0; atual <= 4; atual++){  
        if(precos[atual] < precos[maisBarato]) {  
            maisBarato = atual;  
        }  
    }  
    // ...  
}
```

Fizemos um array de produtos. Agora que não temos mais o preço do `atual` e do `maisBarato`, usaremos o `produto` do `atual` e o seu `getPreco`. Em seguida, vamos inserir o `produto` do `maisBarato` e o `getPreco`. Nosso código ficará assim:

```

public static void main(String[] args) {
    Produto produtos[] = new Produto[5];
    produtos[0] = new Produto ("Lamborghini", 1000000);
    produtos[1] = new Produto("Jipe", 46000);
    produtos[2]= new Produto("Brasilia", 16000);
    produtos[3] = new Produto("Smart", 46000);
    produtos[4] = new Produto("Fusca", 17000);

    int maisBarato = 0;
    for(int atual = 0; atual <= 4; atual++){
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
            maisBarato = atual;
        }
    }
}

```

É possível que você pense: "Ainda não foi criado o `getPreco`. E quando eu crio uma variável membro, deveria já criar o *getter* e o *setter*". No entanto, vamos criá-los apenas quando for preciso. Se for desnecessário, não temos razão para criá-los.

No nosso exemplo, precisaremos do `getPreco`, porque vou usar o produto `atual` e seu preço para, em seguida, comparar com o produto `maisBarato` e seu preço.

Vamos criar o `getPreco`? Entramos na classe `Produto`, digitamos `getPreco + Ctrl + barra de espaço` e o programa vai criar o *getter*.

```

public class Produto {

    private String nome;
    private double preco;

    public Produto(String nome, double preco ) {
        this.nome = nome;
        this.preco = preco;
    }

    public double getPreco() {
        return preco;
    }
}

```

```
}
```

Após fechar a classe `Produto`, vamos comparar os dois preços. Também tenho um `System.out` que imprime o **número** do `maisBarato`, e o `O carro mais barato custa` o valor do `produtos[maisBarato].getPreco`.

```
public static void main(String[] args) {
    Produto produtos[] = new Produto[5];
    produtos[0] = new Produto ("Lamborghini", 1000000);
    produtos[1] = new Produto("Jipe", 46000);
    produtos[2]= new Produto("Brasilia", 16000);
    produtos[3] = new Produto("Smart", 46000);
    produtos[4] = new Produto("Fusca", 17000);

    int maisBarato = 0;
    for(int atual = 0; atual <= 4; atual++){
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
            maisBarato = atual;
        }
    }
    System.out.println(maisBarato);
    System.out.println("O carro mais barato custa"
                       + produtos[maisBarato].getPreco()
    );
}
```

Esse será o preço do carro. Na verdade, já temos o nome do produto e podemos também imprimi-lo: "O carro `produtos[maisBarato].getNome()` é o **mais barato, e custa**".

```
System.out.println(maisBarato);
System.out.println("O carro" + produtos[maisBarato].getNome()
                  + " é o mais barato, e custa"
                  + produtos[maisBarato].getPreco()
);
```

Vamos criar o `getNome`? Entramos no `produto`. Com o comando `get + Ctrl + barra de espaço`, o programa sugerirá um `getNome`: `public String getNome()`.

```
public class Produto {
```

```
private String nome;
private double preco;

public Produto(String nome, double preco ) {
    this.nome = nome;
    this.preco = preco;
}

public double getPreco() {
    return preco;
}

public String getNome() {
    return nome;
}

}
```

Quando colocamos o `get.Nome`, o programa imprimirá o nome e o preço do produto.

Vale revisar o que fizemos até agora: criamos uma classe `Produto` e um array com todos os elementos. Em seguida, quando fizemos a comparação de preços listados, comparamos os preços dos produtos entre si. Quando queremos imprimir, necessito do nome e o preço do produto. Paramos de usar um array de `double` e começamos a usar um array de `produtos`. Porém, a comparação continuou sendo de elementos numéricos.

Vamos testar o código? Quando nós rodarmos o programa, vai aparecer na tela que o `carro 2` é o mais barato e custa R\$ 16.000.

```
2
0 carro Brasília é o mais barato, e custa 16000.0
```

No código, passamos a usar uma classe e objetos para armazenar os dados relativos aos meus produtos.

1.10 DETALHES DE IMPLEMENTAÇÃO NA LINGUAGEM

Nós temos nosso código Java que cria um array de produtos, faz a busca do menor de todos os valores e nos mostra o menor de todos os preços. O algoritmo está implementado.

Agora, o que queremos saber é: como podemos melhorar o nosso código? Vamos aperfeiçoá-lo, antes de continuar.

```
public static void mais(String[] args) {
    Produto produtos[] = new Produto[5];
    produtos[0] = new Produto ("Lamborghini", 1000000);
    produtos[1] = new Produto("Jipe", 46000);
    produtos[2] = new Produto("Brasília", 16000);
    produtos[3] = new Produto("Smart", 46000);
    produtos[4] = new Produto("Fusca", 17000);

    int maisBarato = 0;
    for(int atual = 0; atual <= 4; atual++){
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
            maisBarato = atual;
        }
    }
    System.out.println(maisBarato);
    System.out.println("O carro" + produtos[maisBarato].getNome()
                      + " é o mais barato, e custa"
                      + produtos[maisBarato].getPreco());
});
```

}

Quando nós criamos um array nas versões mais recentes do Java, em vez de falar o tamanho do array, podemos usar `double precos[] = {1.3, 4.4}`.

Podemos colocar os valores desde o princípio, entre as chaves. Então, vamos declarar o array de produtos dessa maneira:

```
public static void mais(String[] args) {
    Produto produtos[] = {
        new Produto ("Lamborghini", 1000000),
        new Produto("Jipe", 46000),
        new Produto("Brasília", 16000),
        new Produto("Smart", 46000),
        new Produto("Fusca", 17000)
};
```

```

        int maisBarato = 0;
        for(int atual = 0; atual <= 4; atual++) {
            if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
                maisBarato = atual;
            }
        }
        System.out.println(maisBarato);
        System.out.println("O carro " + produtos[maisBarato].getNome()
e()
+ " é o mais barato, e custa"
+ produtos[maisBarato].getPreco());
    }
}

```

A definição de array de produtos fica um pouco mais simples no Java assim. E para separar cada um dos elementos, nós usamos a **vírgula (,)**.

Temos o mesmo código, a mesma equivalência, só que conseguimos deixá-lo mais simples: criamos esses cinco produtos e um array baseado nestes elementos. Agora, temos um código mais bonito!

1.11 REFATORAÇÃO: EXTRAINDO UMA FUNÇÃO

Considerando que nós já temos a criação de um array e nosso algoritmo que detecta o produto com o menor valor, queremos extrair o código:

```

int maisBarato = 0;
for(int atual = 0; atual <= 4; atual++){
    if(produtos[atual].getPreco() < produtos[maisBarato].getPreco())
    {
        maisBarato = atual;
    }
}

```

Este código é bastante utilizado para detectar qual preço é o

mais barato, ou qual é o menor valor de um array. É possível extrair um método (uma **função**) deste recorte.

Moveremos uma parte do código e, depois, vamos substituí-la pela função `int maisBarato = buscaMenor(produtos)` dentro do array. Em seguida, criaremos o método.

```
public class TestaMenorPreco {  
  
    public static void main(String[] args) {  
        Produto produtos[] = {  
            new Produto ("Lamborghini", 1000000),  
            new Produto("Jipe", 46000),  
            new Produto("Brasília", 16000),  
            new Produto("Smart", 46000),  
            new Produto("Fusca", 17000)  
        };  
  
        int maisBarato = buscaMenor(produtos);  
        System.out.println(maisBarato);  
        System.out.println("O carro" + produtos[maisBarato].getNome()  
                           + " é o mais barato, e custa"  
                           + produtos[maisBarato].getPreco());  
  
    }  
  
    private static int buscaMenor(Produto[] produtos) {  
        return 0;  
    }  
}
```

Observe que o **método** recebe o array de produtos, então, apenas daremos um *paste* na parte retirada do código. No fim, é claro que não vamos retornar `0`. Não faria sentido. Retornaremos o `maisBarato`.

```
private static int buscaMenor(Produto[] produtos) {  
    int maisBarato = 0;  
    for(int atual = 0; atual <= 4; atual++) {  
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
```

```
        maisBarato = atual;
    }
}
return maisBarato;
}
```

Depois disso, vamos salvar e rodar o código. O programa imprimirá:

```
2
0 carro Brasilia é o mais barato, e custa 16000.0
```

Nós extraímos uma função que é um algoritmo de busca do menor valor dentro do array.

1.12 BUSCANDO DO INÍCIO AO FIM DE UM ARRAY

Vamos ser cuidadosos. O que acontece se tirarmos um produto da lista de cinco carros? Por exemplo, a Brasília.

```
Produto produtos[] = {
    new Produto ("Lamborghini", 1000000),
    new Produto("Jipe", 46000),
    new Produto("Smart", 46000),
    new Produto("Fusca", 17000)
};
```

Nossa lista fica com quatro produtos. Quando tentarmos rodar o código, o que aparecerá na tela?

```
Exception in thread "main"java.lang.ArrayIndexOutOfBoundsException
: 4
        at br.com.alura.algoritmos.TestaMenorPreco.buscaMenor(TestaMenorPreco.java:24)
        at br.com.alura.algoritmos.TestaMenorPreco.main(TestaMenorPreco.java:13)
```

```
TestaMenorPreco.java
package br.com.alura.algoritmos;

public class TestaMenorPreco {

    public static void main(String[] args) {
        Produto produtos[] = {
            new Produto("Lamborghini", 1000000),
            new Produto("Jipe", 46000),
            new Produto("Smart", 46000),
            new Produto("Fusca", 17000)
        };

        int maisBarato = buscaMenor(produtos);
        System.out.println(maisBarato);
        System.out.println("O carro " + produtos[maisBarato]);
    }
}
```

Figura 1.37: Problemas com o teste

O programa tentou acessar o produto na posição 4, porque o nosso `for` vai até a posição 4, inclusive. Porém, não temos mais cinco produtos. Agora, temos quatro elementos, que vão até a posição 3. Temos um problema.

Nosso array tem um número fixo e o ideal é que o número seja do **tamanho** da nossa lista, menos **um**:

```
for(int atual = 0; atual <= produtos.length - 1); atual ++)
```

Considerando que agora temos apenas 4 elementos, vou incluir a variável `termino = (produtos.length - 1);`.

```
private static int buscaMenor(Produto[] produtos) {
    int maisBarato = 0;
    int termino = produtos.length - 1;
    for(int atual = 0; atual <= termino; atual++){
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
            maisBarato = atual;
        }
    }
    return maisBarato;
}
```

Vamos testar nosso código? Quando todos os cinco produtos estiverem listados, nós vamos executar o programa e aparecerá que o carro mais barato é a Brasília.

Em seguida, apagamos a Brasília da lista, que ficará com quatro elementos. Então, vou rodar o código, que vai indicar o Fusca como o carro mais barato.

3

0 carro Fusca é o mais barato, e custa 17000.0

Isto significa que o nosso código funcionou.

The screenshot shows the Eclipse IDE interface. In the center-left, there's a code editor window titled "TestaMenorPreco.java" containing Java code. An arrow points from the text "produtos[mai" in the code to the "Console" tab. In the "Console" tab, the output is displayed: "0 carro Fusca é o mais barato, e custa 17000.0". The Eclipse toolbar and various views are visible around the central workspace.

```
Java - produtoa/src/com/alura/algoritmos/TestaMenorPreco.java - Eclipse - /Users/alura/Documents/guiherme/workspace3
TestaMenorPreco.java
+ produtos[mai
}
private static int buscaMenor(Produto[] produtos)
{
    int maisBarato = 0;
    int termino = produtos.length - 1;
    for(int atual = 0; atual <= termino; atual++)
        if(produtos[atual].getPreco() < produtos[maisBarato])
            maisBarato = atual;
    }
    return maisBarato;
}

}
0 carro Fusca é o mais barato, e custa 17000.0
```

Figura 1.38: Testando o código novamente

Independentemente do tamanho do array, o programa vai verificá-lo por completo, para então encontrar o menor valor possível. Nós **generalizamos** nossa função de busca de menor valor.

1.13 BUSCANDO SOMENTE EM UM TRECHO ESPECÍFICO DE UM ARRAY

Nós aprendemos a criar a função que verifica **todos** os elementos de um array para, em seguida, encontrar aquele com o menor valor. No exemplo dos carros, nós detectamos o menor preço.

Se trabalhássemos com um array de *double*, nós compararíamos os valores. Se fôssemos analisar os dados sobre provas de alunos, faríamos uma comparação das notas. Nós comparamos o que queremos e encontramos o menor valor de todos.

O que acontece se a nossa lista termina com diversos valores inválidos? Por exemplo, é comum criarmos um array com vários elementos, mas nem todos são válidos:

```
public static void main(String[] args) {  
    Produto produtos[] = {  
        new Produto ("Lamborghini", 1000000),  
        new Produto("Jipe", 46000),  
        new Produto("Brasília", 16000),  
        new Produto("Smart", 46000),  
        new Produto("Fusca", 17000)  
        null,  
        null,  
        null,  
        null,  
        null  
    };
```

Nós já temos cinco itens válidos, porém podemos incluir outros elementos sem preencher. Em Java, isto é bem incomum, porque na prática nós podemos usar um *arraylist* (outros elementos que representem um array). Mas em diversas linguagens, podemos não saber o tamanho exato (válido) da nossa lista.

No exemplo dos carros, nosso array tem nove elementos, porém apenas cinco são válidos. O programa vai comparar `null` com `null`, o que vai resultar em `Nullpointer` e isto não é bom. Por isso, de alguma maneira, nós temos de indicar para a nossa função onde ela deve terminar.

O término do código é `produtos.length - 1`. Isto significa que, quando o produto for `5`, o processo deve parar no `4`. Mas podemos encontrar casos diferentes, em que um array pode conter valores vazios, além de linguagens em que não temos como saber o seu tamanho ou quando ele acaba.

Se não temos estes dados, não teremos `length`. Sendo assim, como calcularemos o `termino`? É impossível calculá-lo.

```
private static int buscaMenor(Produto[] produtos) {
```

```

int maisBarato = 0;
int termino = produtos.length - 1;
for(int atual= 0; atual <= termino; atual++) {
    if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
        maisBarato = atual;
    }
}
return maisBarato;
}

```

Em todas estas situações, quando mandamos fazer uma busca dentro de uma lista, devemos indicar de quantos elementos ela dispõe. No nosso caso, preciso dizer que temos 5 elementos.

```
int maisBarato = buscaMenor(produtos, 5)
```

Com as alterações, nosso código ficará assim:

```

int maisBarato = buscaMenos(produtos, 5)
System.out.println(maisBarato);
System.out.println("O carro" + produtos[maisBarato].getNome()
+ " é o mais barato, e custa"
+ produtos[maisBarato].getPreco()
));

private static int buscaMenor(Produto[] produtos) {
    int maisBarato = 0;
    int termino = produtos.length - 1;
    for(int atual= 0; atual <= termino; atual++) {
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
            maisBarato = atual;
        }
    }
    return maisBarato;
}

```

Isso significa que nós estamos informando para a função (o nosso método) qual é o tamanho do array.

```
private static int buscaMenor(Produto[] produtos, int termino) {
```

Com isso, eliminamos a linha:

```
int termino = produtos.length - 1;
```

Na verdade, já podemos indicar o `termino` direto. Como o `termino` é `4`, podemos inseri-lo diretamente no meu código.

```
int maisBarato = buscaMenos(produtos, 4)
System.out.println(maisBarato);
System.out.println("O carro" + produtos[maisBarato].getNome()
+ " é o mais barato, e custa"
+ produtos[maisBarato].getPreco()
));
```

No Java, nós podíamos usar `produtos.length`. Porém, em diversas linguagens nós não podemos fazer isso, porque não sabemos o `length` do nosso array. O responsável pela busca precisa nos informar o tamanho da lista. É o que alteramos no nosso código: informamos quantos produtos tem dentro do array. Isto é, estou falando para minha função (meu método) qual é o tamanho do meu do array.

```
int maisBarato = buscaMenos(produtos, 4)
System.out.println(maisBarato);
System.out.println("O carro" + produtos[maisBarato].getNome()
+ " é o mais barato, e custa"
+ produtos[maisBarato].getPreco()
));

private static int buscaMenor(Produto[] produtos, int termino) {
    int maisBarato = 0;
    for(int atual = 0; atual <= termino; atual++) {
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
            maisBarato = atual;
        }
    }
    return maisBarato;
}
```

Como temos cinco produtos, ele vai da posição `0` até a posição `4`. Vamos indicar isto no código?

```
int maisBarato = buscaMenor(produtos, 0, 4)
```

Vamos falar tudo de uma vez e indicar também o início e o `termino`:

```

private static int buscaMenor(Produto[] produtos, int inicio, int
termino) {
    int maisBarato = inicio;
    for(int atual = inicio; atual <= termino; atual++){
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
            maisBarato = atual;
        }
    }
    return maisBarato;
}

```

Fizemos uma busca do menor valor possível, de uma determinada posição a outra do array, do início até o fim, ou do `0` ao `length - 1`. Ele vai começar em uma parte do nosso array e vai terminar em outra, do `int inicio` até o `int termino`.

```

int maisBarato = buscaMenos(produtos, 0, 4)
System.out.println(maisBarato);
System.out.println("O carro" + produtos[maisBarato].getNome()
+ " é o mais barato, e custa"
+ produtos[maisBarato].getPreco());
};

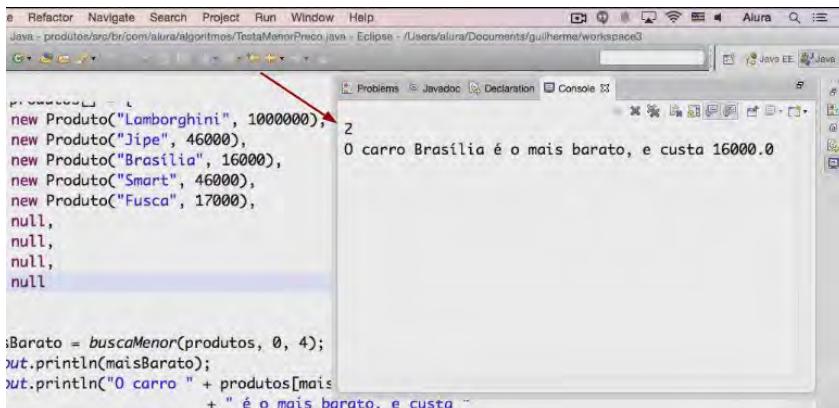
private static int buscaMenor(Produto[] produtos, int inicio, int
termino) {
    int maisBarato = inicio;
    for(int atual = inicio; atual <= termino; atual++) {
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
            maisBarato = atual;
        }
    }
    return maisBarato;
}

```

Por que vamos especificar o `inicio` e o `termino`? Porque queremos generalizar nossa função e fazer com que ela funcione em diversas linguagens (incluindo aquelas que não suportam os objetos de um array, ou quando não sabemos quantos são os elementos).

A nossa função funcionará em todas essas situações, porque o programa vai varrer exatamente entre as posições determinadas, em busca do menor valor de todos. Se estamos verificando do `0` até `4`

inclusive, vamos levar em consideração da Lamborghini até o Fusca. Quando rodarmos o programa, vamos obter o mesmo resultado: o carro 2 é o mais barato. Se retirarmos os elementos null , o resultado será sempre o mesmo.



The screenshot shows the Eclipse IDE interface. In the top menu, 'File', 'Refactor', 'Navigate', 'Search', 'Project', 'Run', 'Window', 'Help' are visible. Below the menu, the title bar says 'Java - produção/sr/com/alura/algoritmos/TestaMenorPreco.java - Eclipse - /Users/alura/Documents/guilherme/workspace3'. The left pane shows Java code:`private void testaMenorPreco() {
 Produto[] produtos = {
 new Produto("Lamborghini", 1000000),
 new Produto("Jipe", 46000),
 new Produto("Brasília", 16000),
 new Produto("Smart", 46000),
 new Produto("Fusca", 17000),
 null,
 null,
 null,
 null
 };

 int menorPreco = buscaMenor(produtos, 0, 4);
 System.out.println(menorPreco);
 System.out.println("O carro " + produtos[menorPreco] +
 " é o mais barato, e custa " + produtos[menorPreco].getPreco());
}`The right pane shows the 'Console' tab with the output:

```
2
O carro Brasília é o mais barato, e custa 16000.0
```

A red arrow points from the word 'menorPreco' in the code to the value '2' in the console output.

Figura 1.39: Resultado do teste

A minha função é genérica o suficiente para funcionar em diversas linguagens e em diversas situações. Tanto se estamos programando em C ou em Java e tenho diversos nulls (valores que quero ignorar) no meu array, independentemente se estão no começo ou no fim. Descartamos os elementos desnecessários e analisamos apenas um determinado pedaço para descobrir o menor valor de todos.

A função está mais poderosa agora.

1.14 O PRÓXIMO DESAFIO: OS MAIS BARATOS

Já sei identificar o elemento mais barato ou o mais caro de um conjunto. Consigo encontrar o elemento menor ou maior. Mas e se estou interessado nos dois maiores?

Por exemplo, quem são os quatro melhores times da primeira

rodada de uma competição para fazer uma segunda fase só entre eles? Quem foram os 3 primeiros colocados em uma competição? Quais são os 5 produtos mais baratos?

Nosso algoritmo só é capaz de detectar o menor ou o maior. Vamos correr atrás de algoritmos ainda mais interessantes para resolver esses problemas.

CAPÍTULO 2

COMO A ORDENAÇÃO PODE NOS AJUDAR

2.1 OS N MAIS BARATOS

Na prática, será que nós só estamos interessados em saber o valor mais barato ou o mais caro? Ou também estamos interessados em descobrir o **grupo** dos produtos mais baratos, o **grupo** dos mais caros, o **grupo** dos que fizeram mais pontos.

Às vezes, queremos saber duas informações, por exemplo: quem ganhou com **mais** pontos e quem perdeu com **menos** pontos. Porém, é possível que uma busca nos traga inúmeros resultados. Como quando queremos descobrir quais são os 10 carros mais baratos, ou os cinco primeiros times que vão para a próxima fase da competição. Ou quando queremos informações sobre um grupo com vários produtos.

Como nós podemos alterar o nosso processo para marcar quais são os dois carros mais baratos? Dê uma olha na nossa lista de carros novamente:



Figura 2.1: Lista de carros

Ao observarmos a lista, logo vamos concluir que a Brasília e o Fusca são os carros mais baratos. Qual foi o processo que fizemos mentalmente para descobrir que os dois eram os mais baratos?

O mesmo aconteceria se quiséssemos detectar quais são os três carros mais baratos. A Brasília, o Fusca e o Smart (ou o Jipe) são os carros mais baratos. Em que estávamos pensando à medida que analisávamos cada produto?

Observe a lista de carros e tente imaginar qual foi o processo que realizamos ao analisar cada um dos itens.

2.2 ALGORITMO BÁSICO (E RUIM?) PARA ENCONTRAR O N MAIS BARATO

Qual é o processo que faremos ao analisar cada um dos elementos para descobrir quais dos produtos são o `maisBarato` e o `segundomaisBarato`? Para descobrir qual é o `maisBarato`, nós vamos precisar de uma variável que armazene o produto com o menor preço naquele momento.

Vamos precisar também ter registrado (mentalmente ou anotado em um papel) qual é o `segundoMaisBarato`. Isto significa que vou precisar de outra variável e, assim, anotar o primeiro e o segundo mais barato.

Analisaremos cada elemento e identificaremos "esse é o mais barato" ou "esse é o segundo mais barato agora". Para isso, teremos de usar o `if` várias vezes.



Figura 2.2: Segundo mais barato

No exemplo dos carros, se desejarmos encontrar quais são os três carros mais baratos, precisaremos usar três `ifs` e o código terá mais uma variável: `terceiroMaisBarato`.



Figura 2.3: Novas variáveis

Sempre que quisermos saber se a Lamborghini é o carro mais barato, ou o segundo, talvez, o terceiro carro mais barato, teremos sempre de passar pelo `for`. Em consequência, será preciso utilizar o `if` várias vezes se quisermos criar um grupo dos produtos mais baratos.

Por exemplo, se desejarmos identificar o grupo dos cinco carros com o menor preço, vamos precisar utilizar `if` cinco vezes. O mesmo vai acontecer se quisermos descobrir quais são os 10 carros mais baratos. Para encontrar a resposta, vamos precisar usar 10 `if`s. Imagine o tamanho do nosso código se quisermos descobrir os 50 produtos mais baratos de uma lista.

Não parece ser a melhor solução colocar inúmeras variáveis para

verificar se o produto é o primeiro, o segundo, o terceiro, quarto ou quinto mais barato. Não parece a melhor solução usar infinitos `if`s para identificar os elementos com o menor valor.

Será que podemos encontrar uma outra solução para encontrar o produto mais barato?

2.3 A IMPORTÂNCIA DA ORDENAÇÃO

Continuamos com a questão de como descobrir quais são os produtos mais baratos e os mais caros, os times ou competidores que vão para a próxima fase da competição, os n elementos **mais** ou **menos, maiores** ou **menores**.

Temos um tipo de classificação quando determinamos que "isso é menor do que aquilo". Como responder a pergunta sem colocar infinitas variáveis e `ifs` no meu código?

Com 500 variáveis no código, escrevemos e escrevemos. Em algum momento, vamos terminar. Porém, desejamos aproveitar melhor nosso tempo e não queremos inserir inúmeras vezes o `if`. O que podemos fazer?

Vamos dar uma olhada na lista de carros:



Figura 2.4: Lista de carros

É mais difícil encontrar o maior e o menor valor na nossa lista, porque os elementos estão misturados. Mas e se tivéssemos uma lista diferente? Uma lista que fosse ordenada do menor para o maior

preço?



Figura 2.5: Lista de carros ordenada

Com a nova lista, vamos responder a mesma pergunta de antes: qual é o carro mais barato? É possível identificá-lo com maior facilidade, afinal, os três primeiros elementos são os mais baratos.

Se queremos descobrir quais são os dois carros mais baratos, será o mesmo: os dois primeiros carros têm os menores preços. Caso nossa lista tenha 1 milhão de elementos e eles forem ordenados desta maneira, os 30 primeiros também serão os 30 mais baratos. Se quiséssemos os dois carros mais caros? Seriam os dois últimos da lista. Observe que a ordem faz toda diferença.

No momento em que os elementos de uma lista (seja carros, produtos, times ou alunos) estão ordenados do menor para o maior, fica fácil responder a perguntas como:

- Quem são os dois menores?
- Quem são os dois maiores?
- Quem são os três menores?
- Quem são os cinco maiores?

Basta selecionarmos os três primeiros ou os cinco últimos, ou os dois primeiros e os dois itens do fim da lista. Com a ordenação, você descobre os maiores e os menores elementos rapidamente.

Perguntas como "quais são os 500 menores e maiores?" não precisariam ser respondidas com 500 `if`s no nosso código. Porém,

basta que os produtos estejam ordenados para que encontremos a resposta rapidamente.

Outro exemplo seria se quiséssemos encontrar o produto mais barato. Caso nossa lista esteja em ordem, sabemos que o elemento com o menor preço será aquele que estiver na primeira posição. Quando o array está ordenado (uma coleção de itens ordenados), é simples identificar quais são os elementos maiores e menores. O desafio está na ordenação dos itens.

No exemplo dos carros, nós ordenamos os elementos manualmente. Movemos os produtos e conseguimos colocá-los em novas posições.

Como conseguimos ordenar um array? Este é o desafio que vamos resolver: como ordenar nossas listas de maneira inteligente e rápida?

Outras questões:

- Como criar uma ordem para cartas de baralho, provas de alunos ou resultados de uma eleição?
- Como podemos ordenar itens e, depois, aperfeiçoar a ordenação?
- Qual o mais barato?
- Qual carro está posicionado no meio da lista?

2.4 O PRÓXIMO DESAFIO: ORDENAÇÃO

Quando as coisas estão ordenadas, o processo de busca se torna mais rápido. Essa é a grande sacada da ordenação: ela nos permite resolver tarefas do cotidiano quase imediatamente.

O nosso desafio será descobrir **como criar uma ordem para os produtos que estão desordenados**.

CAPÍTULO 3

SELECIONANDO E ORDENANDO

3.1 COMO ORDENAR?

Nós queremos ordenar o nosso array de produtos. E como podemos fazer isto? Temos os elementos:



Figura 3.1: Lista de carros

Escreva os cinco elementos em um papel, na mesma ordem. Em seguida, tente ordená-los do produto mais barato para o mais caro. Como você ordenou os carros? Responda e eu mostrarei o meu processo de ordenação, daqui a pouco.

3.2 ORDENANDO AO SELECIONAR O MAIS BARATO

Ordenou os seus produtos? Eu ordenarei os meus agora. Tentei encontrar uma maneira de fazer isso:

- Lamborghini custa R\$ 1.000.000

- Jipe custa R\$ 46.000
- Brasilia custa R\$ 16.000
- Smart custa R\$ 46.000
- Fusca custa R\$ 17.000

De todos, a **Brasília** é o mais barato. Logo, este será o primeiro carro na nossa lista. Em seguida, continuamos buscando o carro com menor preço. O Fusca custa R\$ 17.000 é o segundo mais barato. Vamos mudá-lo de posição.

Tanto o Smart quanto o Jipe custam R\$ 46.000. Vamos movê-los na lista também. A Lamborguini custa R\$ 1.000.000, sendo o carro mais caro dos cinco elementos. Então, será o elemento no fim da nossa lista.

Os elementos estão ordenados. Como eu criei a ordem?

Vamos descrever passo a passo o que eu fiz durante o processo: no começo a Lamborghini estava no início. Mas será que ela realmente é o carro com o menor preço? Vamos analisar se temos outros carros mais baratos que ela.

- Jipe custa R\$ 46.000
- Brasilia custa R\$ 16.000
- Smart custa R\$ 46.000
- Fusca custa R\$ 17.000

Em comparação com a Lamborghini, o Jipe é mais barato, mas a Brasilia é ainda mais. O Smart também é mais barato, porém a Brasilia tem o menor preço. O Fusca é barato, mas a Brasilia continua sendo mais barata. Na verdade, vários carros são mais baratos do que a Lamborghini. E a **Brasilia** é o elemento mais barato.

Isto significa que depois da Lamborghini, todos os carros são mais baratos. Não faz sentido que ela seja o primeiro item da lista.

Então, a Brasília passa a estar no início, porque é o produto mais barato.

Agora nós sabemos que quem ocupa a primeira casa é o carro mais barato. Não precisamos mais nos preocupar com ele.

Seguimos para o próximo carro. Vamos ver quem merece estar na próxima posição? Da segunda casa adiante, qual carro é o mais barato? É o **Fusca**. Trocamos o Fusca de lugar e agora sabemos qual é o segundo carro mais barato.

Quero encontrar o terceiro carro mais barato. Aquele que merece ocupar a terceira casa e será o elemento com o menor preço entre os restantes. Qual é o mais barato? Vamos selecionar o **Smart**.

Para finalizarmos, qual é o carro mais barato da quarta casa para o fim? O Jipe. Então, vamos trocá-lo com a Lamborguini. Com a nossa nova lista, simplificamos o processo de busca.

A ideia de ordenarmos os elementos a partir do mais barato é para simplificarmos a busca do menor preço de acordo com a posição. Vamos refazer nossos passos?

3.3 ANALISANDO CADA PASSO

A partir da primeira casa, fizemos a pergunta: qual é o carro mais barato? Identificamos que era a Brasília. Vamos trocá-la de lugar. Encontrar um produto com referência na posição era algo que já havíamos feito anteriormente.

Seguimos com a pergunta: a partir da segunda posição, qual é o mais barato? O Fusca. Nós já escrevemos o algoritmo `maisBarato` a partir de uma posição. O método "encontre o menor a partir de determinada posição" nós já temos. O carro mais barato desde a terceira casa é o Smart, então, trocamos o produto de posição.

O carro mais barato a partir da quarta posição é o Jipe. Movemos o elemento de lugar e resolvemos a questão de como encontrar o carro mais barato, com a ordenação. Porém, não encontramos o `maisBarato` apenas uma vez. Respondemos a mesma pergunta cinco vez.

Quando encontramos o carro mais barato, trocamos o elemento de lugar. Em seguida, encontramos o segundo mais barato e o trocamos de lugar. Fazemos o mesmo com o terceiro, quarto e quinto elementos. Quando chegamos no último, já sabíamos que ele era o carro mais caro de todos.

Então, o que fizemos? Varremos nosso array procurando o mais barato a partir de uma determinada posição. Nós perguntamos: "A partir desse carro, qual é o mais barato? É este." Trocamos o elemento de lugar na lista, e assim ficamos felizes com o resultado.

Vamos passar isto para o código? É o nosso próximo passo.

3.4 IMPLEMENTANDO EM JAVA A ORDENAÇÃO PELA SELEÇÃO DO MENOR VALOR

Chegou a hora de tentarmos implementar nosso processo: a sequência de passos que ordena o nosso array.

```
private static int buscaMenor(Produto[] produtos, int inicio, int termino) {
    int maisBarato = inicio;
    for(int atual = inicio; atual <= termino; atual++){
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
            maisBarato = atual;
        }
    }
    return maisBarato
}
```

Vamos começar a criar o processo: após entrarmos no pacote `br.com.alura.algoritmos`, criaremos a nova classe `TestaOrdenacao`. Ela também terá um método `main`.

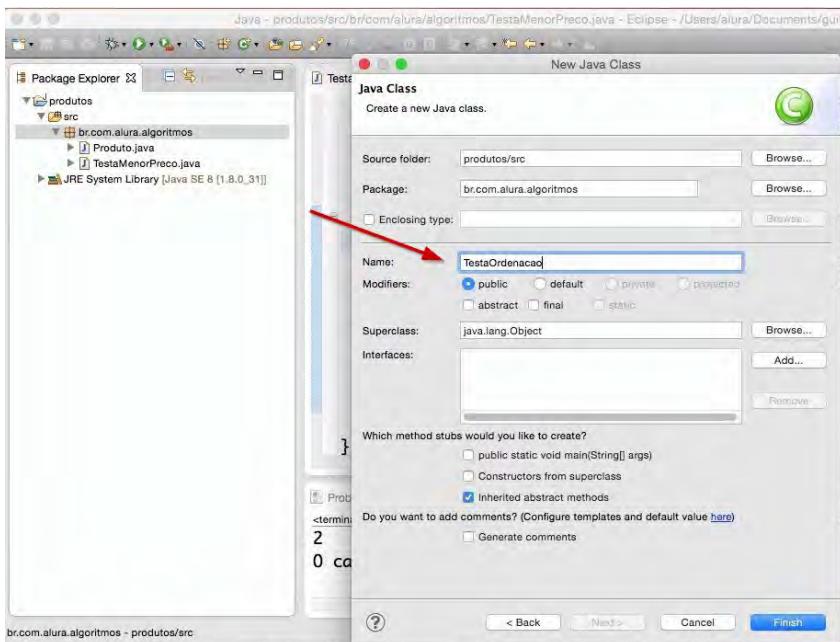


Figura 3.2: Classe TestaOrdenacao

Dentro do método `main`, vou copiar um pedaço do código, em que temos nosso array de produtos:

```
package br.com.alura.algoritmos;

public class TestaOrdenacao {

    public static void main(String[] args) {
        Produto produtos[] = {
            new Produto("Lamborghini", 1000000),
            new Produto("Jipe", 46000),
            new Produto("Brasília", 16000),
            new Produto("Smart", 46000),
            new Produto("Fusca", 17000)
        };
    }
}
```

Temos os elementos (Lamborghini, Jipe, Brasília, Smart e Fusca) e gostaríamos de ordená-los. Você se lembra de como nós fizemos isto? Nós analisamos cada produto da nossa lista com o `for` que já usamos anteriormente.

```
for(int atual = 0; atual < produtos.length; atual++)
```

Então, queremos verificar até a última casa com produtos. Para ordenar os elementos, nós observamos o produto da primeira casa e fazemos a pergunta: "A partir daqui, qual dos produtos é o menor de todos?". Encontramos a resposta e, então, movemos o elemento mais barato para o início da lista.

Você lembra de que já escrevemos a função que busca o menor elemento? Era `buscaMenor` :

```
private static int buscaMenor(Produto[] produtos, int inicio, int termino) {  
    int maisBarato = inicio;  
    for(int atual = inicio; atual <= termino; atual++){  
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {  
            maisBarato = atual;  
        }  
    }  
    return maisBarato;  
}
```

Como funciona `buscaMenor` ? Nós colocamos: os produtos (`produtos`), o início (`inicio`) e o fim (`termino`).

```
private static int buscaMenor(Produto[] produtos, int inicio, int termino)
```

Então, se dissermos para o programa "busque o menor produto (`produtos`), a partir da posição atual (`atual`) até o fim do nosso array (`produtos.length`)", ele buscará o menor de todos a partir da primeira posição.

```
int menor = buscaMenor(produtos, atual, produtos.length)
```

Agora que encontramos o menor produto, quero trocá-lo de posição para a casa que seja do menor preço. Isto é: o `produtoAtual`, que é o elemento na posição `produtos[Atual]`.

```
Produto produtoAtual = produtos[atual];
```

O `produtoMenor` é o elemento na posição `produtos[menor]`.

```
Produto produtoMenor = produtos[menor];
```

Vou fazer a inversão: o `produtos[atual]` é o `produtoMenor` e o `produtos[menor]` é o `produtoAtual`.

```
for(int atual = 0; atual < produtos.length; atual++) {  
    int menor = buscaMenor(produtos, atual, produtos.length);  
    Produto produtoAtual = produtos[atual];  
    Produto produtoMenor = produtos[menor];  
    produtos[atual] = produtoMenor;  
    produtos[menor] = produtoAtual;  
}
```

O que nós fizemos? Passamos por cada casa com produto a partir da **primeira casa**, e verificamos qual produto tem menor preço. Após encontrarmos o menor, trocamos os elementos de lugar. Seguimos para o produto da **segunda casa** e buscamos qual elemento é o menor a partir dele. Descobrimos qual é e o movemos de lugar.

Repetimos o mesmo processo com os elementos da terceira, quarta e quinta casa. Assim, fomos encontrando os produtos com menor preço e ordenando como "primeiro menor", "segundo menor", "terceiro menor". Quando identificamos todos os produtos, nosso array estava ordenado corretamente.

Vamos imprimir o resultado?

Para isso, criaremos um `for` que passe por todos os produtos e imprima as informações de todos:

```
for(Produto produto : produtos) {  
    System.out.println(produto.getNome() + "custa" + produto.getPr
```

```
eco());  
}
```

Vamos imprimir a informação dos produtos. Para rodar o nosso programa, vamos clicar no botão direito e depois, em *Run As* e *Java Application*.

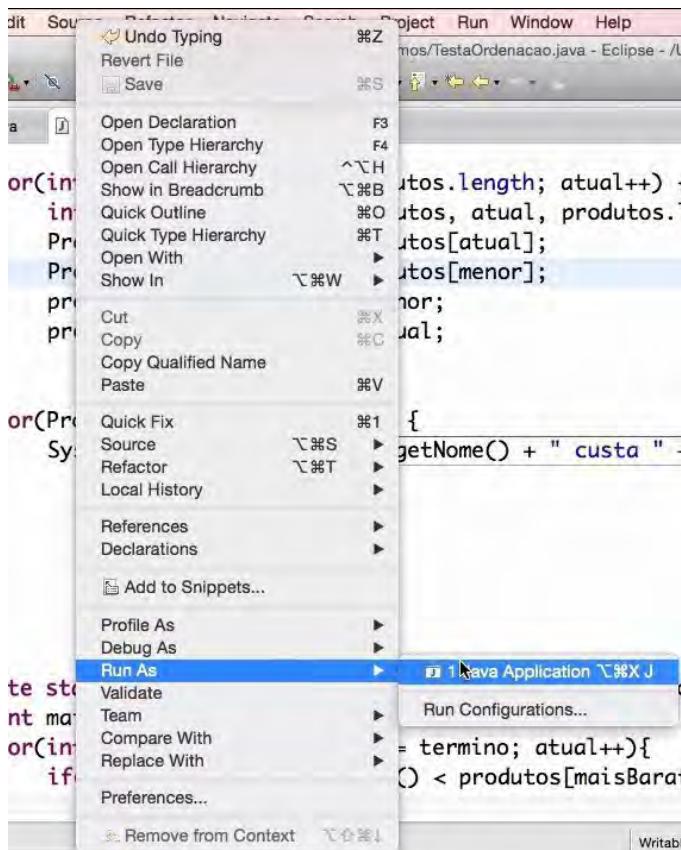


Figura 3.3: Java, Run As

Porém, o programa vai mostrar na nossa tela uma mensagem, avisando que algo não deu certo no processo.

```
Exception in thread "main"java.lang.ArrayIndexOutOfBoundsException  
: 5  
at br.com.alura.algoritmos.TestaOrdenacao.buscaMenor(Testa
```

```
Ordenacao.java: 3)
    at br.com.alura.algoritmos.TestaOrdenacao.main(TestaOrdenacao.java: 15)
```

O que fizemos errado? Em algum lugar, o programa tentou acessar a posição 5. A mensagem nos mostra que foi no buscaMenor .

The screenshot shows the Eclipse IDE interface. The top menu bar includes 'Source', 'Navigator', 'Search', 'Project', 'Run', 'Window', and 'Help'. Below the menu is a toolbar with icons for file operations like Open, Save, and Run. The left side shows a project tree with 'Java - produtos' selected. The main workspace contains a code editor with Java code and a 'Problems' view. The code editor has a red underline under 'produtos.length' in the first line. The 'Problems' view displays an error message: '<terminated> TestaOrdenacao [Java Application] /Library/javaVirtualMachines/dt1.8.0_31.jdk/Contents/jre/lib/rt.jar!/java/lang/IndexOutOfBoundsException: 5' followed by several stack trace entries: 'br.com.alura.algoritmos.TestaOrdenacao.buscaMenor(TestaOrdenacao.java: 5)' and 'br.com.alura.algoritmos.TestaOrdenacao.main(TestaOrdenacao.java:15)'. The code itself is a selection from the 'buscaMenor' method.

```
int atual = 0; atual < produtos.length; c
int menor = buscaMenor(produtos, atual, r
    roduto produtoAtual = produtos[atual];
    roduto produtoMenor = produtos[menor];
    rodutos[atual] = produtoMenor;
    rodutos[menor] = produtoAtual;

    roduto produto : produtos) {
    ystem.out.println(produto.getNome() + "
```

Figura 3.4: BuscaMenor em Java

Do lado do erro, temos um link que nos leva direto para o buscaMenor :

```
private static int buscaMenor(Produto[] produtos, int inicio, int
termino) {
    int maisBarato = inicio;
    for(int atual = inicio; atual <= termino; atual++){
        if(produtos[atual].getPreco() < produtos[maisBarato].getPr
eo())){
            maisBarato = atual;
        }
    }
    return maisBarato;
}
```

Observe que a variável atual percorre até termino que é igual a 5, pois atual <= termino .

```
for(int atual = inicio; atual <= termino; atual++){
```

Isto significa que o programa verifica até a posição 5 .

O <= (menor ou igual) faz muita diferença no nosso algoritmo. Como a nossa função busca exatamente até determinado elemento, quando a chamamos não queremos que verifique `produtos.length`. De fato, o que desejamos que ela faça é que vá até `produtos.length - 1`.

Você se lembra de que, na nossa classe `TestaMenorPreco`, nossa última posição era 4 (que era o tamanho do array menos 1)?

```
Produto produtos[] = {  
    new Produto ("Lamborghini", 1000000),  
    new Produto("Jipe", 46000),  
    new Produto("Brasília", 16000),  
    new Produto("Smart", 46000),  
    new Produto("Fusca", 17000)  
};  
  
int maisBarato = buscaMenos(produtos, 0, 4)  
System.out.println(maisBarato);  
System.out.println("O carro" + produtos[maisBarato].getNome()  
    + " é o mais barato, e custa"  
    + produtos[maisBarato].getPreco());
```

Voltamos para a classe `TestaOrdenacao`. Na nossa função, a verificação deve ser feita de atual (a primeira posição) até `produtos.length - 1` (a última posição). Não é o tamanho do nosso array. Um erro que ocorre, porque nossa função está utilizando <= (menor ou igual).

```
for(int atual =0; atual < produtos.length; atual++) {  
    int menor = buscaMenor(produtos, atual, produtos.length - 1);  
    Produto produtoAtual = produtos[atual];  
    Produto produtoMenor = produtos[menor];  
    produtos[atual] = produtoMenor;  
    produtos[menor] = produtoAtual;  
}
```

Dependendo da maneira como for implementada, algumas vezes veremos a função com < (menor), outras com = (igual).

Nós escolhemos implementar a função com `<=`, por isso vamos usar a última posição do array (`produtos.length - 1`).

Vários livros fazem da mesma forma. Caso outro livro mostre uma implementação diferente, que use `<`, não precisaremos do `-1` e utilizaremos apenas `produtos.length`. Tudo depende da teoria e do livro que você estiver usando. No caso, estamos seguindo um exemplo com `<=`, logo, vamos usar `produtos.length - 1`.

Agora vamos testar o nosso código e o programa vai nos mostrar nosso array ordenado:

- Brasília custa 16000.0
- Fusca custa 17000.0
- Smart custa 46000.0
- Jipe custa 46000.0
- Lamborghini custa 1000000.0

The screenshot shows a Java IDE interface with two tabs: 'TestaOrdenacao.java' and 'Console'. The code in 'TestaOrdenacao.java' is a selection sort algorithm. A red arrow points from the text 'Brasilia custa 16000.0' in the console output to the line 'if atual < produtos.length - 1' in the code editor. The 'Console' tab displays the sorted array elements: 'Brasilia custa 16000.0', 'Fusca custa 17000.0', 'Smrt custa 46000.0', 'Jipe custa 46000.0', and 'Lamborghini custa 1000000.0'.

```
TestaOrdenacao.java
it atual = 0; atual < produtos.length - 1; c
it menor = buscaMenor(produtos, atual, f
`oduto produtoAtual = produtos[atual];
`oduto produtoMenor = produtos[menor];
`odutos[atual] = produtoMenor;
`odutos[menor] = produtoAtual;

`oduto produto : produtos) {
    system.out.println(produto.getNome() + "
```

```
Brasilia custa 16000.0
Fusca custa 17000.0
Smrt custa 46000.0
Jipe custa 46000.0
Lamborghini custa 1000000.0
```

Figura 3.5: Array ordenado

Vamos recapitular o que nós fizemos?

Nós temos um array de produtos, e verificamos em cada casa. Em seguida, observamos todos os elementos à direita, encontramos o mais barato e trocamos de lugar. Depois, passamos a ignorar o produto, porque ele já está ordenado.

Continuamos com a pergunta: "a partir do atual, qual é o mais barato?". Descobrimos o elemento e o passamos para as primeiras

posições. Repetimos o processo, apenas com os produtos ainda não ordenados.

Nós sempre verificamos a partir do `atual` e vamos até onde termina o array, que finaliza no último elemento (`produtos.length - 1`). Fazemos isto, porque no `for` nós buscamos `<=` ao último elemento.

```
for(int atual = inicio; atual <= termino; atual++) {
```

No fim, o nosso código imprimirá os nossos elementos em ordem.

3.5 ALGORITMOS E O MENOS 1

Nós escrevemos o código da nossa ordenação, após selecionarmos o maior elemento de todos, o segundo maior, o terceiro maior etc. Fomos identificando um de cada vez, até ordenarmos o array. Agora podemos responder rapidamente diversas perguntas diferentes:

- Quem é o primeiro ou o segundo elemento mais barato?
- Quem é o terceiro mais caro?
- Qual é o do meio? Qual é o do meio menos 1?
- Qual é o mais caro?

Podemos responder de forma rápida todas estas perguntas. Observe o nosso código:

```
for(int atual = 0; atual < produtos.length; atual++) {  
    int menor = buscaMenor(produtos, atual, produtos.length - 1);  
    Produto produtoAtual = produtos[atual];  
    Produto produtoMenor = produtos[menor];  
    produtos[atual] = produtoMenor;  
    produtos[menor] = produtoAtual;  
}
```

```
for(Produto produto : produtos) {  
    System.out.println(getNome() + " custa " + produto.getPreco())  
};  
}
```

Será que nós conseguimos melhorar o código ainda mais? Nós estamos passando por todas as casas dos produtos (0, 1, 2, 3 e 4). Porém, quando os 4 primeiros produtos estão ordenados, a última casa fica com o item que sobra. Por isso, é muito comum que o nosso `for` termine em `produtos.length - 1`. Porque quando sobra apenas o último elemento para ser ordenado, ele já ocupa a casa correta. Ele também foi ordenado.

O último elemento que sobra no processo de ordenação é o mais caro ou o maior de todos. Então, é comum que o algoritmo seja implementado com `-1`. Assim, cada elemento ocupará a sua posição correta.

3.6 EXTRAINDO A ORDENAÇÃO POR SELEÇÃO DE ELEMENTOS

Indicamos o produto mais barato, o segundo mais barato, o terceiro mais barato do nosso código. E com isso, temos um array organizado e podemos extrair uma função de ordenação.

```
for(int atual = 0; atual < produtos.length - 1; atual++) {  
    int menor = buscaMenor(produtos, atual, produtos.length - 1);  
    Produto produtoAtual = produtos[atual];  
    Produto produtoMenor = produtos[menor];  
    produtos[atual] = produtoMenor;  
    produtos[menor] = produtoAtual;  
}
```

Lembre-se: uma função de ordenação receberá o array que será ordenado. Vamos recortar o código:

```
for(int atual = 0; atual < produtos.length - 1; atual++) {  
    int menor = buscaMenor(produtos, atual, produtos.length - 1);  
};  
    Produto produtoAtual = produtos[atual];
```

```

        Produto produtoMenor = produtos[menor];
        produtos[atual] = produtoMenor;
        produtos[menor] = produtoAtual;
    }
}

```

E fazer com que ele ordene os nossos produtos:

```
ordena(produtos);
```

Criamos o método `ordena` com a parte recortada:

```

private static void ordena(Produto[] produtos) {
    for(int atual = 0; atual < produtos.length - 1; atual++) {
        int menor = buscaMenor(produtos, atual, produtos.length - 1);
        Produto produtoAtual = produtos[atual];
        Produto produtoMenor = produtos[menor];
        produtos[atual] = produtoMenor;
        produtos[menor] = produtoAtual;
    }
}

```

Como nosso array troca as posições dos elementos que estão dentro, vamos ordená-lo. Ao testarmos novamente o código, veremos que funciona:

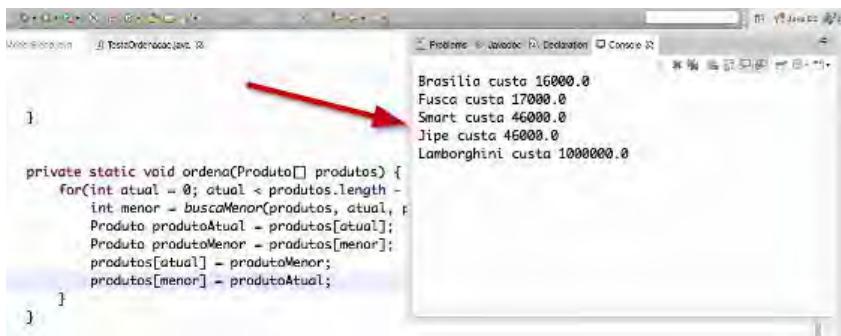


Figura 3.6: Método `ordena`

Lembrando de que, em diversas linguagens, seja porque não sabemos o limite ou quantos elementos temos dentro do array, é comum que uma função de ordenação receba o tamanho `.length`. É comum recebermos os valores `produtos.length` e

`produtos.length -1` como argumento. Isto significa que o tamanho do array (`produtos.length`) costuma ser usado em funções de ordenação.

```
ordena(produtos, produtos.length);

for(Produto produto : produtos) {
    System.out.println(produto.getNome() + " custa " + produto.getPreco());
}
```

Como isso acontece em várias linguagens, vamos substituir `produtos.length` por tamanho no nosso código:

```
private static void ordena(Produto[] produtos, int tamanho) {
    for(int atual = 0; atual < tamanho - 1; atual++) {
        int menor = buscaMenor(produtos, atual, tamanho - 1);
        Produto produtoAtual = produtos[atual];
        Produto produtoMenor = produto[menor];
        produtos[atual] = produtoMenor;
        produtos[menor] = produtoAtual;
    }
}
```

Usamos o termo `tamanho`, porém isto é uma ilusão. Pode ser que existam 50 posições, mas que na verdade tenhamos apenas 10 elementos. Logo, em vez de `tamanho`, chamaremos `quantidadeDeElementos`, porque é a quantidade de elemento que temos no array. Então, iremos de 0 até a `quantidade de elementos menos 1`. O nosso código ficará assim:

```
private static void ordena(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos - 1; atual++) {
        int menor = buscaMenor(produtos, atual, quantidadeDeElementos - 1);
        Produto produtoAtual = produtos[atual];
        Produto produtoMenor = produto[menor];
        produtos[atual] = produtoMenor;
        produtos[menor] = produtoAtual;
    }
}
```

Em casos em que não sabemos com exatidão a quantidade de elementos que temos dentro do array, usamos o `length` como parâmetro para a nossa função. Em Java, nós temos como saber e podemos usar `produtos.length`.

```
ordena(produtos, produtos.length);

for(Produto produto : produtos) {
    System.out.println(produto.getNome() = " custa " + produto.getPreco());
}
```

Em outras linguagens, na definição ou no formato de uma ordenação, é comum recebermos este argumento. Por isso, estamos criando o código detalhadamente para que ele funcione corretamente. Em seguida, vamos testar e ver o que está acontecendo.

3.7 VISUALIZANDO A TROCA DE POSIÇÕES DOS ELEMENTOS DURANTE A SELEÇÃO

Vamos testar passo a passo o que está acontecendo com o nosso código. Para isso, colocaremos alguns `System.out`s a mais para ordenarmos o nosso array. Quando passarmos por cada uma das casas, colocamos esses `System.out`s para que o programa imprima "Estou na casinha 0, 1, 2, 3".

```
System.out.println("Estou na casinha " + atual)
```

Porém, não vamos passar na casinha 4. Por ser a última, sabemos que ela é o maior elemento de todos.

```
private static void ordena(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos - 1; atual++) {
        System.out.println("Estou na casinha " + atual)

        int menor = buscaMenor(produtos, atual, quantidadeDeElementos)
    }
}
```

```

tos - 1);
    Produto produtoAtual = produtos[atual];
    Produto produtoMenor = produto[menor];
    produtos[atual] = produtoMenor;
    produtos[menor] = produtoAtual;
}
}

```

Além de passarmos pelas casas, o que faremos depois? Vamos trocar os elementos de lugar:

```
System.out.println("Trocando " + atual + " com o " + menor);
```

Quais elementos serão trocados? Nós trocaremos o `getNome()` com o `produtoMenor.getNome()`.

```
System.out.println("Trocando "+ produtoAtual.getNome() + " " + pro-
dutoMenor.getNome());
```

Estamos trocando uma posição por outra, o que significa "trocar esse carro com aquele".

```

private static void ordena(Produto[] produtos, int quantidadeDeEle-
mentos) {
    for(int atual = 0; atual < quantidadeDeElementos - 1; atual++)
    {

        System.out.println("Estou na casinha " + atual)

        int menor = buscaMenor(produtos, atual, quantidadeDeElemen-
tos - 1);
        System.out.println("Trocando " + atual + " com o " + menor)
    }

    Produto produtoAtual = produtos[atual];
    Produto produtoMenor = produto[menor];

    System.out.println("Trocando "+ produtoAtual.getNome() + " "
" + produtoMenor.getNome());

    produtos[atual] = produtoMenor;
    produtos[menor] = produtoAtual;
}
}

```

E veremos as trocas acontecerem! A quantidade de alterações deverá ser a mesma que o número de casinhas menos 1. Ou seja, se

temos cinco casas, apenas verificaremos as casas 0, 1, 2, 3. A casa 4 não será analisada.

Ao rodarmos o programa, poderemos conferir as trocas que aconteceram:

```
Estou na casinha 2
Trocando 2 com o 3
Trocando Lamborghini Smart
Estou na casinha 3
Trocando 3 com o 4
Trocando Lamborghini Jipe
Brasilia custa 16000.0
Fusca custa 17000.0
Smart custa 46000.0
Jipe custa 46000.0
Lamborghini custa 1000000.0
```

Estamos na casinha 0 e trocamos a Lamborghini pelo 2 (a Brasília). A alteração faz sentido. A Brasília era o carro mais barato e a Lamborghini estava no início da lista, por isso trocamos os dois carros de lugar.

Quem ficou na casinha 1? Foi o Jipe. Agora na casinha 1, vou trocar o Jipe com o Fusca, porque buscamos o menor de todos, a partir da posição atual. Colocamos o Fusca no início e movemos o Jipe para o fim.

Passamos para a casinha 2, onde está a Lamborghini. Vamos trocá-la de posição com o Smart, que é o carro mais próximo, com o menor preço.

Na casinha 3, eu tenho a Lamborghini. Vamos trocá-la de lugar com o Jipe. Após movê-los, sobra apenas a casinha 4. Esta não é preciso revisar, porque já sabemos qual elemento é o maior de todos. O Lamborghini é o produto mais caro.

Em seguida, imprimo os resultados. No total, fizemos as trocas das quatro casinhas: 0, 1, 2 e 3. Agora veremos o processo

visualmente, já que o código foi implementado.

3.8 SIMULANDO NO PAPEL O ALGORITMO DE SELEÇÃO DE MENORES ELEMENTOS PARA ORDENAÇÃO

Vamos revisar o nosso algoritmo de ordenação?

Nós fizemos um laço que passava por cada casa com a variável `atual`, selecionando qual elemento era o menor a partir de determinada posição. Quem é o menor a partir da casinha 0? É a Brasília. Então, trocaremos o produto atual da casinha 0 para o produto da casinha menor.

Passamos para o elemento 1. Qual é o menor a partir dele? É o Fusca. Vamos selecioná-lo e fazer a troca de posições. Com o laço, fomos resolvendo nosso problema com os outros produtos.

```
for(int atual = 0; atual < quantidadeDeElementos - 1; atual++) {  
    int menor = buscaMenor(produtos, atual, quantidadeDeElementos  
    - 1);  
    Produto produtoAtual = produtos[atual];  
    Produto produtoMenor = produto[menor];  
  
    produtos[atual] = produtoMenor;  
    produtos[menor] = produtoAtual;  
}
```

Isto quer dizer que a nossa função de ordenação recebe não só o array com os elementos, mas também a quantidade de itens que temos dentro dele. No nosso caso, a quantidade é 5, o que significa que vamos trabalhar da casinha 0 até a 3. O que sobrar na casinha 4 será o produto mais caro de todos.

Vamos trabalhar com outras variáveis também: `atual` e `menor`. Precisamos saber em que casinha estamos e qual é o menor a partir da posição `atual`. Eu criei duas variáveis auxiliares:

`produtoAtual` e `produto menor`. Elas referenciarão os produtos para que as trocas de posições sejam possíveis.

Em seguida, vamos simular os algoritmos: a variável `atual` começou com 0. A variável `menor`, a partir da posição 0, será 2.

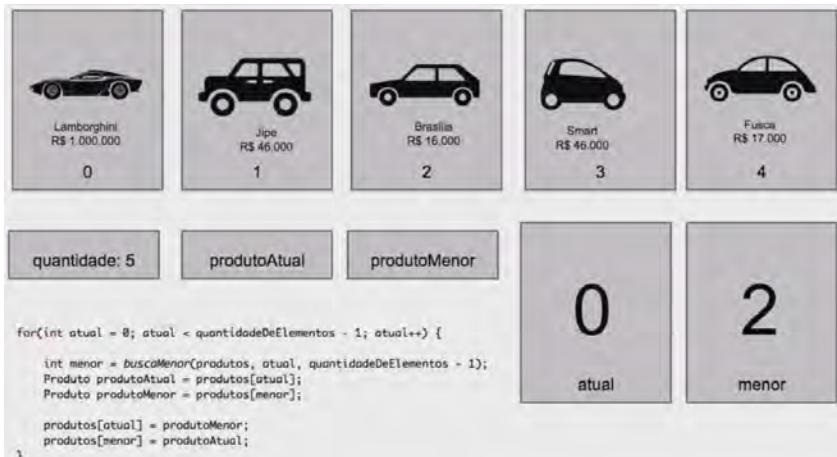


Figura 3.7: Simulação do algoritmo de seleção — passo 1

Nosso `produtoAtual` estará apontando para a Lamborghini. Já o `produtoMenor` apontará para a Brasília. Quando invertemos as posições dos dois elementos da lista, a Brasília passa a ocupar o lugar da Lamborghini.

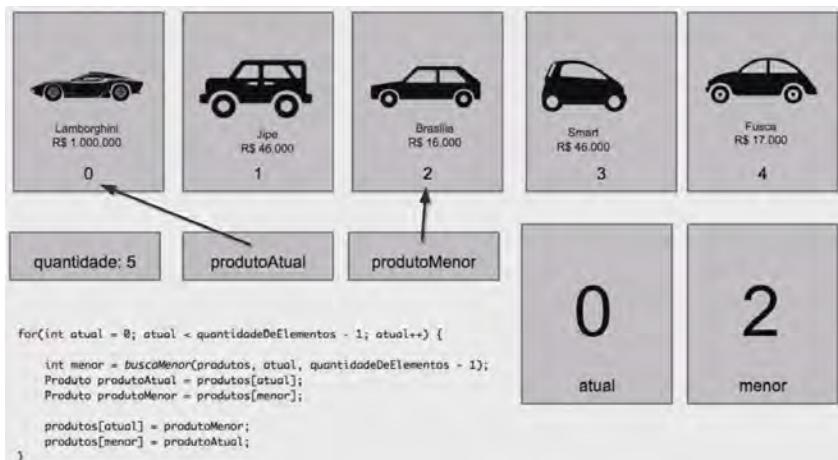


Figura 3.8: Simulação do algoritmo de seleção — passo 2

A casinha 1 será a nossa próxima variável `atual`. A partir desta posição, verificaremos qual elemento é o menor. Identificamos que, a partir da posição 1, o menor é 4. Vamos fazer o mesmo que fizemos anteriormente: trocaremos os elementos de lugar e substituiremos o Fusca pelo Jipe.

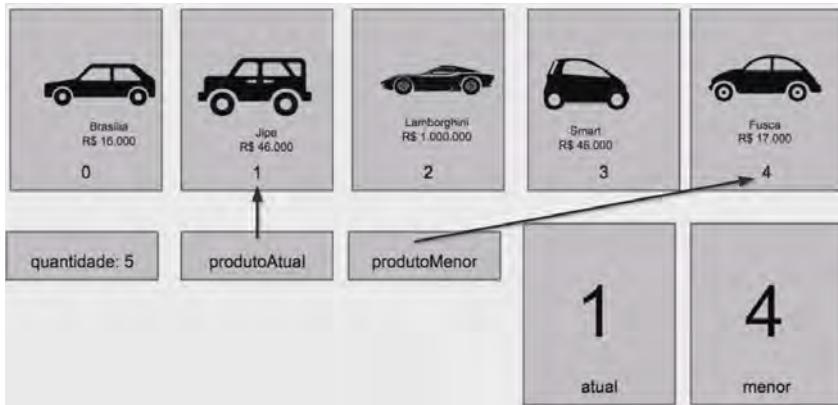


Figura 3.9: Simulação do algoritmo de seleção — passo 3

Passamos para a casinha 2. Qual é o menor elemento a partir da segunda posição? É o Smart. Vamos referenciar o `produtoAtual` e

o `produtoMenor`, e finalizamos invertendo os produtos de lugar. Vamos bem até aqui!

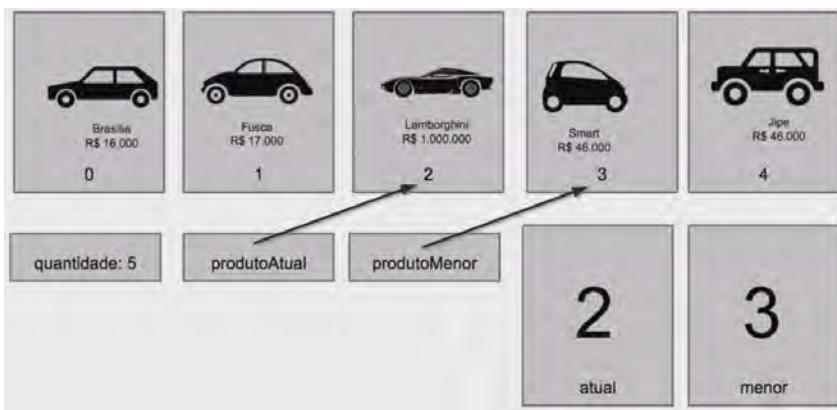


Figura 3.10: Simulação do algoritmo de seleção — passo 4

Agora estamos na posição 3. Fazemos novamente a pergunta: qual é o menor elemento a partir da minha posição `atual`? O menor é o 4. Voltamos a referenciar o `produtoAtual` e o `produtoMenor`, para depois trocarmos a Lamborghini e o Jipe de posição.

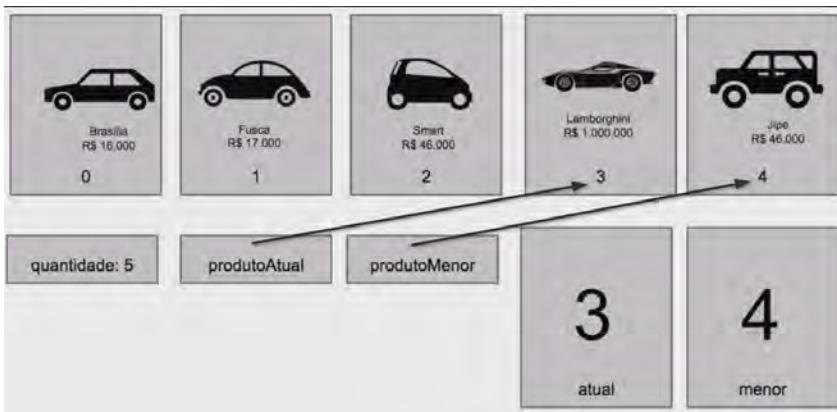


Figura 3.11: Simulação do algoritmo de seleção — passo 5

Como a Lamborghini já ficou no fim da nossa lista, não precisaremos verificar a última posição. Se passarmos pela casinha 4, sairemos do nosso laço. Então, já encontramos a solução do nosso problema e os produtos estão ordenados.

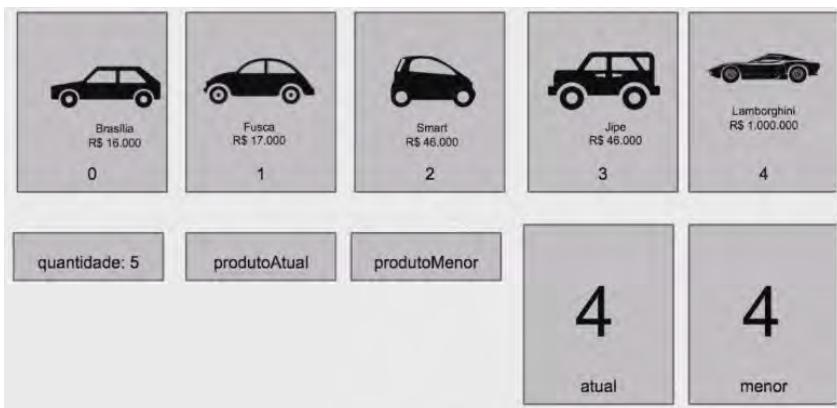


Figura 3.12: Simulação do algoritmo de seleção — passo 6

3.9 SELECTION SORT

O algoritmo que nós usamos para resolver o problema de selecionar o menor a partir de uma parte do nosso array e trocá-lo de posição é chamado de **ordenação por seleção** (*Selection Sort*). Ele seleciona o menor a partir do instante atual e permite que o reposicionemos na lista.

O *Selection Sort* é capaz de resolver o nosso problema de ordenação. Ele passa por cada elemento e pergunta "Quem deve estar nesta posição? É esse". Então, coloca cada item em uma ordem.

Além disso, ele utiliza a nossa função para encontrar o `menor`. Isso significa que, ao implementarmos primeiro a função de seleção do menor, simplificamos o nosso algoritmo de ordenação .

O algoritmo de ordenação, o *Selection Sort*, se baseia no algoritmo de seleção do menor.

3.10 O PRÓXIMO DESAFIO: QUESTIONANDO A VELOCIDADE DE UM ALGORITMO

Nós vimos como é o processo de ordenação dos carros e produtos, em geral. Também já conseguimos ordenar. Por exemplo, se quiséssemos ordenar os resultados das provas do Enem ou de um concurso público, poderíamos utilizar o mesmo processo.

E se quiséssemos descobrir quem ficou em primeiro lugar no campeonato de futebol de acordo com a pontuação alcançada? Poderíamos seguir o mesmo processo de ordenação: selecionamos o primeiro (aquele que obteve menos pontos) e o último (o elemento que mais ganhou pontos). Com este tipo de ordenação, podemos usar o algoritmo usado até agora, o processo de seleção, o **Selection Sort**.

Porém, tem alguma coisa errada aqui! Toda vez que tentamos ordenar uma lista de elementos como no exemplo dos cinco carros, precisamos comparar o atual com todos os outros itens seguintes para reposicioná-los. Nós sempre passamos por todos os elementos. Isto significa que, quando fizermos o `for` dentro dele, terei de verificar cada elemento. Parece trabalhoso.

Se tivermos um `for` de 0 a 100, para cada um dos elementos teremos de criar outros `for`s que passem por dentro de cada um deles. Ficamos com a sensação de que é preciso fazer muita coisa. Vamos formalizar esta "sensação" e entender melhor o que queremos dizer com "muita coisa". Também entenderemos o que significa um algoritmo **lento**.

É correto termos a impressão de que podemos realizar o processo de ordenação mais rapidamente. É nesta tentativa de fazer as coisas de uma forma mais rápida e melhor que nós buscaremos novas alternativas para o *Selection Sort*.

Vamos procurar uma nova maneira de ordenação dos nossos elementos e, a partir disto, poderemos comparar qual é o mais rápido ou o mais lento realmente.

Esse será o nosso próximo passo: encontrar uma maneira de ordenar elementos de uma forma diferente da que utilizamos até agora. Uma maneira que as pessoas também usem no cotidiano, por exemplo, quando jogam baralho.

CAPÍTULO 4

INSERINDO E ORDENANDO: O INSERTION SORT

4.1 ORDENANDO CARTAS DE BARALHO

Queremos encontrar uma nova maneira de **ordenar**, de posicionar do menor para o maior. Independente de meus elementos serem produtos, notas de alunos, pontuação de um time de futebol, votação de políticos brasileiros etc. Nossa objetivo é ordenar para descobrir quais são os maiores e os menores elementos.

Vamos pensar em um **jogo de baralho**, no qual recebemos cinco cartas. Algumas variações de poker são jogadas com esta quantidade de cartas. Vamos usá-lo como exemplo, mas desconsiderando as regras e a relevância que cada uma delas pode ter no jogo.

Em uma partida de poker, eu recebi as seguintes cartas do naipe de copas: 7, 4, 10, 2 e 8:

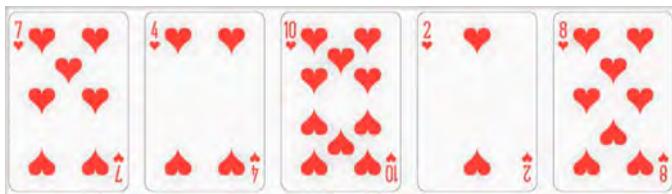


Figura 4.1: Jogo de poker

O meu grande desafio será colocar as cartas recebidas em ordem!

Quando jogamos baralho, observe que é comum sentirmos o impulso de ordenar as cartas que estão na nossa mão. À medida que as recebemos, começamos naturalmente a movê-las de posição, dizendo mentalmente "essa vem para cá e essa vai para lá".

A ordenação das cartas de baralho é feita passo a passo, porque temos muitos elementos para ordenar. Começamos com uma carta e a colocamos em uma posição que supomos que seja a correta. Depois, seguimos ajustando a ordem à medida que recebemos novas cartas.

É isso que farei adiante: ordenar as cinco cartas que acabamos de receber. Vamos realizar o processo lentamente?

Primeiro, tenho de observar minhas cartas. Verifiquei que tenho 7, 4, 10, 2 e 8 na minha mão. Então, percebi que o 4 é menor do que o 7 e, por isso, vou movê-lo para o início da fileira de cartas.

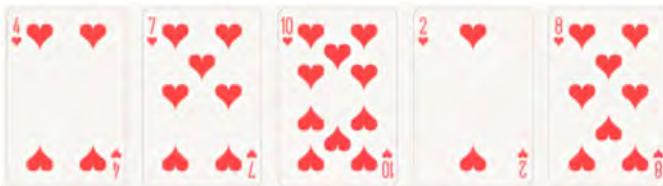


Figura 4.2: Ordenando as cartas do baralho — passo 1

Segui para o 10, mas não precisei modificar a ordem. Quando verifiquei o 2, concluí que ele é menor do que todas as cartas anteriores, logo, movi a carta para o começo.



Figura 4.3: Ordenando as cartas do baralho — passo 2

Sobrou a última carta. Tenho 8 no fim. Porém, eu sei que esta não é a posição que corresponde ao elemento, porque ela é menor do que 10. Então, vou trocá-las de lugar.

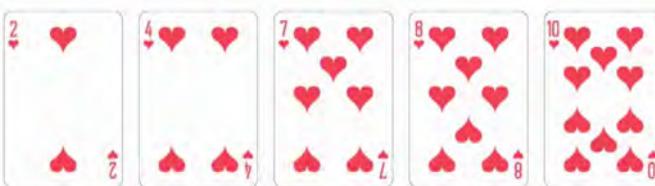


Figura 4.4: Ordenando as cartas do baralho — passo 3

Depois disso, será o suficiente. Não precisarei fazer novas alterações.

O que eu fiz para ordenar as cartas: observei as cartas na minha mão e troquei suas posições.

Identificamos que a segunda era menor do que a primeira, por isso, trocamos as duas de lugar. Passamos para a terceira, momentaneamente ela estava em uma boa posição. Na quarta, percebemos que ela não estava no lugar correto e a movemos para a esquerda até chegar ao início da fileira. Finalmente, na última carta, precisamos trocá-la de lugar com a anterior. Com isso, nossa ordenação estava finalizada. Terminamos.

Essa foi a forma como eu fiz a ordenação na minha mão. E você, o que fez com as suas cartas quando as recebeu na ordem que citei?

4.2 SIMULANDO PASSO A PASSO A ORDENAÇÃO DE CARTAS DE BARALHO

Revisaremos passo a passo o que fiz enquanto ordenava as cartas do baralho. Tenho a experiência de ter ordenado cartas várias vezes, quase as posso organizar automaticamente. Porém, no que estou pensando durante do processo?

Vamos agora observar detalhadamente o algoritmo (o processo) que executamos. O programa só compreenderá `for` s e `if` s, laços e condições.

Quais laços e condições executei no meu algoritmo? Quais elementos usei para fazer comparações? Voltamos para o exemplo das cartas.

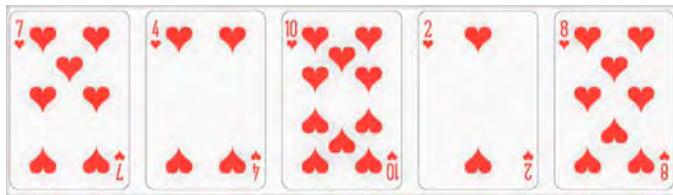


Figura 4.5: Jogo de poker

Eu recebi estas cartas. Comecei verificando minha primeira carta: 7. Não posso fazer muita coisa tendo apenas esta.

Passamos para a segunda. Como nosso processo passa por cada elemento, o computador vai precisar de uma variável que diga "essa é a carta que estamos ordenando agora". Como posso organizar até o 4? Observo a carta e a comparo com a outra que recebi.

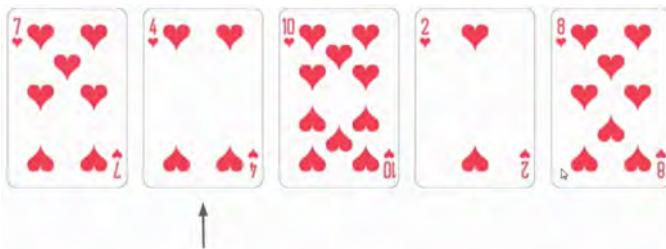


Figura 4.6: Simulando a ordenação de cartas de baralho — passo 1

Nosso segundo elemento é maior ou menor do que o primeiro?
É menor. Então vou movê-lo para o início.

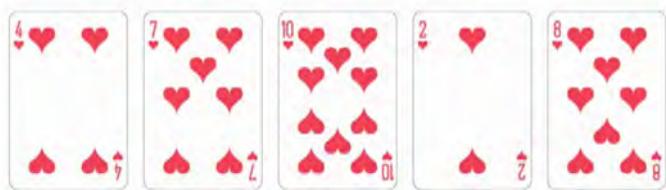


Figura 4.7: Simulando a ordenação de cartas de baralho — passo 2

O próximo elemento será o 10. As cartas anteriores na fileira são maiores do que a atual? Se for, precisarei trocá-la de lugar. Porém, a carta 10 é a maior das três e não será preciso movê-la.
Momentaneamente, as três primeiras cartas estão ordenadas.

Sigo para a carta 2 e a comparo com a anterior.

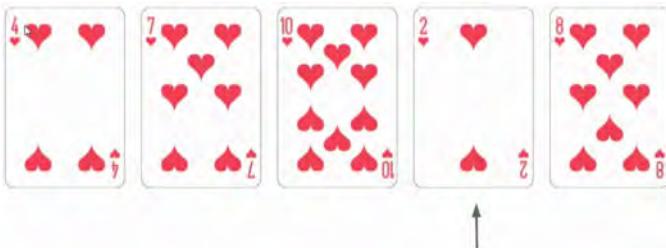


Figura 4.8: Simulando a ordenação de cartas de baralho — passo 3

Ela é maior ou menor do que a 10? É menor. Mas percebo que as demais cartas recebidas também são maiores, então, vou trocar a posição do 2 até que esteja no começo da fileira.

Agora as quatro primeiras cartas estão em ordem, fica faltando a última. Faço a comparação entre o 8 e o 10, e percebo que preciso trocá-las de lugar. Faço a mudança e termino o processo de ordenação dos elementos.

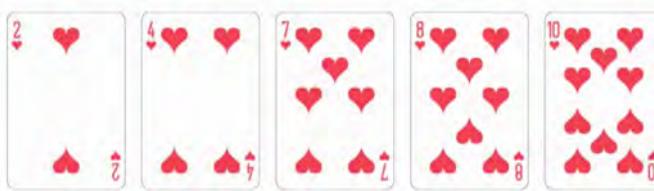


Figura 4.9: Simulando a ordenação de cartas de baralho: elementos organizados

Como foi feito meu processo? Analisei cada elemento (minha variável atual), sempre comparando com a carta anterior para identificar qual era a menor: se as duas fossem iguais, tanto faz minha decisão.

Podemos perceber que o processo passou por duas partes:

1. Passei por todas as cartas e identifiquei quais eram as menores.
2. Ao identificar qual era a menor, passei a procurar o lugar correto para inseri-la ordenadamente.

Isto significa que trabalhei com laços diferentes: um que analisava os elementos para frente, e outro que fazia a verificação para trás. É isto que faremos adiante.

4.3 PENSANDO NO ALGORITMO DE

ORDENAÇÃO

Vamos criar o nosso algoritmo?

Temos os cinco produtos do exemplo que já trabalhamos, os carros, e agora queremos ordená-los. Sabemos, de diversas linguagens, que precisaremos da variável `quantidade`, referente à quantidade de elementos — não importando se ela é do mesmo tamanho do array ou se é um parâmetro a mais. Precisamos conhecer a quantidade de elementos. No nosso caso, sabemos que é 5.

Também já percebemos que devemos verificar nossa lista da esquerda para direita, porque se organizarmos cartas de baralho, por exemplo, nós as receberemos uma de cada vez. Por isso, precisaremos da variável `atual`, que varrerá do primeiro até o último elemento.

Faremos o nosso laço que vai de **0 até 5 (exclusive)**, deixando o quinto elemento de fora. Nossa próximo passo será começar a fazer a nossa análise.

Nós vamos analisar o elemento atual, a carta que acabamos de receber. Quando vou analisar, também preciso comparar o preço do atual com o do produto anterior.

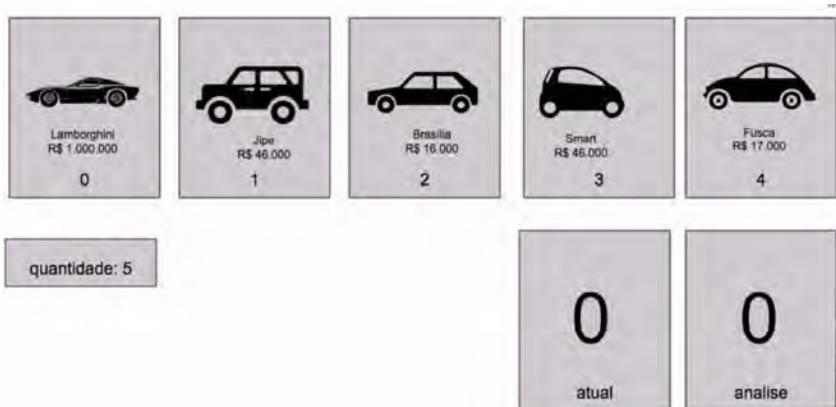


Figura 4.10: Análise — passo 1

Isto significa que compararemos o valor de quem está na posição analise com o valor daquele que estiver posicionado anteriormente na lista. Quando analisamos o produto 0 (a Lamborghini), analise é igual a 0. Depois, vamos verificar qual item veio antes (o elemento na posição analise - 1).

Precisamos considerar ainda se o item analisado é mais barato do que o elemento posicionado antes. Se o preço for menor, existe algo errado na ordenação e teremos de reposicioná-los. Trocarei as posições de quem está sendo analisado com quem está situado antes.

Após efetuar a troca, preciso fazer a mesma comparação com o produto que agora está localizado antes: analise -1 . Diminuo 1 para poder analisar os dois elementos anteriores.

```
atual de 0 até 5 (exclusive) {
    analise = atual
    enquanto(produtos[analise] < produtos[analise - 1]){
        troca produtos, analise, analise - 1
        analise = analise - 1
    }
}
```

Vamos testar o resultado do algoritmo e confirmar se realmente

foi o que descrevemos há pouco? Começamos com `atual = 0` , `analise = atual` , logo `analise = 0` . Considerando que:

```
enquanto(produtos[analise] < produtos[analise - 1]);
```

Quando o produto analisado for 0, quem será o produto `analise - 1` da minha lista? O programa não vai aceitar, porque nenhum elemento ocupa a posição `-1` . Não podemos deixar que isso aconteça. Precisamos ser específicos sobre isto no nosso `enquanto` .

Antes de verificar se o produto analisado é mais barato, precisamos nos certificar sobre a existência de um produto anterior. Se `analise` for igual a 0, não precisaremos analisar o elemento posicionado antes.

Então, se `analise` for maior do que 0 e o elemento que estiver posicionado antes for mais caro, faremos a troca.

```
enquanto(analise > 0 *E*  
         produtos[analise] < produtos[analise - 1])
```

Nosso algoritmo ficará assim:

```
atual de 0 até 5 (exclusive) {  
    analise = atual  
    enquanto(analise > 0 *E*  
             produtos[analise] < produtos[analise - 1]) {  
        troca produtos, analise, analise - 1  
        analise = analise - 1  
    }  
}
```

Se fizermos a simulação com o primeiro elemento (a Lamborghini), ele não se encaixará no nosso algoritmo, porque `analise = 0` e não teremos o produto da **posição - 1**.

Passaremos para `atual` igual a 1. Assim, `analise` é igual a `atual` , que é maior do que 0.

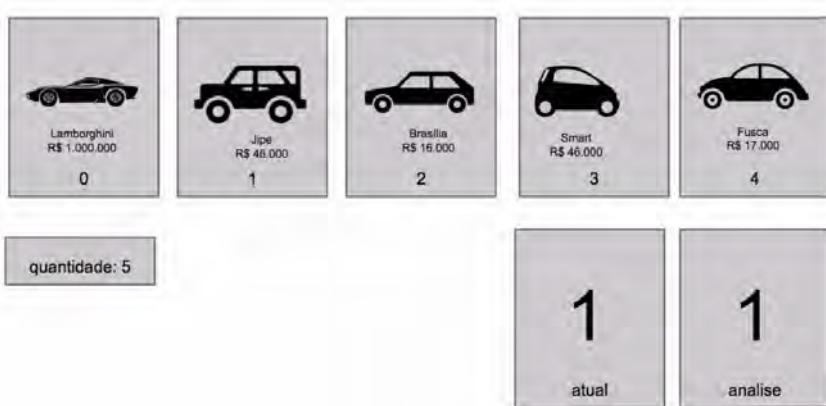


Figura 4.11: Análise — passo 2

Enquanto analise for maior do que 0, a condição será atendida. O **produto 1** (Jipe) é menor do que o **produto 0** (Lamborghini)? Sim. As duas condições são verdadeiras. Por isso, trocaremos os elementos de posição. Em seguida, diminuiremos 1 na analise .

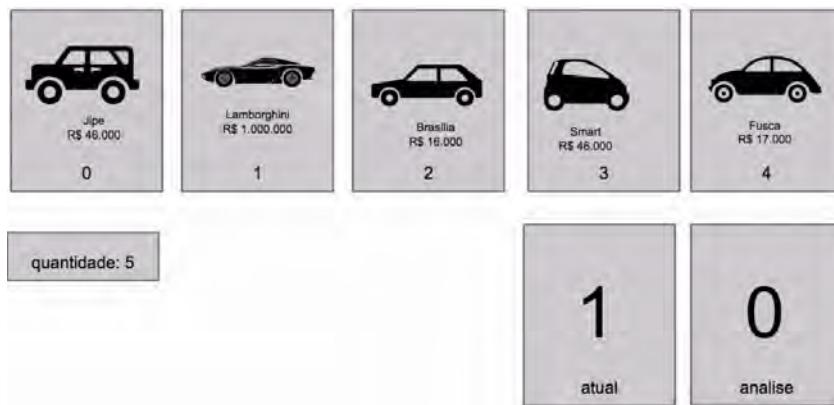


Figura 4.12: Análise — passo 3

Porém, o nosso laço pede que analise seja maior do que 0 para continuarmos as comparações. Então, paramos por aqui.

Seguimos para o próximo elemento, `atual = 2`. Vamos analisar a Brasília e descobrir onde ela deve ser inserida.

`analisé = atual`

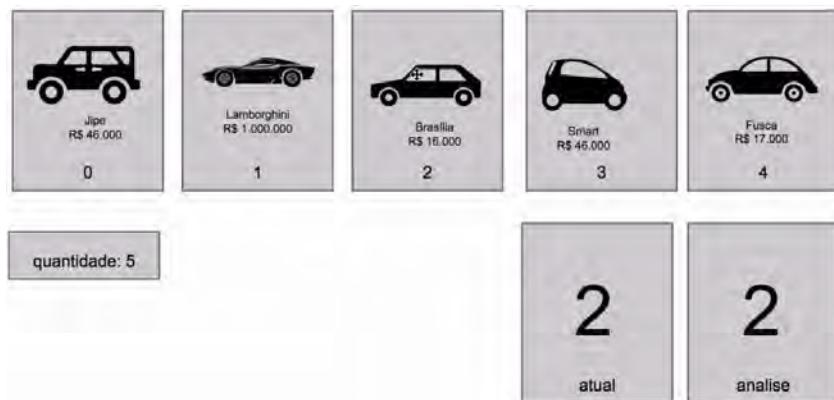


Figura 4.13: Análise — passo 4

A `analisé` é igual a 2, atendendo a condição de ser maior que 0. Também percebemos que o preço do `atual` é menor que o anterior. Outra condição atendida, então vamos trocar os elementos de posição.



Figura 4.14: Análise — passo 5

Agora, analise menos 1 é maior do que 0. O preço do analisado é menor do que o elemento anterior. O que fazemos? Repositionamos os dois carros.



Figura 4.15: Análise — passo 6

Porém, se novamente diminuirmos 1 de analise , ele será igual a 0 e ficará fora do nosso laço. Então, sem novas alterações. Passamos para o próximo atual , que é igual a 3.

A analise terá o mesmo valor e atenderá a condição de ser maior do que 0. O produto analisado (Smart) atende também à condição de ser menor que o anterior. Então, trocaremos os elementos de lugar.

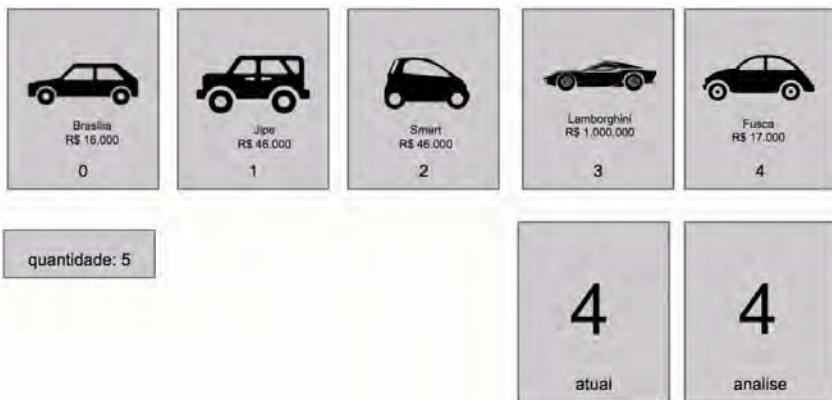


Figura 4.16: Análise — passo 7

Continuamos com analise igual a 2, o que atende a primeira condição. No entanto, o preço do Smart não é menor do que o Jipe. Por isso, não faremos alterações na ordenação.

Vamos analisar o próximo produto. A analise é igual a 4 e atende à condição de ser maior do que 0.

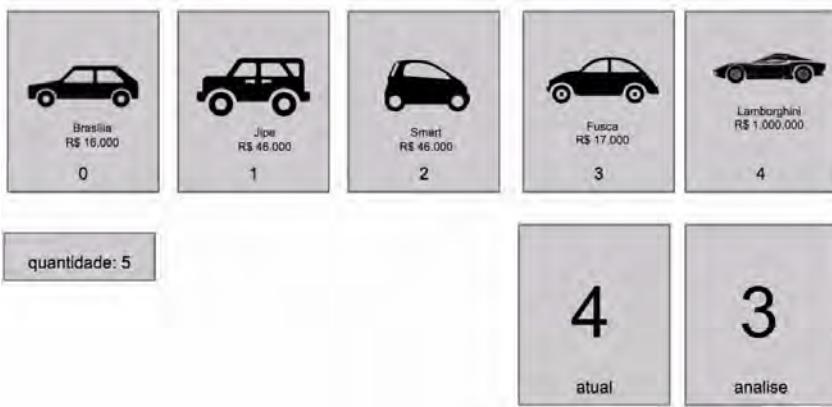


Figura 4.17: Análise — passo 8

O `produtos[analise]` (Fusca) é menor que o `produtos[analise - 1]`? Sim. Logo, trocaremos os produtos de

posição.

Diminuiremos 1 da analise e continuamos com nosso laço. O produto 3 é maior que zero e tem o preço menor que o **produto 2**. Vamos reposicioná-los.

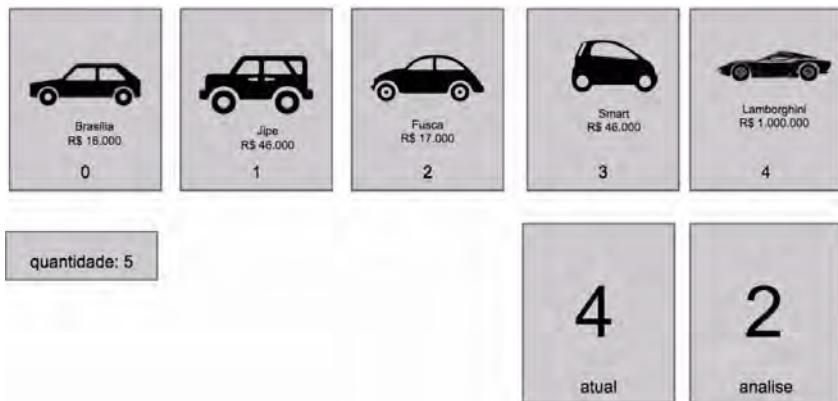


Figura 4.18: Análise — passo 9

Novamente, diminuiremos 1 da analise , que atende à condição de ser maior do que 0, enquanto o preço do Fusca é menor do que o Jipe. Vamos trocá-los de posição.

Diminuiremos 1 da analise . Será maior do que 0, porém o preço do produto não é menor do que o da Brasília.

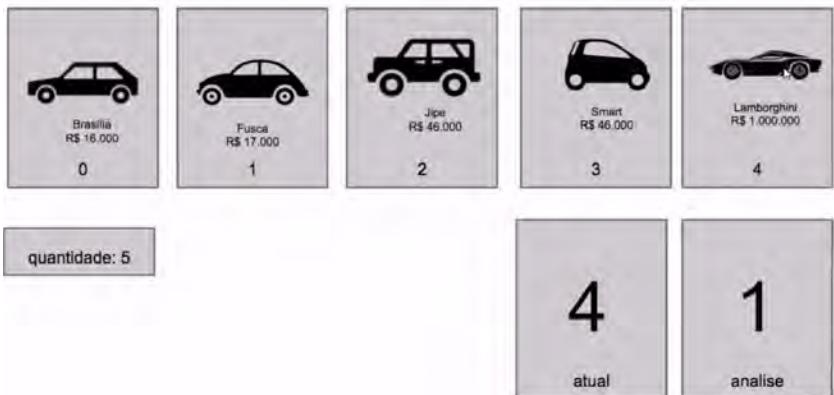


Figura 4.19: Análise — passo 10

Nosso `produtos[análise]` (Fusca) não será menor que `produtos[análise -1]` (Brasília). Seguiremos para o próximo, sem alterar a nossa ordenação.

A variável `atual` será igual a 5. Porém, definimos que 5 é `exclusive`. Então, nossos elementos já estão ordenados.

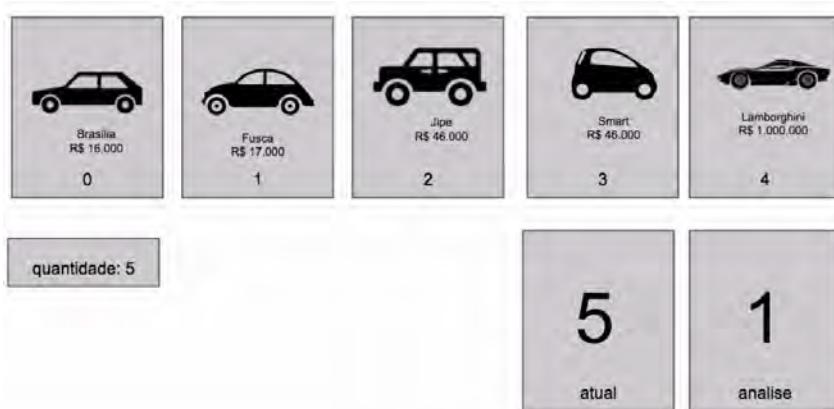


Figura 4.20: Análise — passo 11

O que foi feito? Criei um laço da esquerda para a direita e analisei cada item, identificando onde ele deveria ser incluído para

depois inseri-lo em ordem. O processo foi semelhante ao que fizemos com o exemplo das cartas de baralho.

À medida que fui recebendo uma nova carta, observei as que já estavam na minha mão e identifiquei onde deveria inseri-la. Repeti o mesmo processo até ordenar todos os elementos. Fui inserindo aos poucos as minhas cartas na fileira.

Vamos traduzir este processo para Java?

4.4 IMPLEMENTANDO A ORDENAÇÃO POR INSERÇÃO

Nós já temos um algoritmo de ordenação (o *Selection Sort*), que foi implementado com o método chamado **ordena**.

```
ordena(produtos, produtos.length);
```

Vou renomear o método `ordena` para `selectionSort`, pois teremos outras diversas ordenações:

```
selectionSort(produtos, produtos.length);
```

Queremos fazer uma nova ordenação que chamaremos de `novoSort`. Ela também receberá o nosso array e o tamanho da nossa lista.

```
selectionSort(produtos, produtos.length);
novoSort(produtos, produtos.length);
```

Vamos comentar a linha anterior:

```
//selectionSort(produtos, produtos.length);
```

Depois ordenaremos o que precisa ser ordenado. O método será igual e terá a mesma assinatura do `selectionSort`:

```
private static void novoSort(Produto[] produtos, int quantidadeDeElementos) {
```

Nós queremos implementar um novo laço que passe por todos os elementos. Então, o nosso `for` será:

```
for(int atual = 0; atual < quantidadeDeElementos; atual++);
```

Assim passaremos por todos os itens do array.

Por exemplo, selecionei o elemento da **posição 2** e, a partir dele, quero analisar os elementos anteriores. Vamos compará-los para encontrar a posição em que o elemento atual deve ser inserido. Isto significa que sempre compararemos o elemento analisado com os das posições anteriores, ou seja, a partir daquela posição:

```
int analise = atual;
```

E o que nós levaremos em conta na análise? Além do produto (`produtos[analise]`), vamos considerar seu preço (`getPreco()`). Caso ele seja menor do que o elemento posicionado anteriormente (`produtos[analise -1]`), o produto não está no lugar correto e precisa ser reposicionado. Enquanto (`while`) os elementos não estiverem ordenados, precisaremos seguir trocando os itens de posição.

```
while(produtos[analise].getPreco() < produtos[analise -1].getPreco());
```

Precisamos nos certificar de que os elementos serão trocados de posição. O `produtoAnalise` será o `produtos[analise]`, e o `produtoAnaliseMenos1` será `produtos[analise -1]`.

```
Produto produtoAnalise = produtos[analise];
Produto produtoAnaliseMenos1 = produtos[analise -1];
```

Como será feita a troca dos elementos? Trocaremos `produtos[analise]` e `produtos[analise -1]` de posições.

```
produtos[analise] = produtoAnaliseMenos1;
produtos[analise -1] = produtoAnalise;
```

Começamos a analisar cada casinha e fizemos as seguintes considerações: "esse elemento é mais caro ou mais barato do que o

elemento à esquerda? Se ele for mais caro, não precisaremos continuar com nossa análise, porque ele já está posicionado corretamente. Porém, se o preço dele for menor, precisaremos trocá-lo de lugar".

Após realizarmos a primeira troca, precisaremos continuar com as verificações para identificar se o valor também é menor do que os anteriores. Seguiremos pelos itens da esquerda (analise --) até encontrar a posição devida do elemento, na ordenação.

O nosso código ficará assim:

```
private static void novoSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos - 1; atual++)
    {
        int analise = atual;
        while(produtos[analise].getPreco() < produtos[analise - 1]
            .getPreco()) {
            Produto produtoAnalise = produtos[analise];
            Produto produtoAnaliseMenos1 = produtos[analise - 1];
            produtos[analise] = produtoAnaliseMenos1;
            produtos[analise - 1] = produtoAnalise;
            analise--;
        }
    }
}
```

Ao analisar a posição atual, verificamos se o preço do elemento é menor e se a posição é a devida. Fizemos as trocas quando necessário e seguimos a análise para o próximo (anterior) item. Repetimos o processo até encontrarmos a posição em que ele deveria ser inserido (o que acontece quando o preço do produto é mais caro do que o valor do elemento situado antes na lista). Quando o valor é maior, encontramos a posição justa e paramos o processo.

Vamos testar o nosso código e ver o que acontece? Quando rodamos o programa, a saída será um `Exception -1`.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
        at br.com.alura.algoritmos.TestaOrdenacao.novoSort(TestaOrdenacao.java:27)
        at br.com.alura.algoritmos.TestaOrdenacao.mais(TestaOrdenacao.java:15)
```

Ao acessarmos o `TestaOrdenacao.java:15` linha 27, descobriremos que o nosso problema está na ordenação das posições analise e analise -1.

Quando analise for igual a 0, analise -1 será igual a -1. Logo, não poderemos acessar o elemento. Isso significa que existe um limite de análise. E para que o programa não tente acessar a posição -1, precisamos criar uma condição e definir que analise deve ser maior do que 0 (`analise > 0`).

```
private static void novoSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos - 1; atual++) {
        int analise = atual;
        while(analise > 0 && produtos[analise].getPreco() < produtos[analise - 1].getPreco()) {
            Produto produtoAnalise = produtos[analise];
            Produto produtoAnaliseMenos1 = produtos[analise - 1];
            produtos[analise] = produtoAnaliseMenos1;
            produtos[analise - 1] = produtoAnalise;
            analise--;
        }
    }
}
```

Isso porque, se o analise for igual 1, o analise -1 será igual a 0. Enquanto não ultrapassarmos o limite da esquerda, vamos continuar reposicionando os elementos.

Agora se testarmos o algoritmo, o programa rodará corretamente e imprimirá os resultados.

```
Brasília custa 16000
Fusca custa 1700
Jipe custa 46000
```

```
Smart custa 46000
Lamborghini custa 1000000
```

Revisando rapidamente, nós começamos da esquerda para direita, passando por cada uma das casinhas. Recebemos cada carta (ou cada produto) e comparamos os elementos com os que tínhamos recebido antes. A partir disto, encontramos a posição em que deveríamos inseri-los.

No exemplo do baralho, quando já tínhamos três cartas e a quarta foi recebida, analisamos as anteriores para definir a posição em que deveríamos colocá-la. Nossa laço foi construído sempre comparando os elementos na posição atual com os itens anteriores. Quando o item recebido não atendia à condição de ser menor, ou chegávamos ao limite definido na esquerda, nossa análise chegava ao fim.

Para cada um dos nossos elementos, respeitando o `for` do `atual`, encontramos a posição correta na ordem. Para cada carta de baralho que recebemos, encontramos a posição adequada na lista.

Em seguida, simularemos este processo com calma.

4.5 LOGANDO AS INFORMAÇÕES DA ORDENAÇÃO POR INSERÇÃO

Nós já implementamos o código da nova ordenação. Vamos ver se ele funciona corretamente?

```
private static void novoSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos - 1; atual++) {
        int analise = atual
        while(analise > 0 && produtos[analise].getPreco() < produtos[analise - 1].getPreco()) {
            Produto produto Analise = produtos[analise];
            produtos[analise] = produtos[analise - 1];
            produtos[analise - 1] = produto;
            analise--;
        }
    }
}
```

```

        Produto produtoAnaliseMenos1 = produtos[analise -1];
        produtos[analise] = produtoAnaliseMenos1;
        produtos[analise -1] = produtoAnalise;
        analise--;
    }
}
}
}

```

Adicionaremos o `System.out` que já havíamos utilizado antes.

```
System.out.println("Estou na casinha " + atual);
```

Em cada uma das casinhas que serão analisadas, vamos acrescentar o `System.out` e informar nossa posição.

```

private static void novoSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos - 1; atual++) {
        System.out.println("Estou na casinha " + atual);

        int analise = atual
        while(analise > 0 && produtos[analise].getPreco() < produtos[analise - 1].getPreco()) {
            Produto produtoAnalise = produtos[analise];
            Produto produtoAnaliseMenos1 = produtos[analise -1];
            produtos[analise] = produtoAnaliseMenos1;
            produtos[analise -1] = produtoAnalise;
            analise--;
        }
    }
}

```

O que acontece durante o processo: quando começamos a analisar os elementos, fazemos mudanças nas posições. Então, nós queremos imprimir as trocas. Cada uma delas é feita entre um item e outro. E o que estamos trocando? As posições `analise` com `analise -1`.

```
System.out.println("Estou na casinha " + analise + " com " + (analise -1));
```

Também já podemos incluir o nome das variáveis:

```
private static void novoSort(Produto[] produtos, int quantidadeDeE
```

```

lementos) {
    for(int atual = 0; atual < quantidadeDeElementos - 1; atual++)
    {
        System.out.println("Estou na casinha " + atual);

        int analise = atual
        while(analise > 0 && produtos[analise].getPreco() < produtos[analise - 1].getPreco()) {
            System.out.println("Estou trocando " + analise + " com "
" + (analise -1));

            Produto produto Analise = produtos[analise];
            Produto produtoAnaliseMenos1 = produtos[analise -1];
            System.out.println("Estou trocando " + produtoAnalise.
getName()
                                + " com " + produtoAna
liseMenos1.getName());
            produtos[analise] = produtoAnaliseMenos1;
            produtos[analise -1] = produtoAnalise;
            analise--;
        }
    }
}

```

As variáveis ainda estão com nomes grandes. Nós vamos melhorá-los. Porém, antes veremos o algoritmo rodando. Mudaremos as posições até que todos os elementos estejam inseridos no lugar correto.

Vamos ver as trocas acontecendo?

```

Estou na casinha 0
Estou na casinha 1
Estou trocando 1 com 0
Estou trocando Jipe com Lamborghini
Estou na casinha 2
Estou trocando 2 com 1
Estou trocando Brasília com Lamborghini
Estou trocando 1 com 0
Estou trocando Brasília com Jipe
Estou na casinha 3
Estou trocando Smart com Lamborghini
Estou na casinha 4
Estou trocando 4 com 3
Estou trocando Fusca com Lamborghini
Estou trocando 3 com 2
Estou trocando Fusca com Smart

```

```
Estou trocando 2 com 1
Estou trocando Fusca com Jipe
Brasilia custa 16000.0
Fusca custa 17000.0
Smart custa 46000.0
Jipe custa 46000.0
Lamborghini custa 1000000.0
```

O programa passou pela casinha 0 e recebeu a Lamborghini. Tem algum produto posicionado antes da casinha 0? Não. Então, ele não trocou os produtos de lugar.

Depois, o programa recebeu o segundo carro e observou que o Jipe é mais barato do que a Lamborghini. Os carros foram trocados de posição. Por que o algoritmo não tentou seguir com as trocas? Pois quando ele chega à posição 0, precisa parar.

Vamos imprimir o resultado deste array e ver passo a passo o processo. O código seguinte já imprime todos os produtos.

```
for(Produto produto : produtos) {
    System.out.println(produto.getNome() + " custa " +
        produto.getPreco());
}
```

Adicionaremos uma nova impressão e extrairemos um método. Clico em **Extract Method**, que vamos chamar de **imprime(produtos)**.

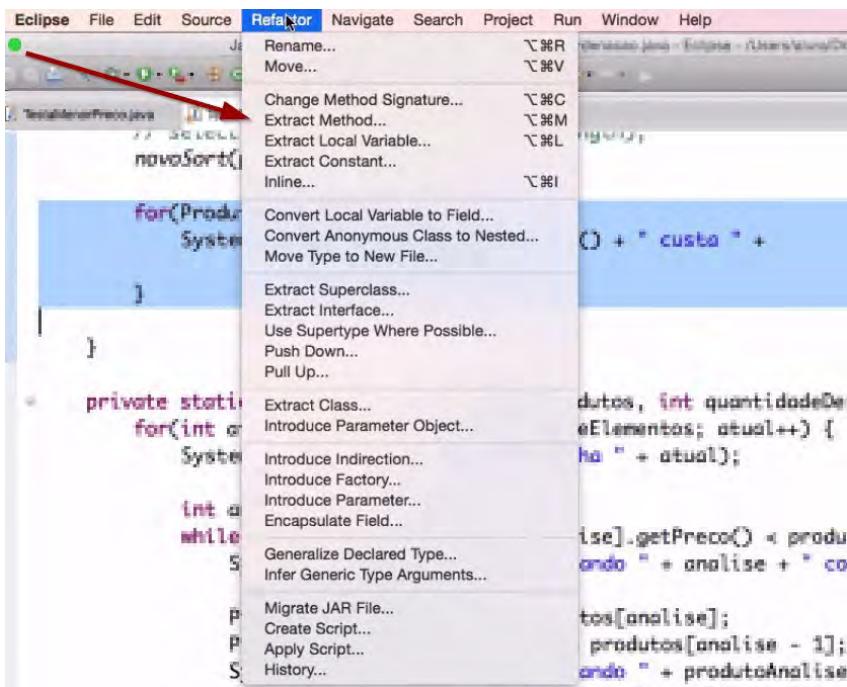


Figura 4.21: Extract method

Dentro do nosso novo *Sort*, vamos imprimir os produtos no fim de cada nova rodada. Logo, adicionaremos alguns `System.out`s:

```

imprime(produtos);
System.out.println();
System.out.println();
System.out.println();
System.out.println();

```

Ao rodarmos o algoritmo, o resultado será:

```

Estou na casinha 0
Lamborghini custa 1000000.0
Jipe custa 46000.0
Brasília custa 16000.0
Smart custa 46000.0
Fusca custa 17000.0

```

Primeiro, estou na casinha 0 (da Lamborghini). Não terei outra

casinha para compará-la e a ordem permanecerá a mesma, com todos os produtos na mesma posição. Porém, quando eu for analisar a casinha 1, será diferente.

```
Estou na casinha 1
Estou trocando 1 com 0
Estou trocando Jipe com Lamborghini
Jipe custa 46000.0
Lamborghini custa 1000000.0
Brasilia custa 16000.0
Smart custa 46000.0
Fusca custa 17000.0
```

Vou analisar o Jipe com a Lamborghini. Qual elemento é o mais barato? O Jipe. Por isso, o programa me informa que trocará o Jipe com a Lamborghini. E por que ele não continua com as trocas? Porque ele chegou ao limite da esquerda, a casinha 0.

Em seguida, o programa imprime o nosso array com os dois produtos reposicionados. Vou para o próximo.

```
Estou trocando 1 com 0
Estou trocando Jipe com Lamborghini
Jipe custa 46000.0
Lamborghini custa 1000000.0
Brasília custa 16000.0
Smart custa 46000.0
Fusca custa 17000.0
```

Estou na casinha 2, na qual está a Brasília (que custa R\$ 16.000). Comparo o produto com a Lamborghini (que custa R\$ 1.000.000) e percebo que a Brasília é menor. Troco os dois produtos de posição. Em seguida, comparo a Brasília com o Jipe e farei uma nova troca.

Quando ela ocupar a posição 0, chegaremos ao limite da esquerda e o programa não fará mais alterações. Continuarei a análise na casinha 3.

```
Estou na casinha 3
Estou trocando 3 com 2
Estou trocando Smart com Lamborghini
Brasília custa 16000.0
```

```
Jipe custa 46000.0
Smart custa 46000.0
Lamborghini custa 1000000.0
```

Esta é a casinha do Smart. Comparo o preço do produto com o da Lamborghini. Qual tem o menor valor? O Smart. Farei a análise de preços com o Jipe. Porém, a nossa comparação deve identificar o produto **menor**.

```
while(analise > 0 && produtos[analise].getPreco() < produtos[analise -1].getPreco())
```

Então, o que farei é verificar: o valor R\$ 46.000 é menor do que R\$ 46.000? Não. Logo, o programa ficará no Jipe, sem realizar a segunda troca. Em seguida, ele imprime como ficou a nova ordem: Brasília, Jipe, Smart, Lamborghini e Fusca.

Agora vou para a última casinha.

```
Estou na casinha 4
Estou trocando 4 com 3
Estou trocando Fusca com Lamborghini
Estou trocando 3 com 2
Estou trocando Fusca com Smart
Estou trocando 2 com 1
Estou trocando Fusca com Jipe
Brasília custa 16000.0
Fusca custa 17000.0
Jipe custa 46000.0
Smart custa 46000.0
Lamborghini custa 1000000.0
```

Estou na casinha 4. O programa vai comparar o Fusca com os outros elementos até encontrar o lugar correto para inseri-lo. Serão feitas trocas de posições com a Lamborghini, com o Smart e com o Jipe. No entanto, quando a análise comparar o Fusca com a Brasília, não haverá trocas. O programa detém o processo aqui.

A ordem final dos elementos será: Brasília, Fusca, Jipe, Smart e Lamborghini. Como ficou o nosso algoritmo?

Ele passa em todas as casinhas e, a partir de cada uma delas,

analisa os elementos anteriores. Quando os elementos verificados atendiam às condições, eram trocados de lugar. O processo foi repetido diversas vezes até que as trocas não fossem mais necessárias. O reposicionamento se tornava desnecessário em duas condições:

- Quando a análise chegava à posição limite à esquerda;
- Quando o preço do elemento anterior era menor do que o analisado.

Temos assim a ordenação que insere cada elemento na posição adequada.

4.6 PEQUENAS REFATORAÇÕES E MELHORIA DO CÓDIGO

Nós acompanhamos o nosso algoritmo rodando passo a passo. Observe que percebemos um detalhe: na primeira casinha, quando o `atual` é 0, `análise` também será igual 0.

```
int análise = atual;
```

Se `análise` é igual a 0, ele nem entrará no `while`.

```
while(análise > 0 && produtos[análise].getPreço() < produtos[análise-1].getPreço())
```

É óbvio. Quando você observa o primeiro produto e procura onde inseri-lo, não existe outro item para compará-lo. É desnecessário começar da posição 0. Por isso, é comum que o algoritmo seja iniciado a partir do 1. Então, para o `novoSort`, modificamos o `atual` para começar a partir do 1:

```
for(int atual = 1; atual < quantidadeDeElementos; atual++) { }
```

Da mesma forma, é impossível começar a ordenação quando

recebemos a primeira carta de baralho. Também não podemos fazer nada quando analisamos o primeiro produto, ou o primeiro preço de uma lista. Por isso, nosso algoritmo está ignorando a primeira casinha. Se fizermos o teste, veremos que tudo está funcionando corretamente.

Temos como resultado a seguinte ordenação:

```
Brasília custa 16000.0
Fusca custa 17000.0
Jipe custa 46000.0
Smart custa 46000.0
Lamborghini custa 1000000.0
```

Observe que estamos trocando dois produtos de diferentes posições dentro do `while` (`analise` e `analise -1`):

```
while(analise > 0 && produtos[analise].getPreco() < produtos[analise-1].getPreco()) {

    Produto produtoAnalise = produtos[analise];
    Produto produtoAnaliseMenos1 = produtos[analise -1];

    produtos[analise] = produtoAnaliseMenos1;
    produtos[analise -1] = produtoAnalise;
}
```

A seguir, no `selectionSort`, também é feita a troca entre o `atual` por `menor`:

```
private static void selectionSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos -1; atual++)
    {

        int menor = buscaMenor(produtos, atual, quantidadeDeElementos -1);
        Produto produtoAtual = produtos[atual];
        Produto produtoMenor = produtos[menor];

        produtos[atual] = produtoMenor;
        produtos[menor] = produtoAtual;
    }
}
```

Trocar duas posições de um array é uma tarefa muito comum. Então, precisamos saber fazer isto de diversas maneiras.

Vamos extrair esse código que troca posições, para deixar meu algoritmo mais simples. Substituímos isto:

```
Produto produto Analise = produtos[analise];
Produto produtoAnaliseMenos1 = produtos[analise -1];
System.out.println("Estou trocando " + produtoAnalise.
getNome()
+ " com " + produtoAna
liseMenos1.getNome());
produtos[analise] = produtoAnaliseMenos1;
produtos[analise -1] = produtoAnalise;
```

Por isto:

```
troca(produtos, analise, analise -1);
```

Nosso laço ficará assim:

```
while(analise > 0 && produtos[analise].getPreco() < produtos[analise - 1].getPreco()) {
    System.out.println("Estou trocando " + analise + " com " + (analise -1));
    troca(produtos, analise, analise -1);
    analise--;
}
```

Usamos o comando `Ctrl + 1` com essa nova linha de código selecionada e vamos criar o método (`Create method 'troca(Produto[], int, int)'`):

```
private static void troca(Produto[] produtos, int analise, int i)
{}
```

O programa criou o método que recebe os produtos `analise` e `i`. Como os nomes não ficaram bons, vamos alterá-los para `primeiro` e `segundo`:

```
private static void troca(Produto[] produtos, int primeiro, int se
gundo) {
}
```

Vamos colar o código que extraímos anteriormente e incluir as variáveis `primeiro` e `segundo`:

```
private static void troca(Produto[] produtos, int primeiro, int segundo) {
    Produto primeiroProduto = produtos[primeiro];
    Produto segundoProduto = produtos[segundo];
    System.out.println("Estou trocando " + primeiroProduto.getNome()
()                                         + " com " + segundo
oProduto.getNome());
    produtos[primeiro] = segundoProduto;
    produtos[segundo] = primeiroProduto;
}
```

Então, estamos trocando o `primeiroProduto` com o `segundoProduto`. No nosso `novoSort`, estamos trocando o `analise` com `analise -1`, logo:

```
troca(produtos, analise, analise -1);
```

No caso do `selectionSort`, a troca é feita entre o `atual` e o `menor`, dentro do array de `produtos`.

```
private static void selectionSort(Produto[] produtos, int, quantidadeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos -1; atual++)
    {
        System.out.println("Estou na casinha " + atual);

        int menor = buscaMenor(produtos, atual, quantidadeDeElementos -1);
        System.out.println("Trocando " + atual + " com o " + atual);
    }

    troca(produtos, atual, menor);
}

}
```

Observe que na nossa função já vamos imprimir os nomes das variáveis que estamos trocando.

```
private static void troca(Produto[] produtos, int primeiro, int segundo) {
```

```

        Produto primeiroProduto = produtos[primeiro];
        Produto segundoProduto = produtos[segundo];
        System.out.println("Estou trocando " + primeiroProduto.getNome
())
                                         + " com " + segund
oProduto.getNome());
        produtos[primeiro] = produtoAnaliseMenos1;
        produtos[segundo] = produtoAnalise;
    }
}

```

Temos a seguinte linha imprimindo a posição no `while` :

```
System.out.println("Estou trocando " + analise + " com " + (analis
e -1));
```

Poderíamos incluí-la dentro do algoritmo, já com as variáveis `primeiro` e `segundo`.

```

private static void troca(Produto[] produtos, int primeiro, int se
gundo) {
    System.out.println("Estou trocando " + primeiro + " com " + se
gundo);
    Produto primeiroProduto = produtos[primeiro];
    Produto segundoProduto = produtos[segundo];
    System.out.println("Estou trocando " + primeiroProduto.getNome
()
                                         + " com " + segund
oProduto.getNome());
    produtos[primeiro] = produtoAnaliseMenos1;
    produtos[segundo] = produtoAnalise;
}
}

```

Para não deixarmos nenhuma repetição, excluiremos também o trecho do `selectionSort` :

```
System.out.println("Trocando " + atual + " com o " + menor);
```

O `selectionSort` ficará assim:

```

private static void selectionSort(Produto[] produtos, int, quantid
adeDeElementos) {
    for(int atual = 0; atual < quantidadeDeElementos -1; atual++)
    {
        System.out.println("Estou na casinha " + atual);

        int menor = buscaMenor(produtos, atual, quantidadeDeElemen
tos -1);

```

```
        troca(produtos, atual, menor);
    }

}
```

Então, vamos definir quais elementos serão trocados entre si, para depois serem reposicionados. O nosso código ficará assim:

```
private static void troca(Produto[] produtos, int primeiro, int se
gundo) {
    System.out.println("Estou trocando " + primeiro + " com " + se
gundo));
    Produto primeiroProduto = produtos[primeiro];
    Produto segundoProduto = produtos[segundo];
    System.out.println("Estou trocando " + primeiroProduto.getNome
())
                    + " com " + segundoProduto.get
Nome());
    produtos[primeiro] = segundoProduto;
    produtos[segundo] = primeiroProduto;
}
```

No algoritmo `novoSort`, que detecta a posição em que o elemento será inserido, fazíamos uma troca com o item anterior. Assim como no algoritmo `selectionSort`, que seleciona qual item merece estar em cada casinha, também fazíamos uma troca e reposicionávamos os elementos. Nós realizávamos trocas nos dois algoritmos. Logo, vamos extraí-los desta função.

Novamente, rodaremos o algoritmo e conferiremos se funcionou corretamente.

```
Brasilia custa 16000.0
Fusca custa 17000.0
Jipe custa 46000.0
Smart custa 46000.0
Lamborghini custa 1000000.0
```

Temos a ordenação. Agora vamos tirar o `novoSort` e colocar o `selectionSort`.

```
selectionSort(produtos, produtos.length);
//novoSort(produtos, produtos.length);
```

Rodaremos o algoritmo para verificar se também funciona corretamente. O resultado será:

```
Estou na casinha 2
Estou trocando 2 com 3
Estou trocando Lamborghini com Smart
Estou na casinha 3
Estou trocando 3 com 4
Estou trocando Lamborghini com Jipe
Brasilia custa 16000.0
Fusca custa 17000.0
Smart custa 46000.0
Jipe custa 46000.0
Lamborghini custa 1000000.0
```

A ordem está correta. Voltaremos para o algoritmo em que estamos trabalhando, o `selectionSort`:

```
//selectionSort(produtos, produtos.length);
novoSort(produtos, produtos.length);
```

Um detalhe foi melhorado no nosso algoritmo: nós passamos a utilizar o `atual` a partir da posição `1`. Porém, percebemos que são muito comuns **trocas** nos arrays, e que é interessante deixá-los organizadas em um único lugar. Extraímos tudo para uma função, que vai trocar a posição dos elementos dentro do array.

4.7 SIMULANDO NO PAPEL COM O NOSSO CÓDIGO

Nós já implementamos o algoritmo, agora vamos simulá-lo na memória.

```
for(int atual = 1; atual < quantidadeDeElementos; atual++ ) {
    int analise = atual;
    while(analise > 0 &&
        produtos[analise].getPreco() < produtos[analise-1].get
Preco())
        troca(produtos, analise, analise - 1);
        analise--;
    }
}
```

O que nós fizemos primeiro? Trabalhamos com o array e a *quantidade de elementos*.

Precisamos usar a variável `atual` e `analisar` na memória. Vamos começar com `atual` igual a 1. Por quê? Pois como só recebemos uma carta de baralho, não existe dúvida sobre a posição em que o elemento 0 deve ser inserido. Por isso, começamos com o segundo elemento, o **Jipe**.

```
int analise = atual;
```

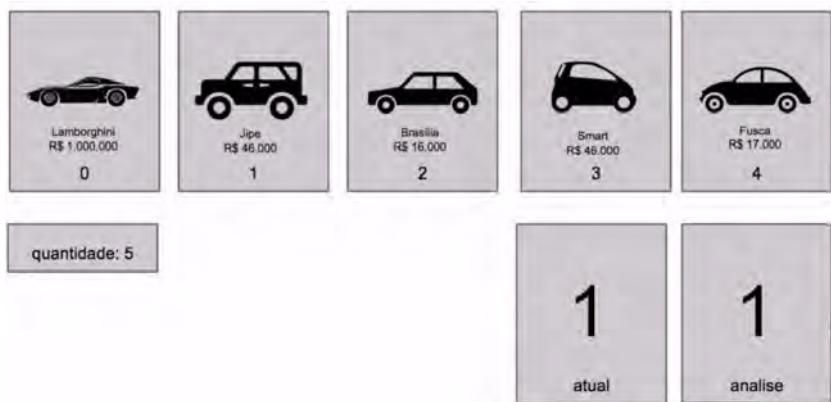


Figura 4.22: Simulação da análise — passo 1

O `atual` é igual a 1. `Analise` é igual a `atual` e também será igual a 1. Ela é maior do que 0? Sim. O preço do `produto[analise]` (o Jipe) é menor do que o `produtos[analise-1]` (a Lamborghini)? Sim, o Jipe é mais barato. Então, trocaremos os dois produtos de posição.

Continuaremos com `analise--`. Diminuiremos 1 da `analise`, que será igual a 0 e não atende a primeira condição (`while(analise > 0)`). Não faremos novas alterações.

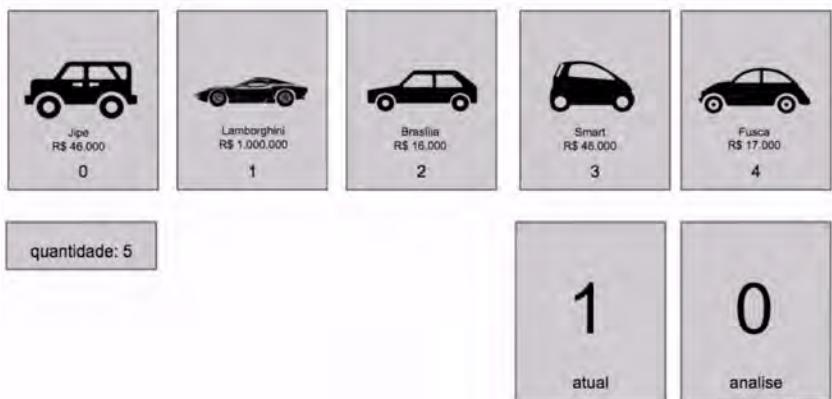


Figura 4.23: Simulação da análise — passo 2

Passamos para o próximo elemento na posição 2, a **Brasília**.

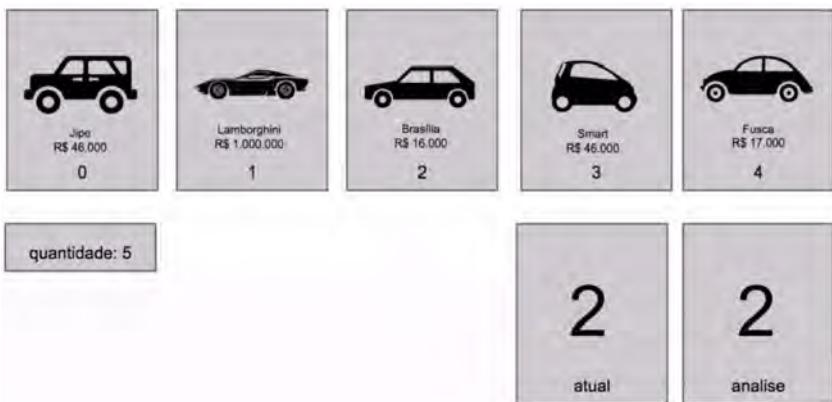


Figura 4.24: Simulação da análise — passo 3

A variável `atual` será igual a 2. Ela é menor do que a `quantidadeDeElementos`? Sim. Como `análise` é igual a `atual`, será maior que 0. Vamos verificar se atende à segunda condição: o preço da Brasília é menor do que a Lamborghini? Sim. Trocaremos os elementos de lugar. Depois, vamos diminuir 1 da `análise`.

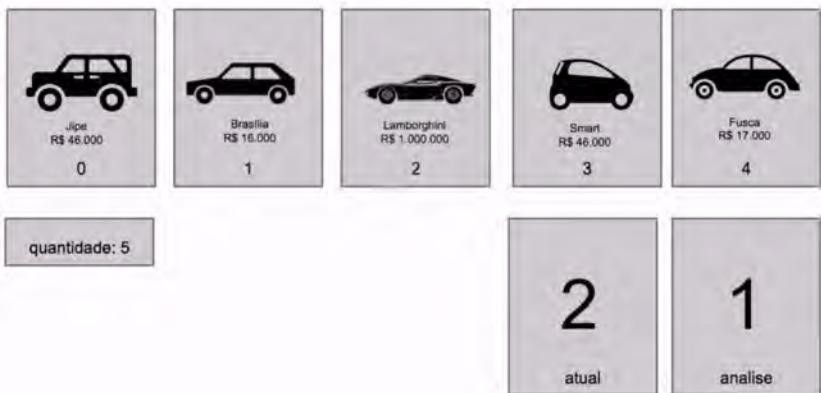


Figura 4.25: Simulação da análise — passo 4

`análise` é igual 1. Verificaremos se a **Brasília** está posicionada adequadamente. O preço do produto (que custa R\$ 16.000) é menor do que o `Jipe` (que custa R\$ 46.000)? Sim. Trocaremos os elementos de lugar e a `Brasília` estará posicionada na **primeira casa**.

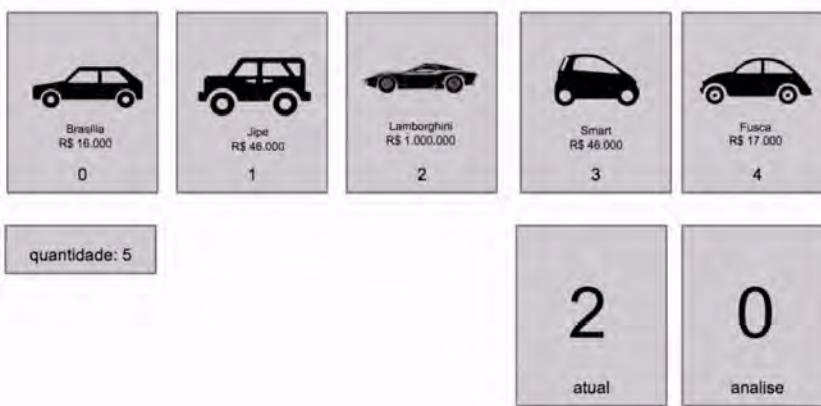


Figura 4.26: Simulação da análise — passo 5

Ela já ocupa a posição correta? Se avançarmos para o elemento anterior, `análise` será igual a 0. Como não atende à primeira condição, a ordem ficará como está.

Seguimos para o próximo produto, o **Smart**. O **atual** será igual a 3.

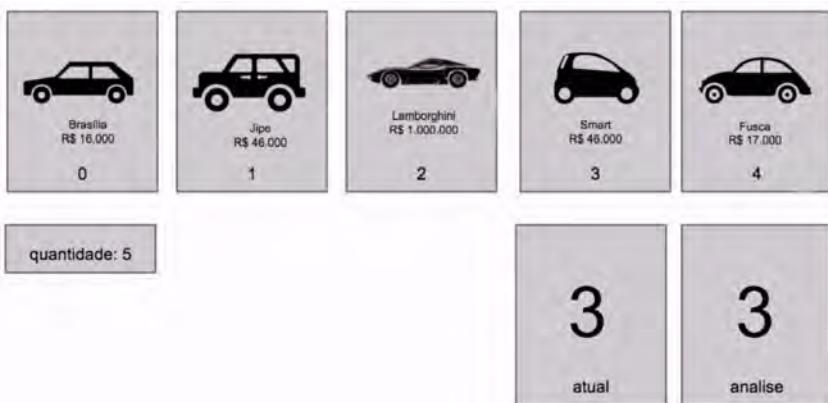


Figura 4.27: Simulação da análise — passo 6

Aonde o produto merece ser inserido? Até o momento, só podemos compará-lo com os quatro itens recebidos anteriormente. Começamos com **análise** igual a 3 e a variável é maior do que 0.

`produtos[análise].getPreco()` é menor do que `produtos[análise-1].getPreco()`. Como atende as duas condições do `while`, vamos trocá-los de lugar.

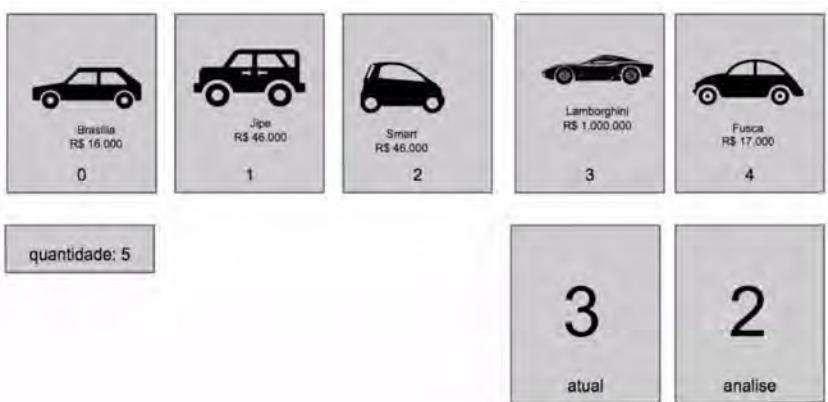


Figura 4.28: Simulação da análise — passo 7

Continuaremos com a análise do **Smart**. Agora vamos compará-lo com o **Jipe**. analise será igual a 2 e maior que 0. No entanto, o preço do produto não é menor do que o preço do anterior. Os produtos vão permanecer na mesma posição.

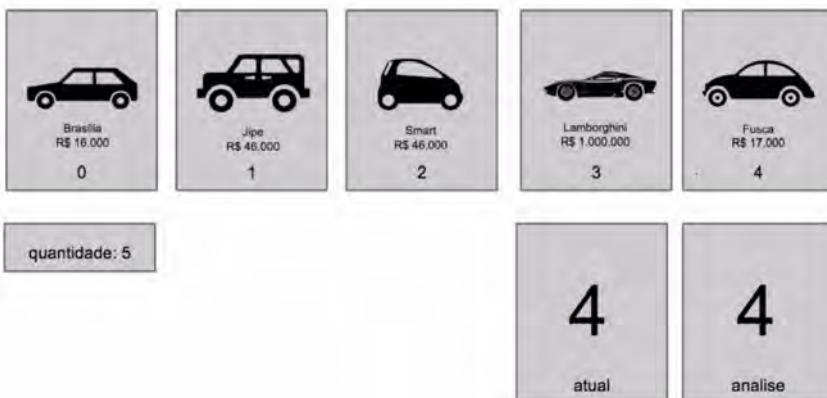


Figura 4.29: Simulação da análise — passo 8

Vamos para o próximo elemento, atual será igual a 4, que é igual a 5 (e menor que a quantidadeDeElementos). análise será igual a 4 e o produto analisado será o **Fusca**.

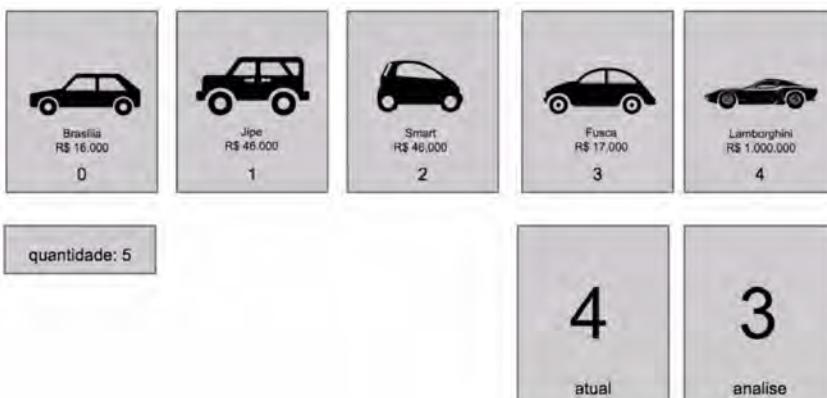


Figura 4.30: Simulação da análise — passo 9

`analise` é maior que 0? Sim. Seguimos para a próxima condição: `produtos[analise].getPreco()` é menor do que `produtos[analise-1].getPreco()`? Sim, porque o preço do Fusca é menor do que da Lamborghini. Vamos reposicioná-los na lista.

Diminuiremos 1 da `analise`, que será igual a 3. A variável é maior que 0 e o preço do Fusca é menor do que o Smart. Vamos modificar a ordem.

Seguimos para `analise` é igual a 2.

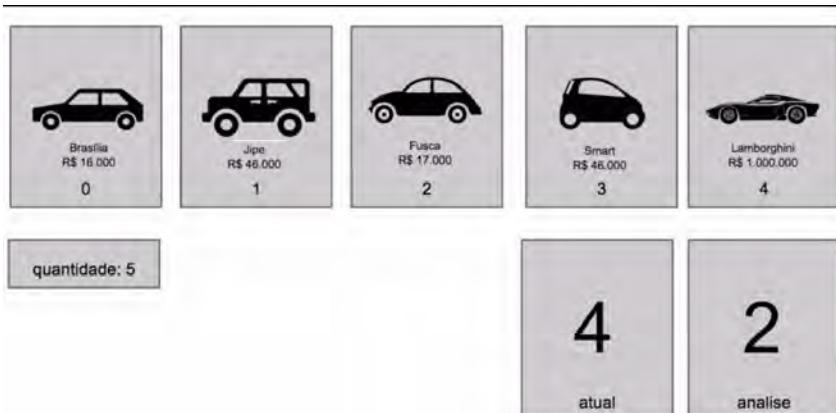


Figura 4.31: Simulação da análise — passo 11

O produto atende as duas condições. Como o preço do Fusca é menor do que Jipe, vamos trocá-los de lugar. Agora `analise` é igual a 1.

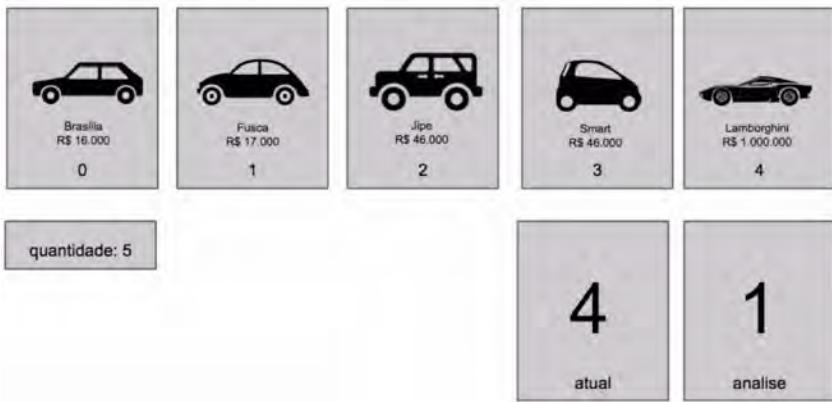


Figura 4.32: Simulação da análise — passo 12

Também atendeu a primeira condição, porém o preço do Fusca é maior do que a **Brasília** e, por isso, não atende à segunda condição. Isso significa que o Fusca merece ficar na posição 1.

Seguimos para `atual` igual a 5.

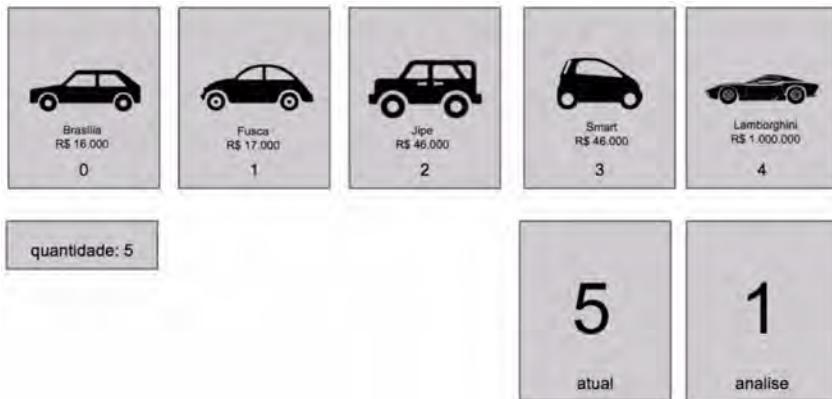


Figura 4.33: Simulação da análise — passo 13

Ele respeita o `for` que especifica que `atual` precisa ser menor que `quantidadeDeElementos`? Não. Então, paramos por aqui. Nossa algoritmo já está ordenado.

Agora que nosso array está ordenado, podemos obter várias informações sobre os elementos, além do benefício de tê-lo organizado. A ordenação nos possibilita resolver diversos problemas do mundo real:

- Encontrar quais são os maiores e os menores;
- Os melhores e os piores.

Com uma ordem (uma classificação), conseguimos dizer qual elemento é o primeiro, o segundo e o terceiro. Conseguimos identificar qual é o 10º ou o 15º maior item, devido à posição dos elementos que estão ordenados no array. Todos esses sistemas podem ser implementados.

Conseguimos isto, porque nós introduzimos um segundo algoritmo que passa por cada elemento. E a partir da posição dos itens analisados, observamos os anteriores e identificamos onde ele merece ser inserido. Assim, posicionamos os elementos de acordo com a ordem.

4.8 INSERTION SORT

Agora que já temos o nosso algoritmo, vamos ver o que acontece quando simulamos na nossa memória (nós também armazenamos variáveis na mente) ou no computador? Usaremos no exemplo as variáveis `atual` e `análise` e cartas de baralho.

Assim que recebo a primeira carta, ainda não posso fazer nada com ela. A minha análise começa apenas quando recebo a segunda. Com os dois elementos na minha mão, comparo a minha carta analisada com a anterior. Percebo que ela é maior e, por isso, continuará na mesma posição.

Recebo uma nova carta, que será a próxima a ser analisada (minha variável `análise`) e será a terceira na fileira. Chega mais

uma, já tenho quatro cartas na minha mão. Comparo-a com o elemento anterior (minha analise anterior) e decido movê-la para a posição 3. Sigo comparando-a com as cartas anteriores, e então identifico que ela é menor do que as outras. Ela passa a ocupar a segunda posição na minha fileira.

Recebo mais uma carta e a analiso, sempre em comparação com as anteriores. Decido que a minha variável vai ocupar a posição 2. Agora tenho as cinco cartas ordenadas, todas do naipe de Copa: 3, 5, 6, 9 e 10.

Observe que, quando recebemos cartas de baralho, vamos aos poucos encontrando a posição adequada para cada uma. O que fazemos durante o processo é inserir os itens na posição correta. Este algoritmo receberá o nome de **Insertion Sort**. É ele que nos faz inserir os elementos na posição adequada.

4.9 O PRÓXIMO DESAFIO: ASSENTANDO O CONHECIMENTO

Agora que já vimos diversos trechos de algoritmos que compõem dois algoritmos de ordenação, chegou a hora de analisarmos um pouco mais de tudo o que vimos. Como nos comunicamos com um algoritmo? Como podemos usar algoritmos para solucionar problemas em que não havíamos pensado anteriormente?

ALGUNS COMPONENTES DE UM ALGORITMO

5.1 ALGORITMOS: ENTRADA E SAÍDA

Vamos verificar de novo os três problemas que tentamos resolver. Resolvemos o problema de um array de vários produtos, com início e fim, e que começa do 0 e vai até o 4. Qual elemento é o menor deles? É o que está na casinha 0, 1, 2, 3 ou 4?

Essa é uma das diversas perguntas que podemos fazer para um conjunto de elementos. E já sabemos respondê-la: basta entregar os elementos para o programa e pedir para que ele indique qual é o menor.

Analisamos também outro tipo de pergunta. Dado um array com diversos elementos (alunos, políticos, times de futebol etc.), pedimos para que o programa coloque-os em ordem. Queremos que ele devolva os elementos de uma forma ordenada, do menor para o maior.

Então, entramos com um array de vários produtos (elementos) para o nosso programa (**a entrada**). Ele nos retorna uma lista (**a saída**), com os itens na ordem correta. No nosso caso, resolvemos o problema de ordenação primeiro pelo *Selection Sort* para um conjunto de carros.

Por fim, fizemos a mesma pergunta agora para as cartas de

baralho: "se dermos esses elementos, por exemplo, várias cartas de baralho, você consegue devolver todas elas (ou todos esses elementos) de uma maneira ordenada? Porém, criando a ordem de uma maneira diferente desta vez". Na vida real, se pensarmos, cada pessoa ordena cartas de forma diferente: alguns usam *Selection Sort*, outros usam *Insertion Sort*. Depende de cada um de nós escolher qual processo lhe parece mais natural.

Observe que todos os problemas resolvidos até agora tinham uma entrada: um conjunto de dados, com várias informações. Além disso, o programa tinha de nos dar uma saída, seu resultado.

Todo algoritmo tem uma entrada e uma saída. O importante é que ele resolva a questão, transformando os dados de entrada em um resultado de saída. O algoritmo nada mais é do que uma maneira de se transformar dados de entrada na saída correta.

Se o problema é de encontrar o menor, "como o algoritmo resolve e acha o menor preço" não parece relevante à primeira vista. Primeiro, é importante ele estar correto, devolver o valor certo.

No problema de ordenação, "como ele vai ordenar o meu array?" não parece ser relevante a primeira vista. Mas é importante estar correto, devolver a ordem correta. Quando questionamos então se um algoritmo está correto, estamos analisando sua *corretude*.

Já as duas perguntas levantadas ("como o algoritmo faz isso") trazem a tona questões de diferenças em desempenho e em consumo de memória. Questões que temos de analisar no próximo capítulo.

5.2 REDUZINDO UM PROBLEMA A OUTRO

Nós resolvemos o problema da ordenação com duas soluções diferentes:

- Selecionando quem deveria ficar em cada casinha;
- Encontrando a casinha onde deveríamos inserir um elemento.

As duas formas são utilizadas naturalmente pelas pessoas, sem que elas percebam que fizeram a escolha entre uma maneira e a outra, em situações diferentes. Porém, como são inventados algoritmos como estes? Será que existem apenas os dois já conhecidos e ninguém mais tenha inventado outra maneira?

A verdade é que existem diversos outros. E o processo de inventar um algoritmo é um processo difícil. Além de criá-lo, precisamos provar que ele funciona e que ele é rápido. É uma tarefa trabalhosa.

Por isso, ao ser dado um problema, optamos muitas vezes por analisá-lo e tentamos identificar uma solução conhecida: "esse é um problema de ordenação, então vou usar aquele algoritmo que é muito bom para ordenar!".

Ou seja, olhar com outros olhos o problema. Por exemplo, se quero descobrir quais foram os políticos mais votados em uma eleição, posso verificar qual é o algoritmo que vai identificar quais receberam um maior número de votos. Para isto, basta olhar para os elementos e ver a necessidade de uma ordenação.

Nesse mesmo exemplo, se quero saber quais foram os quatro políticos com maior votação em uma eleição, basta ordená-los e, em seguida, selecionar os quatro primeiros elementos. Analisei a questão com outros olhos em vez de me limitar ao problema aparente "tenho de encontrar os quatro maiores".

Pensamos de outra maneira: ao ordenar todos os candidatos, foi possível identificar os quatro com maior número de votos. Além de observar por outros ângulos, nós reduzimos o problema a outro,

cuja solução já era conhecida. Sabíamos como solucionar uma ordenação, então apenas foi preciso selecionar os quatro maiores. Podemos também definir outros critérios: os cinco piores ou os três maiores, e simplifico-os em uma única ordenação.

Sempre que alguém nos pedir os 15 candidatos mais votados ou os três cursos com maior nota, ordenaremos todos os cursos e pronto, estará acabado. Podemos identificar este tipo de informação. Sem grandes dificuldades, implementamos a ordenação, depois a executamos e informamos rapidamente quais são os três primeiros elementos.

Fica ainda mais fácil se a quantidade de itens for pequena. A grande sacada está em reduzir o problema.

Por exemplo, se simplificarmos a questão em uma ordenação, já saberemos que é possível resolvê-la com o *Selection Sort* — que foi reduzido a uma busca do menor.

```
for(int atual = 0; atual < quantidadeDeElementos -1; atual++) {  
  
    int menor = buscaMenor(produtos, atual, quantidadeDeElementos - 1);  
    Produto produtoAtual = produtos[atual];  
    Produtos produtoMenor = produtos[menor];  
  
    produtos[atual] = produtoMenor;  
    produtos[menor] = produtoAtual;  
}
```

Nós buscamos o menor dentro do algoritmo. Nós simplificamos a implementação do *Selection Sort* usando um algoritmo já conhecido. Dentro do *Selection Sort*, usamos o algoritmo de encontrar o menor.

Então, quando reduzimos um problema para a utilização de um algoritmo que já conhecemos, nós simplificamos a maneira de resolver o nosso empecilho. Pode ser que ela seja muito performática, com a melhor memória, ou pode ser que não. Isto vai

depender da análise do nosso algoritmo, que faremos já no próximo capítulo.

Agora estamos interessados em identificar a melhor maneira de simplificar um problema, após observá-lo de diversos ângulos e perceber a melhor maneira para resolvê-lo. A estratégia é observar um problema por diferentes ângulos e encontrar um aspecto em que a solução já é conhecida, descobrir um algoritmo que seja conhecido e que é capaz de resolvê-lo.

Ou talvez, encontrar mais de um algoritmo, como foi o caso do *selection sort*: nós criamos um laço, usamos o `buscaMenor` diversas vezes e resolvemos o problema da ordenação. A sacada para solucionar os problemas do cotidiano é: observar um problema, encontrar uma nova forma de atacá-lo e, depois, reduzi-lo a algoritmos já conhecidos.

Analiso um problema e penso: "para solucionar isso aqui, basta fazer um `for` e buscar o menor" ou "basta ordenar e encontrar os três maiores". A estratégia é reduzir o problema a coisas que já sabemos fazer.

5.3 O PRÓXIMO DESAFIO: COMPARAR NOSSOS ALGORITMOS

Agora que já temos duas soluções distintas para um mesmo problema, precisamos ser capazes de dizer como cada um deles se comporta e qual quero escolher em uma situação ou outra. Preciso ser capaz de compará-los. Para isso, existe toda uma área de pesquisa da ciência da computação. Nossa próximo passo é entrarmos no mundo da análise de algoritmos.

CAPÍTULO 6

COMPARANDO ALGORITMOS E A ANÁLISE ASSINTÓTICA

Considerando que analisamos diversos algoritmos e já sabemos implementá-los, o que é preciso para compará-los? Como podemos dizer qual está bom e qual não está? Se alguém desejar seguir a carreira de Ciências da Computação e quiser criar novos algoritmos, como poderá afirmar que o seu algoritmo de ordenação é melhor do que outro já existente?

6.1 COMO ANALISAR O DESEMPENHO DE ALGORITMOS?

Um possível critério seria analisar quanto cada um deles utiliza de memória. Um algoritmo seria melhor do que o outro segundo esse critério se ele usasse menos memória.

No nosso caso, usaremos um outro critério, o mais utilizado de todos: quanto tempo é necessário para esse algoritmo resolver nosso problema?

Essa é a comparação de **desempenho de velocidade**, que nos informa o quanto rápido pode ser um algoritmo. Porém, como vamos medir a velocidade? Em segundos, em minutos ou em horas? Qual unidade usaremos?

O computador é capaz de fazer contas, por isso vamos medi-la a partir da quantidade de operações que o computador precisa fazer para encontrar a resolução de um algoritmo. Se precisarmos fazer 10 ou 9 operações, não faz muita diferença na velocidade. No entanto, se a variação for entre 10 e 1 bilhão de operações, a diferença será enorme!

É desta forma que vamos avaliar o desempenho dos dois algoritmos, comparando a ordem de grandeza do número de operações que devem ser feitas. Se em um algoritmo precisamos fazer "uma dezena" de operações e no outro "centenas de milhares", diremos que o segundo é pior que o primeiro.

6.2 OS ALGORITMOS

Temos três algoritmos que queremos analisar, e vamos atacá-los um a um.

1. `buscaMenor` busca o menor elemento:

```
private static int buscaMenor(Produto[] produtos, int inicio, int termino) {
    int maisBarato = inicio;
    for(int atual = inicio; atual <= termino; atual++) {
        if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()) {
            maisBarato = atual;
        }
    }
    return maisBarato;
}
```

2. `selectionSort` ordena um array:

```
private static void selectionSort(Produto[] produtos, int quantidadeDeElementos)
{
    for(int atual = 0; atual < quantidadeDeElementos - 1; atual++) {
        int menor = buscaMenor(produtos, atual, quantidadeDeElementos - 1);
        troca(produtos, atual, menor);
    }
}
```

```
    }
}
```

3. `insertionSort` ordena um array:

```
private static void insertionSort(Produto[] produtos, int q
uantidadeDeElementos) {
    for (int atual = 1; atual < quantidadeDeElementos; atual
++) {
        System.out.println("Estou na casinha " + atual);

        int analise = atual;
        while(analise > ) && produtos[analise].getPreco() < p
rodutos[analise -1]) {
            troca(produtos, analise, analise -1);
            analise--;
        }
    }
}
```

Claro que cada um pode ser utilizado em diferentes situações. O `buscaMenor` serve para encontrar o menor (ou o maior) de todos. Já o `insertionSort` e o `selectionSort` são algoritmos que ordenam o nosso array.

Qual deles parece ser mais rápido e qual parece ser o mais devagar? Qual é a sua opinião?

6.3 ANALISANDO O BUSCAMENOR

Vamos começar analisando o algoritmo de `buscaMenor`. Ele recebeu um array com cinco elementos e fizemos a busca do **0 até o 4 inclusive**.

O nosso algoritmo passará por cada um dos elementos. Trata-se de um `for` que passa por todos os elementos, desde o primeiro até o quinto. Nós faremos as operações do algoritmo para cada um dos elementos, ou seja, se temos cinco elementos, cinco operações. Se temos 100 elementos, por exemplo, teremos 100 operações. Será o mesmo se tenho 1.000.000 ou 5.000.000 de elementos. Basicamente,

disto se trata o nosso algoritmo.

Dentro do nosso `for`, faremos quantas operações? Apenas uma? Se fizermos uma operação, serão as cinco operações de fora para cada uma das de dentro, cinco vezes um, o que resultará em cinco. Porém, se fizermos duas ações dentro do `for`, devido à quantidade de elementos, teremos 5 operações de fora vezes 2 operações de dentro, assim 5 vezes 2 são 10 operações. Será o dobro.

Isto é, se tenho um laço, multiplico o número de operações internas pelo número de vezes que executarei este laço. Vamos verificar passo a passo:

```
for(int atual = inicio; atual<= termino; atual++){}
```

O `for` passa pelo array inteiro. Se passar de 0 até 4, ele vai executar cinco vezes o código dentro do `for`, que é esse:

```
if(produtos[atual].getPreco() < produtos[maisBarato].getPreco()){  
    maisBarato = atual  
}
```

Para um array ordenado em ordem crescente:

10 26 29 36 76

Nosso algoritmo passa pelo `if` 5 vezes, mas não entra nele em nenhuma delas. Mas o que aconteceria em um array que já esteja ordenado em ordem decrescente?

76 36 29 26 10

Nesse caso, não só validaríamos a condição do `if` 5 vezes, mas entrariíamos nele todas as vezes, executando a operação `maisBarato = atual` 5 vezes. E em um array qualquer como no caso a seguir?

76 29 26 36 10

Passaríamos pelo `if` as mesmas 5 vezes, mas entramos na operação interna somente quando encontramos um novo número menor. Isto é, sempre que um elemento for menor que todos os elementos da sua esquerda. No caso anterior, os números 29, 26 e 10 possuem essa condição, totalizando 3 execuções de `maisBarato = atual`.

Isto significa que, dependendo da "sorte" que temos, um array chegará para nós de forma já ordenada crescente. Para cada um dos elementos, faremos apenas um `if`, e não entraremos no laço, o que resulta em cinco operações.

Porém, se dermos "azar", além do `if`, teremos de aplicar o `maisBarato = atual` para cada um dos cinco itens. Serão 10 operações.

O número total ficará entre 5 (melhor caso) e 10 (melhor caso). Isto significa que, se tivermos 100 elementos, teremos entre 100 e 200 operações. Se tivermos 500 elementos, ficaremos entre 500 e 1000 operações. O mesmo ocorrerá se forem 1.000.000 de elementos. Portanto, se tivermos n elementos, a quantidade de operações estará entre n (melhor caso) e $2n$ (pior caso).

Falta então entender qual a diferença entre n e $2n$ operações. Isso é algo gigante? Pequeno? Qual o impacto disso no nosso algoritmo?

Primeiro vamos desenhar um gráfico que mostre a quantidade de operações que serão feitas à medida que o número de elementos aumente. Por exemplo, se temos 100 elementos, quantas operações teremos? Quantas operações teremos se tivermos 1000 ou 100.000 elementos?

6.4 CRIANDO UMA TABELA DE OPERAÇÕES

POR DESEMPENHO DE ALGORITMO

Observamos que nosso algoritmo rodará cerca de n operações, ou até $2n$ operações por elemento ($2n$), para buscar o menor elemento. Vamos usar alguns números para verificar como isto ficaria?

Para o primeiro caso, se temos 1 elemento, qual será o valor de n ? Exatamente 1.

	A	B	C
1	Elementos	n	
2		1	1
3			
4			

Figura 6.1: Operações por tipo de algoritmo — passo 1

E se fosse $2n$ (duas operações por elemento)? O resultado seria 2 operações.

	A	B	C	D
1	Elementos	n	$2n$	
2		1	1	2
3				
4				

Figura 6.2: Operações por tipo de algoritmo — passo 2

E se tivéssemos 2 elementos, o dobro da quantidade anterior, como preencheríamos as outras colunas? Os outros resultados também seriam o dobro.

	A	B	C	D	E
1	Elementos	n	2n		
2		1	1	2	
3		2	2	4	
4					
5					

Figura 6.3: Operações por tipo de algoritmo — passo 3

Isto significa que, à medida que dobrarmos o número de elementos, o número de operações também vai dobrar:

	A	B	C	
1	Elementos	n	2n	
2		1	1	2
3		2	2	4
4		4	4	8
5		8	8	16
6		16	16	32
7		32	32	64
8				

Figura 6.4: Operações por tipo de algoritmo — passo 4

Quando cresce o número de elementos, o número de operações crescerá proporcionalmente. Se dobrarmos o número de elementos, o número de operações também dobrará.

	A	B	C	D
1	Elementos	n	2n	
2		1	1	2
3		2	2	4
4		4	4	8
5		8	8	16
6		16	16	32
7		32	32	64
8		64	64	128
9		128	128	256
10		256	256	512
11		512	512	1024
12		1024	1024	2048
13		2048	2048	4096
14		4096	4096	8192
15		8192	8192	16384

Figura 6.5: Operações por tipo de algoritmo — passo 5

6.5 GRÁFICO DE UM ALGORITMO LINEAR

Vamos inserir os dados da tabela em um gráfico para vermos como será aproximadamente o nosso número de operações? À medida que o nosso array for crescendo e quisermos, por exemplo, encontrar o menor preço de 8.192 produtos, conseguiremos ver no gráfico quantas operações o algoritmo terá de fazer. Vamos visualizar no gráfico o crescimento do algoritmo corretamente?

Primeiro, temos a nossa última figura anterior, *Operações por tipo de algoritmo — passo 5*. Então, selecionaremos todos os elementos e clicaremos em *Insert* e *Chart*. O programa abrirá o *Chart Editor*. Vou definir qual será a coluna e a linha que serão o cabeçalho e o rodapé.

Chart Editor



Figura 6.6: Char Editor

Nosso gráfico terá linhas simples.

Chart Editor

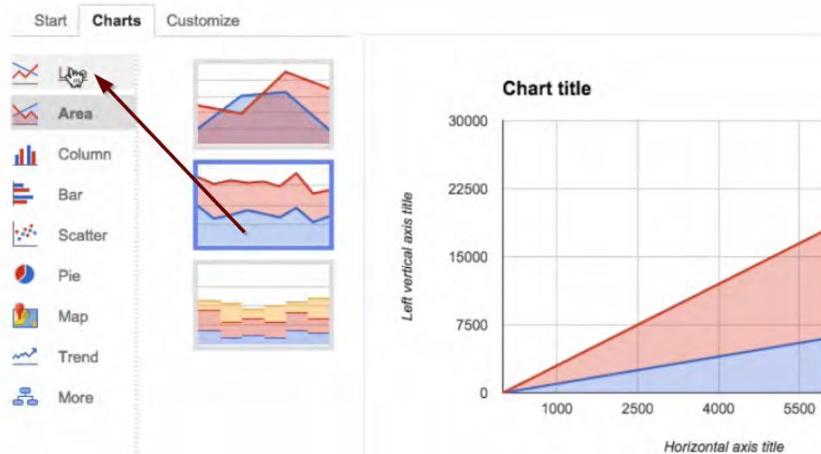


Figura 6.7: Chart Editor — passo 2

Em seguida, pedimos para inserir o gráfico:

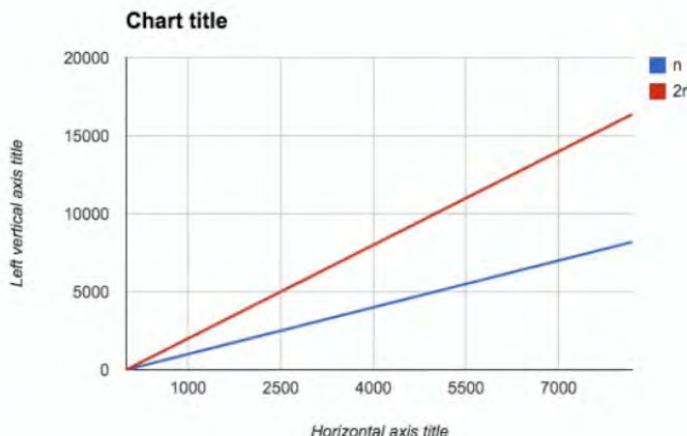


Figura 6.8: Gráfico do algoritmo

Vamos incluir novos títulos. Embaixo, alteraremos o título do eixo horizontal (*Horizontal axis title*) por **Elementos** (referente ao número de elementos que cresce aos poucos). Na lateral, vamos alterar o eixo vertical (*Left vertical axis title*) por **Operações** (referente ao número de operações). Isto significa que, se o número de elementos cresce, o número de operações também vai crescer.

Este é o algoritmo. E como ele cresce? Vamos analisar o n , a linha debaixo do gráfico.

À medida que o número de elementos cresce, o número de operações também aumentará. O n é linear, por isso o número de operações seguirá uma linha. O mesmo acontecerá com $2n$, que terá o dobro do número de operações, mas seguirá uma linha.

Naturalmente, $2n$ será mais lento e terá uma demora duas vezes maior do que n , que é mais rápido. No entanto, ambos crescem como uma linha. Isto significa que, à medida que dobrarmos o número de elementos, dobraremos também o número de operações. Se triplicarmos o número de elementos, triplicaremos também o número de operações. O número de operações sempre terá um

aumento proporcional e linear.

Este é o nosso algoritmo que buscar o menor. Por isso, o nosso gráfico receberá o título de **Busca menor**. Ele buscará o menor elemento, que está situado entre as linhas n e $2n$ do gráfico.

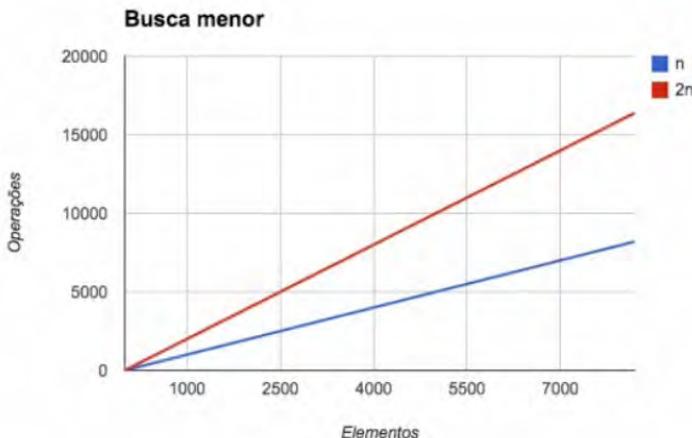


Figura 6.9: Gráfico do buscaMenor

Sabemos que é um algoritmo válido. Agora analisaremos outros algoritmos para poder compará-los entre si.

6.6 ANALISANDO O SELECTION SORT

Nós vimos o algoritmo de menor elemento e, para executá-lo, percebemos que tivemos de passar por todos os elementos. Para cada um deles (para cada um dos n elementos), executávamos um `if` e um código que estava dentro dele.

Nós executávamos entre 1 e 2 operações para cada um dos n elementos. Isto é, nós executávamos n ou $2n$ operações. Podemos ter uma ideia de quantas operações este algoritmo precisa. Vamos analisar outro algoritmo: o `selectionSort`.

Nós já havíamos analisado ele por cima, afinal, ele era baseado no `buscaMenor`. Ele passava por cada uma das casinhas e buscava o menor a partir dali. Vamos analisar uma simulação dele?



Figura 6.10: Analisando o Selection Sort

Na hora de executarmos o nosso algoritmo, o que nós fazíamos?

Executávamos o `atual` indo de **0 até 4** — um `for` para cada elemento. Se tínhamos cinco elementos, tínhamos um `for` para cinco. Se tínhamos 100 elementos, tínhamos um `for` de 100. Se tínhamos 1.000.000 de elementos, tínhamos um `for` de 1.000.000. Se tínhamos n elementos, o nosso `for` precisava passar por cada um dos n elementos. Sabemos que o nosso algoritmo é um `for` que vai do *0 até 4*.

Porém, a partir da **posição 0**, mandávamos buscar o menor desde a **posição 1**. O mesmo acontecia quando buscávamos a partir da **posição 2**, e a busca pelo menor iniciava da **posição 3**. O processo era feito até chegarmos no quarto elemento.

Nós já conhecemos o `buscaMenor` e sabemos o quanto lento ou rápido ele é. O `buscaMenor` está entre n e $2n$ operações, para cada um dos elementos. Porém, se o que estamos fazendo é executar um

`for` que passe por cada um dos n elementos, o que faremos é executar um outro algoritmo que realizará n ou $2n$ operações.

Quanto é o total? O resultado será aproximadamente n^2 ou $2n^2$.

Vamos analisar o código? Veremos o nosso `for` de n elementos:

```
for(int atual = 0; atual < quantidadeDeElementos - 1; atual++)
```

E dentro dele, executaremos o algoritmo:

```
int menor = buscaMenor(produtos, atual, quantidadeDeElementos - 1);
```

Isso significa que nós executamos n vezes o algoritmo que demora n ou $2n$ operações. Isto significa que dá n^2 ou $2n^2$ operações.

Temos de executar também uma troca para cada um deles, que igualmente demorará algumas operações. Porém, vamos descartá-la para ver como ficam estas operações que já estamos trabalhando.

Vamos analisar apenas como será com n^2 e $2n^2$, que é o tempo de execução do algoritmo de ordenação, e compará-lo com o `TestaOrdenacao` que demora n ou $2n$.

Vamos analisar o gráfico destes dois algoritmos? Primeiro, gostaria que você tentasse rascunhar o nosso algoritmo que demora n , e depois o que vai demorar n^2 ou $2n^2$. Tente rascunhá-lo e em seguida veremos como ficará.

6.7 CRIANDO A TABELA DE OPERAÇÕES DE UM ALGORITMO QUADRÁTICO

Chegou a hora de desenharmos um gráfico para o número de operações que teremos, de acordo com o algoritmo `selectionSort`. Vamos analisá-lo?

Criaremos uma nova planilha e chamaremos a primeira coluna de **Elementos**. Já vimos que, no `selectionSort`, o número de operações que teremos será n^2 . Vamos preencher a segunda coluna da tabela com n^2 .

Se tivermos um elemento, n^2 será igual a 1. Sabemos que o nosso número de operações seria entre n^2 e $2n^2$, então vamos preencher a terceira coluna com $2n^2$. Significa ele poderá ter 2 vezes o número de elementos.

	A	B	C	D
1	Elementos	n^2	$2 \cdot n^2$	
2		1	1	2
3				
4				
5				

Figura 6.11: Operações de algoritmo quadrático — passo 1

O que acontecerá se tivermos 2 elementos? Neste caso, vamos executar entre 4 e 8 operações.

	A	B	C	D
1	Elementos	n^2	$2 \cdot n^2$	
2		1	1	2
3		2	4	8
4				
5				

Figura 6.12: Operações de algoritmo quadrático — passo 2

E se tivermos o dobro do número de elementos? Se tivermos 4 elementos, teremos mais do que o dobro de operações.

Se tivermos 8 elementos, também teremos mais do que o dobro de operações. Observe que, à medida que dobrarmos o número de elementos, o número de operações cresce, explode no gráfico, afinal

ele será elevado ao quadrado.

	A	B	C	D
1	Elementos	n^2	$2 \cdot n^2$	
2		1	1	2
3		2	4	8
4		4	16	32
5		8	64	128
6		16	256	512
7				

Figura 6.13: Operações de algoritmo quadrático — passo 3

Seguimos dobrando o número de elementos e observando o que acontece com as operações.

	A	B	C	D
1	Elementos	n^2	$2 \cdot n^2$	
2		1	1	2
3		2	4	8
4		4	16	32
5		8	64	128
6		16	256	512
7		32	1024	2048
8		64	4096	8192
9				

Figura 6.14: Operações de algoritmo quadrático — passo 4

O que percebemos é o crescimento gigantesco dos resultados.

6.8 O GRÁFICO DE UM ALGORITMO QUADRÁTICO

Chegou o momento de criarmos o gráfico do algoritmo. Clicamos em **Insert**, e depois **Chart**. Em seguida, selecionaremos o rodapé e o cabeçalho. Escolhemos o gráfico de linha e depois o inserimos.

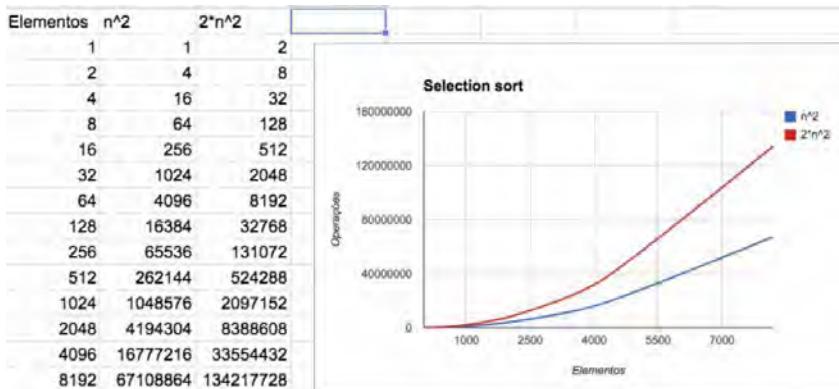


Figura 6.15: Gráfico de um algoritmo quadrático

Vamos alterar o título do gráfico para **Selection Sort**. A seguir, denominaremos o eixo horizontal como **Elementos** e o vertical como **Operações**. À medida que o número de elementos cresce, o número de operações também aumentará. Porém, observe o tanto que o último cresce!

Quando tivermos 4.000 elementos, teremos cerca de 20.000.000 operações sendo executadas. Quando o número de elementos alcançar 8.192, teremos 134.217.728 operações. Observe que, se tivermos, por exemplo, que colocar 8192 políticos em ordem, o computador terá de fazer 134.217.728 operações matemáticas.

Parece um número muito elevado! Será que de fato são necessárias tantas operações? Para responder tal pergunta, precisaremos comparar o algoritmo quadrático com o linear, e assim concluir como funciona o crescimento de cada um.

6.9 COMPARANDO O DESEMPENHO DOS ALGORITMOS

Vamos comparar os algoritmos que nós analisamos. Vamos criar uma nova aba e inserir os elementos do nosso buscaMenor , incluindo n e $2n$, n^2 (n^2) e $2*n^2$ ($2n^2$).

A	B	C	D	E
Elementos	n	$2n$	n^2	$2*n^2$
1	1	2	1	2
2	2	4	4	8
4	4	8	16	32
8	8	16	64	128
16	16	32	256	512
32	32	64	1024	2048
64	64	128	4096	8192
128	128	256	16384	32768
256	256	512	65536	131072
512	512	1024	262144	524288
1024	1024	2048	1048576	2097152
2048	2048	4096	4194304	8388608
4096	4096	8192	16777216	33554432
8192	8192	16384	67108864	134217728

Figura 6.16: Comparação de desempenho de algoritmos em tabela

Após preenchermos todas as células da tabela, teremos o número de operações e já poderemos gerar o gráfico. Selecionaremos todos os dados e, em seguida, vamos clicar em **Insert** e **Chart**. Vou selecionar a coluna e a linha que serão o cabeçalho e o rodapé do nosso gráfico, que será inserido com linhas.

Agora conseguiremos visualizar graficamente o número de operações que estão acontecendo. Faremos alterações nos títulos da

imagem, identificando o eixo inferior com o título **Elementos** e eixo vertical com **Operações**. Nossa gráfica receberá o nome **Algoritmos**.

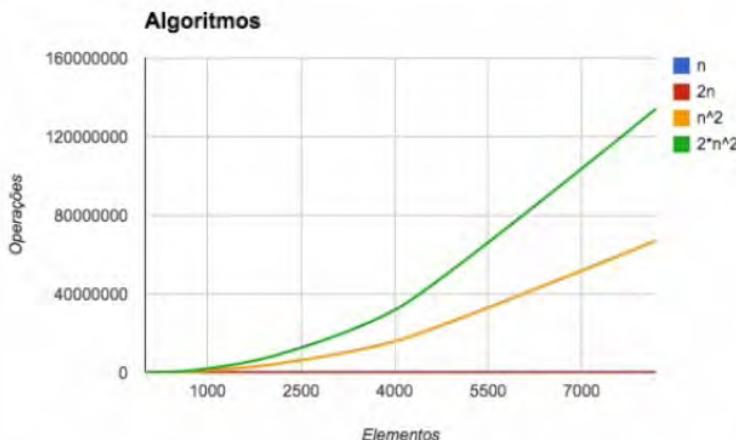


Figura 6.17: Comparação de desempenho de algoritmos em gráfico

No entanto, observe que tem algo estranho no gráfico. Nós não conseguimos visualizar a linha n , nem a linha $2n$ direito, porque as duas são muito "baixas" e permanecem próximas do eixo inferior. Como os valores das linhas quadráticas são muito superiores, mal conseguimos ver as outras linhas. O algoritmo quadrático cresce tão rápido e vai tão "alto" que, quando fazemos a comparação, não conseguimos localizar os lineares.

Por isso, vamos excluir alguns elementos da tabela para conseguirmos visualizar as duas linhas. Que tal visualizarmos até 32 elementos? 27? 5.000? 5.000 parece muito, pois teria de fazer uma tabela enorme. Vou tentar analisar somente até o caso de 64 elementos, e verificamos se será o suficiente para algumas conclusões.

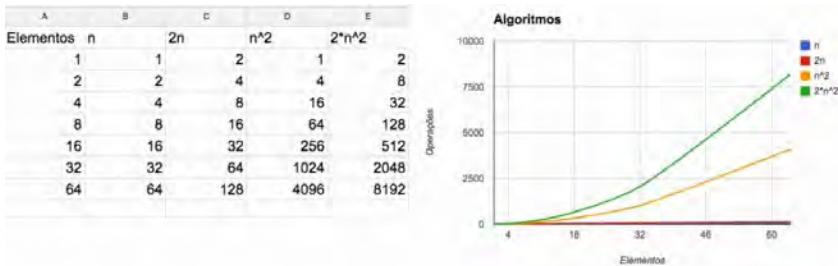


Figura 6.18: Comparação do desempenho de algoritmos — gráfico 2

Como a distância entre a linha que mais cresce e a que menos cresce é muito grande, não é possível ainda visualizar a diferença entre todas elas. Repare que as duas de baixo estão "coladinhas". Então precisaremos excluir mais elementos da tabela para enxergar as duas linhas que faltam. Vamos tentar o limite de 16 elementos.

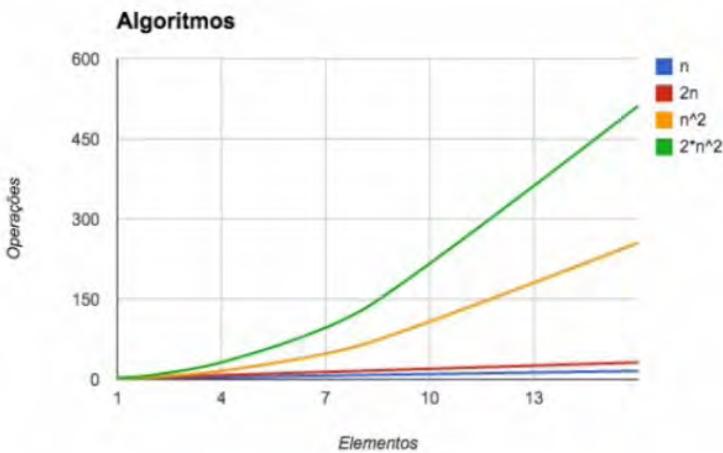


Figura 6.19: Comparação do desempenho de algoritmos — gráfico 3

Agora conseguimos visualizar um pouco melhor. As linhas quadráticas logo se distanciam das linhas que representam algoritmos lineares, crescendo absurdamente. No entanto, analisando a velocidade dos dois algoritmos, o quadrático é muito

lento em relação ao linear.

Quando temos apenas 16 elementos, se usarmos um algoritmo linear, faremos entre 16 e 32 contas. Porém, se estivermos usando um algoritmo que é quadrático, faremos entre 256 e 512 operações. A diferença entre os dois algoritmos é absurda.

Observe também que a diferença no resultado entre n e $2n$ não parece ser tão grande. O mesmo acontece com os valores do quadrático. Não há tanta diferença entre as duas linhas. Provavelmente, se incluíssemos uma terceira opção, um algoritmo cúbico por exemplo, o número de operações seria ainda maior e a diferença entre as duas linhas não seria tão expressiva.

Naturalmente, se dobrarmos os valores, significa que ele será duas vezes pior. No entanto, a diferença entre as linhas de um mesmo algoritmo não é a mesma quando modificamos de linear para quadrático. Isto significa que, quando analisamos de forma geral os algoritmos, nós não nos preocupamos tanto com o fator de multiplicação, como temos em $2n$ ou $2n^2$, ou mesmo $2n^5$.

O resultado será irrelevante em uma comparação geral. O interesse não está no fator de multiplicação, mas sim, na potência.

Observe que a **POTÊNCIA 1** linear cresce como uma linha, enquanto a **potência 2** quadrática deslancha no nosso gráfico e vai embora. As **potências 3 e 4** vão ainda mais adiante. É na **potência** que estamos interessados.

Por isso, se o nosso foco for no desempenho dos algoritmos, vamos preferir os lineares (e só depois os quadráticos e outros com crescimento pior).

Inclusive, podem existir algoritmos ainda mais rápidos (ou lentos) do que os comparados na nossa análise.

6.10 COMO COMPARAR O DESEMPENHO DO ALGORITMO

Uma vez que a potência é quem dita a regra de crescimento do nosso gráfico, ao compararmos dois algoritmos em relação ao desempenho de velocidade, ignoraremos o **fator de multiplicação 2** e analisaremos apenas o n e o quadrático.

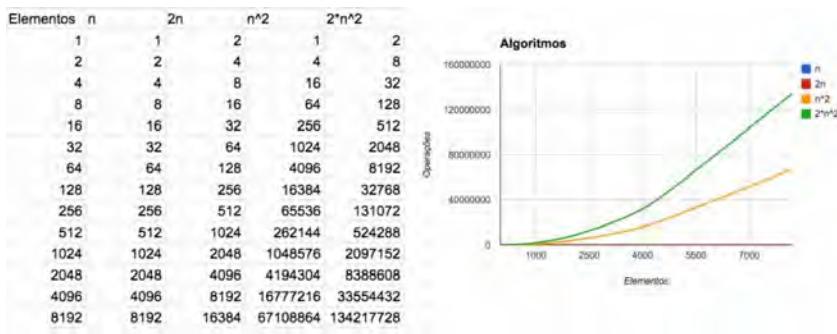


Figura 6.20: Comparaçāo do desempenho de algoritmos quando com muitos elementos

Nós observamos que, quando analisamos um número elevado de elementos, as linhas do algoritmo quadrático deslanchavam. Entre n e $2n$, não existe diferença. Qual é a conclusão que podemos tirar do gráfico quando temos a intenção de encontrar o menor produto de uma lista? Com certeza, ordenar a lista não é uma boa ideia. Para colocar os elementos em ordem apenas com o objetivo de encontrar o menor de todos, é preferível usar o algoritmo `buscaMenor`, porque este é linear.

Porém, se estamos interessados em saber quais são os três elementos mais baratos, os cinco mais caros ou o produto localizado na metade da lista, aparentemente é mais simples ordenar a lista.

Inclusive, porque o `buscaMenor` não resolveria o problema. Essa é a sacada: o `buscaMenor` só pode ser utilizado para encontrarmos o **menor** e o **maior** elemento.

No entanto, quando ordenamos a lista, podemos resolver diversos problemas. Então, primeiro vamos verificar: "para resolver o problema, usar o `buscaMenor` será útil? Ou ordenar é a melhor opção? Qual algoritmo resolverá mais rápido?".

Caso os dois algoritmos resolvam o nosso problema, a melhor escolha será utilizar o `buscaMenor`, por este ser linear e crescer mais devagar em comparação com o quadrático. Se temos poucos elementos, por exemplo 4, fazer 8 ou 32 operações não faz diferença para o computador, como veremos em seguida.

Mas a sacada principal, quando compararmos dois algoritmos, é analisarmos se ele é linear ou quadrático — ou outras classificações. Ele será linear quando tiver um `for` que passe por todos os elementos. No entanto, quando tivermos um `for` com outro laço dentro, que também passe por todos elementos, teremos a quantidade de elementos multiplicada pela quantidade de elementos. Neste caso, o algoritmo será quadrático, ou seja, sendo dividido ou multiplicado por 2.

Vamos compará-los agora no computador, e ver quanto tempo demoraria para resolver os nossos problemas.

6.11 COMPARANDO O DESEMPENHO DO ALGORITMO EM UM COMPUTADOR

Sabemos que, quando temos um algoritmo linear, ele crescerá como uma linha no gráfico. Se temos um algoritmo quadrático, ele terá um crescimento de forma quadrática. Mas e o computador? Quantas operações ele executará?

No mundo real, quando marcamos um encontro, ao falarmos do tempo, nós não dizemos "daqui a cinco operações, estarei lá" ou "me encontre em cinquenta operações". Nós usamos segundos, minutos etc.

A	B	C	D	E
Elementos	n	2n	n^2	$2*n^2$
1	1	2	1	2
2	2	4	4	8
4	4	8	16	32
8	8	16	64	128
16	16	32	256	512
32	32	64	1024	2048
64	64	128	4096	8192
128	128	256	16384	32768
256	256	512	65536	131072
512	512	1024	262144	524288
1024	1024	2048	1048576	2097152
2048	2048	4096	4194304	8388608
4096	4096	8192	16777216	33554432
8192	8192	16384	67108864	134217728

Figura 6.21: Comparando o desempenho do algoritmo em um computador — passo 1

Então, vamos imaginar um computador que execute 8 mil operações por segundo. Criaremos uma nova coluna na tabela para representar esse computador capaz de rodar 8.000 operações por segundo, e vou chamar o computador de *Comp8000*.

Vamos verificar quanto tempo ele (*Comp8000*) demora para rodar um algoritmo linear (n). Então, chamaremos a coluna de *Comp8000 n*. Para executar uma única operação ($n=1$), o programa vai demorar 0,000125 segundos neste computador. Será uma fração muito pequena de segundo e o processo será muito rápido.

A	B	C	D	E	F
Elementos	n	2n	n^2	$2*n^2$	$Comp8000\ n$
1	1	2	1	2	0.000125
2	2	4	4	8	0.00025
4	4	8	16	32	0.0005
8	8	16	64	128	0.001
16	16	32	256	512	0.002
32	32	64	1024	2048	0.004
64	64	128	4096	8192	0.008
128	128	256	16384	32768	0.016
256	256	512	65536	131072	0.032
512	512	1024	262144	524288	0.064
1024	1024	2048	1048576	2097152	0.128
2048	2048	4096	4194304	8388608	0.256
4096	4096	8192	16777216	33554432	0.512
8192	8192	16384	67108864	134217728	1.024

Figura 6.22: Comparando o desempenho do algoritmo rodando em um computador — passo 2

Se preenchermos o resto da tabela (número de operações dividido por 8000), o computador vai demorar só 1 segundo para 8.192 operações.

Agora se tivermos um algoritmo quadrático, com o *Comp8000* reagirá? O quão rápido ele seria? Vamos criar uma nova coluna, agora chamada *Comp8000 n²*, e nela vamos dividir n^2 por 8.000. Quando executarmos apenas uma operação, o tempo será o mesmo utilizado por outro algoritmo (0,000125 segundos).

A	B	C	D	E	F	G
Elementos	n	2n	n^2	$2 \cdot n^2$	Comp8000 n	Comp8000 n^2
1	1	2	1	2	0.000125	0.000125
2	2	4	4	8	0.00025	0.0005
4	4	8	16	32	0.0005	0.002
8	8	16	64	128	0.001	0.008
16	16	32	256	512	0.002	0.032
32	32	64	1024	2048	0.004	0.128
64	64	128	4096	8192	0.008	0.512
128	128	256	16384	32768	0.016	2.048
256	256	512	65536	131072	0.032	8.192
512	512	1024	262144	524288	0.064	32.768
1024	1024	2048	1048576	2097152	0.128	131.072
2048	2048	4096	4194304	8388608	0.256	524.288
4096	4096	8192	16777216	33554432	0.512	2097.152
8192	8192	16384	67108864	134217728	1.024	8388.608

Figura 6.23: Comparando o desempenho do algoritmo rodando em um computador — passo 3

À medida que o número de operações aumenta, a demora também será um pouco maior. Inicialmente, como o tempo é inferior a 1 segundo, a diferença é "irrelevante". Porém, com 512 elementos, o computador demorará 32 segundos para executar o algoritmo e, para fazer o maior número de operações da nossa tabela, precisará de 8388,608 segundos para executá-lo.

Estimando o valor por hora, a demora será de 2h33.

Enquanto o *Comp8000* leva 1 segundo para um algoritmo linear, o *Comp8000* vai demorar mais de 2 horas para executar um algoritmo quadrático.

É por isso que consideramos que multiplicar n por 2 não é o grande critério quando fazemos uma análise geral do algoritmo. Com 8 mil elementos, a diferença entre n e $2n$ vai saltar de 1 para 2 segundos. No entanto, estamos mais interessados na diferença entre

1 segundo e 2 horas.

Como o computador executa 8 mil operações por segundo, poderíamos argumentar que, para o usuário final, a diferença de frações de segundo é irrelevante. Logo com 32 ou 64 elementos, a diferença entre os dois algoritmos não será grande, já que ambos terão uma demora de menos de 1 segundo. Podemos implementar qualquer um, o que for da preferência do programador. Porém, entre a demora de 32 segundos e 0,06 segundo, teremos de analisar qual dos algoritmos será mais útil.

No algoritmo quadrático, serão necessárias 2 horas para a execução, versus 1 segundo do algoritmo linear. Isto é, o algoritmo linear cresce de forma linear, enquanto o algoritmo quadrático terá um crescimento "gigantesco" (quadrático).

6.12 A ANÁLISE ASSINTÓTICA

Observe que a análise do nosso algoritmo não está focada se os valores são 1 segundo ou 2 segundos, 1 minuto ou 2 minutos. Mas sim o quanto rápido esse tempo cresce à medida que o número de elementos cresce.

Estamos interessados no número de operações que o computador faz a longo prazo, quando as linhas deslancham no gráfico, para analisarmos se o crescimento será linear, quadrático ou outras possibilidades. Quando fazemos uma análise que leva tais aspectos em consideração, ela é chamada de **assintótica**.

Ao fazermos uma **análise assintótica** do algoritmo, identificamos se ele é linear ou quadrático. É possível que, para valores pequenos, a diferença entre um e outro seja irrelevante. Porém, quando temos um aumento nos valores, as diferenças se tornarão mais expressivas. Por exemplo, no **Comp 800 n²**, o tempo

para execução de operações passa das 2 horas.

A	B	C	D	E	F	G
Elementos	n	2n	n^2	$2*n^2$	Comp8000 n	Comp8000 n^2
1	1	2	1	2	0.000125	0.000125
2	2	4	4	8	0.00025	0.0005
4	4	8	16	32	0.0005	0.002
8	8	16	64	128	0.001	0.008
16	16	32	256	512	0.002	0.032
32	32	64	1024	2048	0.004	0.128
64	64	128	4096	8192	0.008	0.512
128	128	256	16384	32768	0.016	2.048
256	256	512	65536	131072	0.032	8.192
512	512	1024	262144	524288	0.064	32.768
1024	1024	2048	1048576	2097152	0.128	131.072
2048	2048	4096	4194304	8388608	0.256	524.288
4096	4096	8192	16777216	33554432	0.512	2097.152
8192	8192	16384	67108864	134217728	1.024	2.330168889

Figura 6.24: Quando usar um algoritmo ou outro

Mas poderíamos pensar que a demora seria menor com um computador mais rápido. Mesmo que o computador usado seja duas vezes mais potente, ainda precisaríamos de 1 hora para encontrar a solução do problema. Se o computador fosse quatro vezes mais rápido (e mais caro também), teríamos ainda uma demora de meia hora, enquanto seria possível executar em 1 segundo a mesma quantidade de operações com outro algoritmo.

Por maior que seja a rapidez do computador, o desempenho de um algoritmo linear para outro continuará sendo imensa. O que precisamos descobrir é se o algoritmo linear vai resolver o nosso problema.

Observe que esta é a gravidade dos algoritmos quadráticos ou de outros ainda piores, e de usarmos um algoritmo qualquer, sem pensarmos em qual seria a melhor solução. Esta é grande vantagem de trabalharmos com algoritmos mais espertos e mais rápidos. Ele nos permite trabalhar com o mesmo computador, mantendo os

gastos baixos. Você consegue usar o seu computador de forma mais rápida e eficiente.

Se fosse preciso usar o método quadrático o tempo inteiro, seria preciso investir uma grande quantia de dinheiro em computadores absurdamente potentes. Imagine se, para resolver um algoritmo quadrático, tivéssemos de comprar um computador quadraticamente mais potente.

Como temos uma limitação na potência dos computadores, é muito importante que os algoritmos sejam mais rápidos e consumam menos de processamento do nosso computador. Por isso, sempre vamos favorecer algoritmos que seguirão uma tendência linear do que a quadrática.

Nós vimos como se comporta um algoritmo buscaMenor (linear) e Ordenacao (quadrático). Então, paramos para pensar: "se vamos apenas buscar o menor, não faz sentido ordenarmos nossos elementos".

No entanto, se o objetivo for identificar diversos pedaços de uma lista, por exemplo, os três menores ou os dois maiores, talvez seja mais simples ordenar todos os itens e depois fazer inúmeras pesquisas com base nos elementos já ordenados. Depende da análise de qual método vale mais a pena. Porém, se o nosso objetivo for apenas encontrar o menor, certamente o algoritmo linear será a melhor opção.

Mas também trabalhamos com outro algoritmo: o insertionSort . Falamos dele justo quando percebemos que algo não funcionava bem no selectionSort , que parecia ficar lento quando adicionávamos muitos elementos.

O `insertionSort` é linear ou quadrático? Qual resposta vamos obter se fizermos uma análise similar a que fizemos até agora?

6.13 ANALISANDO O INSERTION SORT

Vamos analisar o algoritmo `insertionSort`. Como ele funciona mesmo? Nós criamos um laço que varre do começo até o fim de uma lista, da esquerda para direita. Ele passa por todos os elementos n vezes — porque temos n elementos.

Voltaremos ao exemplo das cartas de baralho, que tinham cinco itens. O que era feito com cada um delas?

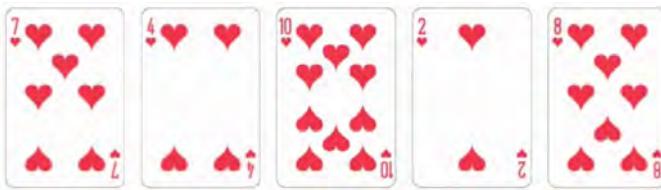


Figura 6.25: Analisando o Insertion Sort — passo 1

Quando precisávamos comparar o primeiro elemento, a carta 7, com todos os outros itens posicionados antes, percebíamos que não havia nada. Então, já começamos a análise a partir do elemento na segunda posição, a carta 4. Então, comparávamos a carta com todas as outras atrás e as reposicionávamos.

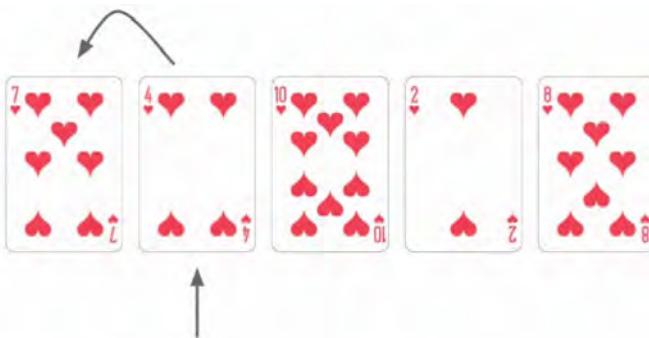


Figura 6.26: Analisando o Insertion Sort — passo 2

Depois, seguíamos para a carta 10 e a comparávamos com todas as posicionadas antes dela. Não precisávamos mudar a ordem.

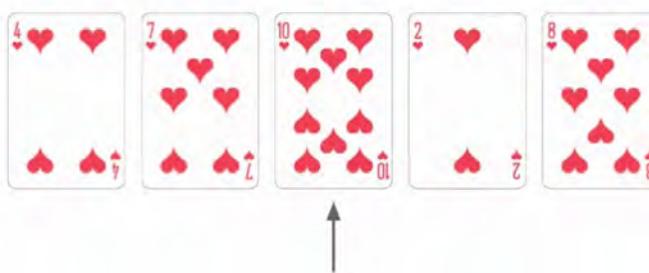


Figura 6.27: Analisando o Insertion Sort — passo 3

Seguíamos a análise para a carta 2 e a comparávamos com todas atrás dela.

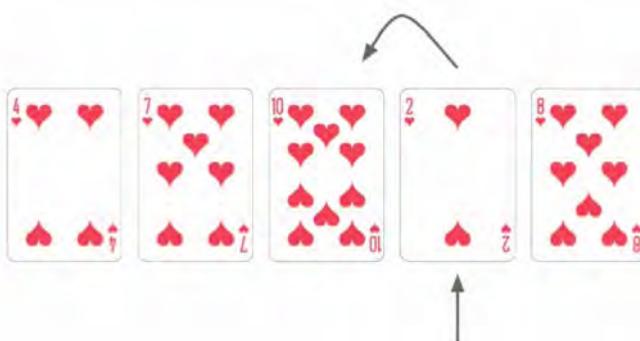


Figura 6.28: Analisando o Insertion Sort — passo 4

Trocávamos para a carta de posição até identificar qual era a mais adequada.

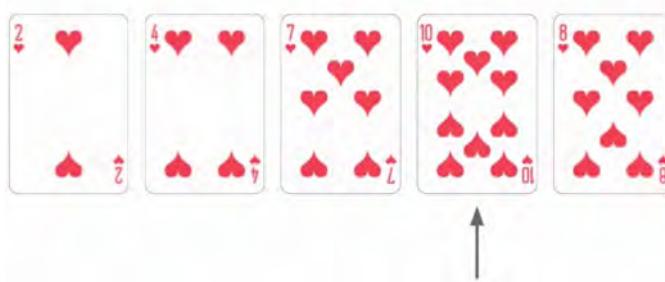


Figura 6.29: Analisando o Insertion Sort — passo 5

A nossa análise chegava até a carta 8 e novamente a comparávamos com as que estavam posicionadas atrás.

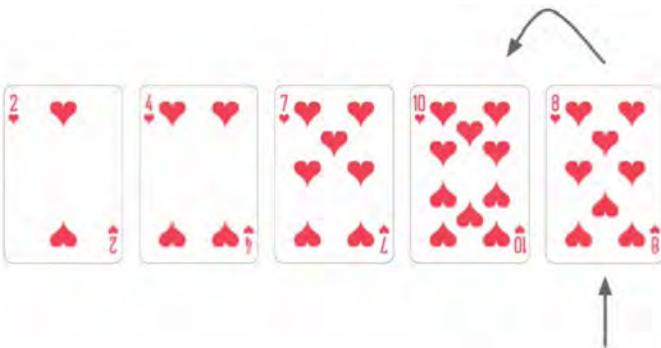


Figura 6.30: Analisando o Insertion Sort — passo 6

Até que todas estivessem na posição correta.

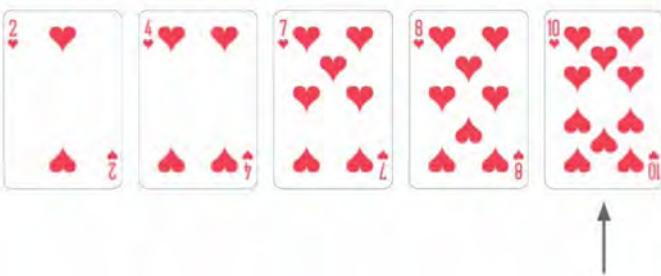


Figura 6.31: Analisando o Insertion Sort — passo 7

Observe que o nosso laço passou por todos os itens n vezes e, dentro dele, criamos um `for` que varreu todos os elementos da esquerda. Ou seja, um laço precisava passar por todos os elementos até a direita, enquanto o outro, no pior caso, passava por todos os itens posicionados à esquerda.

Ainda que tivéssemos uma variação nas operações, não faria muita diferença para o algoritmo — mesmo que fosse uma divisão, multiplicação, ou a soma de uma constante. A diferença só existe quando multiplicamos ou dividimos por n . É o que estamos fazendo.

Vamos para o código e analisaremos o algoritmo `insertionSort`:

```
private static void insertionSort(Produto[] produtos, int quantidadeDeElementos)
    for(int atual=1; atual < quantidadeDeElementos; atual++) {
        System.out.println("Estou na casinha " + atual);

        int analise = atual;
        while(analise > 0 && produtos[analise].getPreco() < produtos[analise -1].getPreco())
            troca(produtos, analise, analise -1);
        analise--;
    }
```

O algoritmo passa por todos os elementos, exceto o primeiro. Então, poderíamos deduzir que seria preciso trabalhar com $n-1$. Nós já comentamos anteriormente que, quando fazemos uma análise de longo prazo, se vamos ou não subtrair é irrelevante. A **análise assintótica** vai ignorar esta característica. Isto significa que vamos passar pelos n elementos.

Como é especificado no `while`, para cada um dos n elementos, o laço varrerá para a esquerda.

```
while(analise > 0 && produtos[analise].getPreco() < produtos[analise -1].getPreco())
    troca(produtos, analise, analise -1);
    analise--;
```

Isto significa que ele vai repetir o processo enquanto analisar do elemento 1 até 5. Logo, teremos n multiplicado por n . Ou seja, assim como o `selectionSort`, o algoritmo `insertionSort` também é quadrático. Ele também terá um desempenho terrível quando considerarmos o tempo de execução das operações.

Mas será que não existem outros algoritmos de ordenação e de busca, que podem ser até mais rápidos do que os quadráticos? Ou será que o mundo está fadado a fazer as coisas com o mesmo método? Sempre precisaremos de computadores absurdamente

potentes para fazer a ordenação de um número elevado de elementos? Devem existir outros algoritmos. Veremos adiante quais são eles.

O importante é sabermos como analisar estes algoritmos, para poder compará-los. Até o momento, aprendemos que: temos três algoritmos, um que busca o menor elemento e dois que fazem ordenação. Aprendemos a analisá-los e a identificar o crescimento de cada um deles, para afirmar "esse cresce bem e este outro não será útil se tivermos muitos elementos. Vou ter de procurar um novo algoritmo".

Além da maneira linear e quadrática, existem outras maneiras com que um algoritmo pode crescer. Será que existem outras maneiras?

6.14 ALGORITMOS QUE RODAM COM O TEMPO CONSTANTE

Nós observamos que um algoritmo linear comparado com um quadrático vai performar muito melhor. Enquanto a linha do quadrático crescerá rapidamente no gráfico, a linear vai permanecer bem próxima ao eixo inferior. Quase nem a percebemos, mesmo quando o número de elementos passa de 8 mil elementos.

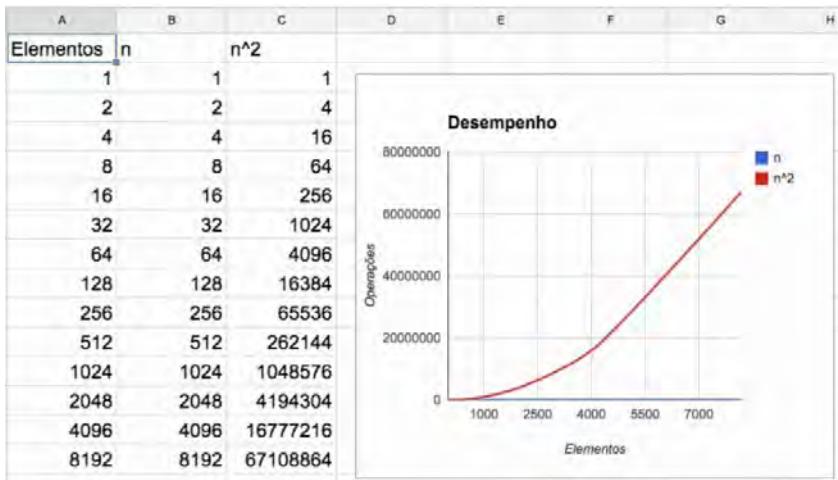


Figura 6.32: Algoritmos com tempo constante — passo 1

Porém, será que existe algum algoritmo mais rápido do que o linear? Pode haver um que seja realmente rápido.

Por exemplo, se queremos descobrir qual elemento está posicionado no meio de um array? Temos uma lista com cinco casas, o elemento buscado será o que estiver na posição do meio. Praticamente, executamos uma operação. Será muito rápido! Temos de descobrir apenas qual elemento está no meio, porém não precisamos fazer inúmeros cálculos.

O mesmo se quisermos descobrir qual é o elemento que está no começo ou no fim do array. Estas operações apenas pedem que "informe quem está aqui ou ali". Dizemos que tais operações têm **tempo constante**. Elas dependem de um número constante e, por isso, são muito rápidas. Imagine que, se o tempo de demora for 1 no início, também será 1 quando chegar ao fim.

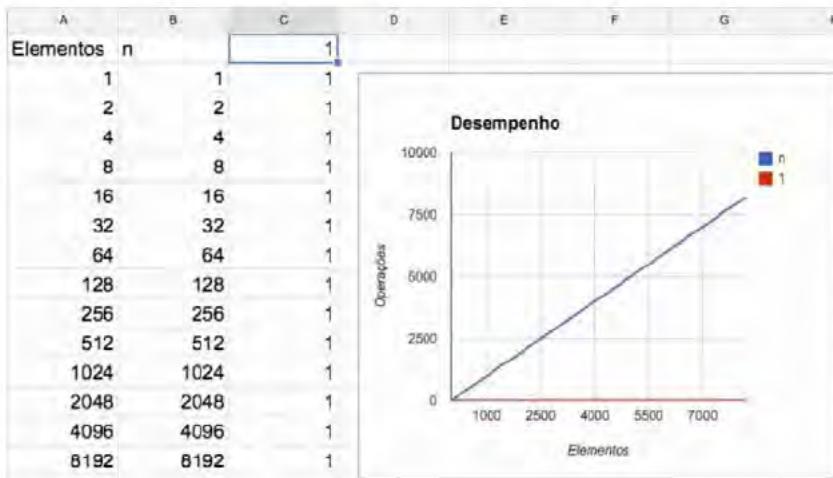


Figura 6.33: Algoritmos com tempo constante — passo 2

Mesmo que o algoritmo linear (linha na diagonal) cresça como uma linha no gráfico, observe como o algoritmo com o tempo constante (linha de baixo) permanecerá reto. Então, o algoritmo constante (que é 1) será mais rápido do que um algoritmo linear (que é n).

Os algoritmos constantes trabalhados até agora podem identificar:

- O elemento do meio;
- O primeiro elemento;
- O último elemento.

Parecem questões menores, porém, descobriremos utilidades para elas. Por exemplo, já vimos que se temos um array ordenado, quanto tempo vamos precisar para indicar o carro que tem o preço do meio? Se os elementos estão ordenados, basta selecionar o item no meio da lista. Isto é, se o array estiver ordenado, o algoritmo para selecionar o item do meio será constante.

Por exemplo, se precisarmos selecionar os três elementos mais

baratos ou os cinco mais caros, também usaremos um algoritmo constante. Caso o array esteja ordenado, todas estas perguntas podem ser respondidas rapidamente. No entanto, precisaremos usar também bons algoritmos de ordenação.

Na comparação entre algoritmos, os constantes são mais rápidos do que os lineares.

6.15 ALGORITMOS COM DESEMPENHO BASEADO EM LOG

Existe o algoritmo constante e o linear. Será que não temos um algoritmo além destes dois? Podem existir vários!

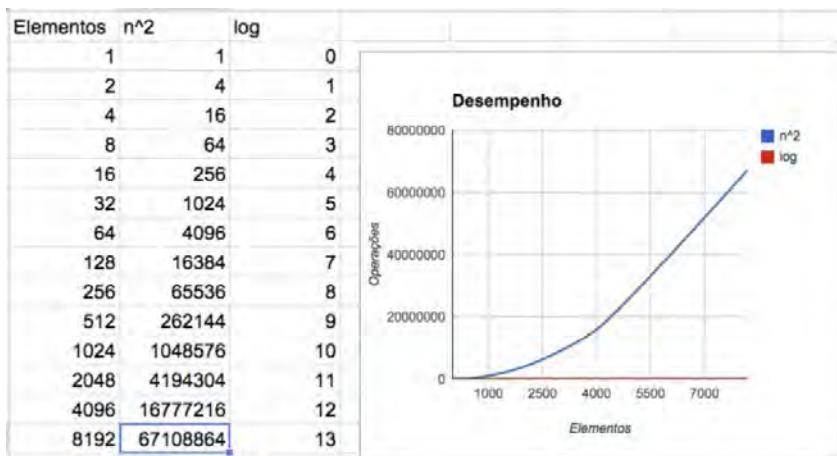


Figura 6.34: Algoritmos n logn — passo 1

Outro algoritmo interessante será usado quando procuramos um nome em uma lista telefônica ou em uma agenda, em que os contatos já estão ordenados. Ele não será constante, porque não sabemos exatamente onde estará posicionado o nome da pessoa. Nós apenas tentamos adivinhar a posição correspondente.

Este tipo de algoritmo de busca também é um algoritmo mais

rápido do que uma busca simples. Ele não é constante, mas também não é linear. É algo entre os dois. O algoritmo recebe o nome de **logarítmico**, porque depende do *log* do nosso número (*log na base 2*). Nós vamos ver como e quando usar algoritmos do gênero.

E como funciona o algoritmo *log na base 2*? Ele será o *log* do elemento relacionado na tabela, na base 2. Observe os resultados: quando tivermos dois elementos, ele fará 1 operação. Quando trabalharmos com o maior número de elementos da tabela, ele fará 13 operações. Com 8.192 elementos, ele executará 13 operações. É um número muito baixo e um ótimo resultado.

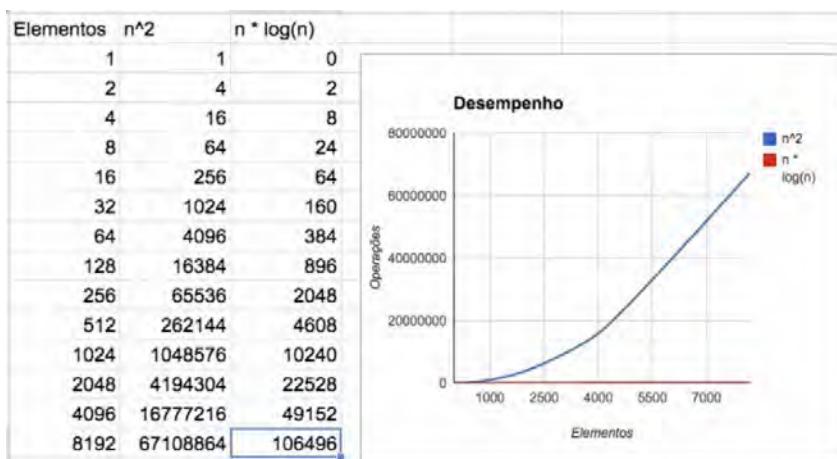


Figura 6.35: Algoritmos $n \log n$ — passo 2

Se tivermos um array de produtos para serem mostrados, poderemos resolver muito rápido as 13 operações.

Voltando ao exemplo da agenda de telefone, provavelmente nós temos menos de 8 mil contatos salvos na agenda. Se fizermos uma busca da maneira como estamos habituados, teremos de executar 8 mil operações. Mas se fizermos uma busca logarítmica, vamos fazer 13 operações — e nós não precisamos fazer uma busca com tantos elementos na nossa agenda. A busca que fazemos em uma agenda

telefônica, em geral, é mais esperta.

Mais adiante, nós ensinaremos ao computador como fazer estas buscas mais espertas, assim como buscamos naturalmente na agenda de contatos.

Quando observamos a implementação de um algoritmo que é logarítmico, percebemos que ele cresce menos no gráfico e, por isso, tem um desempenho melhor do que uma linear. Então, até o momento, analisamos os algoritmos constantes, logarítmicos, lineares e, depois, a quadrática.

6.16 ALGORITMOS $N^*LOG\ N$

Nós conhecemos a logarítmica, a constante, a linear e a quadrática. Vamos incluir esta última novamente na tabela.

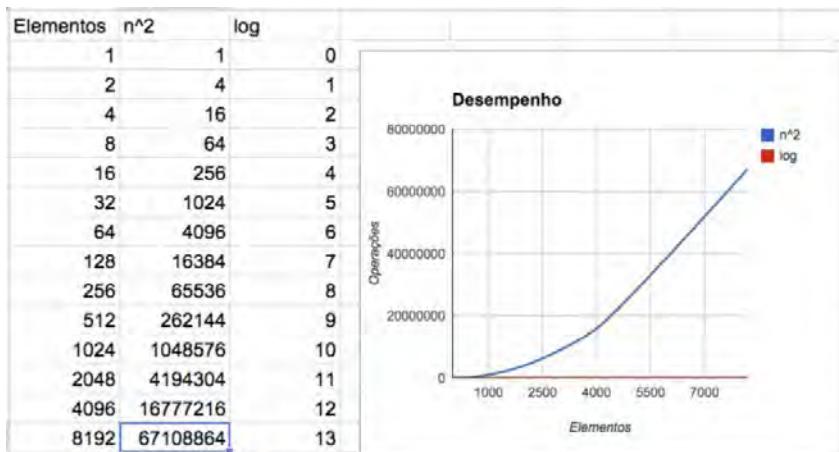


Figura 6.36: Algoritmos $n \log n$ — passo 1

Observe que a quadrática (n^2) comparada com a logarítmica (\log) tem uma diferença gigantesca no gráfico. Em uma fazemos 13 operações, enquanto na outra fazemos mais de 67 milhões de operações.

E será que entre a linear (n) e a quadrática (n^2), não existe outra entre as duas linhas? Sim, temos. Vamos multiplicar o n pelo \log de n ($n \log(n)$).

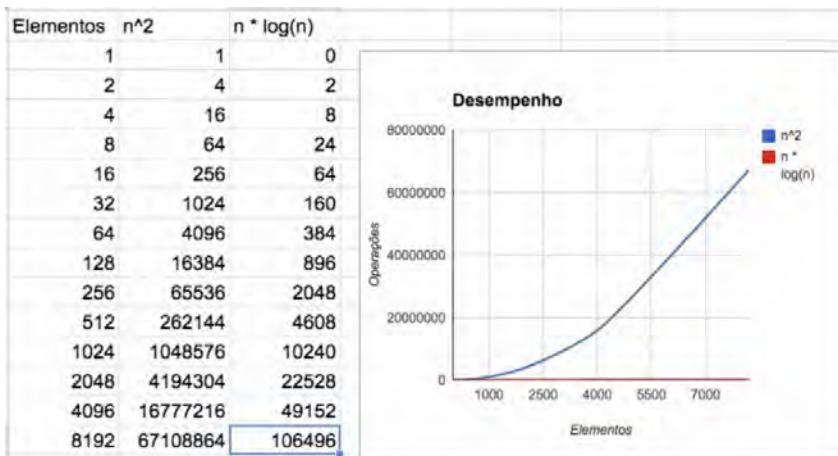


Figura 6.37: Algoritmos $n \log(n)$ — passo 2

Então, nossos resultados serão superiores aos resultados alcançados com a logarítmica. Quando trabalharmos com 8.192 elementos, teremos de executar 106.496 operações. É maior do que o resultado 13, da logarítmica. Porém, ainda é muito inferior do que o número 67.108.864, da quadrática.

É possível perceber no gráfico do $n \log(n)$ que a linha surge e fica bem próxima de 0, enquanto a linha do crescimento quadrático deslancha. Ao compararmos os algoritmos $n \log(n)$ com n^2 , a primeira opção parece ser muito melhor, por isso daremos preferência para ela.

Então, temos os algoritmos constante (1), o linear (n), o linear logarítmico ($n \log(n)$) e o quadrático (n^2). Naturalmente, depois do quadrático, ainda teremos mais opções.

6.17 ALGORITMOS CÚBICOS

Depois do quadrático, o que poderemos ter? Está claro que, se aumentarmos o número da potência, os valores ficarão ainda maiores.

Vamos comparar o algoritmo quadrático (n^2) com o cúbico (n^3)? Os valores da terceira coluna serão o resultado de n^3 .

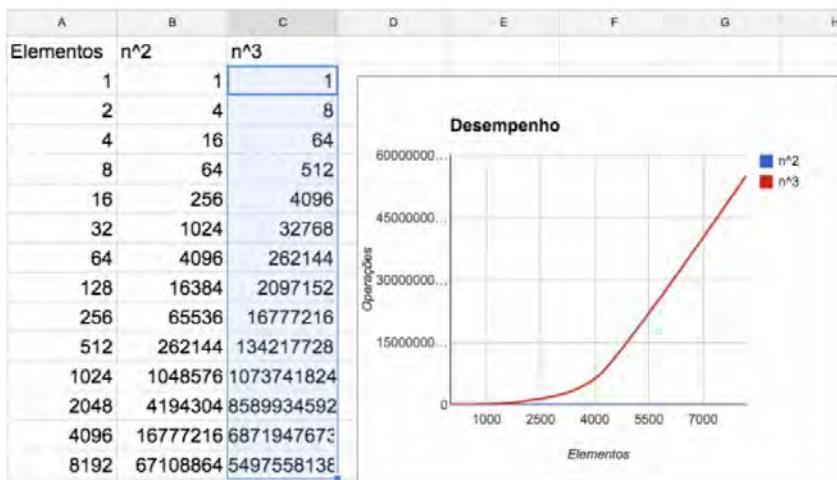


Figura 6.38: Algoritmo cúbico

Observe como a representação do algoritmo cúbico deslancha no gráfico! A diferença entre os algoritmos é bastante expressiva. Nitidamente, o cúbico terá um desempenho pior do que o quadrático.

Para 8.192 elementos, enquanto n^2 faz 67.108.864 operações, n^3 fará 54.905.581.388 — um valor tão grande que nem é possível visualizá-lo inteiro na célula da tabela. Por isso, rodar um algoritmo cúbico no computador é inviável. Pelo menos, usando um semelhante ao do exemplo, que faz 8 mil operações por segundo.

6.18 ALGORITMO EXPONENCIAL

Depois do cúbico, será que não teremos mais nada? Sim, ainda podemos elevar à quarta, à quinta, à sexta potência. E o desempenho vai piorando em cada um deles. Evitaremos usá-los, porque será impossível rodá-los.

Anteriormente, vimos que temos o constante (1), o linear (n), o linear logarítmico ($n\log(n)$) e o quadrático (n^2), depois o cúbico (n^3). Mas, será que existe outra categoria em que os resultados crescerão muito rápido e que também explodirão no gráfico? Sim. 2 elevado a n (2^n) terá um desempenho ainda pior em relação a todos os outros.

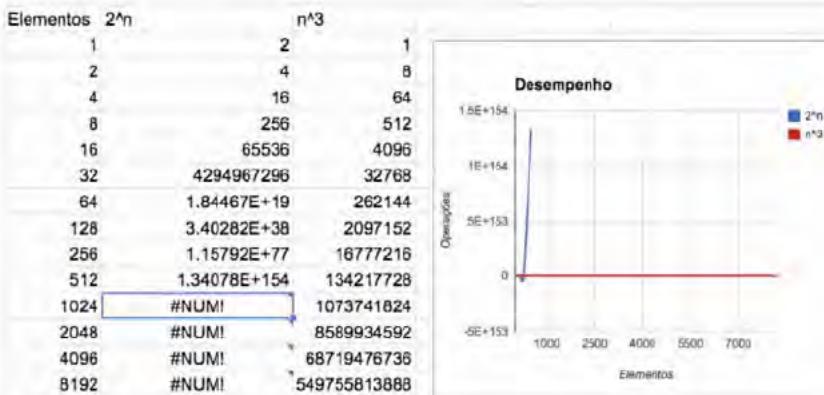


Figura 6.39: Algoritmo exponencial

À medida que fazemos os cálculos, podemos perceber que vão surgindo valores elevados:

- 2 elevado a 16 será igual a 65.536.
- 2 elevado a 32 será igual 4.294.967.296.
- 2 elevado a 64 será igual a 1.84467E+19.

O resultado de 2 elevado a 64 será um número tão grande, que o programa decidiu escrever em notação científica e apenas indicou a

presença de 19 casas ($E+19$), além do valor apresentado. Ele fez o mesmo com outros valores, até desistir e começar a apresentar a mensagem de erro #NUM!, nas últimas células da coluna, indicando que o número estourou.

No gráfico, o programa foi incapaz de desenhá-lo completamente e a linha do algoritmo exponencial começa a vazar pelo eixo inferior. Isto prova que 2^n é um algoritmo inviável. Quando trabalharmos com 16 elementos, já será difícil executar a quantidade de operações, com 32 elementos, o tempo de demora será absurdo.

Precisaremos encontrar outra solução para não usarmos o algoritmo exponencial. Observe que existem classes de algoritmos que não podem ser usadas e que vão nos fazer procurar novas soluções.

6.19 ANÁLISE ASSINTÓTICA DE UM ALGORITMO

Nós vimos que temos diversos tipos de algoritmos e que a performance deles variará bastante. Por isso, estaremos sempre atentos nos casos em que tivermos dois laços alinhados, porque pode ser difícil utilizar alguns deles.

Com dois laços passando por todos os elementos de um array, teremos de tomar alguns cuidados — em especial com o segundo, para que ele não seja quadrático. Por exemplo, vamos retomar o algoritmo do `insertionSort` que escrevemos anteriormente. O código do método recebia o array de produtos e a quantidade de elementos:

```
private static void insertionSort(Produto[] produtos, int quantidadeDeElementos) {
    for(int atual=1; atual < quantidadeDeElementos; atual++) {
        System.out.println("Estou na casinha " + atual);
```

```

        int analise = atual;
        while(analise > 0 && produtos[analise].getPreco() < produtos[analise -1].getPreco()) {
            troca(produtos, analise, analise -1);
            analise--;
        }
    }
}

```

A função `troca` é exatamente mudar um elemento de posição com outro.

```
troca(produtos, analise, analise -1);
```

Será um algoritmo constante, porque ele simplesmente faz esta troca. Também analisamos outro algoritmo, que passa por todos os elementos e é linear.

```

while(analise > 0 && produtos[analise].getPreco() < produtos[analise -1].getPreco()) {
    troca(produtos, analise, analise -1);
    analise--;
}

```

Ainda que passe apenas na metade ou no dobro dos elementos, ele continuará sendo linear.

Descobrimos que, entre os algoritmos analisados, existia um **logarítmico**. Levando em consideração o desempenho, depois do **linear**, tínhamos o **linear logarítmico** e o **quadrático** (como o `insertionSort` e o `selectionSort`).

Quando inseríamos mais um `for` passando por fora do laço e que também varria todos os elementos, ele se tornava **cúbico**. Com mais um `for`, ele era elevado à potência 4, e com outro `for` adicionado, era elevado à potência 5.

Analisamos também o **exponencial** (2^n) no qual a linha crescia muito rápido no gráfico.

Todos estes valores (constante, linear, logarítmico, quadrático, cúbico e exponencial) são maneiras de descobrirmos a forma geral da curva que o algoritmo se aproxima. Esta é o que chamamos de **análise assintótica**.

É incomum dizermos que um algoritmo linear é 1. A forma correta é $O(1)$. Isto significa que analisamos o algoritmo e ele é constante. $O(n)$ é linear. $O(n^2)$ é quadrático. $O(n^3)$ é cúbico. $O(2^n)$ significa que é exponencial. Utilizamos a letra **O** maiúscula para dizer como o algoritmo se comporta — qual será o seu desempenho assintoticamente.

6.20 O PRÓXIMO DESAFIO: NOVOS ALGORITMOS

Já aprendemos a comparar algoritmos em relação ao número de operações que executa em determinados cenários. Chegou a hora de ver algoritmos mais complexos, alguns que também usamos no nosso dia a dia sem perceber, e que são mais rápidos do que os que vimos até agora.

CAPÍTULO 7

INTERCALANDO ARRAYS PRÉ-ORDENADOS

O que acontece quando eu recebo cinco cartas de baralho? Em diversos jogos, é interessante que as cartas menores fiquem separadas em uma parte, enquanto as maiores fiquem em outra. Assim, fica mais fácil encontrar os elementos que tenho na mão. Como é costume trabalhar desta maneira, é comum que as pessoas façam esta ordenação manualmente.

É simples ordenar cinco cartas do poker, ou três cartas do truco. Mas quando ordenamos treze cartas do buraco, a dificuldade já aumenta. Quanto maior for a quantidade de itens para serem ordenados, mais trabalhoso será o processo.

Por quê? Porque os algoritmos que as pessoas usam são o *Selection Sort* e *Insertion Sort* e, à medida que o número de elementos cresce, eles começam a ficar muito lentos. Serão muitas operações e trocas, e isto justifica a demora.

Nós não queremos passar por isto. Vamos usar um exemplo que tenha mais de 13 cartas? Imagine um jogo de buraco ou de poker, em que no fim da partida todas as cartas estejam na mesa. E então, você pede para que alguém ordene todas as cartas. Ninguém gostaria de fazer isto.

Outro exemplo seria o jogo da vida. No fim da partida, precisamos que alguém reordene as notas do dinheirinho falso.

Ninguém quer ser a pessoa que executará a tarefa de ordenação.

7.1 DIVIDINDO O TRABALHO

No fim do jogo de baralho, do jogo da vida, ou organizar as notas de 300 alunos, ninguém quer ficar responsável por ordenar, porque são muitos elementos. É trabalhoso fazer um *Selection Sort* e um *Insertion Sort*, porque teremos de executar muitas operações. Será que não existem outros algoritmos ou processos mais rápidos?

Eu gosto de encontrar formas mais simples, e por isso, vou contar um segredo que costuma funcionar e é utilizado por muitas pessoas. Quando o jogo de buraco acaba e todas as cartas estão na mesa, se alguém decide ordenar os itens, devemos agrupar todos eles e dividi-los entre os participantes.

Se temos um monte de cartas — ou de dinheirinho — e tivemos uma partida com quatro jogadores, distribuímos $1/4$ do total de elementos para cada participante e todos serão responsáveis pela ordenação da parte recebida. **Ordenar $1/4$ do total de cartas é menos trabalhoso do que organizar todos os itens!** Após cada um ordenar a sua parte, reagrupamos as cartas em um único monte que já estará organizado.

O que nós fizemos? Quebramos o nosso problema em quatro pedaços (poderiam ter sido um número maior ou menor) e depois de organizados, juntamos as partes. Podemos fazer o mesmo processo com as provas de alunos.

Por exemplo, vamos corrigir as provas do Enem e precisamos ordenar 1 milhão delas. Que trabalhoso executar o *Insertion Sort*! E se encontrássemos uma outra maneira de fazer a ordenação?

Vamos usar a maneira que já conhecemos ao organizarmos as cartas de baralho, o dinheirinho do Jogo da Vida. Usaremos o

exemplo das provas dos nove alunos:



Figura 7.1: Ordenar provas do Enem — passo 1

Em vez de ordenar todos os itens, eu divido os alunos entre duas pessoas: o Aniche e o Alberto. Então, cada um deles será responsável pela ordenação considerando as notas dos alunos. Vamos observar como ficam os grupos do Aniche e do Alberto ordenados:



Figura 7.2: Ordenar provas do Enem — passo 2

7.2 CONQUISTANDO O RESULTADO

Agora que já temos os dois grupos ordenados, o nosso trabalho será uni-los. A sacada quando trabalhamos com um número grande de elementos é dividir a tarefa entre as pessoas. No nosso exemplo, o grupo de aluno foi dividido entre o Aniche e o Alberto, e cada um

teve de organizar a metade.

Então, o problema que quero resolver é: considerando o grupo do Aniche e o grupo do Alberto já ordenados — ou seja, dados os arrays ordenados dos dois —, o que teremos de fazer é intercalar os elementos das duas sequências para ter um array todo organizado.

Assim, dado o grupo do Aniche e o do Alberto, o que farei é intercalar todos os objetos para que eles fiquem ordenados em um único array. Dado dois arrays ordenados, intercale os elementos e monte um array ordenado. Se formos capazes de fazer isto, será um enorme avanço, porque basta dividir a ordenação entre as pessoas e, depois, unir os elementos ordenados — tarefa que já sabemos fazer.

Vamos tentar unir dois arrays de uma maneira mais rápida do que faríamos com outros algoritmos, e assim conseguir ordenar uma quantidade de dados muito maiores. Com o algoritmos que estamos montando, nós seremos capazes. O primeiro passo será: considerando que os grupos do Aniche e do Alberto já estão ordenados, como vamos unir estes dois?

7.3 COMO JUNTAR OU INTERCALAR DUAS LISTAS ORDENADAS

Baseado no que usamos cotidianamente, no jogo de truco, no buraco, no Jogo da Vida, ou em qualquer atividade que exija a distribuição de diversos itens, podemos distribuir entre os participantes para que todos ajudem a organizar, e depois juntamos tudo novamente. Queremos resolver o problema de reagrupar os elementos.



Figura 7.3: Intercalar duas listas ordenadas — passo 1

Considerando os alunos do Aniche e do Alberto, que já foram ordenados, como podemos intercalar os nove alunos? Teremos de trabalhar com a variável de 9.



Figura 7.4: Intercalar duas listas ordenadas — passo 2

Sempre que estamos trabalhando com ordenação em linguagem de programação, que o array já nos diz qual o número total de elementos, nós conseguimos extraír o número total. Como queremos generalizar para qualquer linguagem de programação, precisamos saber o total de alunos, que no caso são 9.

Vamos reagrupá-los em um único array que caiba todos. Então, nosso primeiro passo será criar um array com um tamanho em que caibam os 9 elementos.



Figura 7.5: Intercalar duas listas ordenadas — passo 3

Caso contrário, não teremos como juntar todos os elementos em um único array. Agora que temos um lista com tamanho 9,

podemos começar.

Vamos iniciar com o primeiro elemento de cada grupo ordenado, porque começar pelo meio não faria sentido. Seguiremos a opção mais simples.

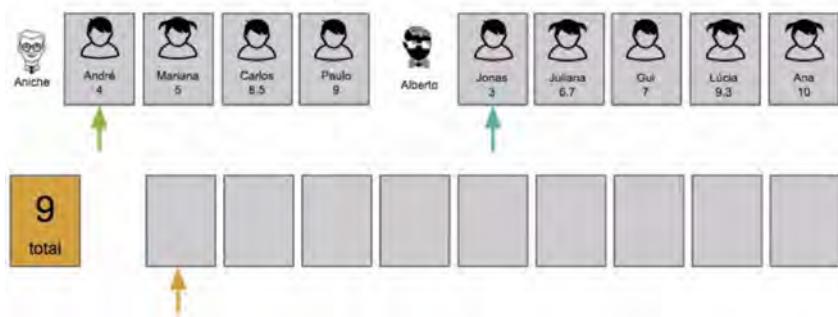


Figura 7.6: Intercalar duas listas ordenadas — passo 4

Vamos comparar o André com o Jonas. Um teve uma nota 4 e o outro teve uma nota 3. O Jonas não se saiu bem nas provas! Ele será o menor do nosso novo array.

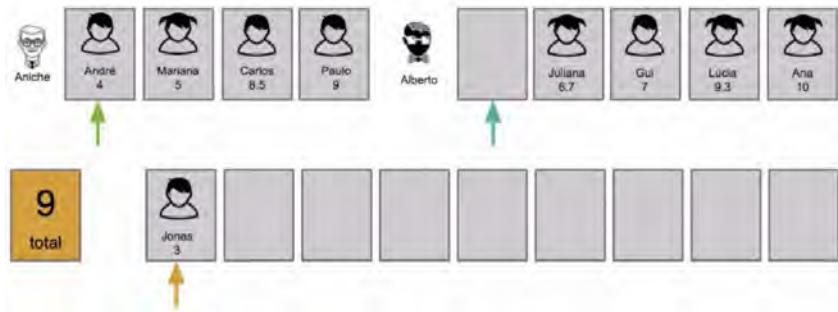


Figura 7.7: Intercalar duas listas ordenadas — passo 5

Como já colocamos o Jonas no início da lista, vamos comparar agora o próximo dos grupos do Aniche e do Alberto: o André e a Juliana. Um aluno tirou uma nota 4, enquanto o outro tirou uma nota 6,7. Qual dos dois tirou uma nota menor? O André.

Vou descer o André para o array.

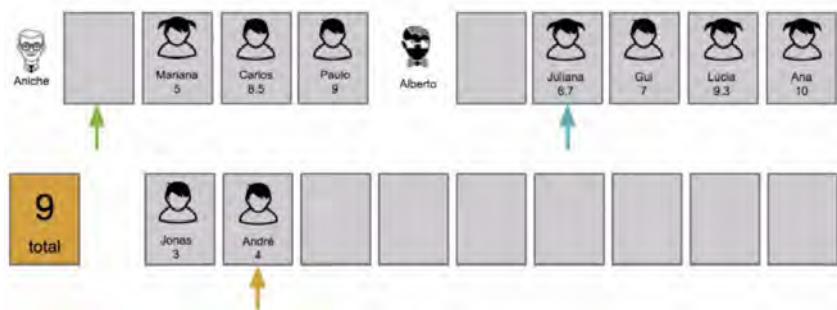


Figura 7.8: Intercalar duas listas ordenadas — passo 6

Seguimos comparando as próximas alunas de cada grupo: Mariana e Juliana. Qual das duas tirou uma nota menor? A Mariana. Vamos descê-la para o array.

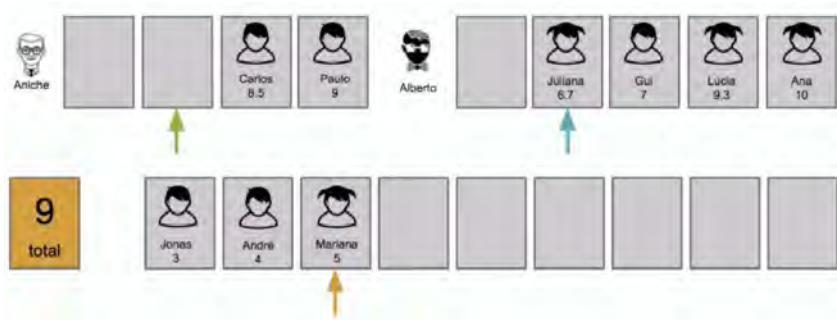


Figura 7.9: Intercalar duas listas ordenadas — passo 7

Os próximos alunos a terem as notas comparadas serão: o Carlos e a Juliana. Quem se saiu pior na prova? A Juliana. Vamos descê-la para o novo array.

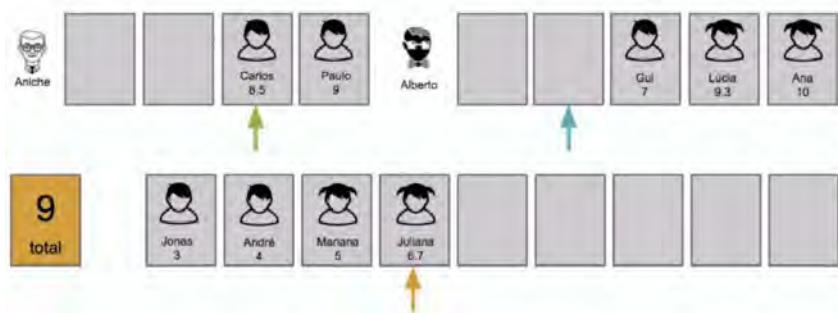


Figura 7.10: Intercalar duas listas ordenadas — passo 8

Compararemos as notas do Guilherme e do Carlos. O Guilherme tirou uma nota 7, enquanto o Carlos tirou 8,5. Quem teve a menor nota? O Guilherme. Então, vamos descer-lo para o array.

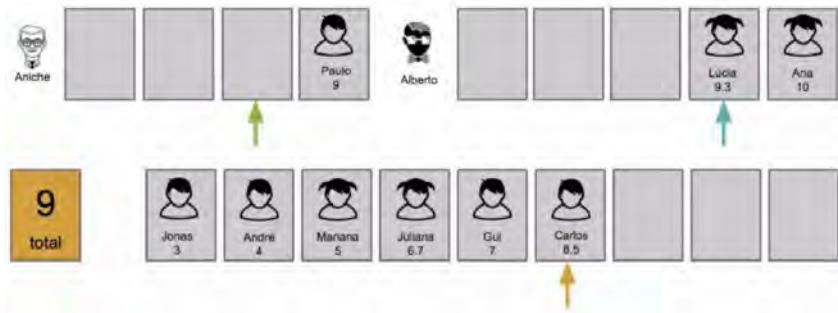


Figura 7.11: Intercalar duas listas ordenadas — passo 9

Seguimos para os próximos alunos. Quem se saiu melhor na prova, o Carlos ou a Lúcia? A Lúcia. Logo, é o Carlos que vai descer para o array.

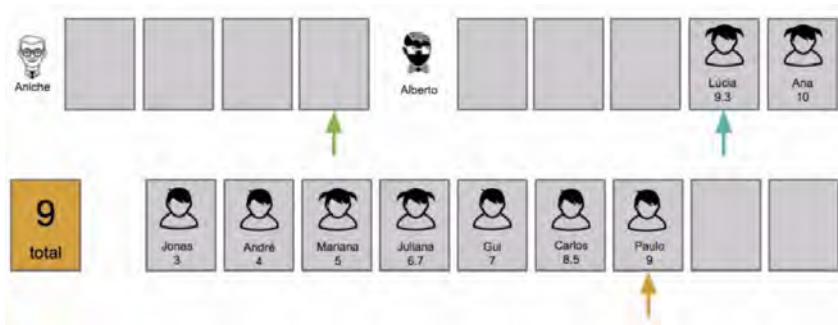


Figura 7.12: Intercalar duas listas ordenadas — passo 10

Comparando o Paulo e Lúcia, quem se saiu pior na prova? O Paulo. Vamos descê-lo para o array.

Então, um por um, fui comparando os alunos, e o menor foi sendo colocado no array. Com os dois alunos restantes, não faz mais sentido procurar o menor. Agora que já descemos todos os alunos de um dos grupos, vamos levar as duas restantes para o array. Colocaremos a Lúcia.

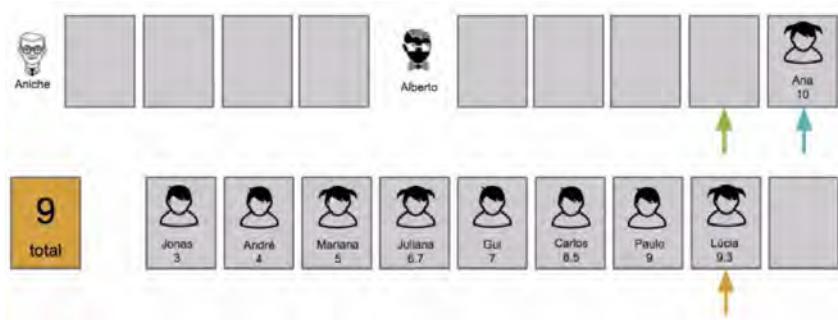


Figura 7.13: Intercalar duas listas ordenadas — passo 11

E depois, a Ana.

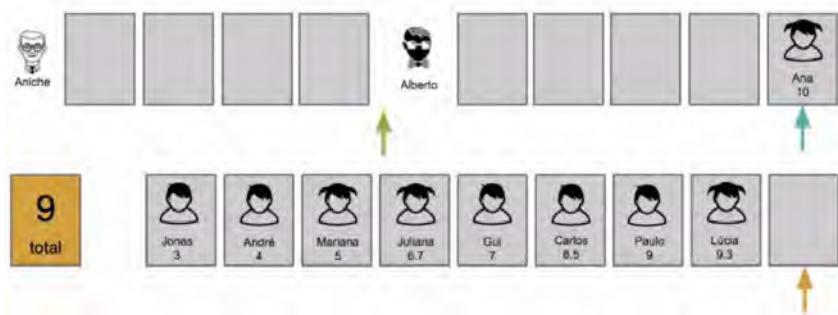


Figura 7.14: Intercalar duas listas ordenadas — passo 12

No fim, quando terminamos as comparações, sobraram duas alunas do grupo do Alberto. Pegamos os dois elementos e descemos para a lista única.



Figura 7.15: Intercalar duas listas ordenadas — passo 13

O que fizemos? Comparamos o elemento de um grupo com o de outro e identificamos qual era o menor. O maior permanecia e o menor descia para o novo array. Seguimos comparando as notas dos alunos e os menores eram colocados na lista abaixo. No fim, temos um array quase completo, porém sobraram algumas alunas no grupo do Alberto. Nós simplesmente descemos as duas e o array ficou ordenado.

Com isto, na próxima vez que formos jogar baralho e todas as

cartas terminarem embaralhadas, já sabemos como organizar: dividimos o monte em partes e distribuímos para os participantes ordenarem com seus próprios algoritmos. No fim, analisamos os grupinhos de cartas e comparamos os itens: "a menor carta é minha, coloca no monte. Agora é a sua, coloca no monte". As menores cartas serão agrupadas em um novo monte, até que não seja mais possível compará-las e as cartas estejam ordenadas.

7.4 SIMULANDO COM AS VARIÁVEIS

Então já sabemos intercalar dois arrays ordenados. Agora, quais variáveis eu preciso para ver isto acontecer?

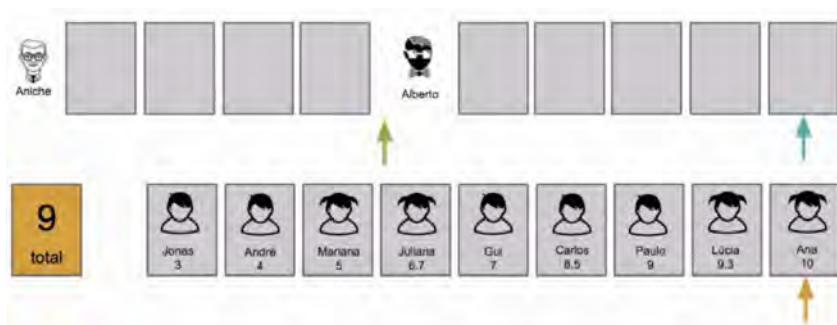


Figura 7.16: Simulando com as variáveis — passo 1

Nós utilizamos três setinhas, três variáveis, com que nós brincamos e cujos valores mudamos, além da variável referente ao total de elementos, que precisávamos saber desde o princípio.

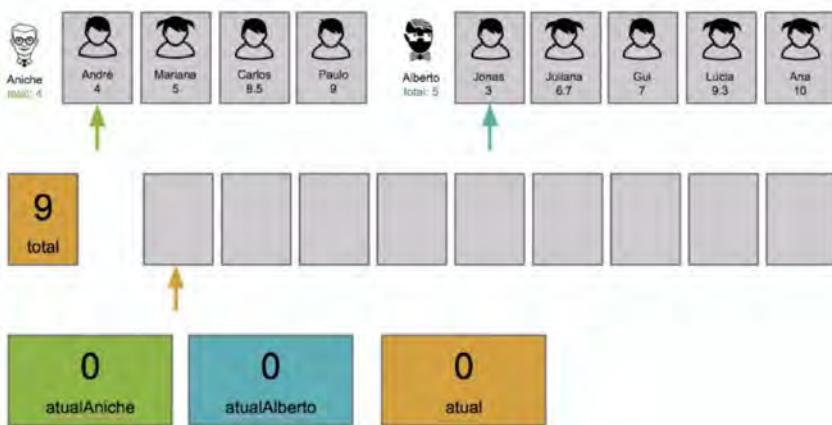


Figura 7.17: Simulando com as variáveis — passo 2

Com o total, nós criamos o tamanho do nosso array. Precisamos descobrir qual era o aluno **atual** do Aniche e qual era o do Alberto. Também precisávamos saber o número total de elementos em cada um dos grupos, e assim vamos brincando com o atual das três listas, incluindo a geral. Como farei isto? Vamos simular de novo o algoritmo que usamos no exemplo, mas agora usaremos as variáveis.

Quando compararmos as notas dos alunos André e Jonas, qual dos dois teve o pior resultado? O Jonas. Então, vamos movê-lo para a posição **atual** do array geral. Em seguida, o que devemos fazer? Vamos somar 1 com a posição **atual**.

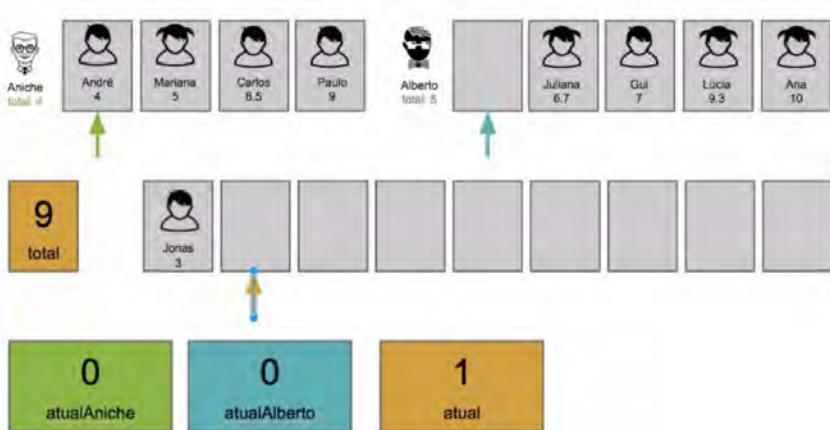


Figura 7.18: Simulando com as variáveis — passo 3

Faremos o mesmo com a atual do Alberto, porque queremos analisar o próximo elemento.

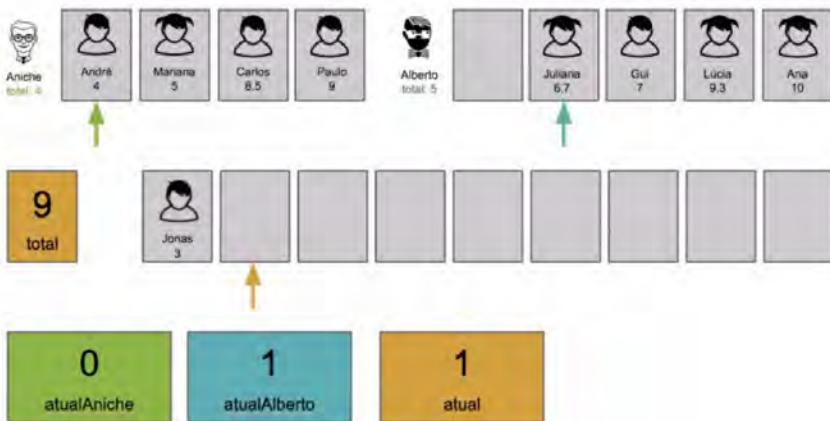


Figura 7.19: Simulando com as variáveis — passo 4

O elemento atual do Aniche (André) é maior ou menor do que o atual do Alberto (Juliana)? É menor. Então, vamos movê-lo para a posição atual do array geral.

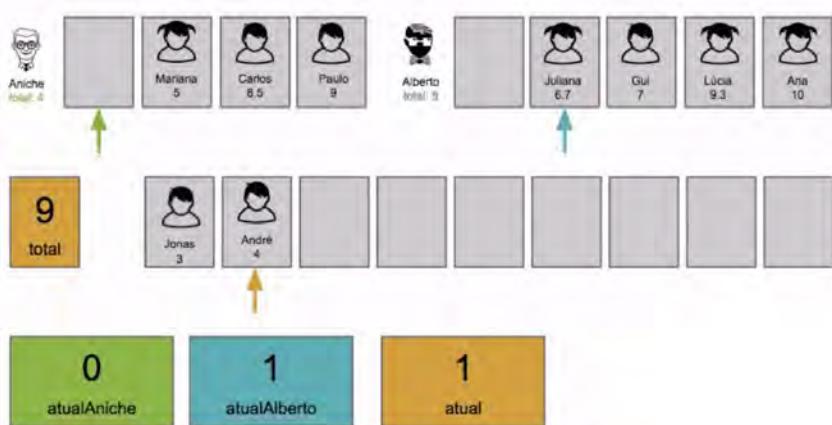


Figura 7.20: Simulando com as variáveis — passo 5

Para continuar, precisamos aumentar 1 do atual e 1 do atual do Aniche, porque queremos analisar o próximo item do grupo dele.

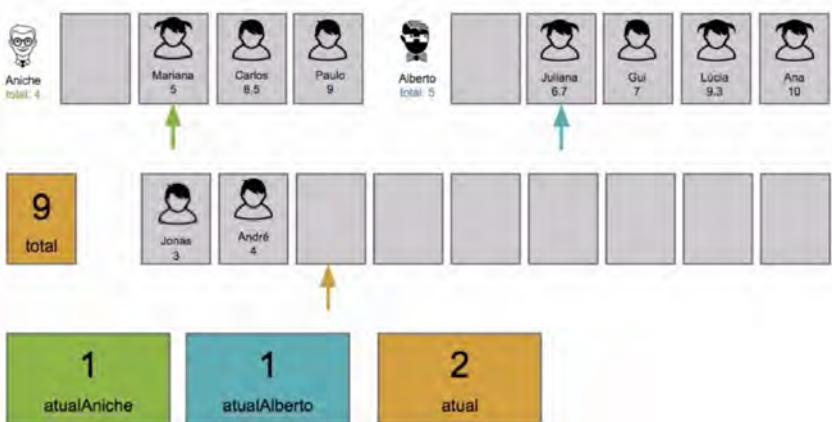


Figura 7.21: Simulando com as variáveis — passo 6

Isto significa que precisamos ter três indicadores apontando para qual posição estamos trabalhando em cada um deste arrays.

Estou trabalhando com o aluno na primeira posição do grupo do Aniche, do grupo do Alberto e do nosso novo array. Depois

vamos caminhando com os indicadores, colocando-os nas posições adequadas. Estas são as variáveis de que precisaremos logo em seguida. Vamos implementar o código?

7.5 CRIANDO O PROJETO E PREPARANDO PARA JUNTAR ORDENADO

Vamos criar o nosso projeto, que será um *Java project*.

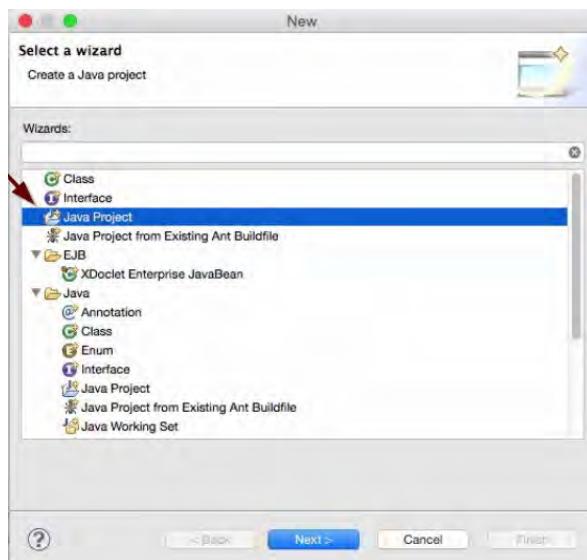


Figura 7.22: Java project

Ele será chamado **Notas**. Dentro, teremos uma classe para representar cada uma das notas que foram tiradas pelos alunos. A classe receberá o nome `Nota`, no pacote `br.com.alura.notas`.



Figura 7.23: Classe Nota

Esta nota tem o nome do aluno (`String aluno`) e tem o valor da nota, que pode ser quebrado (`valor`), como 7,5 ou 9,7, por exemplo. O valor será entre 0 e 10.

```
package br.com.alura.notas;

public class Nota {
```

```
    private String aluno;
    private double valor;
```

```
}
```

Quero adicionar um construtor que receba o nome do aluno e o valor das notas. Ele aceitará tanto o aluno como o valor:

```
public Nota(String aluno, double valor) {
    this.aluno = aluno;
    this.valor = valor;
```

```
}
```

Vamos criar o *getter* apenas quando for necessário. Momentaneamente, é o suficiente. Fecharemos esta classe e, em seguida, criaremos um arquivo de teste.

Nesse arquivo, teremos dois arrays com as notas já ordenadas por dois professores. Depois pediremos para o programa juntar estas notas em um único array. Vamos implementar a **junção**, quando fazemos o **merge**. Vamos juntar, fundir, os dois grupos de notas ordenadas.

Vamos fazer o `TestaMerge`, e testar fundir as listas com o método `main`. Trabalharemos dentro desta classe, digitando diversas notas de alunos.

```
public class TestaMerge {  
  
    public static void main(String [] args) {  
        mariana 5  
        andre 4  
        paulo 9  
        carlos 8.5  
        juliana 6.7  
        jonas 3  
        lucia 9.3  
        ana 10  
        guilherme 7  
    }  
}
```

Temos disponíveis as notas com que vamos trabalhar. Os alunos estão divididos em dois grupos. No primeiro, teremos: Mariana, André, Paulo e Carlos. No segundo, teremos: Juliana, Jonas, Lucia, Ana e Guilherme. O que queremos fazer é juntar os elementos em uma única lista.

O primeiro grupo é do professor Aniche. Então, criaremos um array indicando isto:

```
public class TestaMerge {  
  
    public static void main(String [] args) {  
        Nota[] notasDoAniche = {  
            new Nota("mariana", 5),  
            new Nota("andre", 4),  
            new Nota("paulo", 9),  
            new Nota("carlos", 8.5)  
        };  
    }  
}
```

O professor Aniche foi gentil e já ordenou as notas, do menor para o maior.

```
public class TestaMerge {  
  
    public static void main(String [] args) {  
        Nota[] notasDoAniche = {  
            new Nota("mariana", 5),  
            new Nota("andre", 4),  
            new Nota("carlos", 8.5),  
            new Nota("paulo", 9)  
        };  
    }  
}
```

Recebemos também as notas do segundo grupo, que é do professor Alberto. Vamos indicar no código, que as notas são Alberto (`notasDoAlberto`):

```
Nota[] notasDoAlberto = {  
    juliana 6.7  
    jonas 3  
    lucia 9.3  
    ana 10  
    guilherme 7  
}
```

As notas do Alberto também já foram entregues em ordem. O professor já fez um *Insertion Sort*, ou um *Selection Sort*, e recebemos a ordenação pronta.

```
Nota[] notasDoAlberto = {  
    new Nota("jonas", 3),
```

```
    new Nota("juliana", 6.7),
    new Nota("guilherme", 7),
    new Nota("lucia", 9.3),
    new Nota("ana", 10)

};
```

Então, temos os dois grupos de nota ordenados. O faremos agora? Juntaremos os dois grupos, e fundiremos as duas coleções de itens ordenados. Vamos mostrar que somos capazes de unir todos os elementos em uma única lista ordenada. Isto significa que quero criar uma lista com um ranking final, ordenado pelas notas dos alunos, do menor para o maior.

Queremos ter um rank que vai fundir todas as notas:

```
Nota[] rank = junta(notasDoAniche, notasDoAlberto);
```

Depois que o programa criar meu rank, quero que ele imprima todos os valores ordenados corretamente. Isto é: para cada nota dentro do rank, vou querer imprimir a nota e o nome do aluno (`nota.getAluno()`).

```
for(Nota nota) {
    System.out.println(nota.getAluno());
}
```

Será o `getAluno` que devolverá o próprio aluno :

```
public String getAluno() {
    return aluno;
}
```

Nós precisamos implementar o `junta`. Isto é: dado duas coleções, com os elementos já ordenados, como fazemos para juntar os dois arrays e resolver rapidamente esta ordenação? Se já paralelizamos e dividimos o trabalho entre as pessoas, vamos unir as diversas partes ordenadas. Vamos juntar todos em um único array ordenado.

Este método `junta` recebe o primeiro (`notasDoAniche`) e o

segundo array (`notasDoAlberto`) de notas, e é o método que implementaremos.

7.6 IMPLEMENTANDO O JUNTA/INTERCALA

Temos de implementar agora a função `junta()` que, dado dois arrays ordenados, vai uni-los em uma única lista com todos os elementos. Ele juntará os dois arrays e reorganizará os itens.

Vamos juntar os arrays?

```
private static Nota[] junta(Nota[] notasDoAniche, Nota[] notasDoAlberto) {  
    return null;  
}
```

Precisamos identificar o número de elementos dos dois grupos para criar um novo array que caiba todos eles. Isto significa que:

```
int total = notasDoAniche.length + notasDoAlberto.length;
```

Vamos também criar o resultado, que é a lista final.

```
Nota[] resultado = new Nota[total];
```

Nós também devolveremos o `resultado`. O nosso código ficará assim:

```
private static Nota[] junta(Nota[] notasDoAniche, Nota[] notasDoAlberto) {  
    int total = notasDoAniche.length + notasDoAlberto.length;  
    Nota[] resultado = new Nota[total];  
    return resultado;  
}
```

Ainda falta inserir as notas. Tanto `notasDoAniche` como `notasDoAlberto` já estão ordenados. Vamos incluir os elementos de ambos no nosso array.

Como fizemos a ordenação antes? Verificava qual era a menor nota do grupo do Aniche e, depois, do grupo do Alberto.

Comparava os dois elementos e colocava o menor na lista geral. Seguimos para o próximo elemento de cada grupo, e depois para o próximo, até não ser mais possível comparar os grupos. Então, precisamos começar com a primeira posição de cada lista. Isto significa que:

```
int atualDoAniche = 0;  
int atualDoAlberto = 0;
```

Assim vamos começar pela primeira posição dos dois grupos. Observe a primeira posição de cada um, quem está lá? Vamos chamar de `nota1` a nota que está na posição `atualDoAniche` no array `notasDoAniche`:

```
Nota nota1 = notasDoAniche[atualDoAniche];
```

Qual será a nota 2? Será `notasDoAlberto`, que está na posição `atualDoAlberto`.

```
Nota nota2 = notasDoAlberto[atualDoAlberto];
```

Vamos descobrir qual das duas notas é a menor. Se (`if`) a nota 1 (`nota1.getValor`) for menor do que a nota 2 (`nota2.getValor`), a menor nota será a do Aniche. Se não (`else`), será a do Alberto.

```
Nota nota1 = notasDoAniche[atualDoAniche];  
Nota nota2 = notasDoAlberto[atualDoAlberto];  
if(nota1.getValor() < nota2.getValor()) {  
    // mauricio  
} else {  
    // alberto  
}
```

É isto que queremos comparar. Vamos identificar quando uma ou outra é a menor, para então colocá-la no array geral. Vamos criar o método `getValor`, que vai devolver um `double` (que é o `valor`) e salvar a classe `Nota`:

```
public double getValor() {  
    return valor;
```

```
}
```

Temos nosso `getValor`. Se a `nota1` for menor que a `nota2`, usaremos a `nota1`. Caso a menor seja a `nota2`, ela que será usada. Temos de fazer isto dentro do laço.

Precisamos ir andando dentro do array. Isto significa que todo o código deverá ser executado enquanto pudermos comparar as notas dos dois grupos. Enquanto (`while`) o `atualDoAniche` não ultrapassar o total do grupo (`notasDoAniche.length`) e o `atualDoAlberto` não ultrapassar o total do outro grupo (`notasDoAlberto.length`), podemos seguir colocando itens na nova lista.

```
while(atualDoAniche < notasDoAniche.length &&
      atualDoAlberto < notasDoAlberto.length) {
}
```

Enquanto o `while` for verdadeiro, devo continuar analisando. Vou recortar parte do `if` do código e colá-lo dentro do `while`:

```
while(atualDoAniche < notasDoAniche.length &&
      atualDoAlberto < notasDoAlberto.length) {

    Nota nota1 = notasDoAniche[atualDoAniche];
    Nota nota2 = notasDoAlberto[atualDoAlberto];
    if(nota1.getValor() < nota2.getValor()) {
        // mauricio
    } else {
        // alberto
    }
}
```

Enquanto tivermos elementos para serem analisados, seguimos com o processo de comparação.

Qual é o primeiro elementos de cada grupo? Temos o André, com 4, e o Jonas, com 3. Então, ele cairá no caso do Alberto. A `nota2.getValor` é referente ao Jonas, e é a menor. Então, vamos colocá-la no resultado. Para isso, vamos escrever no nosso código

que o resultado na posição 0 é igual a nota2 .

```
Nota nota1 = notasDoAniche[atualDoAniche];
Nota nota2 = notasDoAlberto[atualDoAlberto];
if(nota1.getValor() < nota2.getValor()) {
    // mauricio
} else {
    // alberto
    resultado[0] = nota2;
}
```

Se o menor fosse o do Aniche, escreveríamos que o resultado na posição 0 é a nota1 .

```
Nota nota1 = notasDoAniche[atualDoAniche];
Nota nota2 = notasDoAlberto[atualDoAlberto];
if(nota1.getValor() < nota2.getValor()) {
    // mauricio
    resultado[0] = nota1;
} else {
    // alberto
    resultado[0] = nota2;
}
```

Agora que analisamos a nota1 e a nota2 , temos de ir para o próximo do Alberto (atualDoAlberto++). Se o do Aniche for o menor, também seguiremos para o próximo (atualDoAniche++).

```
Nota nota1 = notasDoAniche[atualDoAniche];
Nota nota2 = notasDoAlberto[atualDoAlberto];
if(nota1.getValor() < nota2.getValor()) {
    // mauricio
    resultado[0] = nota2;
    atualDoAniche++;
} else {
    // alberto
    resultado[0] = nota2;
    atualDoAlberto++;
}
```

Depois, voltamos para o laço e comparamos novamente os elementos para identificar qual é o menor elemento. Quando descobrimos qual é o menor, o que fazemos com ele? Colocamos na posição 0? Não. Vamos colocando os itens um pouco mais à frente

no array.

Esta posição onde coloco o elemento tem de ir se deslocando. Logo, vamos precisar de uma posição (int atual) que comece no 0. Vamos incluir o atual no resultado também. Tanto faz se é o Aniche ou o Alberto, em ambos usaremos atual++ . Vamos sempre passar para a próxima casa.

```
int atual = 0;

while(atualDoAniche < notasDoAniche.length &&
      atualDoAlberto < notasDoAlberto.length) {

    Nota nota1 = notasDoAniche[atualDoAniche];
    Nota nota2 = notasDoAlberto[atualDoAlberto];
    if(nota1.getValor() < nota2.getValor()) {
        // mauricio
        resultado[atual] = nota2;
        atualDoAniche++;
    } else {
        // alberto
        resultado[atual] = nota2;
        atualDoAlberto++;
    }
    atual++;
}
```

Fizemos o nosso código, a parte inicial do junta() . O que ele fará? Ele vai criar um array que caiba todos os elementos. Depois, vamos analisar o primeiro de cada grupo e identificar qual é o menor dos dois, para então colocá-lo na primeira posição do resultado.

Temos duas notas para comparar? Sim. Após compará-las e descobrir qual é a menor, vamos colocá-la na posição adequada. Em seguida, avançamos para o próximo. Repetimos o processo até analisarmos o total de elemento. Quando acabar, devolveremos para o resultado.

Aparentemente, tudo está pronto. Vamos testar o código?

Rodando-o, o programa deve imprimir o nome de todos os alunos.
Clicamos em *Run As*, e depois em *Java Application*

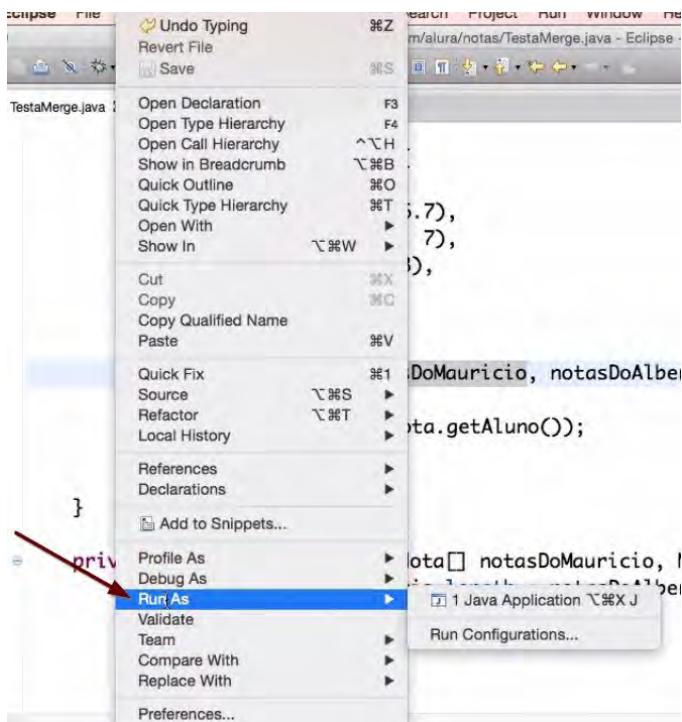


Figura 7.24: Problema de intercalar os dados — Run As

O resultado no console será:

```
jonas
andre
mariana
juliana
guilherme
carlos
paulo
```

No resultado, além de faltar a Ana, apareceu uma mensagem de erro: `NullPointerException`. Nós quase acertamos, mas ficou faltando algum detalhe.

Vamos descobrir qual é este detalhe, em seguida. Por que a Ana não apareceu?

7.7 PROCURANDO UM ERRO EM UM ALGORITMO

Nós vimos que, ao rodar o nosso programa, aconteceu algum erro no fim. Ele imprimiu no resultado sete alunos, porém deixou de fora dois. Quais ficaram faltando?

Vamos conferir detalhadamente qual informação estamos imprimindo? Não nos limitaremos aos nomes (`nota.getAluno`), mas também ao valor das notas (`nota.getValor`).

```
Nota[] rank = junta(notasDoAniche, notasDoAlberto);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() + " " + nota.getValor());
}
```

Vamos rodar de novo o programa e ele vai imprimir novamente a lista de alunos do Jonas até o Paulo e suas notas.

- jonas 3.0
- andre 4.0
- mariana 5.0
- juliana 6.7
- guilherme 7.0
- carlos 8.5
- paulo 9.0

Quais alunos ficaram faltando no resultado? Do primeiro array não faltou nenhum elemento. Porém, do segundo, faltaram a Lúcia e a Ana. Por que será que elas não apareceram?

Vamos tentar imprimir quem passou pelo processo de comparação. Para cada uma das comparações, imprimiremos os resultados com o texto:

```
System.out.println("Estou comparando " + nota1.getAluno() + " com "
+ nota2.getAluno());
```

Vamos voltar a rodar o código com a nova linha e ver o que acontece?

```
while(atualDoAniche < notasDoAniche.length &&
      atualDoAlberto < notasDoAlberto.length) {

    Nota nota1 = notasDoAniche[atualDoAniche];
    Nota nota2 = notasDoAlberto[atualDoAlberto];
    System.out.println("Estou comparando " + nota1.getAluno() + " com "
com " + nota2.getAluno());

    if(nota1.getValor() < nota2.getValor()) {
        // mauricio
        resultado[atual] = nota2;
        atualDoAniche++;
    } else {
        // alberto
        resultado[atual] = nota2;
        atualDoAlberto++;
    }
    atual++;
}
```

O programa vai imprimir diversas comparações:

```
Estou comparando andre com jonas
Estou comparando andre com juliana
Estou comparando mariana com juliana
Estou comparando carlos com juliana
Estou comparando carlos com guilherme
Estou comparando carlos com lucia
Estou comparando paulo com lucia
jonas 3.0
andre 4.0
marianna 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
```

Ele comparou o Paulo com a Lúcia, e depois interrompeu o processo. O Paulo está no primeiro array, enquanto a Lúcia está no

segundo. Ele comparou as duas notas e identificou que a do Paulo era a menor. Se ele escolheu o Paulo, ele aumentou +1 no `atualDoAniche`. Com isto, acabaram os elementos do array do Aniche, e assim ele interrompeu as comparações.

Vamos ver se foi isto que realmente aconteceu? Incluiremos um `System.out.println` no fim do código.

```
System.out.println("Estou saindo");
```

O resultado foi:

```
Estou comparando andre com jonas
Estou comparando andre com juliana
Estou comparando mariana com juliana
Estou comparando carlos com juliana
Estou comparando carlos com guilherme
Estou comparando carlos com lucia
Estou comparando paulo com lucia
Estou saindo
jonas 3.0
andre 4.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
```

Por que ele saiu? Porque estas duas condições do `while` precisavam ser verdadeiras.

```
while(atualDoAniche < notasDoAniche.length &&
      atualDoAlberto < notasDoAlberto.length) {
}
```

Vamos imprimir cada uma delas e conferir se as duas condições são verdadeiras? A primeira será:

```
System.out.println("Estou saindo");
System.out.println(atualDoAniche.length);
```

Para a segunda, faremos a mesma coisa para o Alberto:

```
System.out.println("Estou saindo");
System.out.println(atualDoAniche.length);
System.out.println(atualDoAlberto.length);
```

Isto significa que estamos perguntando: "sobrou alguém no Aniche?" e "sobrou alguém no Alberto?".

Vamos rodar e o resultado será:

```
Estou comparando andre com jonas
Estou comparando andre com juliana
Estou comparando mariana com juliana
Estou comparando carlos com juliana
Estou comparando carlos com guilherme
Estou comparando carlos com lucia
Estou comparando paulo com lucia
Estou saindo
false
true
jonas 3.0
andre 4.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
```

O `false` indica que não sobrou ninguém no Aniche. Já o `true` nos diz que sobraram alunos no Alberto.

Qual foi o nosso problema? Qual foi o *bug*? Nós intercalamos todos os elementos, um por um. Quando chegamos ao fim de um dos arrays, ainda existiam elementos no outro. Com esta condição, pode ter sobrado diversos elementos em cada um dos grupos. Mas nós precisamos incluir estas pessoas, afinal, se só restaram elementos do grupo do Aniche, vamos colocá-los no array. Se sobraram alunos do Alberto, vamos incluí-los lá também. Em breve, é isso o que vamos fazer.

7.8 INTERCALANDO OS ELEMENTOS QUE SOBRARAM

Chegou a hora de aproveitarmos o que sobrou do nosso array. Isto é: nós verificamos tanto em um array como em outro. De repente, acabaram os elementos de um deles, enquanto do outro acabou e alguns itens sobraram. Precisaremos usar esses elementos que restaram.

Se sobrou como no caso em que `atualDoAlberto < notasDoAlberto.length`, o que teremos de fazer? Colocaremos todos que sobraram da lista do Alberto, no array de resultados.

Isto significa que, enquanto (`while`) a condição for verdadeira, vamos inserir o atual no array de resultado.

```
System.out.println("Estou saindo");
System.out.println(atualDoAniche < notasDoAniche.length);
while(atualDOAlberto < notasDoAlberto) {
    resultado[atual] = notasDoAlberto[atualDoAlberto];
}
```

Depois que colocamos o elemento no array de resultado, e andamos com a variável para a direita (`atualDoAlberto++`), vamos colocar isto no código:

```
while(atualDoAlberto < notasDoAlberto) {
    resultado[atual] = notasDoAlberto[atualDoAlberto];
    atual++;
    atualDoAlberto++;
}
```

Enquanto sobrar elementos, vamos seguir copiando os elementos. Vamos rodar a aplicação e ela mostrará o seguinte resultado:

```
Estou saindo
false
- jonas 3.0
- andre 4.0
- mariana 5.0
- juliana 6.7
- guilherme 7.0
```

- carlos 8.5
- paulo 9.0
- lucia 9.3
- ana 10.0

O que faltava era verificarmos se estava sobrando alguém nos arrays. Ou seja, entender qual era o *bug*. Analise o código novamente e tente descobrir qual é o *bug* que ainda precisamos corrigir.

7.9 INTERCALANDO OS ELEMENTOS QUE SOBRARAM, INDEPENDENTE DO LADO

Nós ainda temos um *bug* no nosso código. Ele funciona bem quando sobra algum elemento no Alberto.

```
Nota[] rank = junta(notasDoAlberto, notasDoAniche);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() + " " + nota.getValor());
}
```

E o que acontece se chamarmos a nossa função, com outros valores? Por exemplo, o que acontecerá se trocarmos a posição dos valores na linha? Se dissermos que agora os alunos do Alberto são do Aniche, ou vice-versa.

```
Nota[] rank = junta(notasDoAniche, notasDoAlberto);
```

Vamos testar rodar novamente? Se você mudar a ordem, observe o que acontecerá:

```
Estou saindo
true
- jonas 3.0
- andre 4.0
- mariana 5.0
- juliana 6.7
- guilherme 7.0
- carlos 8.5
- paulo 9.0
```

Voltaremos a ter o mesmo problema: dois alunos ficarão de fora do resultado. Se sobram alunos, é porque a notas deles são maiores do que as do outro array. Precisamos inseri-los no array geral. Por isso, o mesmo `while` que criamos para o `notasDoAlberto`, teremos de fazer para o `notasDoAniche`.

```
System.out.println("Estou saindo");
while(atualDoAlberto < notasDoAlberto.length) {
    resultado[atual] = notasDoAniche[atualDoAniche];
    atualDoAniche++;
    atual++;
}
while(atualDoAlberto < notasDoAlberto.length) {
    resultado[atual] = notasDoAlberto[atualDoAlberto];
    atual++;
    atualDoAlberto++;
}
```

Testaremos novamente o programa e o resultado estará correto.

```
Estou saindo
true
- jonas 3.0
- andre 4.0
- mariana 5.0
- juliana 6.7
- guilherme 7.0
- carlos 8.5
- paulo 9.0
- lucia 9.3
- ana 10.0
```

Não importa mais se sobraram alunos em algum dos arrays. Independente se sobrou no primeiro ou no segundo, caso tenha restado alguém, ele será inserido no resultado.

7.10 PEQUENAS REFATORAÇÕES POSSÍVEIS

Chegou o momento de melhorarmos um pouco o nosso código. Ficaram alguns detalhes sobrando.

```
System.out.println("Estou saindo");
while(atualDoAlberto < notasDoAlberto.length) {
```

```

        resultado[atual] = notasDoAniche[atualDoAniche];
        atualDoAniche++;
        atual++;
    }
    while(atualDoAlberto < notasDoAlberto.length) {
        resultado[atual] = notasDoAlberto[atualDoAlberto];
        atual++;
        atualDoAlberto++;
    }
}

```

Por exemplo, a linha:

```
System.out.println("Estou saindo");
```

Nós já podemos removê-la, porque sabemos que o código funciona bem. Podemos fazer alterações também no `if`:

```

Nota nota1 = notasDoAniche[atualDoAniche];
Nota nota2 = notasDoAlberto[atualDoAlberto];
System.out.println("Estou comprando " + nota1.getAluno() + " com" + nota2.getAluno());

if(nota1.getValor() < nota2.getValor()) {
    // mauricio
    resultado[0] = nota2;
    atualDoAniche++;
} else {
    // alberto
    resultado[0] = nota2;
    atualDoAlberto++;
}
atual++;

```

Também podemos remover os comentários `// mauricio` e `// alberto`.

```

if(nota1.getValor() < nota2.getValor()) {
    resultado[0] = nota2;
    atualDoAniche++;
} else {
    resultado[0] = nota2;
    atualDoAlberto++;
}

```

Observe este trecho:

```
while(atualDoAlberto < notasDoAlberto.length) {
```

```
    resultado[atual] = notasDoAniche[atualDoAniche];
    atualDoAniche++;
    atual++;
}
```

O resultado na posição `atual` é o `notasDoAniche` na posição `atualDoAniche`. Depois somamos +1 no `atualDoAniche` e no `atual`. Temos a opção de escrever tudo isto em uma única linha.

```
while(atualDoAlberto < notasDoAlberto.length) {
    resultado[atual++] = notasDoAniche[atualDoAniche++];
}
```

E removeremos as duas linhas finais. Faremos as mesmas alterações com o Alberto:

```
while(atualDoAlberto < notasDoAlberto.length) {
    resultado[atual++] = notasDoAlberto[atualDoAlberto++];
}
```

As duas formas de escrever estão corretas. Porém, deixarei da maneira como estava antes, porque acredito que o código ficará mais legível de outra forma.

Não será um *Enter* a mais que deixará o código mais difícil de ser mantido. O compilador otimiza este tipo de tarefa, logo, não precisamos nos preocupar com isto. Deixo o computador se responsabilizar.

Mas o nosso código está claro especificando: primeiro será copiada as notas do Aniche, e depois somaremos +1 no `atualDoAniche` e no `atual`. Ficou claro e separado cada passo do processo.

```
while(atualDoAlberto < notasDoAlberto.length) {
    resultado[atual] = notasDoAniche[atualDoAniche];
    atualDoAniche++;
    atual++;
}
```

Escolho deixar desta maneira, mas sabemos que é possível

escrever de outra forma o código.

Continuamos procurando o que podemos melhorar. Temos um `System.out` que também removeremos.

```
while(atualDoAniche < notasDoAniche.length &&
      atualDoAlberto < notasDoAlberto.length) {

    Nota nota1 = notasDoAniche[atualDoAniche];
    Nota nota2 = notasDoAlberto[atualDoAlberto];
    System.out.println("Estou comprando " + nota1.getAluno() + " com" +
        nota2.getAluno());
```

Podemos remover o `System.out`, porque sabemos que o código está funcionando bem.

Agora temos outra questão: precisamos especificar que vamos juntar as notas do Aniche ou do Alberto?

```
Nota[] rank = junta(notasDoAlberto, notasDoAniche);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() + " " + nota.getValor());
}
```

Não importa se as notas são do Aniche, do Alberto, do Paulo ou do Adriano. O importante é que temos o primeiro e o segundo array de notas.

```
private static Nota[] junta(Nota[] notasDoAniche, Nota[] notasDoAlberto) {
    int total = notasDoAniche.length + notasDoAlberto.length;
    Nota[] resultado = new Nota[total];
}
```

Então, vamos substituir `notasDoAniche` e `notasDoAlberto` por `notas1` e `notas2`, respectivamente.

```
private static Nota[] junta(Nota[] notas1, Nota[] notas2) {
    int total = notas1.length + notas2.length;
    Nota[] resultado = new Nota[total];
```

```
}
```

Observe que usar um número para distinguir, como nós fizemos com `notas1` e `notas2`, não é o melhor padrão. Isso porque pode ficar difícil identificar a quem eles se referem. Porém, no nosso exemplo, temos dois grupos e queremos unir os elementos em um único array de notas. Então, a alteração nos nomes faz sentido.

Faremos o mesmo em outros trechos do código:

```
private static Nota[] junta[] notas1, Nota[] notas2) {  
    int total = notas1.length + notas2.length;  
    Nota[] resultado = new Nota[total];  
  
    int atual1 = 0;  
    int atual2 = 0;  
    int atual = 0;  
  
    while(atual1 < notas1.length &&  
          atual2 < notas2.length) {  
  
        Nota nota1 = notas1[atual1];  
        Nota nota2 = notas2[atual2];  
  
        if(nota1.getValor() < nota2.getValor()) {  
            resultado[atual] = nota1;  
            atual1++;  
        } else {  
            resultado[atual] = nota2;  
            atual2++;  
        }  
    }  
}
```

Agora que estamos usando o número 1 e 2 para distinguir os elementos, você pode dizer que o código ficou confuso. Existe alguma outra maneira para renomearmos as variáveis? Não queremos escrever `notasDoAniche`, porque só faria referência ao Aniche, e nós vamos receber as notas de qualquer pessoa.

Logo, temos nossas variáveis e quero juntá-las em um único array, que será o resultado. Nossa código já diz isto. Poderíamos juntar o `++` das variáveis no `while` em uma única linha. Mas

optei em deixar de outra maneira.

Também sabemos que inverter `notasDoAlberto` e `notasDoAniche` não vai interferir no resultado.

```
Nota[] rank = junta(notasDoAniche, notasDoAlberto);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() + " " + nota.getValor());
```

A nossa função que junta os arrays, além de unir os elementos, intercala os valores que estão dentro da lista de uma maneira ordenada. Então, alteraremos o nome da função de `junta()` para `intercala()`.

```
Nota[] rank = intercala(notasDoAniche, notasDoAlberto);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() + " " + nota.getValor());
}
```

A função `intercala` os elementos de uma maneira ordenada, considerando que eles já estavam organizados em cada um dos arrays.

Após renomear as variáveis e funções, vamos verificar se o código continua funcionando? Ao rodarmos novamente, temos o seguinte resultado:

```
- jonas 3.0
- andre 4.0
- mariana 5.0
- juliana 6.7
- guilherme 7.0
- carlos 8.5
- paulo 9.0
- lucia 9.3
- ana 10.0
```

O código roda corretamente, agora com os nomes adequados para a função e para as variáveis que representam o que realmente são.

7.11 O PRÓXIMO DESAFIO: INTERCALANDO EM UM ÚNICO ARRAY

Fomos capazes de intercalar dois arrays já ordenados. Se dividirmos as cartas de baralho — o dinheirinho falso, notas dos alunos ou outros elementos — entre diversas pessoas e cada uma ordenar uma parte do total, intercalar é uma tarefa mais simples. É assim quando temos vários arrays.

Mas o que acontece muitas vezes? Um professor chega e deixa uma pilha de provas. Depois outro professor coloca em cima uma nova pilha. Ou seja, as provas vão sendo amontoadas uma nas outras. Desta forma, não teremos dois array separados e organizados, um do Aniche e outro do Alberto. Na verdade, teremos uma única pilha, um só array, formada por dois grupos ordenados separadamente.



Figura 7.25: O problema de intercalar dados

Não teremos um array de tamanho 4 e outro de tamanho 5, para depois criarmos uma lista de tamanho 9. O que costuma acontecer é que temos um array de tamanho 9, em que os quatro primeiros elementos (que seguem até o "miolo") estão ordenados do menor para o maior, da mesma forma estão ordenados os próximos cinco itens. Começamos do 0 e terminamos no 9. O inicial é 0 e o término é 9, enquanto o miolo é o 4.

Então, na prática, o que acontece com frequência é que recebemos um *array (e não dois), com duas partes ordenadas. Nossa objetivo é intercalar estes dois pedaços. Alguém nos diz: "aqui estão o meu monte de cartas e o seu. Agora encontre uma maneira

de intercalar todos os itens". Nós fazemos um monte único e mandamos intercalar.



Figura 7.26: O problema de intercalar dados

Logo, tudo o que fizemos até agora com dois arrays, teremos de fazer algumas alterações no nosso código para trabalharmos com uma lista única.

CAPÍTULO 8

INTERCALANDO BASEADO EM UM ÚNICO ARRAY

Voltando ao nosso algoritmo de intercalar dois arrays, nós sabemos que recebemos um único array. O total de elementos se refere ao `termíno`.



Figura 8.1: Simulando a intercalação com um array — passo 1

O primeiro elemento é o 0. Então, como nós sabemos qual é a primeira parte e qual é a segunda? Com a variável `miolo`.

Na prática, o que temos agora é algo muito parecido com o que fizemos antes. Temos o `atualAniche` que começa com o valor do `inicial`, e o `atualAlberto` que começa com o valor do `miolo`. O valor do `termino` é 9, o tamanho do array. A variável `atual` também como era antes, começa com 0.

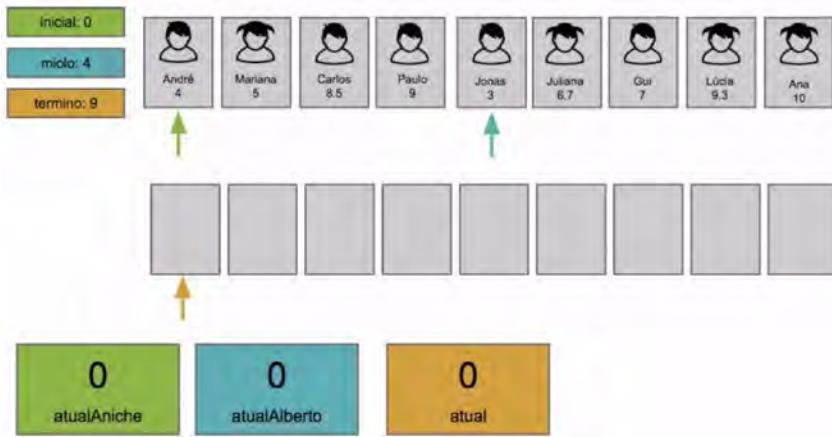


Figura 8.2: Simulando a intercalação com um array — passo 2

A melhor maneira para começarmos é pelo `atualAlberto`, que inicia no `miolo`. Isto significa que começaremos pelo valor 4. Vamos rodar o nosso algoritmo.

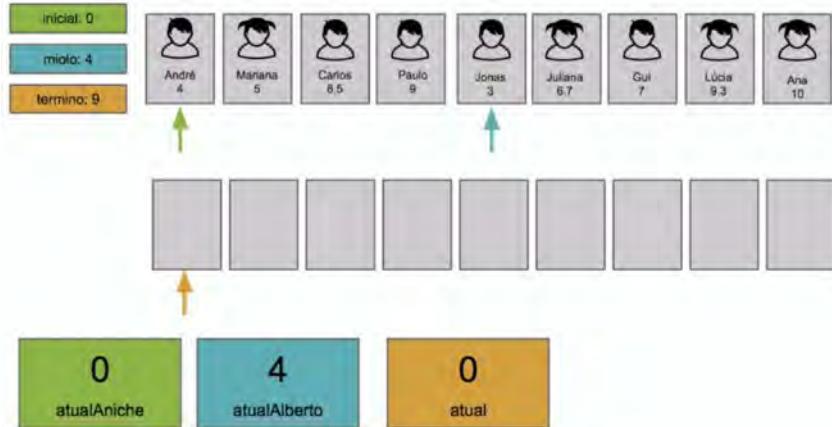


Figura 8.3: Simulando a intercalação com um array — passo 3

Compararemos as notas do André, que está no `atualAniche`, e do Jonas, que está no `atualAlberto`. Qual dos dois é o menor? Você perceberá que é o mesmo algoritmo utilizado antes!

O menor elemento é o Jonas, então vamos movê-lo para o array geral, somamos +1 nas variáveis `atualAlberto` e `atual`.

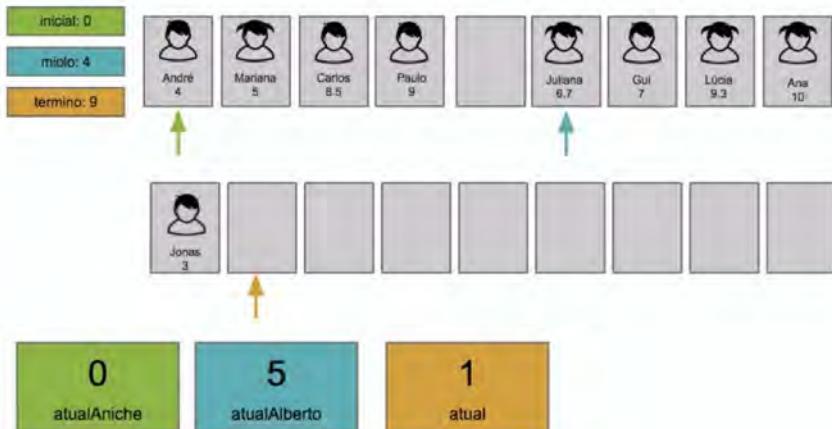


Figura 8.4: Simulando a intercalação com um array — passo 4

Vamos comparar agora o André com a Juliana. Qual é o menor? É o André, por isso vamos movê-lo para o novo array. Em seguida, somaremos +1 no `atualAniche` e no `atual`.

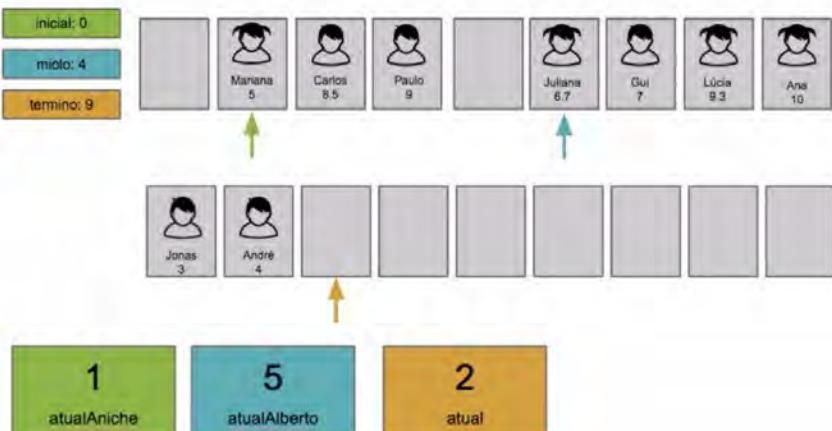


Figura 8.5: Simulando a intercalação com um array — passo 5

Continuaremos o algoritmo como fizemos anteriormente.

Observe o que foi feito: decidimos que o `atualAlberto` começaria pelo `miolo`. Nossa função de ordenação tem de saber agora onde ficará o `inicial`, o `miolo` e o `termino`.

O `atualAniche` vai começar do 0, como já era feito antes. O `atualAlberto` começará pelo `miolo`. Enquanto o `termino` é o tamanho do array. O algoritmo continua o mesmo. A única diferença é que vamos iniciar o `atualAlberto` pelo `miolo`.

Então, em vez de recebermos dois arrays, nós receberemos um único array. Nós seremos capazes de intercalar as duas partes: a primeira que vai de 0 até o `miolo`, e a segunda segue do `miolo` até o fim.

8.1 IMPLEMENTANDO O ÚNICO ARRAY

Nós vimos que, na prática, precisamos suportar uma maneira de intercalar os elementos, mesmo que eles não venham em dois arrays distintos. É possível que eles venham em um único array, em que a primeira e a segunda parte foram ordenadas do menor para o maior. Nós intercalaremos estas partes. Vamos implementar isto?

Vamos criar uma nova classe, chamada de `TestaIntercalaEmUmArray`, que colocará o método `main` para criar todas as notas que havíamos feito anteriormente.

```
public static void main(String[] args) {  
    Nota[] notasDoAniche = {  
        new Nota("andre", 4)  
        new Nota("mariana", 5),  
        new Nota("carlos", 8.5),  
        new Nota("paulo", 9)  
    };  
  
    Nota[] notasDoAlberto = {  
        new Nota("jonas", 3),  
        new Nota("juliana", 6.7),  
        new Nota("guilherme", 7),  
        new Nota("lucia", 9.3),  
    };
```

```
        new Nota("ana", 10)  
    };  
}
```

Porém, em vez de trabalharmos com duas variáveis, teremos apenas uma: primeiro a do Mauricio, depois a do Alberto. Vamos substituir `notasDoAniche` por `notas`.

```
public static void main(String[] args) {  
    Nota[] notas = {  
        new Nota("andre", 4)  
        new Nota("mariana", 5),  
        new Nota("carlos", 8.5),  
        new Nota("paulo", 9)  
        new Nota("jonas", 3),  
        new Nota("juliana", 6.7),  
        new Nota("guilherme", 7),  
        new Nota("lucia", 9.3),  
        new Nota("ana", 10)  
    };  
}
```

Temos as notas, agora nosso objetivo é intercalar todas elas. O código que vamos escrever ficará muito parecido com o `rank` da classe `TestaMerge`. Nós diremos para intercalar as notas, e depois imprimi-las.

```
Nota[] rank = intercala(notas);  
for(Nota nota : rank) {  
    System.out.println(nota.getAluno() + " " + nota.getValor());  
}
```

Falta ainda implementar a função `intercala()` que recebe o array único, em contraste ao que recebia dois arrays. Vou criar o método `intercala`, que receberá as notas.

Qual será o tamanho do array de resultado? Ele será do mesmo tamanho do `notas.length`. No entanto, nós não precisaremos ficar somando os dois, porque já estará tudo em um array. E o programa retornará o próprio `resultado`.

```
private static Nota[] intercala(Nota[] notas) {  
    Nota[] resultado = new Nota[notas.length];  
    return resultado;  
}
```

O que queremos fazer em seguida é a intercalação. Antes, nós tínhamos três variáveis para acompanhar os nossos passos. Uma era `atual`, onde colocaríamos o próximo elemento. O `atual1`, que era o acompanhamento do array da esquerda. E o `atual2`, que era o acompanhamento do array da direita. As duas variáveis começavam com 0.

```
private static Nota[] intercala(Nota[] notas) {  
    Nota[] resultado = new Nota[notas.length];  
    int atual = 0;  
    int atual1 = 0;  
    int atual2 = 0;  
    return resultado;  
}
```

As duas começavam por esta posição, porque estavam separados. No entanto, unimos todos os elementos em um único array, e o `atual2` vai começar a partir da posição 4 (o Jonas).

```
Nota[] notas = {  
    new Nota("andre", 4)  
    new Nota("mariana", 5),  
    new Nota("carlos", 8.5),  
    new Nota("paulo", 9)  
    new Nota("jonas", 3),  
    new Nota("juliana", 6.7),  
    new Nota("guilherme", 7),  
    new Nota("lucia", 9.3),  
    new Nota("ana", 10)  
};
```

Como conseguiremos que o `atual2` saiba que ela precisa começar pela posição 4? Quando criarmos o método `intercala`, precisamos determinar que ele intercale a partir da posição 4.

```
Nota[] rank = intercala(notas, 4);  
for(Nota nota : rank) {
```

```
        System.out.println(nota.getAluno() + " " + nota.getValor());
    }
```

A primeira parte, que segue do 0 até o 3, é referente ao lado da esquerda. Do 4 até o fim, será a parte da direita. Podemos deixar isto explícito: do 0 até o 4 exclusive será a parte da esquerda.

```
Nota[] rank = intercala(notas, 0, 4);
```

Do 4 em diante, o que significa até o `notas.length`, será a parte da direita.

```
Nota[] rank = intercala(notas, 0, 4, notas.length);
```

Assim deixamos bem explícito qual é a parte referente ao lado esquerdo e ao lado direito. Indicamos o `int inicial` na esquerda, o `int miolo`, que é o ponto que separa as partes, e o `int termino` na direita.

```
private static Nota[] intercala(Nota[] notas, int inicial, int mio
lo, int termino) {
    Nota[] resultado = new Nota[notas.length];
    int atual = 0;
    int atual1 = 0;
    int atual2 = 0;
    return resultado;
}
```

Os valores das variáveis serão: `atual` é 0, `atual1` é `inicial`, e `atual2 = miolo`.

```
private static Nota[] intercala(Nota[] notas, int inicial, int mio
lo, int termino) {
    Nota[] resultado = new Nota[notas.length];
    int atual = 0;
    int atual1 = inicial;
    int atual2 = miolo;
    return resultado;
}
```

Agora podemos trabalhar tanto com a parte da esquerda como a da direita, com o `atual1` e o `atual2`.

Em seguida, incluiremos o laço. Enquanto (`while`) o `atual1` for menor do que o tamanho do array da esquerda (o `miolo`) e o `atual2` for menor do que o `termino` , nós seguiremos andando com as variáveis.

```
private static Nota[] intercala(Nota[] notas, int inicial, int mio  
lo, int termino) {  
    Nota[] resultado = new Nota[notas.length];  
    int atual = 0;  
    int atual1 = inicial;  
    int atual2 = miolo;  
    while(atual1 < miolo &&  
        atual2 < termino) {  
  
    }  
    return resultado;  
}
```

Vamos verificar alguns dados. Se `miolo` é igual a 4, então o indicador da direita já deve começar nesta posição. Ele inicia com o Jonas. O indicador da esquerda começa pelo 0, que é o André.

Observe que, quando `atual1` for igual 4 e for referenciar o Jonas, ele não pode continuar. A condição é que o `atual1` seja menor do que o `miolo` , e não menor ou igual (`<=`). Nós já havíamos feito isso antes no `TestaMerge` . O `atual1` era menor do que o lado esquerdo (`notas1.length`).

```
while(atual1 < notas1.length &&  
      atual2 < notas2.length){  
}
```

O mesmo foi feito para o `atual2` . Então, determinaremos isto no `while` do `TestaIntercalaEmUmArray` . O `atual2` será menor do que o fim do lado direito (`termino`).

```
while(atual1 < miolo &&  
      atual2 < termino) {  
  
}
```

O laço vai passar por todos os elementos. O que mais precisamos fazer? Comparar as duas notas, a `notas1`, que é igual a `notas[atual1]`, e `notas2`, que é igual a `notas[atual2]`.

Vamos escrever o `if`: se a `nota1.getValor` for igual ao `notas2.getValor`, o `resultado[atual]` será igual a `nota1`. Se não (`else`), o `resultado[atual]` será igual a `nota2`, e incluímos as duas notas.

```
if(nota1.getValor() < valor2.getValor()) {  
    resultado[atual] = nota1;  
} else {  
    resultado[atual] = nota2;  
}
```

Independente do caso, somaremos +1 ao `atual`. Porém, na esquerda, vamos somar +1 no `atual1` e, na direita, somaremos +1 no `atual2`.

```
if(nota1.getValor() < valor2.getValor()) {  
    resultado[atual] = nota1;  
    atual1++;  
} else {  
    resultado[atual] = nota2;  
    atual2++;  
}  
atual++;
```

Sabemos que não basta comparar enquanto tivermos dois elementos, porque quando termina um dos arrays, sobrarão itens no outro. Precisaremos também dos dois `while`s no fim. Enquanto (`while`) o `atual1` for menor do que o `miolo`, precisamos que todos os elementos sejam copiados. Por isso, `resultado[atual]` será igual a `notas[atual1]`. Depois somaremos +1 no `atual1` e no `atual`.

```
while(atual1 < miolo) {  
    resultado[atual] = notas[atual1];  
    atual1++;  
    atual++;  
}
```

Faremos o mesmo para o `atual2`. O `while` vai indicar que, enquanto ele for menor do que o `termo`, o `resultado[atual]` será igual a `notas[atual2]`. E somaremos +1 no `atual2` e no `atual`.

```
while(atual2< termo) {  
    resultado[atual] = notas[atual2];  
    atual2++;  
    atual++;  
}
```

Nós traduzimos o método que recebia dois arrays, que estavam totalmente separados. Porém, sabemos que, na prática, os elementos virão dispostos em um array único. Ao observá-lo, reconhecemos onde existe uma quebra entre as partes. E precisamos ordenar de uma parte até a outra, intercalando os elementos.

Ao testarmos o código, o resultado será:

- jonas 3.0
- andre 4.0
- mariana 5.0
- juliana 6.7
- guilherme 7.0
- carlos 8.5
- paulo 9.0
- lucia 9.3
- ana 10.0

Os elementos do nosso array foram intercalados corretamente.

8.2 SIMULANDO O MÉTODO INTERCALA EM UM ÚNICO ARRAY

Agora temos uma função que intercala. Ela parece funcionar corretamente, mas vamos testá-la. Nós já a testamos com as variáveis `inicial` igual 0, `miolo` igual 4 e `termo` igual 9. Porém, o que acontecerá se quisermos manter o primeiro elemento, o André, no início da lista?

Imagine os seguintes exemplos: deixaremos a prova de um aluno no início da pilha, porque queremos conversar com ele. Ou queremos deixar a carta do *joker* (o coringa) no alto do monte de baralho. Então, não vamos começar a intercalação a partir da posição 0, mas da 1. Enquanto, `inicial` será igual a 1.

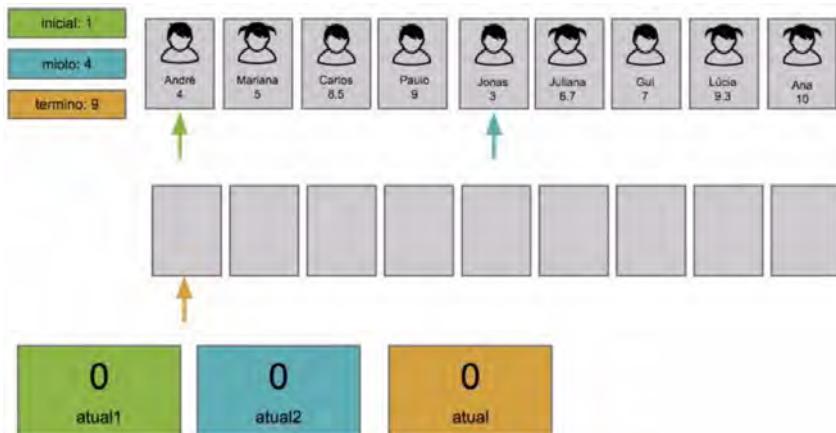


Figura 8.6: Simulando o intercala em um único array — passo 1

Iniciaremos com a Mariana. As variáveis terão os seguintes valores: `atual1` será igual a 1, `atual2` igual a 4, e `atual` igual a 0.

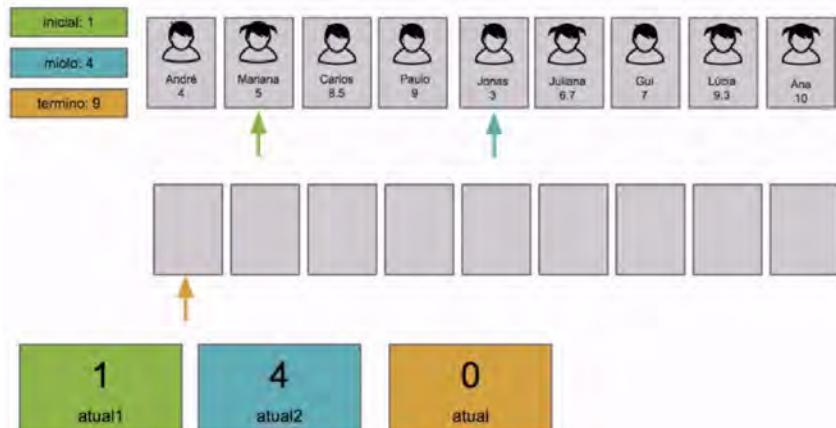


Figura 8.7: Simulando o intercala em um único array — passo 2

Vamos rodar o algoritmo e ver o que acontece.

Ao compararmos os primeiros elementos, identificamos que Jonas é o menor. Vamos movê-lo para o novo array. Também modificaremos as posições dos indicadores e os valores das variáveis. O `atual2` será igual a 5, e o `atual` igual a 1.

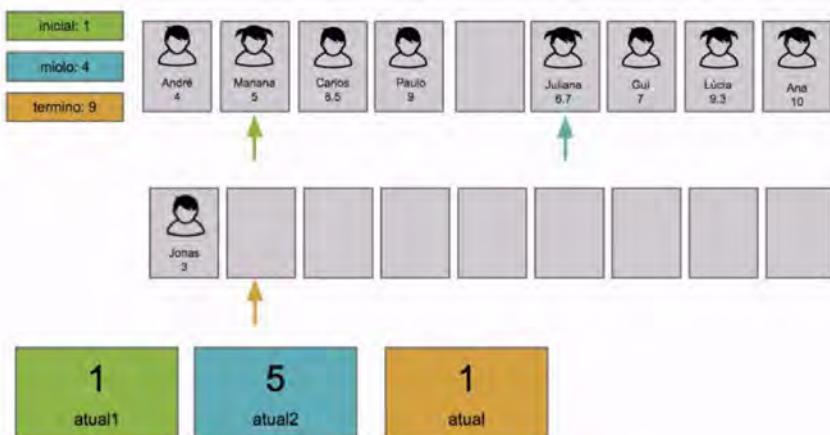


Figura 8.8: Simulando o intercala em um único array — passo 3

Entre Mariana e Juliana, qual é a menor? A Mariana. Vamos colocá-la no array. Alteraremos os valores das variáveis: `atual1` será igual a 2, e `atual` será 2.

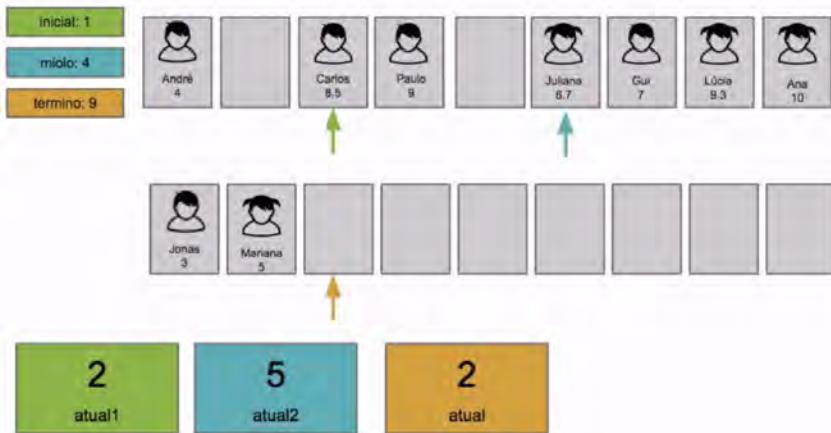


Figura 8.9: Simulando o intercala em um único array — passo 4

Comparando Carlos e Juliana, qual é o menor? A Juliana. É a vez dela de ser movida. Alteramos os valores das variáveis e `atual2` será igual a 6, enquanto `atual` será igual a 3.

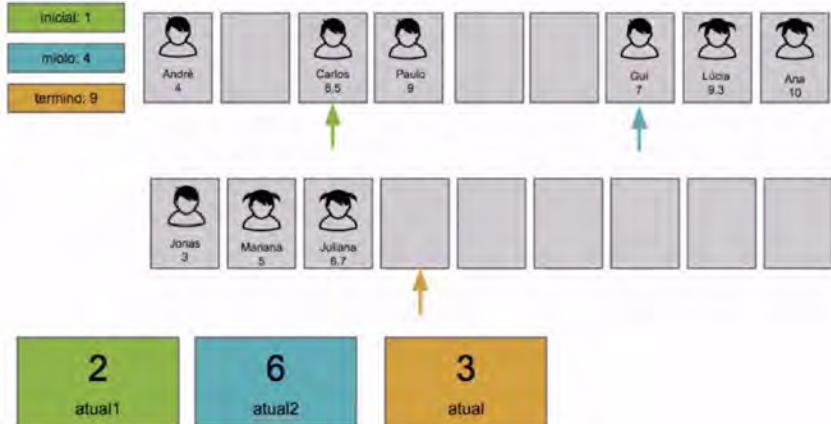


Figura 8.10: Simulando o intercala em um único array — passo 5

Compararemos o Carlos e o Gui. Qual deles é o menor? É o Gui. Vamos movê-lo para o novo array. Andamos com os indicadores e a variável `atual2` será igual a 7, enquanto `atual` será igual a 4.

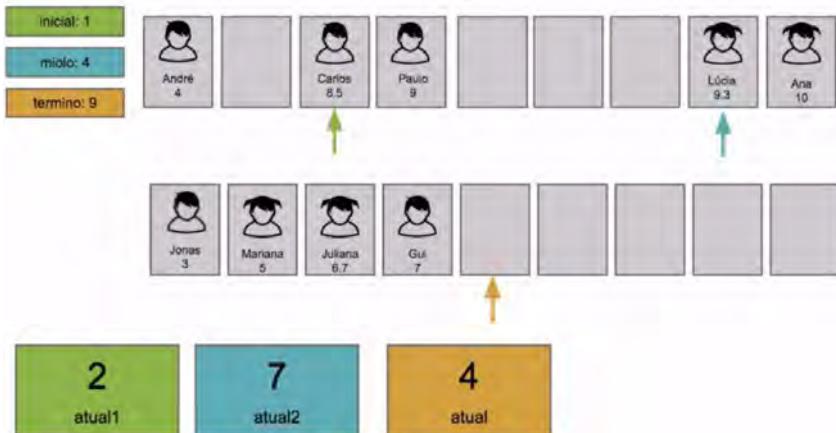


Figura 8.11: Simulando o intercala em um único array — passo 6

Entre Carlos e Lúcia, qual é o menor? O Carlos. Vamos movê-lo para o array. Seguimos alterando os valores das variáveis: `atual1` será igual a 3, e `atual` igual a 5.

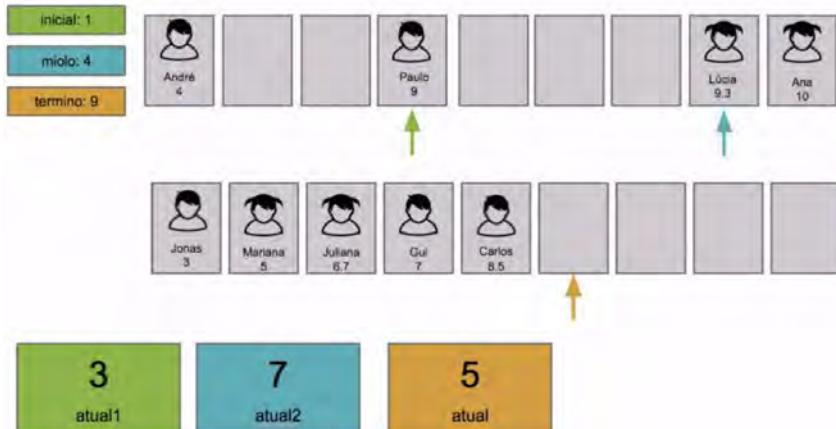


Figura 8.12: Simulando o intercala em um único array — passo 7

Entre Paulo e Lúcia, qual é o menor elemento? É o Paulo. Ele é o próximo a ser inserido no array. Vamos alterar a posição dos indicadores e os valores das variáveis: `atual3` será igual 4, e

atual igual a 6.

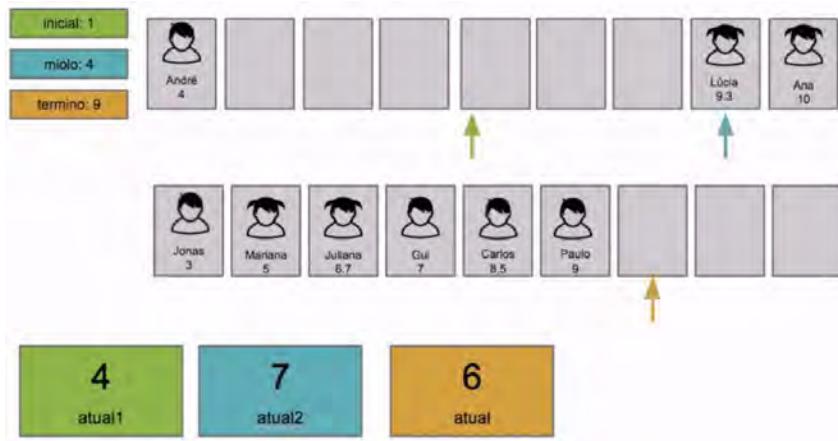


Figura 8.13: Simulando o intercala em um único array — passo 8

Nosso array da esquerda acabou, afinal, `atual1` é igual ao valor do `míolo`. O que devemos fazer? Apenas copiar os elementos que sobraram no outro array. Primeiro moveremos a Lúcia, assim como as posições dos indicadores também serão modificadas.

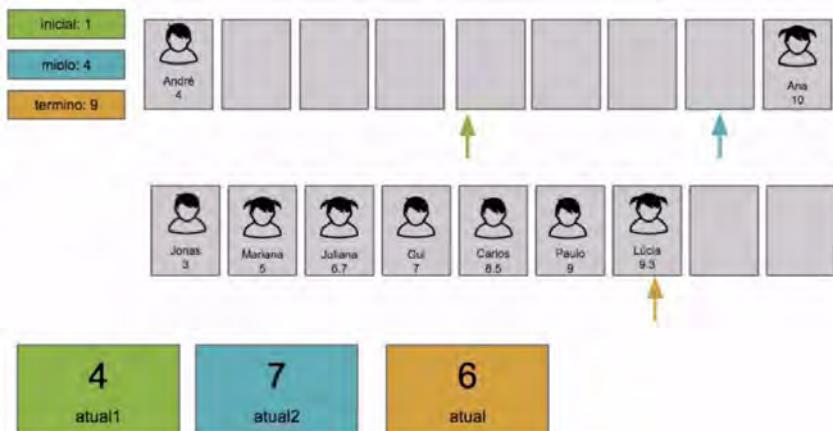


Figura 8.14: Simulando o intercala em um único array — passo 9

Depois, mudamos os valores das variáveis: `atual2` será igual a

8 e atual igual a 7. Ainda temos algum elemento para ser copiado? Sim, a Ana. Vamos movê-la para o novo array.

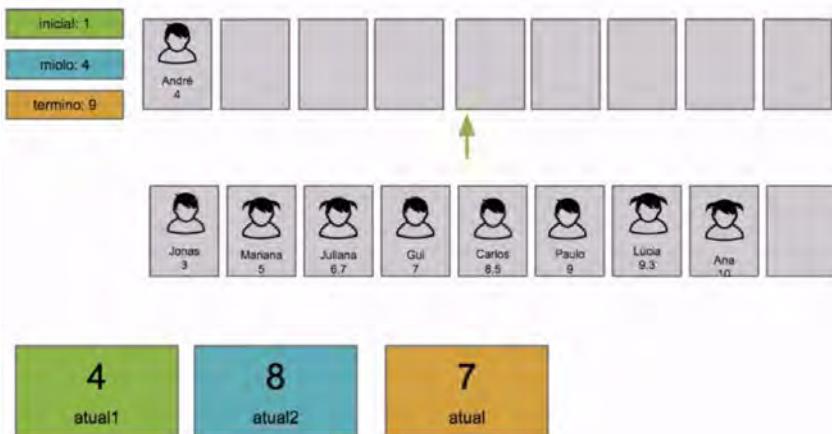


Figura 8.15: Simulando o intercala em um único array — passo 10

Vamos alterar o atual para 8 e o atual2 será igual a 9. A casa que sobra no novo array ficará vazia, porque um dos elementos do array da esquerda permaneceu lá. Assim, terminamos a ordenação.

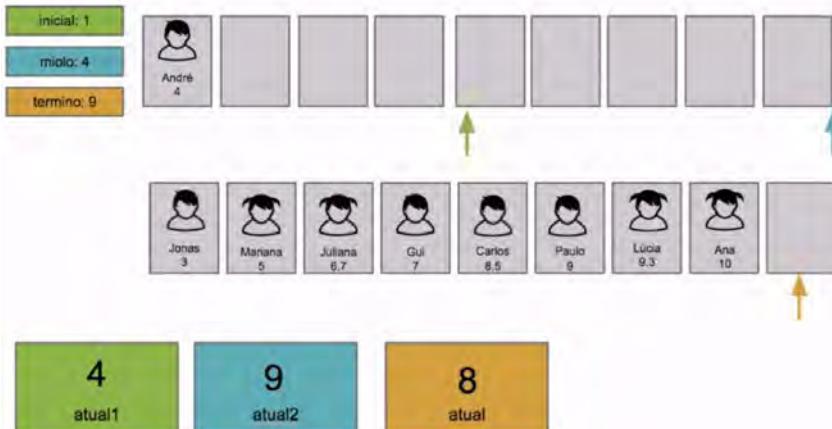


Figura 8.16: Simulando o intercala em um único array — passo 11

Retornamos o array, porém, ele não ficou exatamente como queríamos. O algoritmo não funcionou. Com `inicial` igual a 1, o resultado não está correto. Veremos o que fazer para que o algoritmo funcione.

8.3 O PROBLEMA DE INTERCALAR PARTE DE UM ARRAY

Quando terminei o algoritmo, com o `inicial` igual a 1, ele não funcionou bem. Por quê? Porque ele criou um array de tamanho 9, sendo que ele precisava apenas de espaço para 8. Ele nos devolveu esse array:

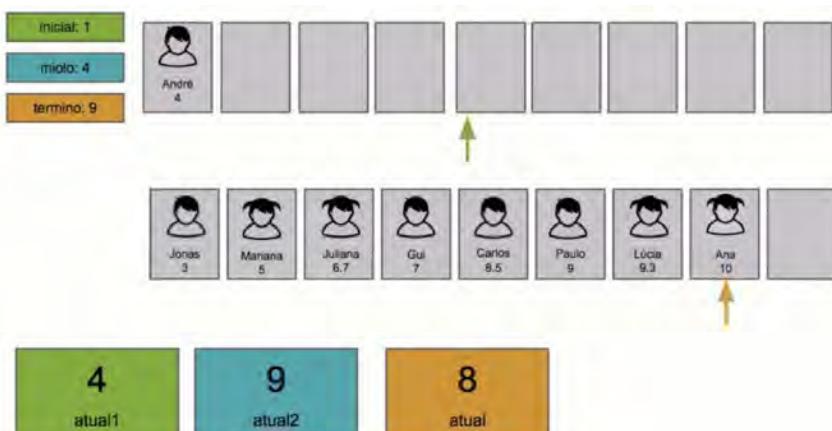


Figura 8.17: O problema de intercalar parte de um array — passo 1

O que nós queríamos é que o primeiro elemento do array da esquerda (o Jonas) permanecesse lá, e que os outros elementos estivessem dispostos no novo array*.

E se modificássemos também o valor do `termino`? Se ele tivesse de terminar antes e sobrasse algum elemento, ele também deveria permanecer na mesma posição. No final, o que restasse deveria, de fato, continuar.

Pedimos que a carta do coringa continuasse no monte e que a prova do aluno trapaceiro permanecesse no começo da pilha, e do melhor aluno, no fim. Queríamos que um determinado elemento permanecesse no início e que outro ficasse no fim. E o algoritmo nos devolveu um array que terminou com um espaço vazio. Isso significa que receberíamos a mensagem de erro *NullPointerException*.

Para resolver o problema, primeiramente temos de evitar criar coisas de que não precisamos. Se precisamos de 8 elementos, por que criamos um array para 9? Foi o primeiro erro cometido.

Antes, ele tinha um tamanho 9, porque nós tínhamos 9 elementos. Quantos elementos temos agora? Apenas 8. Este número surgiu do valor do `termo`, que é igual a 9, menos o `inicial`, que é igual a 1. É assim que sabemos quantos elementos temos no meio e vamos usar.

Então, no momento de criar um novo array, ele não será de tamanho 9, mas de 8. Será o suficiente e não teremos *NullPointerException*.

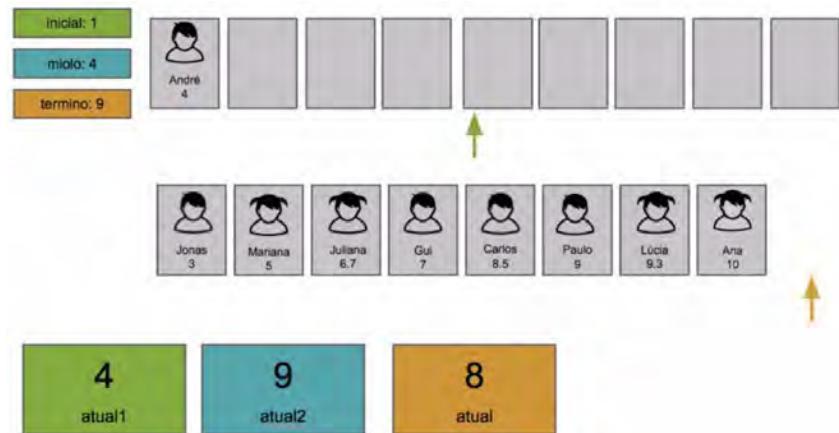


Figura 8.18: O problema de intercalar parte de um array — passo 2

Corrigimos o erro de colocar um elemento a mais no array, o que faremos agora? Vamos colocar os oito elementos de volta ao array em que estavam. E o nosso problema estará resolvido!

Para isto, vamos criar um novo `for` que passe por todo o array de resultado e que copie cada elemento para o array de onde saiu. Também precisaremos utilizar uma variável que passe por todos os elementos, que vai de 0 até 8.

Como fazemos para copiar os itens para o outro array? Podemos selecionar o elemento 0 e inserir na posição 0? Não vai funcionar. O melhor seria colocá-lo na posição seguinte à inicial, a 1. Então, moverei o elemento que está na posição 0 para a posição inicial + 1, que será a 1.

Vamos colocar o elemento na posição 2 na posição inicial + 2, que será a posição 2. Seguimos repetindo o mesmo processo com os demais elementos, até termos movimentado todos os itens para as posições adequadas do outro array. O nosso array original ficou corretamente ordenado.

O que precisamos fazer em seguida? Vamos remover os indicadores que não usaremos, e criaremos um array de tamanho `termo - inicial`. Após intercalar os elementos que fazem parte, basta copiá-los de volta para a origem. No entanto, para conseguirmos fazer isto, precisamos sempre adicionar a posição `inicial` para deslocá-los adequadamente.

8.4 COPIANDO PARTE DO ARRAY EM JAVA

Vamos tentar uma variação do `intercala`? Já que recebemos os parâmetros de `inicio`, o `miolo` e o `termino`, determinaremos que ele ignore o primeiro elemento e que este permaneça na mesma posição.

```
Nota[] rank = intercala(notas, 1, 4, notas.length);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() +" " + nota.getValor());
}
}
```

Vamos tentar rodar o programa? O programa nos mostrará as notas:

```
jonas 3.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
lucia 9.3
ana 10.0
```

```
Exception in thread "main" java.lang.NullPointerException
at br.com.alura.notas.TestaIntercalaEmUmArray
```

Vai aparecer uma mensagem de erro `NullPointerException`. O algoritmo quebrou o programa quando o mandamos começar a partir da posição da Mariana! Vamos verificar o nosso código e entender o que aconteceu?

```
Nota[] rank = intercala(notas, 1, 4, notas.length);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() +" " + nota.getValor());
}
```

Nós vimos que o `intercala` está recebendo os valores 1, 4 e 9. Qual será o tamanho do array que ele criará? Será o mesmo tamanho do array original, 9:

```
private static Nota[] notas, int inicial, int miolo, int termino)
{
    Nota[] resultado = new Nota[notas.length];
}
```

Logo, o `resultado` terá 9 casinhas.

Após executar o código completo, quantos elementos foram copiados? Faremos um `System.out.println` (depois dos *whiles*) e

veremos quanto vale o `atual` :

```
System.out.println(atual);

return resultado;
```

Ao rodarmos o programa, veremos que ele só copiou 8 elementos:

```
8
jonas 3.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
lucia 9.3
ana 10.0
```

```
Exception in thread "main" java.lang.NullPointerException
at br.com.alura.notas.TestaIntercalaEmUmArray
```

Por que ele não copiou todos os elementos? Porque pedimos para começar a partir do `inicial` :

```
int atual1 = inicial;
```

Então, ele ignorou o primeiro elemento e o deixou de fora. Nós ficamos com um array que tem, na verdade, um tamanho 9, mas que na realidade só tem 8 elementos, porque pedimos que ele começasse pelo 1. Observe que, quando formos criar o nosso array `Nota[notas.length]`, não queremos que ele tenha tamanho 9. É suficiente que ele seja tamanho 8. Podemos generalizar o tamanho para `termino - inicial`.

```
private static Nota[] intercala(Nota[] notas, int inicial, int mio
lo, int termino) {
    Nota[] resultado = new Nota[termino - inicial];
```

Isto já resolve parte do problema. O programa já vai imprimir os 8 elementos intercalados corretamente:

```
jonas 3.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
lucia 9.3
ana 10.0
```

No entanto, onde está o primeiro elemento que deveria estar no começo que nós pedimos que ele ignorasse? Por que ele ficou de fora? Porque nós sempre estamos devolvendo um array novo, e não é isto o que queremos. Queremos que ele devolva o mesmo array, com o primeiro elemento. Apenas precisamos que os outros elementos sejam intercalados.

Este é o nosso objetivo: intercalar os elementos que vêm depois do primeiro. Para conseguirmos fazer isto, temos de parar de retornar o mesmo resultado . Ele imprime apenas os 8 elementos que intercalamos. Ele tem os itens ordenados na ordem certa, que estão entre o inicial e o termino , e o resto permanece no notas .

O que vamos fazer? Vamos copiar os elementos do resultado e movê-los para o notas . Criaremos um for , no fim do código, que passará por cada um dos elementos que copiamos:

```
for(int contador = 0; contador < atual ; contador++) {  
}
```

Por que o contador é menor do que atual ? Pois atual é o número de elementos que nós intercalamos. Se intercalamos 8 elementos, vamos querer que todos estejam no nosso array original.

Isto é: selecionaremos o primeiro elemento do resultado[contador] e vamos movê-lo para o notas , na posição que ele pertence(inicial + contador).

```
notas[inicial + contador] = resultado[contador];
```

Se o `contador` é igual a 0 e o `inicial` é igual a 1, então o elemento pertencerá à posição 1. Quando o `contador` for igual a 1 e o `inicial` for 1, ele pertencerá à posição 2. Será o mesmo até terminarmos a intercalação. No fim, apenas teremos de retornar as notas: `return notas;`.

```
for(int contador = 0; contador < atual ; contador++) {  
    notas[inicial + contador] = resultado[contador];  
}  
return notas;
```

Vamos testar se copiamos corretamente os elementos do `resultado` para dentro do `notas`. O programa imprimirá:

```
8  
andre 4.0  
jonas 3.0  
mariana 5.0  
juliana 6.7  
guilherme 7.0  
carlos 8.5  
paulo 9.0  
lucia 9.3  
ana 10.0
```

Ele respondeu que intercalou 8 elementos, deixou o primeiro no lugar e ordenou os seguintes.

Antes, quando nós devolvíamos o array como o valor do `inicial` modificado, o algoritmo não funcionava. A razão é que estávamos criando um array maior do que precisávamos. Depois, criamos um array do tamanho adequado, que seguia até o `termino` menos o `inicial`. É o tamanho que, de fato, gostaríamos de analisar.

Fizemos todo o processo, no fim o nosso `for` pedia para copiar de volta para o `notas` todos os elementos que foram intercalados. Ao rodarmos de novo o algoritmo, teremos o resultado correto.

Agora podemos remover a linha do `System.out` que informa a

quantidade de elementos intercalados:

```
System.out.println(atual);
```

E o nosso array estará ordenado:

```
- andre 4.0
- jonas 3.0
- mariana 5.0
- juliana 6.7
- guilherme 7.0
- carlos 8.5
- paulo 9.0
- lucia 9.3
- ana 10.0
```

Podemos mandar intercalar apenas o trecho que queremos. Estamos intercalando a partir do primeiro elemento, mas ainda temos a opção de intercalar todos os itens. Basta alterar a posição inicial .

```
Nota[] rank = intercala(notas, 0, 4, notas.length);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() + " " + nota.getValor());
}
```

Se testarmos, veremos que todos os elementos serão intercalados.

```
jonas 3.0
andre 4.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
lucia 9.3
ana 10.0
```

Podemos intercalar também todos os elementos, menos o último. Basta subtrair 1 do notas.length .

```
Nota[] rank = intercala(notas, 0, 4, notas.length-1);
for(Nota nota : rank) {
```

```
        System.out.println(nota.getAluno() + " " + nota.getValor());
    }
}
```

Se imprimirmos o resultado, a Ana continuará no fim da lista de elementos. Afinal, ela teve a maior pontuação.

```
jonas 3.0
andre 4.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
lucia 9.3
ana 10.0
```

Vamos testar se subtraíssemos 3 do `notas.length` :

```
jonas 3.0
andre 4.0
mariana 5.0
juliana 6.7
carlos 8.5
paulo 9.0
guilherme 7.0
lucia 9.3
ana 10.0
```

O Guilherme, a Ana e a Lúcia permaneceram na posição, enquanto os outros elementos foram intercalados.

Agora o nosso método `intercala()` intercala adequadamente o trecho determinado.

8.5 O PRÓXIMO DESAFIO: OUTRAS INVOCACÕES AO INTERCALA()

Se invocamos o `intercala()` com os dados pré-ordenados, `início`, `miolo` e `término` como vimos antes, tudo funciona adequadamente. Mas o que acontece se chamarmos o mesmo método com outros valores para `inicio`, `miolo` e `termino` ?

Vamos executar diversos testes de nosso método e entender melhor o que acontece nessas situações extremas.

CAPÍTULO 9

DIVERSAS SIMULAÇÕES DO INTERCALA()

Um outro teste interessante que podemos fazer é: em vez de brincarmos apenas com o `inicial`, vamos alterar também o valor da variáveis `miolo` e `termino`. Não vamos apenas ordenar dois trechos de um array, que já estão ordenados. Vamos complicar mais.

Nosso `inicial` será igual a 0, assim como o `miolo`, e o `termino` será igual a 1. Pedimos para o algoritmo analisar apenas um elemento. Será que funciona?

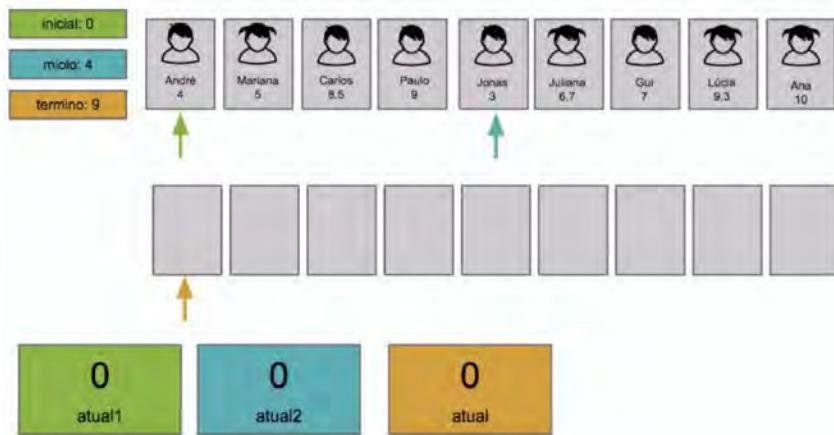


Figura 9.1: Simulando ainda mais o intercala — passo 1

Primeiramente, quantas casinhas serão criadas? Como `termo` menos `inicial` é igual 1, nosso array novo terá apenas **uma** casinha.

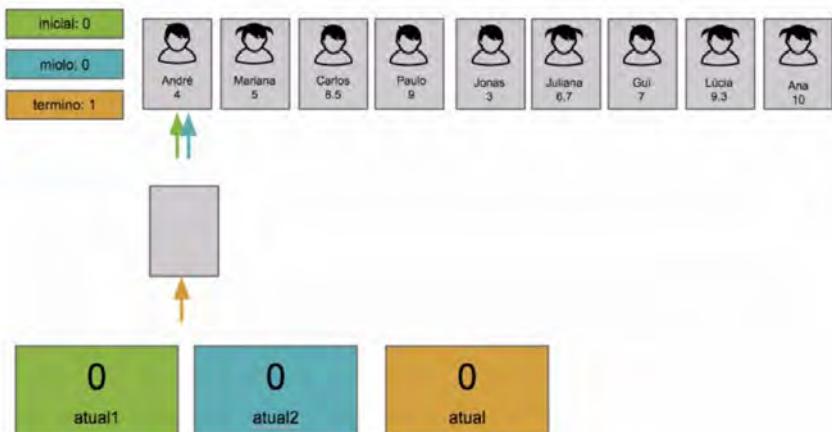


Figura 9.2: Simulando ainda mais o intercala — passo 2

A variável `atual1` será igual a 0, assim como `atual2` e `atual`. Então, temos de atender à condição de que continuaremos enquanto `atual1` for menor do que o `miolo`. Se não atende a primeira condição, a variável é igual ao `miolo`.

Depois seguimos para o segundo laço. Temos mais algum elemento para copiar na lista? Sim. Temos algo para copiar no `atual2`. Vamos mover o André para a nova lista.

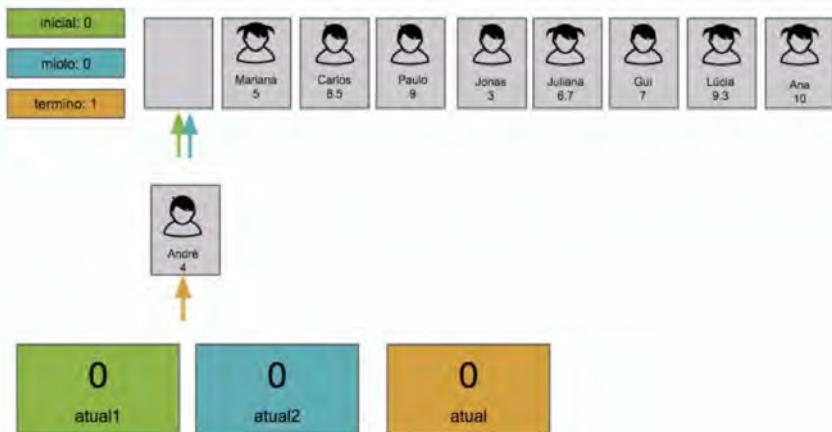


Figura 9.3: Simulando ainda mais o intercala — passo 3

Agora, precisamos copiar o que sobrou de volta para o array de origem.

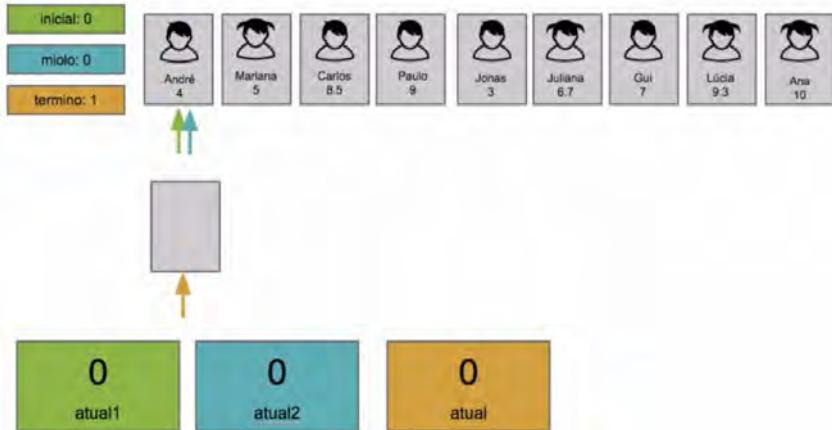


Figura 9.4: Simulando ainda mais o intercala — passo 4

O algoritmo funcionou. Se pedimos para o algoritmo não intercalar nenhum outro elemento, ele funcionará corretamente.

É importante saber que o algoritmo funciona bem nos extremos, com valores beirando o limite do que é válido, para verificarmos que

ele funciona com todos os valores válidos. Os erros mais comuns nas implementações de algoritmos ocorrem nesses extremos, quando uma variável inteira é -1, 0, 1, ou o valor máximo, máximo -1, máximo +1. Lembre-se sempre de testar bastante seu algoritmo.

Vimos que o algoritmo estará correto se passarmos os valores válidos. O primeiro trecho do array tem tamanho 0, porque o `miolo` vai até 0. O segundo trecho do array tem tamanho de 0 até 1. Com estes dados, nosso algoritmo funciona bem? Sim. Ele envia o elemento para o array temporário e move-o novamente para o array original. Tudo funcionou corretamente.

9.1 TAMANHOS VÁLIDOS PARA O INTERCALA

Em vez de pegarmos o array inteiro, ou ignorarmos algum item do começo ou do fim, o que aconteceria se criássemos um grupo pequeno com apenas dois elementos? Será que funcionaria? Vamos testar.

Se selecionarmos dois elementos. O primeiro trecho será da posição 0 e o `miolo` 1, e o segundo trecho será do 1 até o `termino` 2.

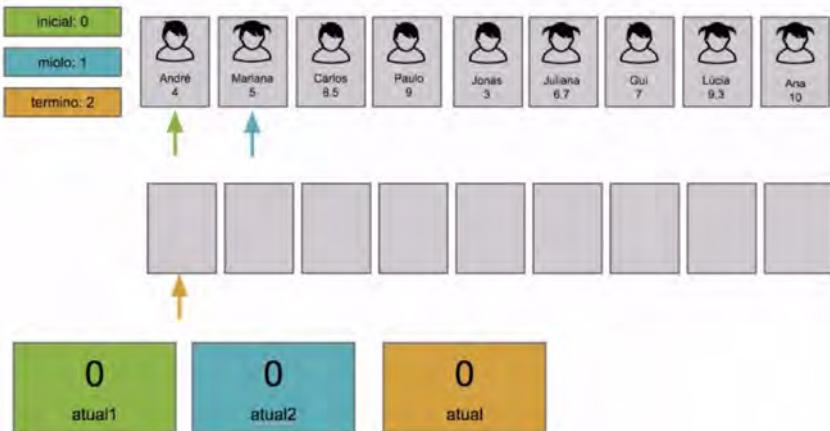


Figura 9.5: Tamanhos válidos para o intercala — passo 1

Qual será o tamanho do array que criaremos? Será o tamanho 2. Em seguida, rodaremos o algoritmo. `atual2` começará com o valor do `miolo` igual a 1.

Como funciona o algoritmo? Vamos comparar o André com a Mariana. Qual é o menor? O André, então vamos movê-lo para o outro array. Depois somaremos +1 no `atual1` e no `atual`.

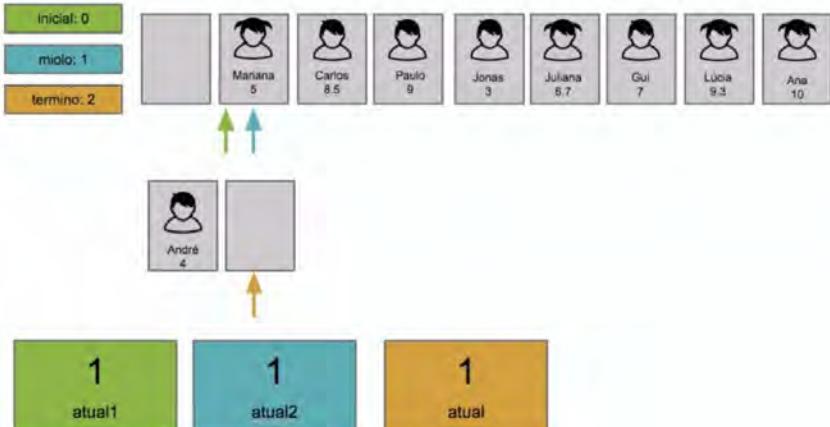


Figura 9.6: Tamanhos válidos para o intercala — passo 2

Sobrou algum elemento no array no primeiro trecho? Não, porque já chegamos ao `miolo`. Sobrou no segundo? Sim, a Mariana. Vamos movê-la para o novo array.

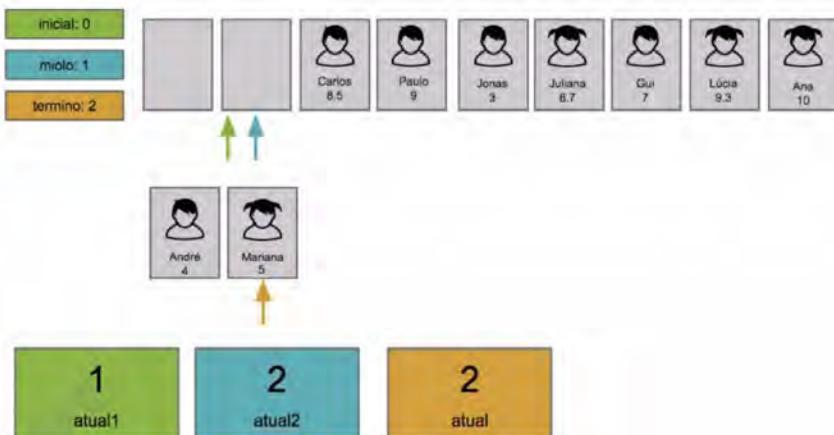


Figura 9.7: Tamanhos válidos para o intercala — passo 3

Depois aumentaremos +1 na variável `atual` e acabou o nosso array. O que faremos agora? Copiaremos de volta os elementos para o array original.

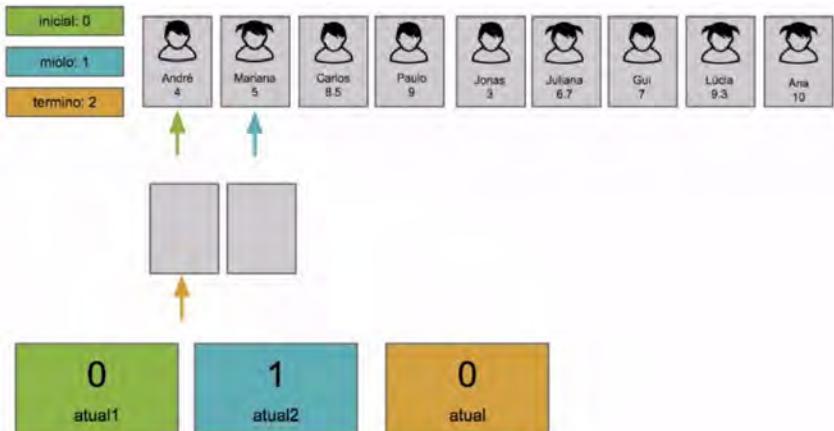


Figura 9.8: Tamanhos válidos para o intercala — passo 4

Para dois elementos, nosso algoritmo funciona. Porém, funcionou porque os elementos já estavam ordenados em posições válidas. Mas e se os dois elementos estiverem com as posições trocadas? Teria funcionado?

Vamos testar com o algoritmo com dois elementos trocados. Começaremos com o `inicial` igual a 3, o `miolo` será igual a 4, e o `termino` 5.

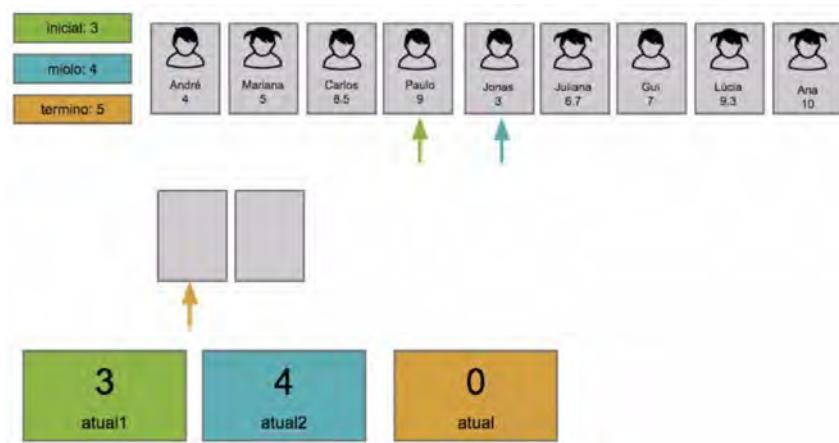


Figura 9.9: Tamanhos válidos para o intercala — passo 5

Vamos rodar o algoritmo. Quantas casas precisaremos criar? Duas. Então, `atual2` será igual ao `miolo` e valerá 4, e `atual1` será igual ao `inicial` e valerá 3.

Ao compararmos Paulo com Jonas, qual é o menor elemento? Jonas. Ele será o primeiro elemento do novo array. Vamos avançar +1 no `atual2` que será igual a 5, e no `atual`, que será igual a 1.

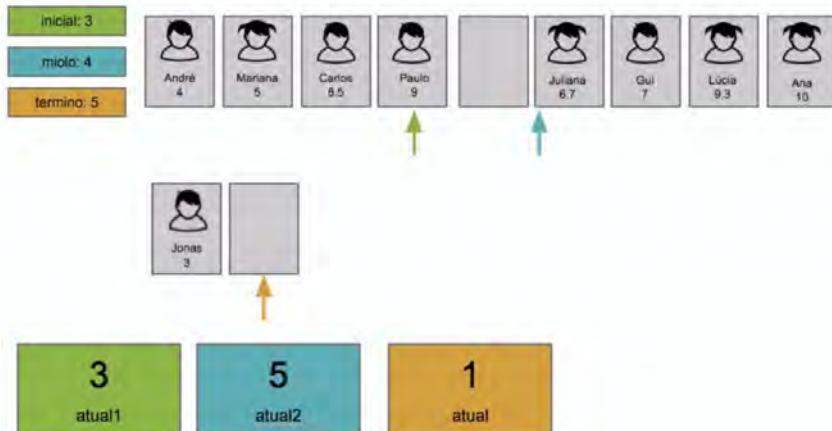


Figura 9.10: Tamanhos válidos para o intercala — passo 6

Agora o `atual2` vai alcançar o `termino`. Acabou. Do segundo trecho, sobrou algum elemento? Sim. Moveremos o Paulo para o outro array e somaremos +1 nas variáveis `atual1` e `atual`.

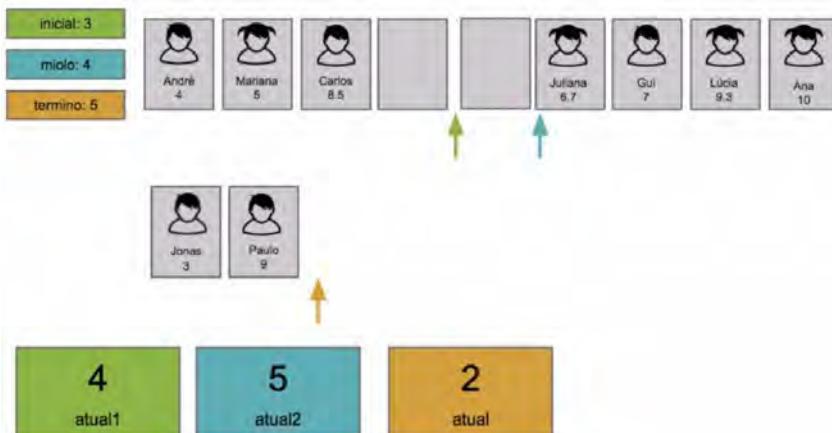


Figura 9.11: Tamanhos válidos para o intercala — passo 7

Como não sobraram outros elementos, vamos retornar os que foram movidos para o array original: Jonas e Paulo.

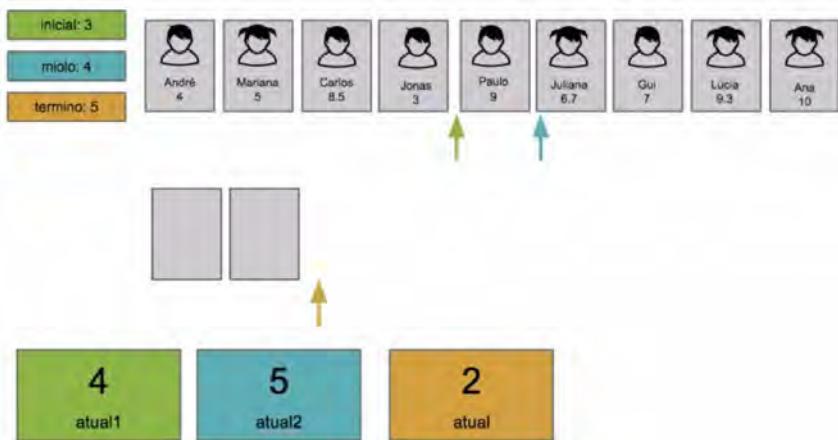


Figura 9.12: Tamanhos válidos para o intercala — passo 8

Após o teste, sabemos que o algoritmo funciona para dois elementos no array, mesmo se eles estiverem trocados. Se os elementos já estiverem na ordem certa, o array ficará correto. Se os elementos estiverem trocados, ele inverterá a posição. Caso só exista um elemento, ele não fará alterações.

Agora sabemos que se trabalharmos com um array de qualquer tamanho, que tenha duas partes ordenadas, independente da quantidade de elementos em cada uma, o algoritmo vai funcionar.

9.2 RODANDO AS VARIAÇÕES DO INTERCALA

Vamos testar outro extremo da nossa função `intercala()`. O que acontecerá se intercalarmos apenas um elemento? Não fará sentido, porque, se isto acontecer, o algoritmo me devolverá exatamente o mesmo resultado.

Porém, se intercalarmos dois elementos, estes serão o outro extremo (pois se intercalarmos apenas um, não fará diferença).

```

public static void main(string[] args) {
    Nota[] notas = {
        new Nota("andre", 4),
        new Nota("mariana", 5),
        new Nota("carlos", 8.5),
        new Nota("paulo", 9),
        new Nota("jonas", 3.0),
        new Nota("juliana", 6.7),
        new Nota("guilherme", 7),
        new Nota("lucia", 9.3),
        new Nota("ana", 10)
    };

    Nota[] rank = intercala(notas, 0, 4, notas.length);
    for(Nota nota : rank) {
        System.out.println(nota.getAluno() + " " + nota.getValor())
    }
}

```

Vamos começar testando os dois primeiros elementos, que já estão ordenados. Então, esquerda será igual a 0, miolo será igual a 1, e tenho 2 elementos que quero intercalar. Vamos escrever isto no código:

```

Nota[] rank = intercala(notas, 0, 1, 2);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() + " " + nota.getValor())
};

```

Ao rodarmos o algoritmo, o resultado será:

```

andre 4.0
mariana 5.0
carlos 8.5
paulo 9.0
jonas 3.0
juliana 6.7
guilherme 7.0
lucia 9.3
ana 10.0

```

A ordem ficou correta. Quando os elementos já estão em ordem, o algoritmo não faz alterações. Mas e se os elementos não estivessem em ordem? Vamos usar dois que não estão em ordem: Paulo e

Jonas.

O Paulo está na posição 3, e o Jonas na posição 4. Como queremos analisar dois elementos, vamos até a posição 5.

```
Nota[] rank = intercala(notas, 3, 4, 5);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() + " " + nota.getValor())
};
```

Se testarmos, o que será que acontece?

```
andre 4.0
mariana 5.0
carlos 8.5
jonas 3.0
paulo 9.0
juliana 6.7
guilherme 7.0
lucia 9.3
ana 10.0
```

No resultado, ele imprimiu primeiro o Jonas e depois o Paulo. O algoritmo ordenou os itens também. A função `intercala` funciona com dois elementos — intercalados ou não. Ela também funcionará para três itens ou array inteiro.

Podemos inclusive testar como funcionará com um elemento. A função vai ordenar o elemento na posição 3, logo o `miolo` também será igual a 3, e só deve ordenar até o 3. Então, vai até o 4 exclusive.

```
Nota[] rank = intercala(notas, 3, 3, 4);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() + " " + nota.getValor())
};
```

O resultado será:

```
andre 4.0
mariana 5.0
carlos 8.5
paulo 9.0
jonas 3.0
juliana 6.7
```

```
guilherme 7.0
lucia 9.3
ana 10.0
```

O elemento permaneceu na mesma posição. O algoritmo não ficou maluco. Vamos ver o que acontece se testarmos com o Jonas, que está na posição 4.

```
Nota[] rank = intercala(notas, 4, 4, 5);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() + " " + nota.getValor())
};
```

A ordem continuará a mesma, e o Jonas também vai continuar na mesma posição. A ordem permaneceria a mesma se testássemos com o primeiro elemento ou qualquer outro do array. Isto significa que o `intercala` funciona para um, dois ou vários elementos.

No nosso caso, `miolo` valerá 4 e o limite será `notas.length`.

```
Nota[] rank = intercala(notas, 0, 4, notas.length);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() + " " + nota.getValor())
};
```

Mas é importante sabermos que o algoritmo funciona independente do número de elementos que intercalarmos. Ele sabe intercalar pequenas e grandes quantidade de itens.

9.3 INTERCALANDO UM TRECHO PEQUENO

Até agora vimos como a função `intercala` é capaz de intercalar elementos de dois arrays ou trechos de uma lista. O algoritmo ordenará todos eles.

Porém, o que acontecerá se o nosso array estiver com todos os itens desordenados? E se, em vez de usarmos dois trechos organizados, tivéssemos um array totalmente desordenado?

O que o nosso algoritmo vai fazer? Sinceramente, não tenho

ideia. Mas ele certamente não funcionará. Vamos testá-lo nestas condições:

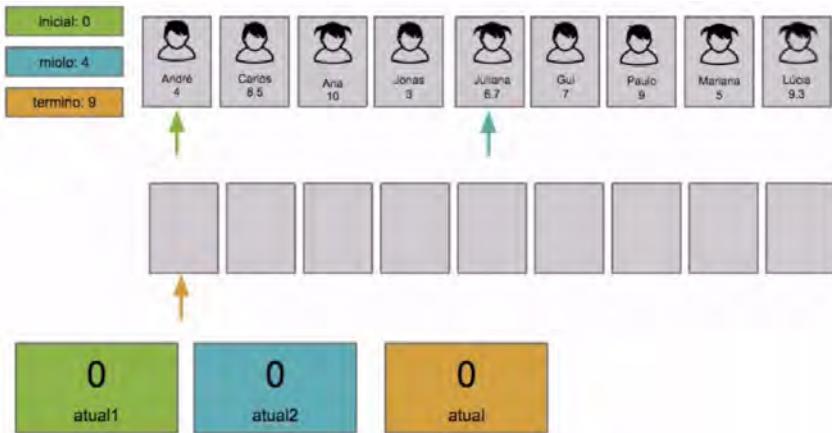


Figura 9.13: Intercala pequeno

Ao compararmos André com Juliana, qual é o menor? O André. Vamos movere-lo para o outro array.

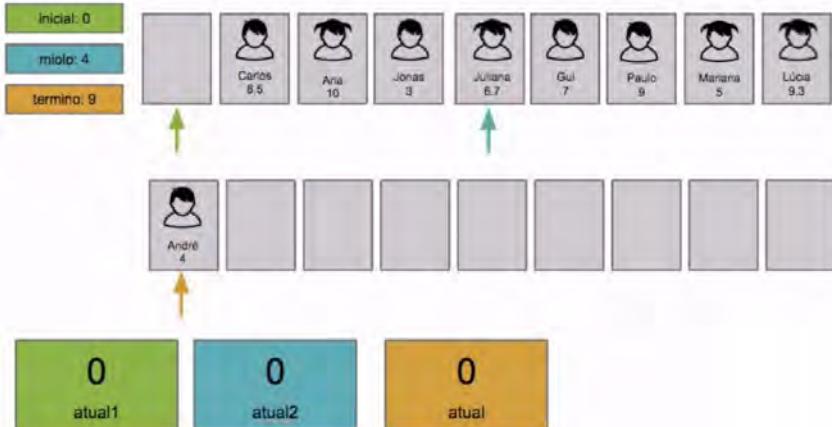


Figura 9.14: Intercala pequeno — passo 2

Depois, entre Carlos e Juliana, qual moveremos? A Juliana.

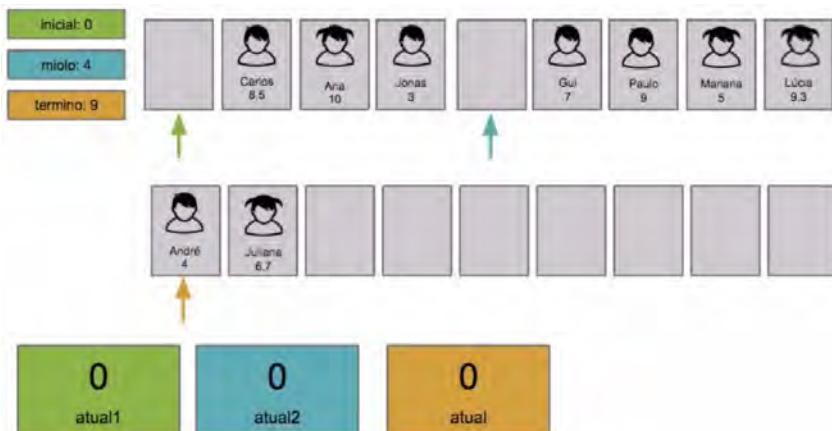


Figura 9.15: Intercala pequeno — passo 3

Comparando Carlos com o Gui, qual é o menor? O Gui. Ele vai para o novo array.

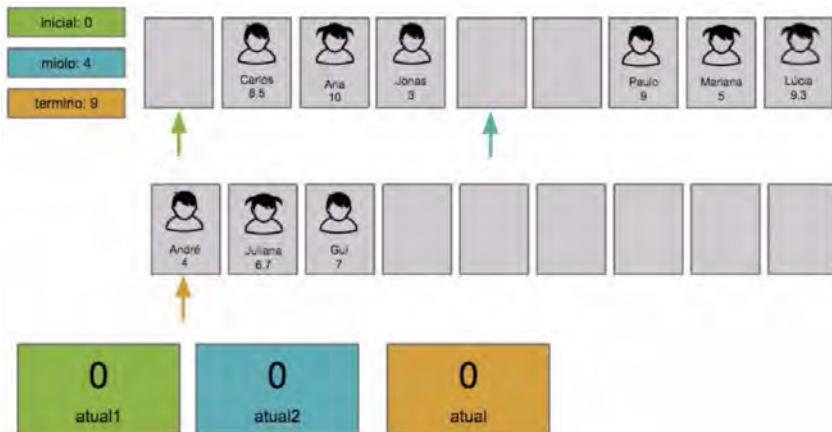


Figura 9.16: Intercala pequeno — passo 4

Qual é o menor, Carlos ou Paulo? O Carlos. Vamos colocá-lo no novo array.

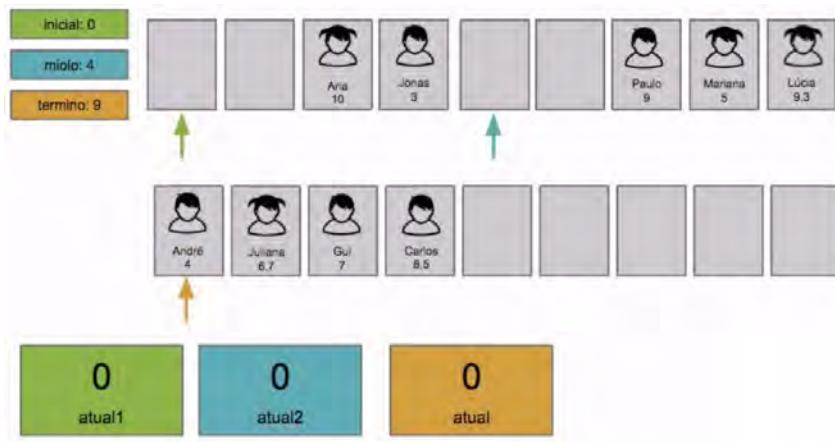


Figura 9.17: Intercala pequeno — passo 5

Qual é o menor, Ana ou Paulo? O Paulo. Vamos movê-lo.

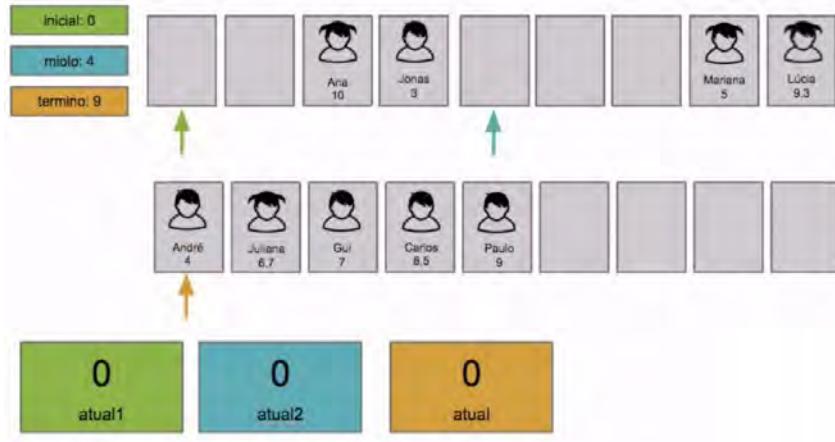


Figura 9.18: Intercala pequeno — passo 6

Qual é o menor, Ana ou Mariana? Mariana. Vamos colocá-la na outra lista.

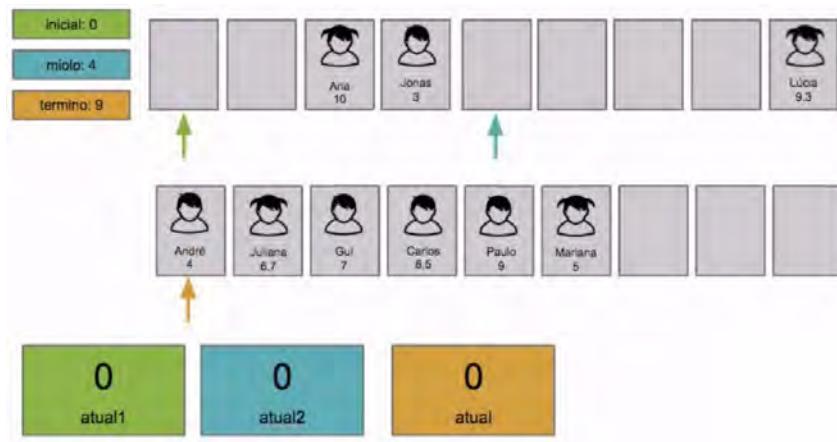


Figura 9.19: Intercala pequeno — passo 7

Qual é o menor, Ana ou Lúcia? Lúcia. Vamos movê-la para o outro array.

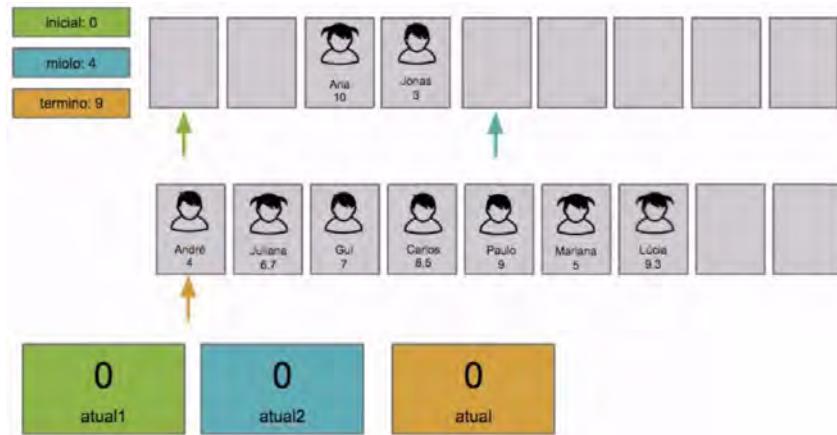


Figura 9.20: Intercala pequeno — passo 8

Depois, sobrará a Ana e o Jonas. Vamos movê-los para o novo array.

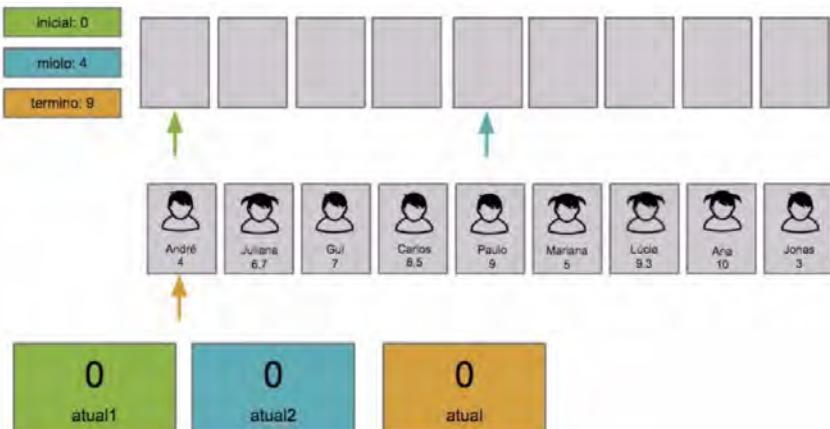


Figura 9.21: Intercala pequeno — passo 9

Observe a ordem que ficou:

André 4
 Juliana 6.7
 Gui 7
 Carlos 8.5
 Paulo 9
 Mariana 5
 Lúcia 9.3
 Ana 10
 Jonas 3

Não funciona. Os elementos ficaram desordenados na lista. Para funcionar corretamente, o algoritmo exige três situações:

1. Ele funcionará no caso original, em que o array tem duas partes ordenadas. Neste caso, conseguimos intercalar corretamente.

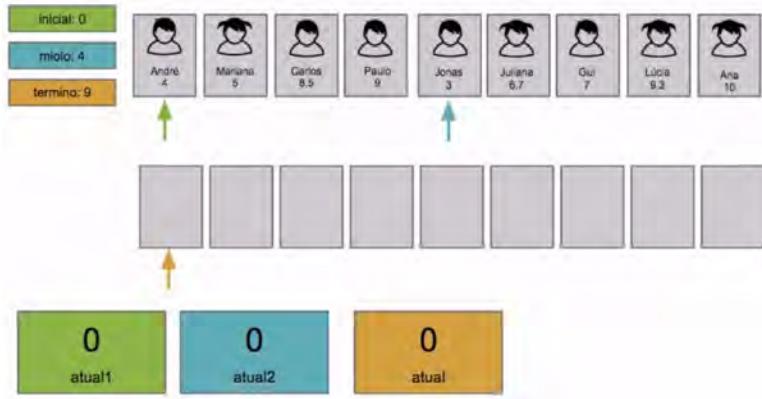


Figura 9.22: Intercala pequeno — passo 10

2. Se deixarmos os elementos misturados, ele funcionará se pedirmos para ordenar apenas um item. Quando isto acontece, o algoritmo compara o elemento com ele mesmo. Se ordenarmos apenas o André, vamos movê-lo para o novo array.

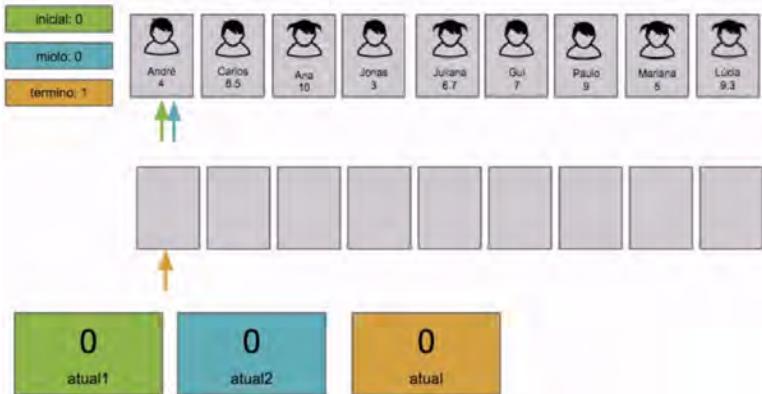


Figura 9.23: Intercala pequeno — passo 11

E depois será devolvido para o original. Neste caso, o algoritmo funciona.

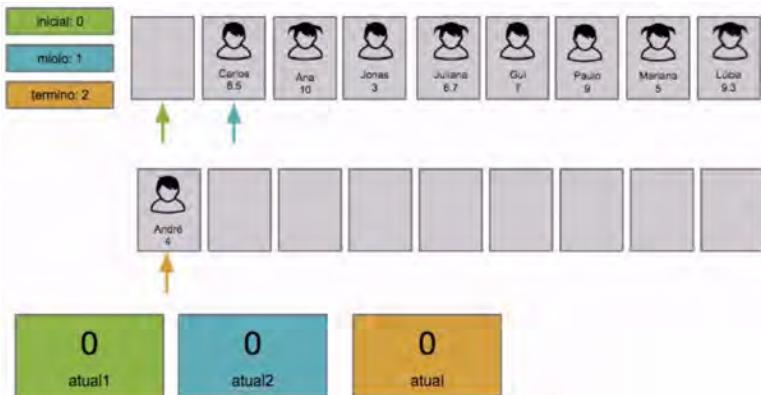


Figura 9.24: Intercala pequeno — passo 12

3. Podemos intercalar dois elementos já ordenados, mesmo que os demais estejam desordenados. Nós já havíamos testado este caso. Vamos movê-los para o novo array, e depois devolveremos.

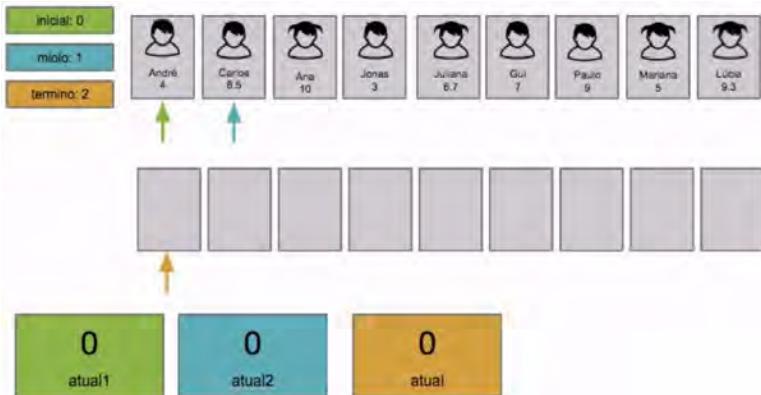


Figura 9.25: Intercala pequeno — passo 13

Então, sabemos que, com um elemento, o algoritmo funciona e, como dois elementos ordenados, também. Mas e se os dois elementos não estivessem ordenados? Como no caso da Ana com o

Jonas, por exemplo.

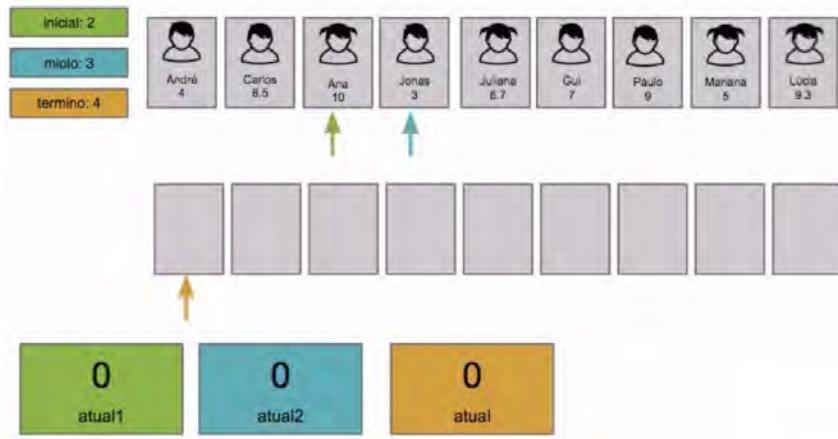


Figura 9.26: Intercala pequeno — passo 14

O restante do array é irrelevante. Ao compararmos a Ana com o Jonas, veremos que o Jonas é o menor. Vamos mover-lo para o novo array. Em seguida, vai a Ana. Depois, devolveremos os dois elementos para o array original.

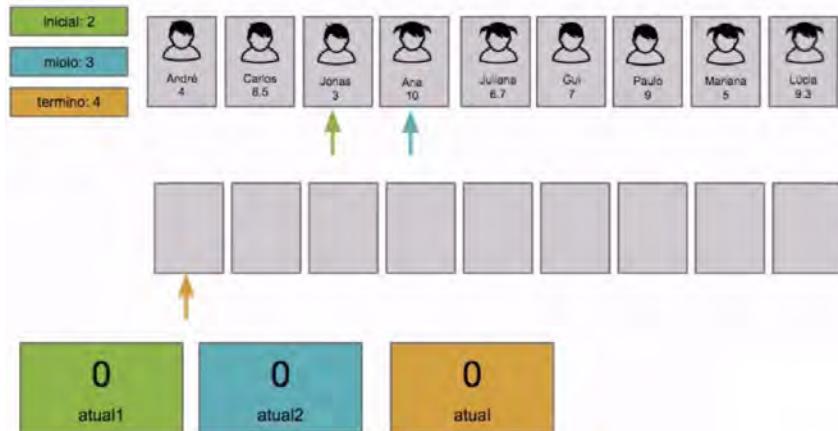


Figura 9.27: Intercala pequeno — passo 15

Logo, para **um** elemento, o **intercala** funciona. Para dois

elementos, ordenados ou não, o algoritmo também funciona. No caso de um array, dividido em dois trechos com os elementos ordenados, o `intercala` funciona corretamente.

9.4 O PRÓXIMO DESAFIO: INTERCALANDO DIVERSAS VEZES SEGUIDAS

Agora que sabemos que o `intercala` funciona adequadamente, o que será que acontece se invocarmos ele dezenas de vezes seguidas? Seguindo um padrão bem definido? Em breve, veremos que uma tarefa pequena como o `intercala` pode ser capaz de compor um dos algoritmos mais famosos de ordenação.

CAPÍTULO 10

O ALGORITMO MERGE SORT

Vimos que o `intercala` funciona em quatro situações diferentes. Se tenho um único elemento, que não podemos fazer nada, então até poderíamos descartar esta situação. Se tenho dois elementos e queremos trocá-los de posição ou deixá-los como estão. Estas já são duas situações em que funciona. Ou se temos um array com dois trechos já ordenados e queremos intercalar os elementos, neste caso, nosso algoritmo também funciona.

Então ele funcionará mesmo com uma quantidade pequena e desordenada (dois elementos), ou com uma quantidade grande e ordenada.

Vamos rodar o nosso algoritmo com o seguinte array que tem os elementos totalmente desordenados. Se testarmos apenas com os primeiros itens, a lista permanecerá na ordem em que está.

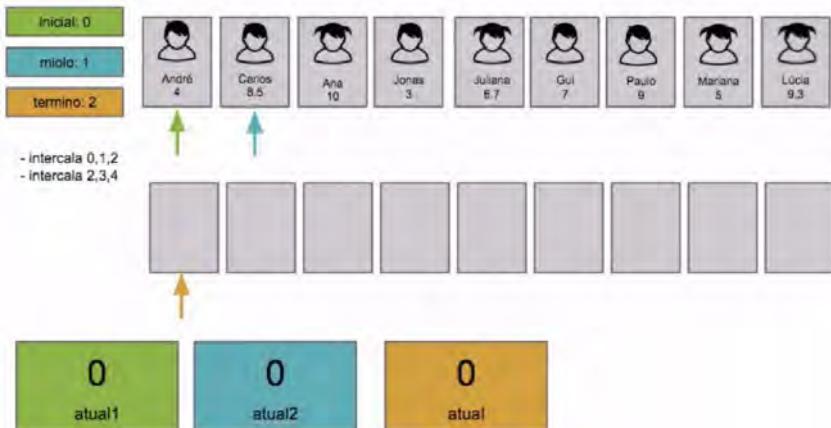


Figura 10.1: Intercalando diversas vezes seguidas — passo 1

Vamos testar o `intercala 0,1,2`. O que acontecerá? Tanto o André como o Carlos vão para baixo.

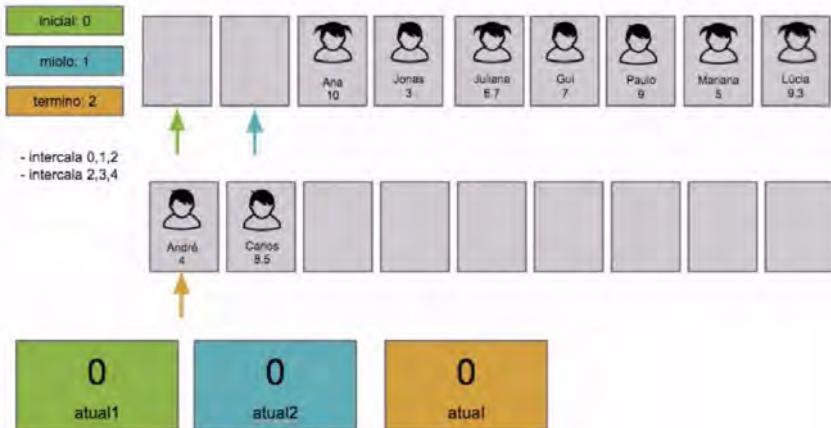


Figura 10.2: Intercalando diversas vezes seguidas — passo 2

Em seguida, voltarão para o array de origem.

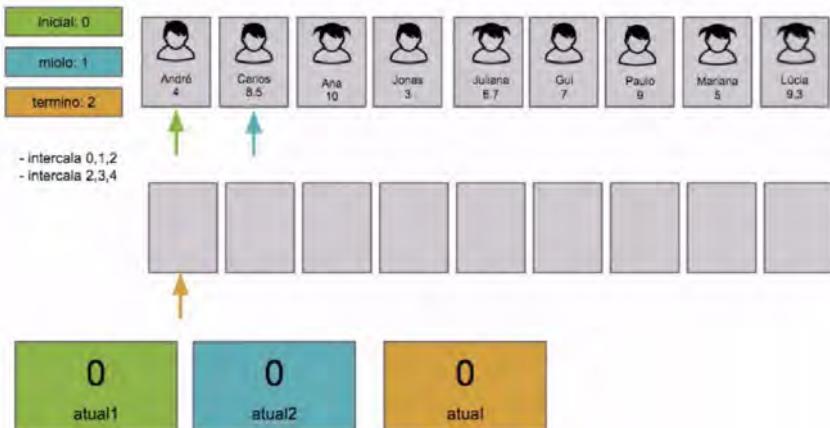


Figura 10.3: Intercalando diversas vezes seguidas — passo 3

Nosso algoritmo funcionou. Vamos tentar intercalar os elementos 3 e 4 (Ana e Joana). O que acontecerá? Se rodarmos os algoritmos para estes elementos, primeiro moveremos o Jonas, e em seguida a Ana.

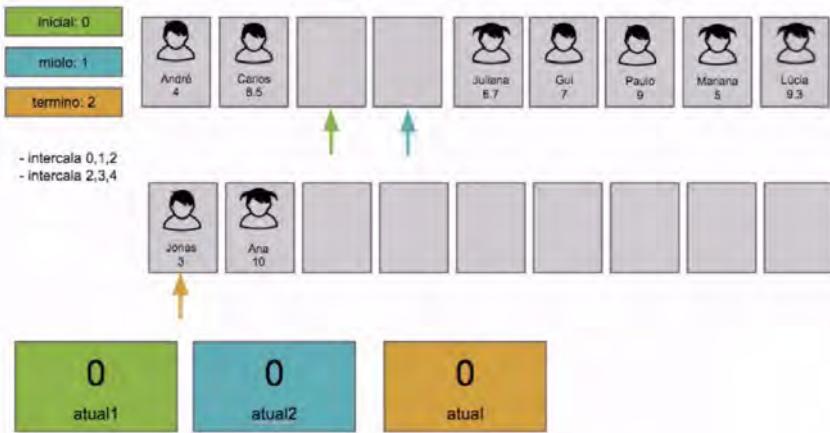


Figura 10.4: Intercalando diversas vezes seguidas — passo 4

Depois os dois voltarão para o antigo array.

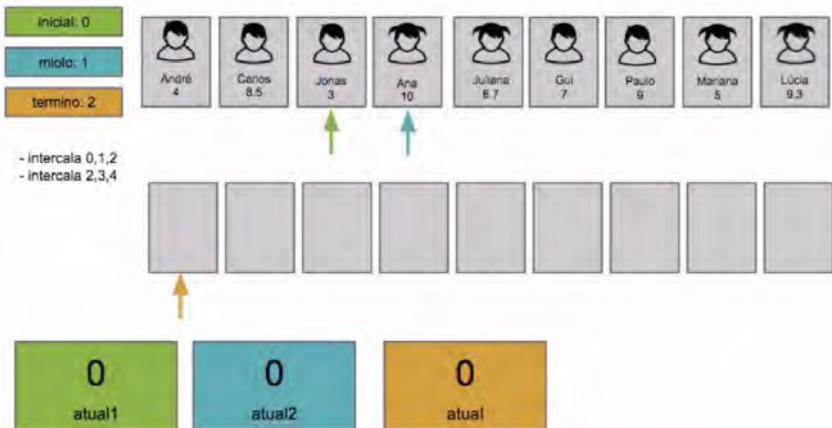


Figura 10.5: Intercalando diversas vezes seguidas — passo 5

Os quatro primeiros elementos já estão melhor posicionados. Se os dois primeiros estão na ordem certa e os dois seguintes também, o que acontecerá se chamarmos o `intercala` com o `inicial` igual a 0, com o `miolo` igual a 2, e o `termino` igual a 4?

Temos uma primeira parte ordenada (do 0 até 2), assim como a segunda também (do 2 até o 4). Se os dois trechos pequenos foram intercalados, eles foram ordenados. Vamos agora intercalar o total destas duas partes. Logo, dado estes dois primeiros passos, vamos seguir para o próximo, que é o `intercala 0, 2, 4`.

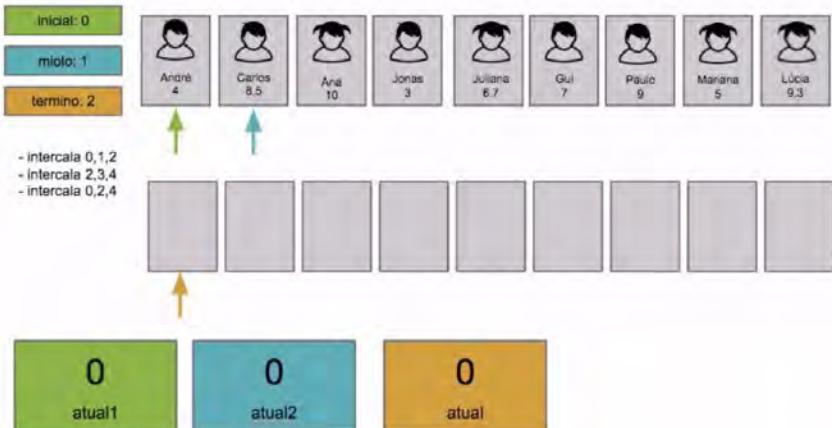


Figura 10.6: Intercalando diversas vezes seguidas — passo 6

Vamos simular os três `intercala` novamente? Começaremos do 0. Comparamos 0 e 1, André e Carlos. Qual é o menor? André. Desceremos ele para o outro array e, depois, com o Carlos.

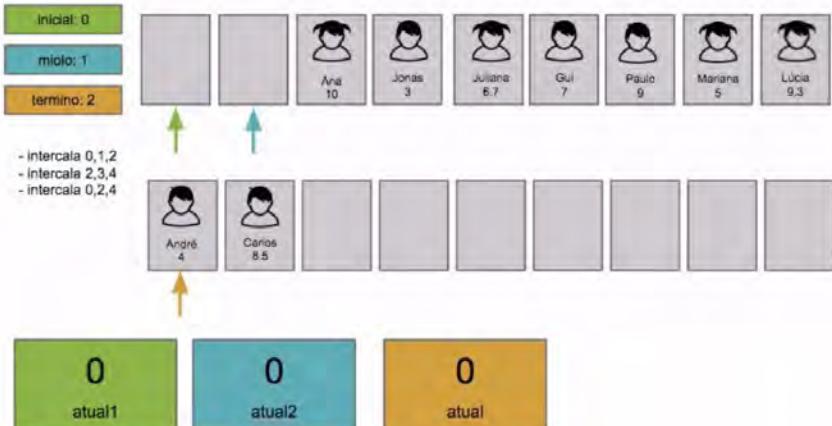


Figura 10.7: Intercalando diversas vezes seguidas — passo 7

Em seguida, sobem os dois novamente. Pronto, os dois estão intercalados.

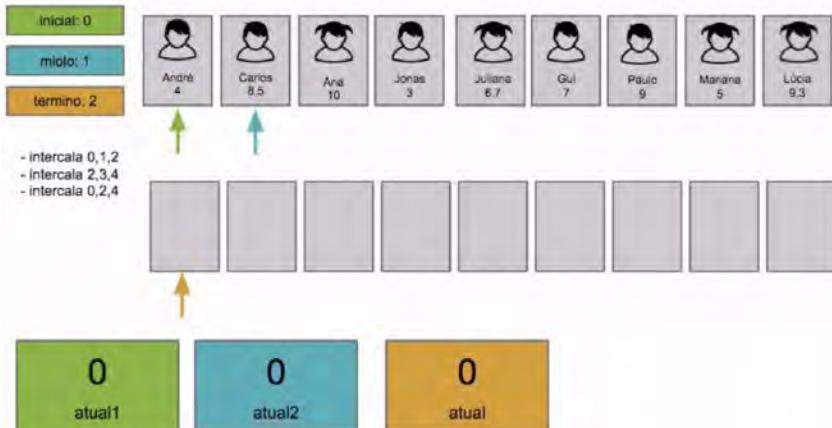


Figura 10.8: Intercalando diversas vezes seguidas — passo 8

Agora o Jonas e a Ana serão os próximos a serem movidos para o novo array.

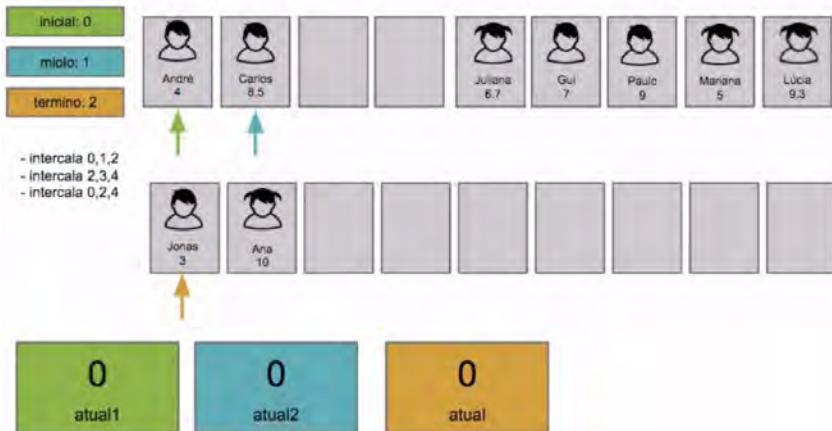


Figura 10.9: Intercalando diversas vezes seguidas — passo 9

Em seguida, subiremos os dois para o array anterior. Os dois estão intercalados.

Fizemos duas chamadas bem pequenas do `intercala`. Vamos fazer em seguida uma chamada um pouco maior, com `inicial`

igual a 0, o miolo igual a 2, e o termino igual a 4. Simularemos o algoritmo. Como as duas partes menores estão ordenadas, ordenaremos os quatro elementos.

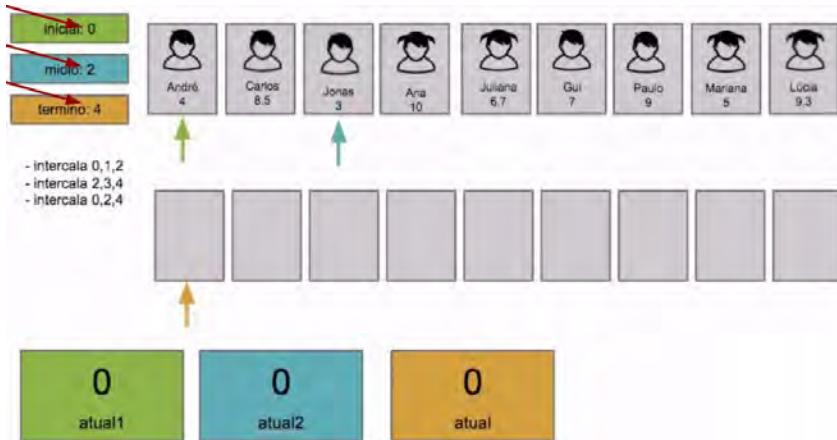


Figura 10.10: Intercalando diversas vezes seguidas — passo 10

Testaremos o algoritmo. Mas quantas casinhas vamos precisar? Apenas 4. Depois, modificaremos os valores da variável `atual2`, que será igual a 2.

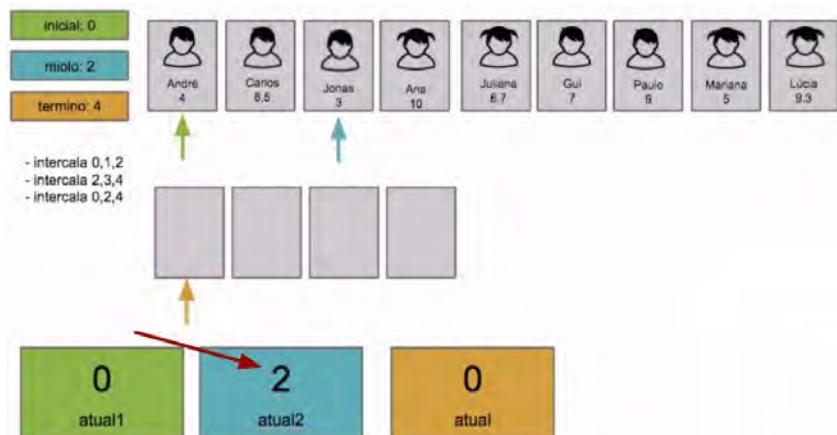


Figura 10.11: Intercalando diversas vezes seguidas — passo 11

Qual elemento é o menor, André ou Jonas? O Jonas. Vamos movê-lo para o outro array.

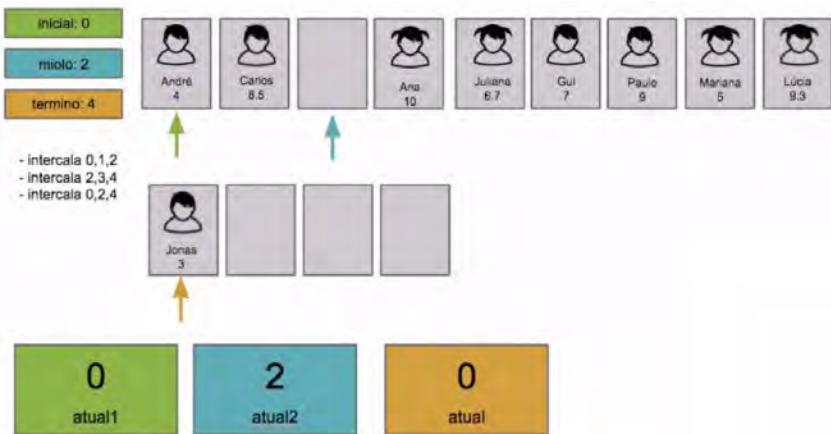


Figura 10.12: Intercalando diversas vezes seguidas — passo 12

Somaremos +1 nas variáveis `atual2` e `atual`. Qual é o menor elemento, André ou Ana? O André. Vamos movê-lo para o outro array.

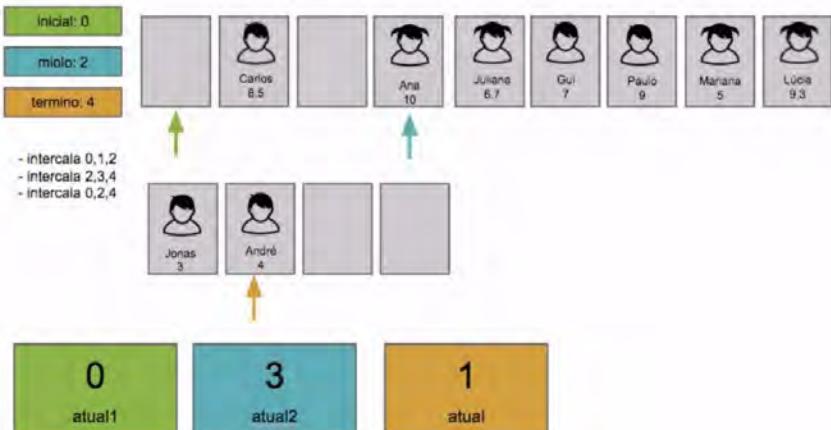


Figura 10.13: Intercalando diversas vezes seguidas — passo 13

Vamos aumentar +1 no `atual1`, que será igual a 1, e no

atual , que será igual a dois. Qual é o menor elemento, Carlos ou Ana? O Carlos. Vamos movê-lo para o outro array.

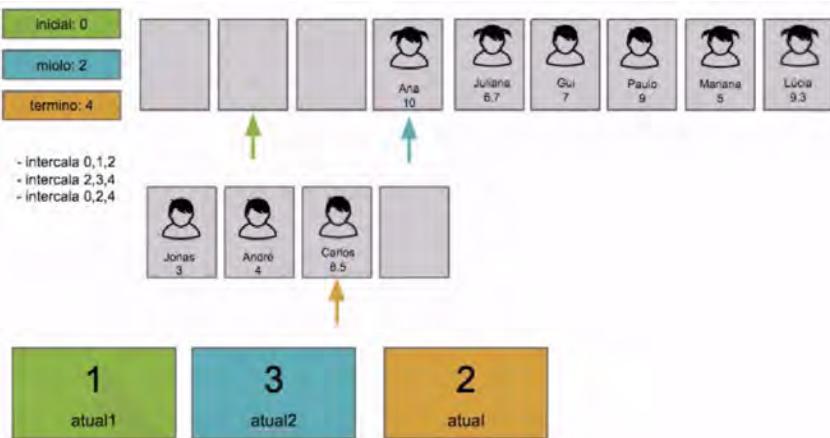


Figura 10.14: Intercalando diversas vezes seguidas — passo 14

Aumentaremos +1 no atual e no atual1 . Observe que a nossa análise acabou, porque o atual1 já alcançou o miolo . Acabou a primeira parte do nosso *array, sobrando um elemento na segunda, a Ana. Vamos copiá-la para o outro array.

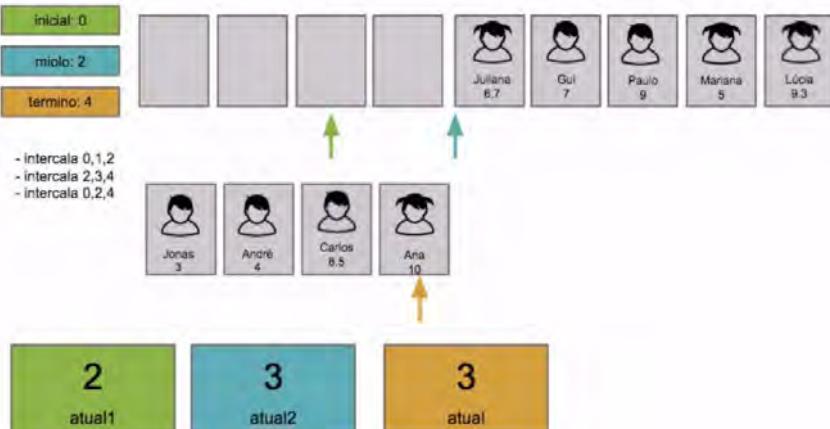


Figura 10.15: Intercalando diversas vezes seguidas — passo 15

Alteraremos os valores das variáveis `atual2` e `atual`, que serão igual a 4. Agora apenas temos de mover os elementos do novo array para o de origem.

O que aconteceu? Os quatro elementos (Jonas, André, Carlos e Ana) estão ordenados.

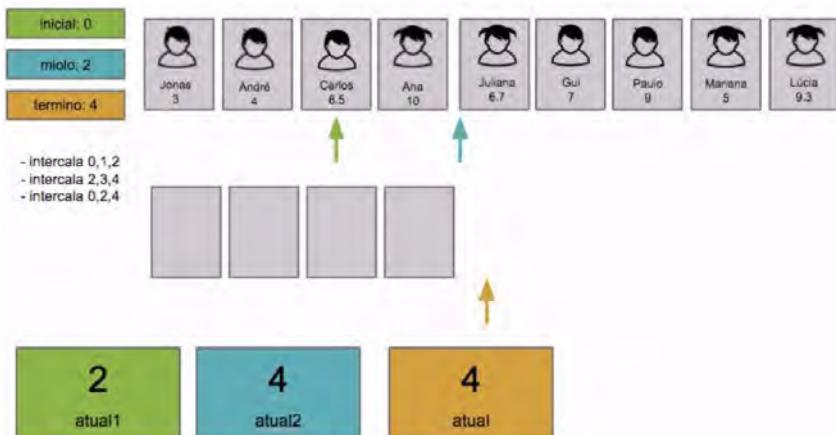


Figura 10.16: Intercalando diversas vezes seguidas — passo 16

Nós começamos com as menores pontuações e fomos até as maiores. Intercalmamos os dois primeiros, depois apenas os dois finais e finalizamos com os quatro. O que podemos concluir, então? O que podemos concluir que acontecerá se seguirmos intercalando com os elementos seguintes da mesma maneira? E se intercalarmos depois oito elementos, o que vai acontecer? Ou se intercalarmos os nove?

O que nós fizemos foi chamar o `intercala` para o elemento 1 e 2, depois para o 3 e 4. Seguimos intercalando os quatro elementos. Agora chamaremos o `intercala` do 4, 5 e 6. Depois vamos intercalar 6, 7 e 8. Então, intercalaremos os elementos da segunda parte. O próximo passo será intercalar os 8 elementos. E depois copiamos o último. Se simularmos tudo isso, o que acontecerá?

Vamos fazer a simulação. Não alteraremos as variáveis, apenas moveremos os elementos para a outra lista. Veremos o que acontece se intercalarmos dos menores para os maiores elementos.

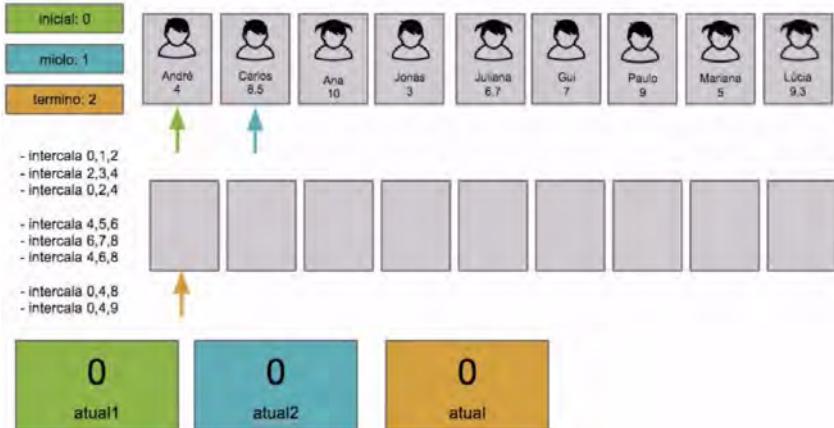


Figura 10.17: Intercalando diversas vezes seguidas — passo 17

No começo, vamos intercalar a cada dois elementos e depois os quatro. Seguiremos para a segunda parte do array e repetiremos o processo com os outros quatro. Após a segunda parte ordenada, intercalaremos os oito. Vamos ver o que acontece quando dividimos o nosso problema em pedaços pequenos?

Iniciaremos pelo `intercala 0,1,2`. Ao compararmos o André com o Carlos, identificaremos que o André é o menor. Vamos movê-lo para o outro array.

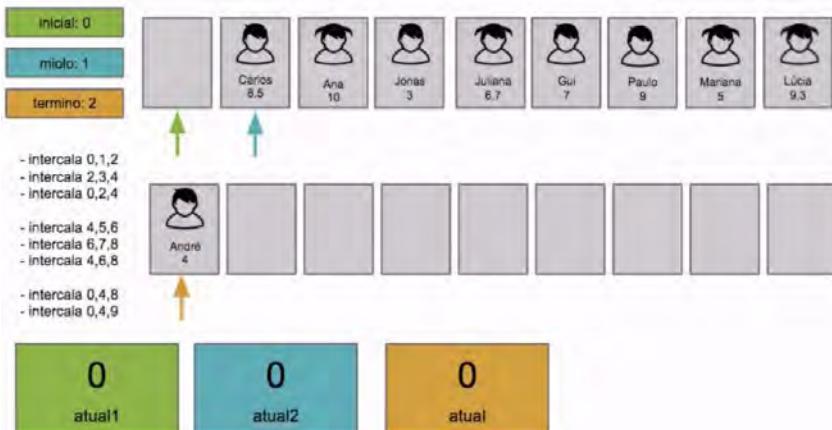


Figura 10.18: Intercalando diversas vezes seguidas — passo 18

Em seguida, o Carlos descerá também.

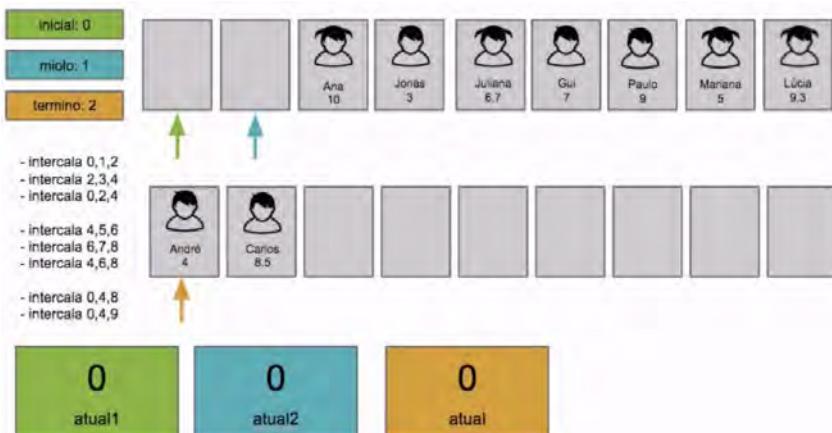


Figura 10.19: Intercalando diversas vezes seguidas — passo 19

Depois, os dois elementos vão voltar para o array de origem.

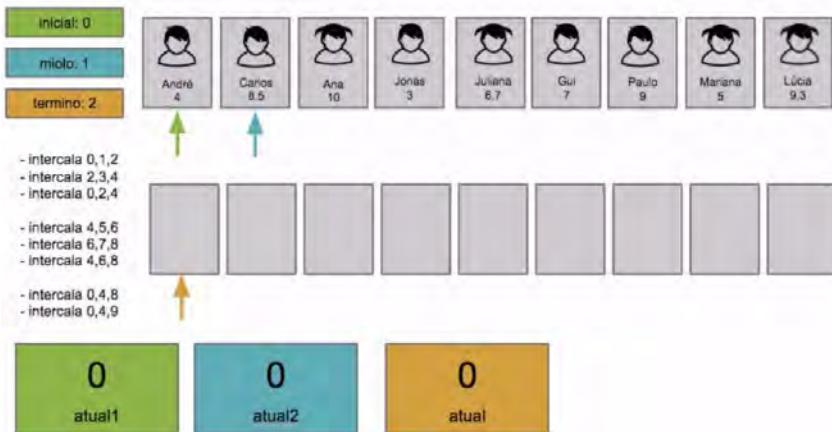


Figura 10.20: Intercalando diversas vezes seguidas — passo 20

Agora é a vez de intercalarmos a Ana e o Jonas. Primeiro o Jonas desce, seguida pela Ana.

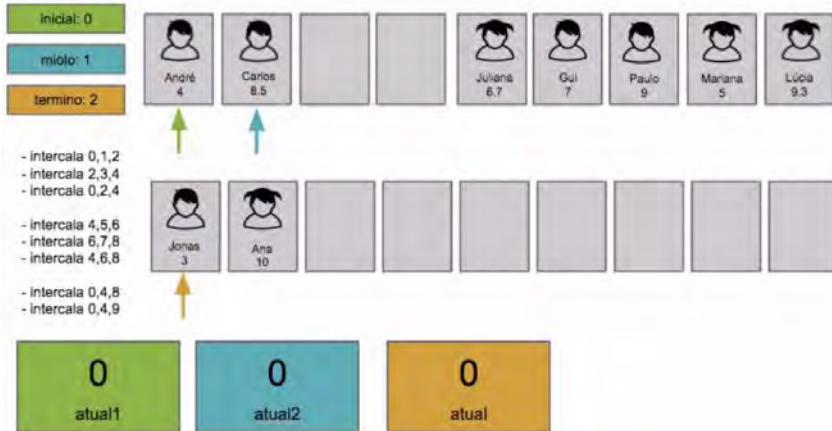


Figura 10.21: Intercalando diversas vezes seguidas — passo 21

Então, os dois sobem.

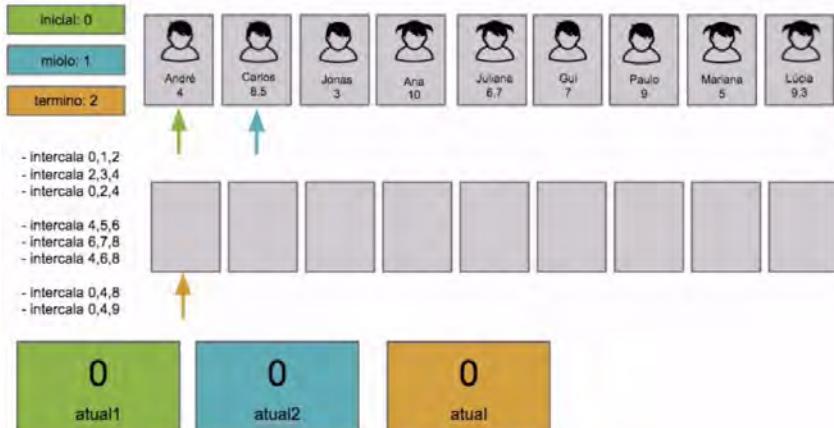


Figura 10.22: Intercalando diversas vezes seguidas — passo 22

Vamos intercalar os quatro primeiros elementos. O que faremos? Intercalaremos o André com o Jonas. O Jonas vai descer.

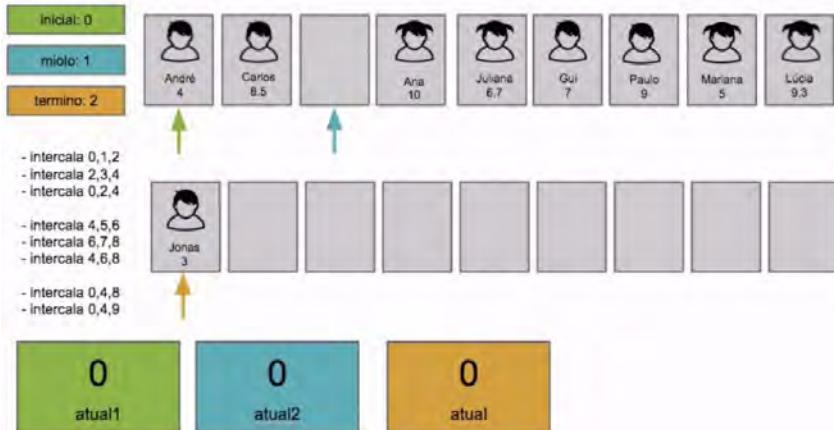


Figura 10.23: Intercalando diversas vezes seguidas — passo 23

Depois o André.

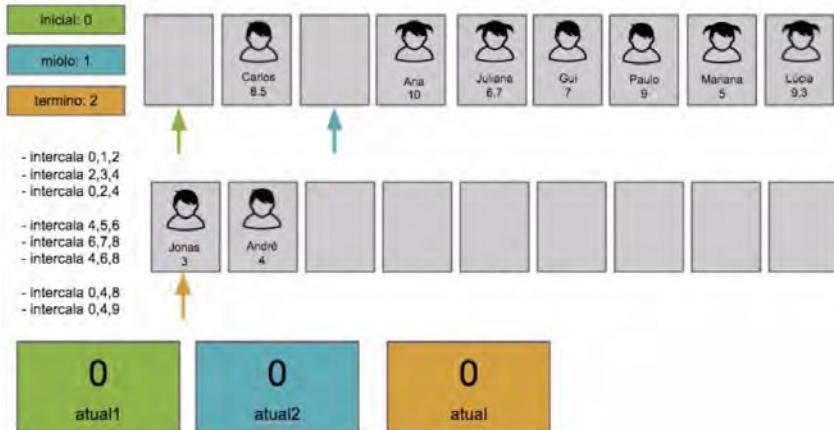


Figura 10.24: Intercalando diversas vezes seguidas — passo 24

O Carlos.

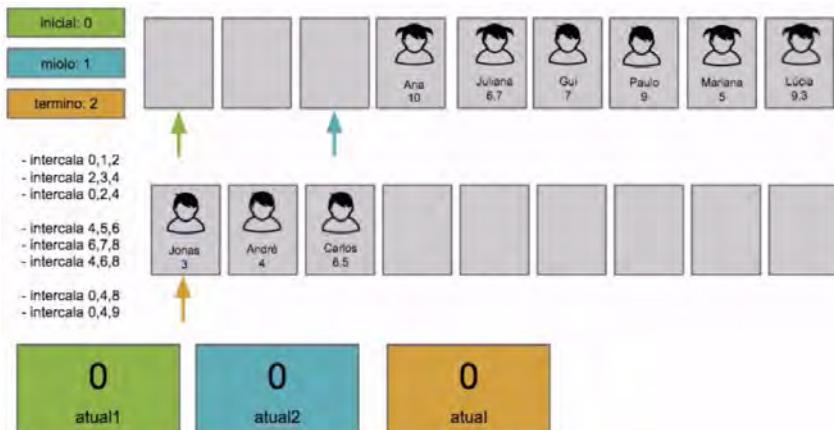


Figura 10.25: Intercalando diversas vezes seguidas — passo 25

E a Ana também descerá.

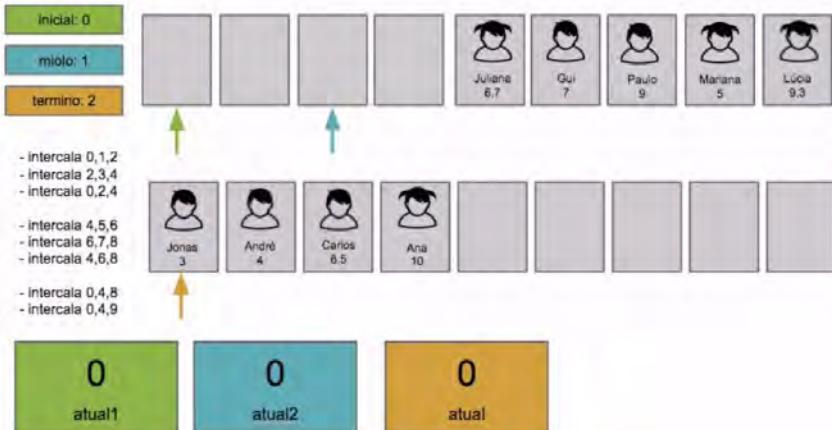


Figura 10.26: Intercalando diversas vezes seguidas — passo 26

Logo, subiremos todos os quatro elementos.

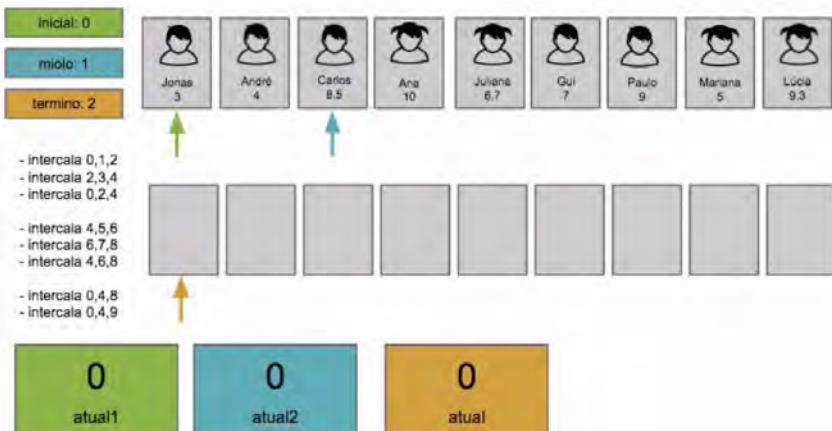


Figura 10.27: Intercalando diversas vezes seguidas — passo 27

Precisaremos intercalar a segunda parte do array. Começaremos novamente intercalando a cada dois elementos.

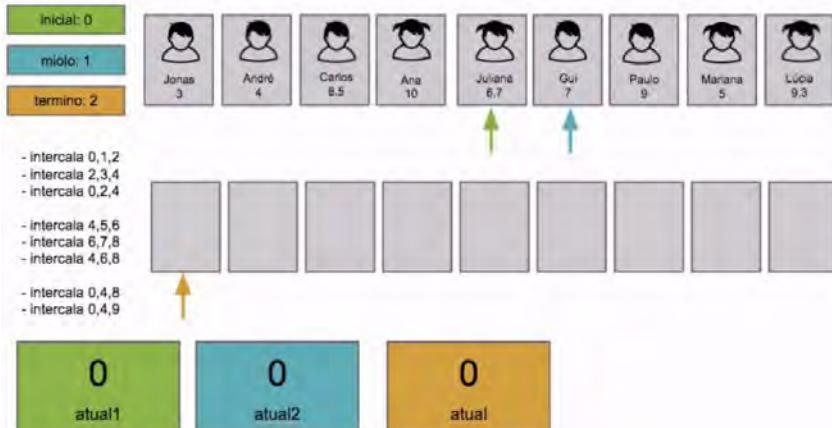


Figura 10.28: Intercalando diversas vezes seguidas — passo 28

Ao compararmos Juliana e Gui, qual é o menor? A Juliana. Ela descerá para o novo array.

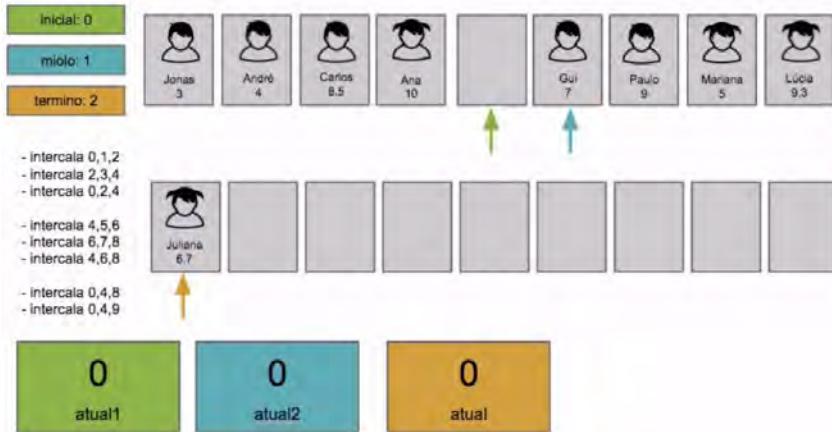


Figura 10.29: Intercalando diversas vezes seguidas — passo 29

Depois será o Gui.

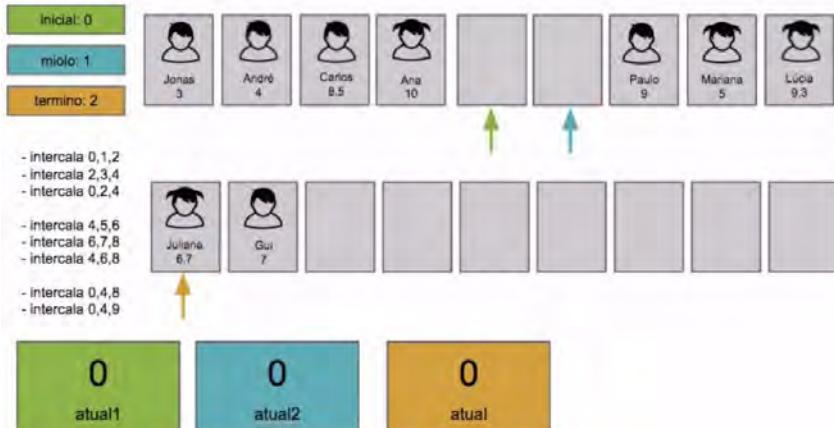


Figura 10.30: Intercalando diversas vezes seguidas — passo 30

Qual é o próximo passo? Os dois subirão.

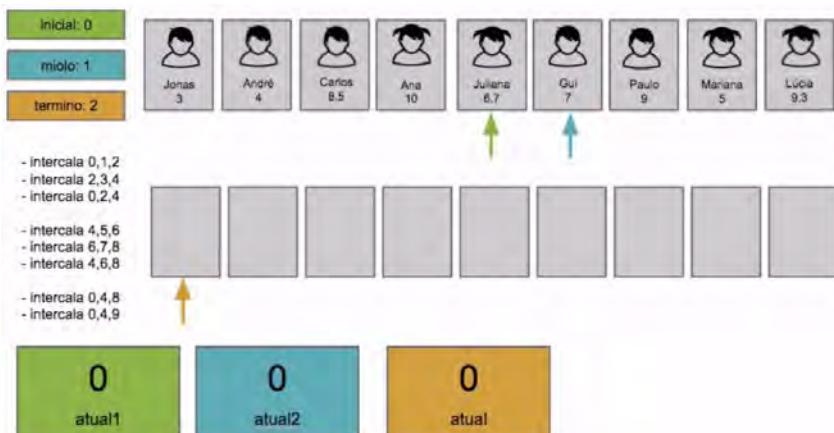


Figura 10.31: Intercalando diversas vezes seguidas — passo 31

Vamos intercalar o Paulo e a Mariana. A Mariana é o menor elemento e vai descer. E depois o Paulo.

Depois, sobem os dois para o array de origem.

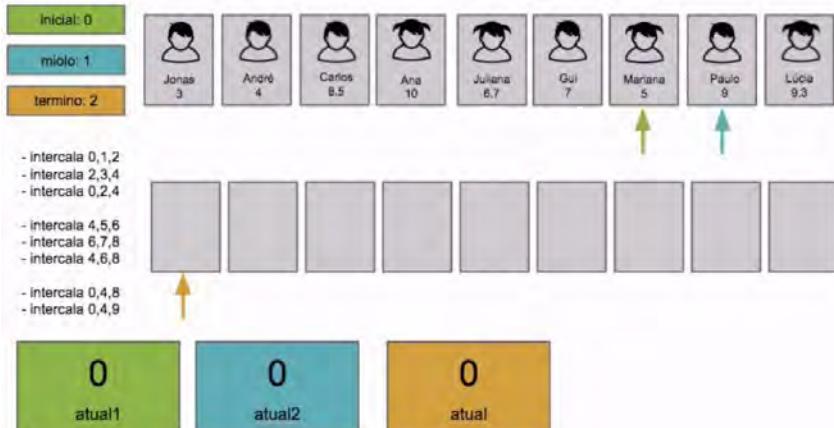


Figura 10.32: Intercalando diversas vezes seguidas — passo 32

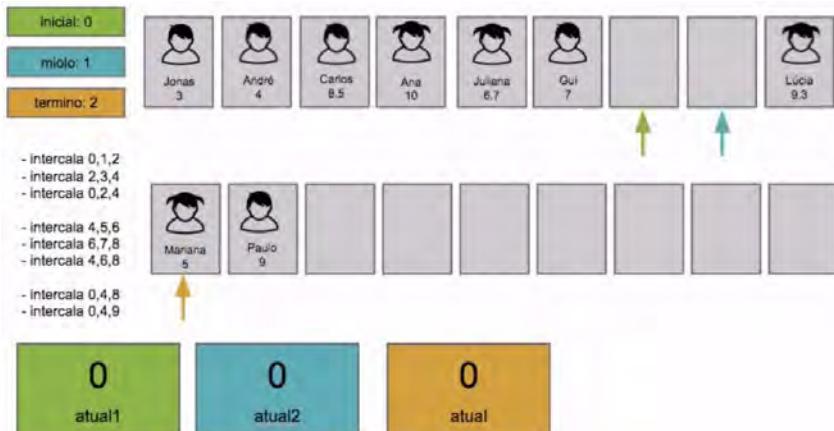


Figura 10.33: Intercalando diversas vezes seguidas — passo 33

Os quatro primeiros elementos da esquerda já foram ordenados. E ordenamos de dois em dois os da direita. Vamos agora intercalar os quatro elementos. Entre a Juliana e a Mariana, qual é a menor? A Mariana. Ela descerá.

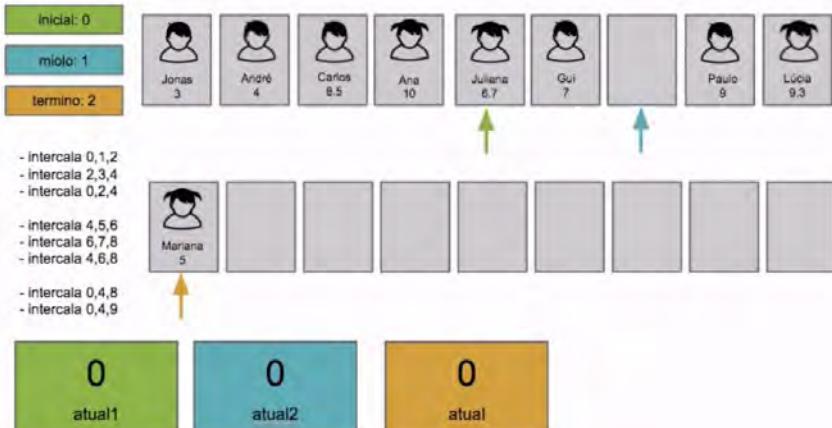


Figura 10.34: Intercalando diversas vezes seguidas — passo 34

Depois a Juliana.

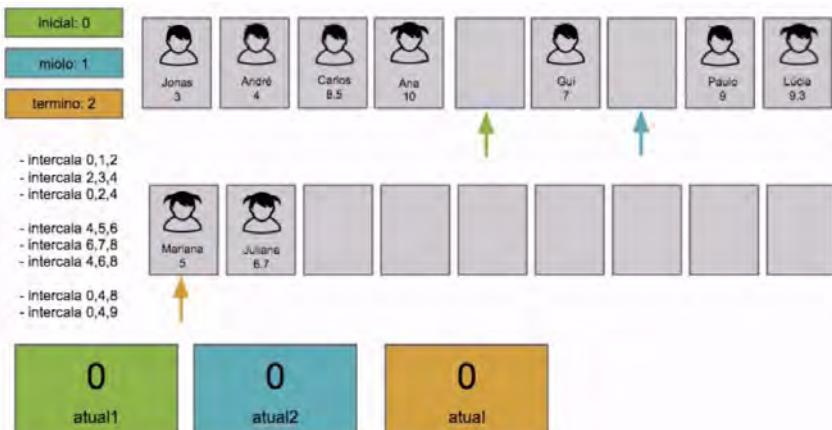


Figura 10.35: Intercalando diversas vezes seguidas — passo 35

Em seguida, será a vez do Gui (Guilherme, eu mesmo).

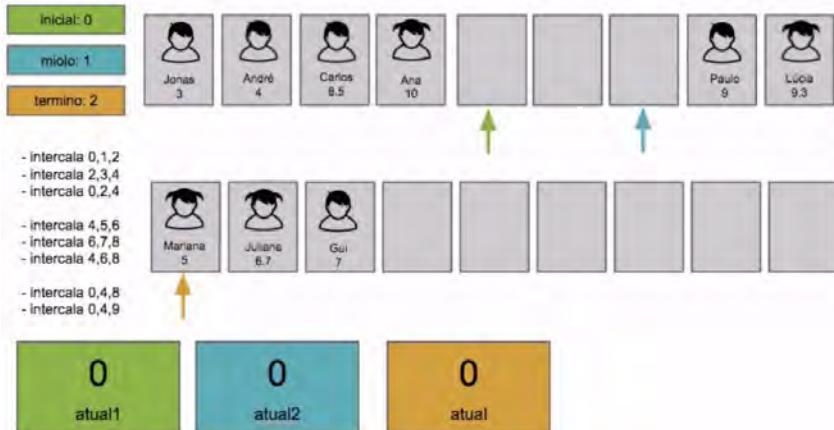


Figura 10.36: Intercalando diversas vezes seguidas — passo 36

E do Paulo.

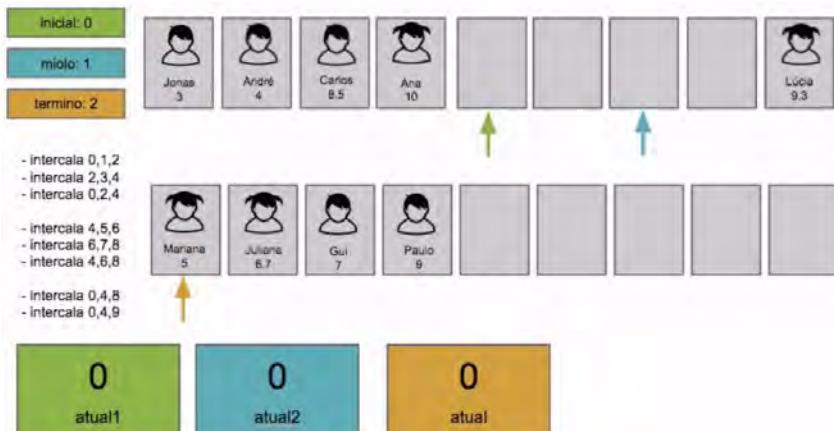


Figura 10.37: Intercalando diversas vezes seguidas — passo 37

Qual é o próximo passo? Todos subirão, porque já foram ordenados.

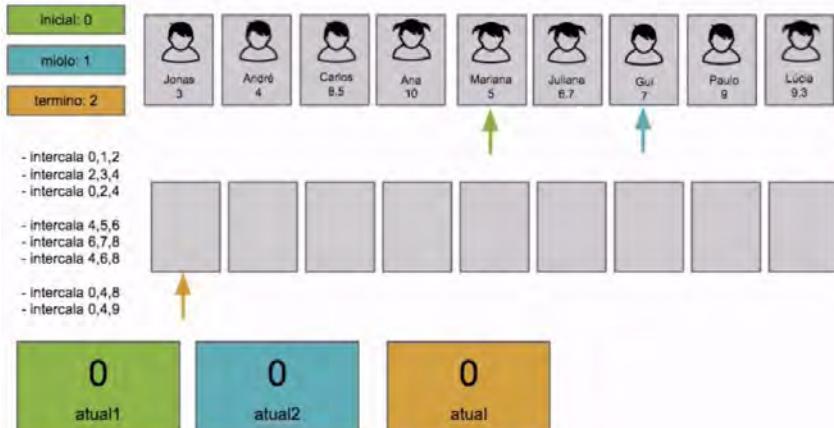


Figura 10.38: Intercalando diversas vezes seguidas — passo 36

Agora que temos tanto os quatro elementos do lado esquerdo como os quatro do lado direito ordenados, poderemos intercalar os oito.

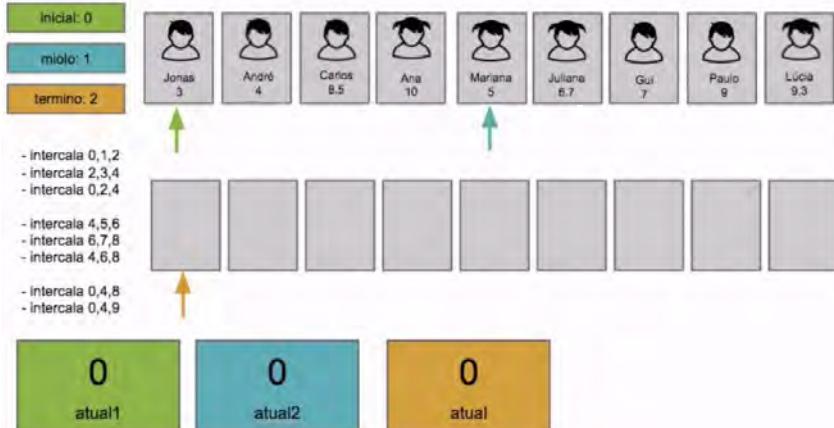


Figura 10.39: Intercalando diversas vezes seguidas — passo 38

Primeiramente, entre Jonas e Mariana, qual é o menor? O Jonas. Vamos movê-lo.

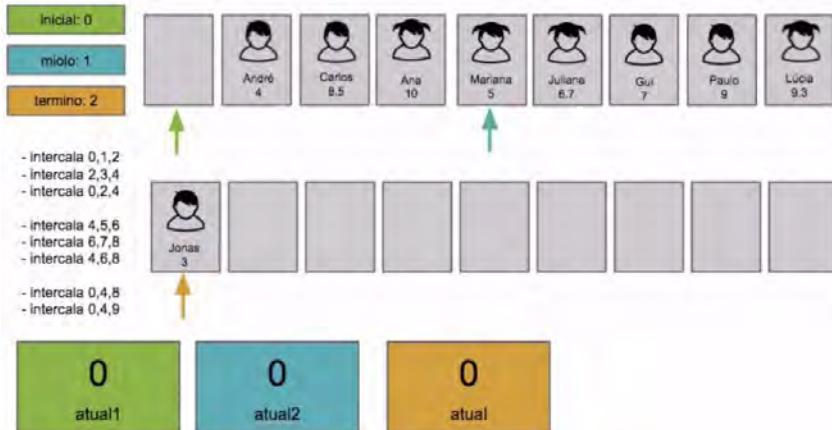


Figura 10.40: Intercalando diversas vezes seguidas — passo 39

Entre André e Mariana, qual é o menor? O André. Vamos movê-lo.

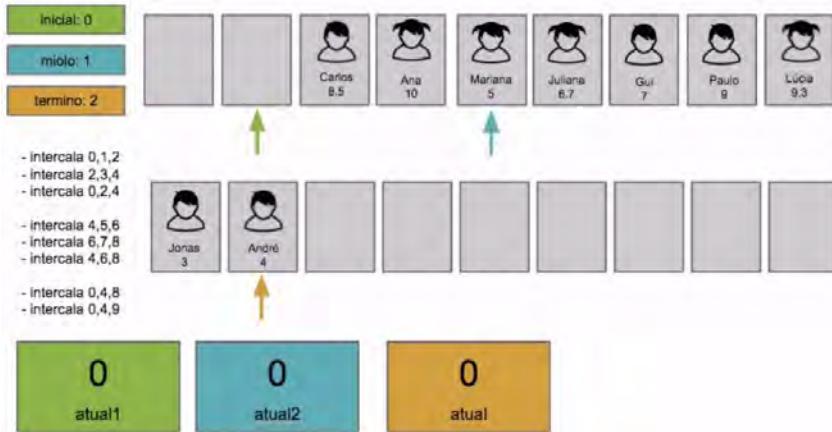


Figura 10.41: Intercalando diversas vezes seguidas — passo 40

Qual é o menor elemento, Carlos ou Mariana? A Mariana. Vamos movê-la.

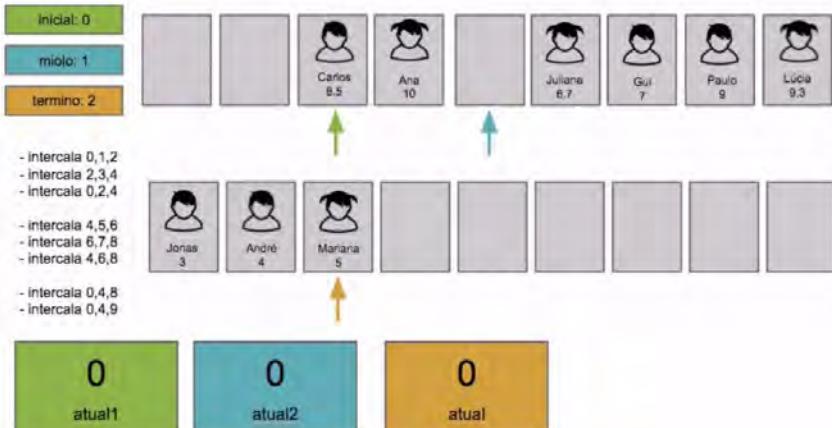


Figura 10.42: Intercalando diversas vezes seguidas — passo 41

Depois, Carlos ou Juliana? A Juliana. Vamos movê-la.

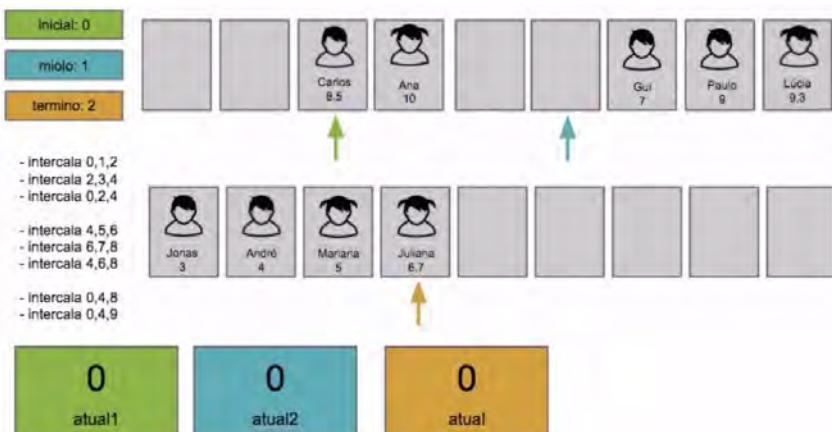


Figura 10.43: Intercalando diversas vezes seguidas — passo 42

Carlos ou Gui, qual é o menor? O Gui. Vamos movê-lo.

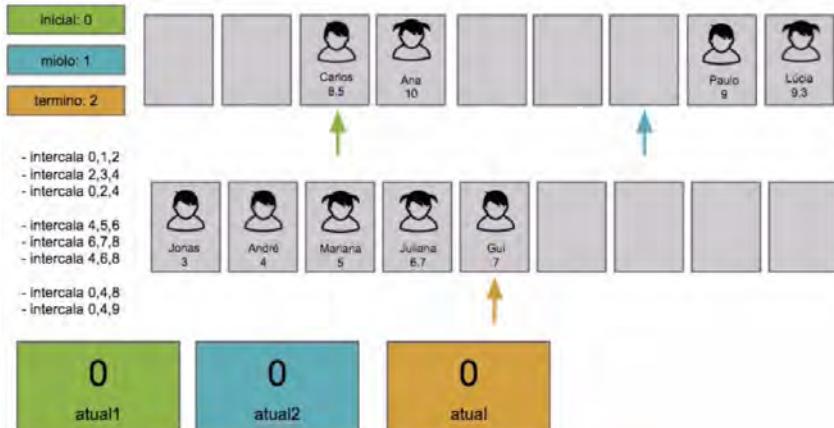


Figura 10.44: Intercalando diversas vezes seguidas — passo 43

Entre Carlos e Paulo? O Carlos. Vamos movê-lo.

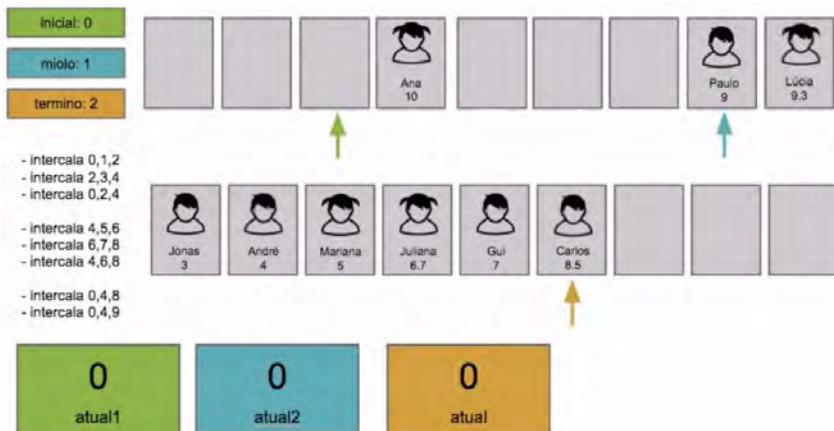


Figura 10.45: Intercalando diversas vezes seguidas — passo 44

Ana ou Paulo, qual é o menor? O Paulo. Vamos movê-lo.

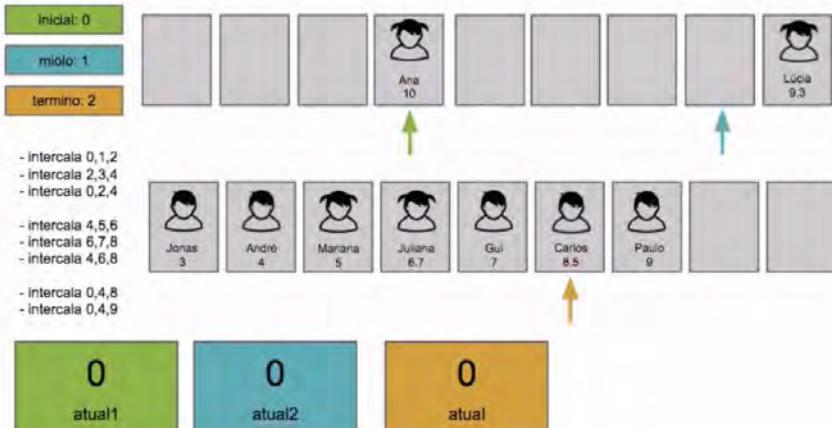


Figura 10.46: Intercalando diversas vezes seguidas — passo 45

Sobrou a Ana da parte esquerda. Ela descerá para o novo array.

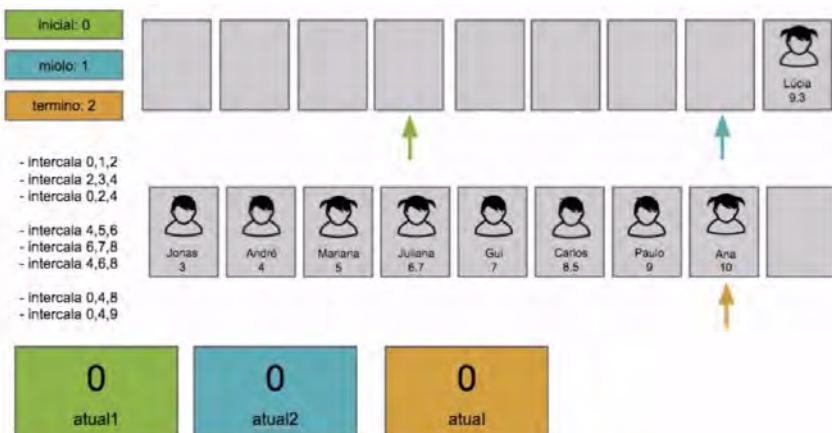


Figura 10.47: Intercalando diversas vezes seguidas — passo 46

Depois vamos subir com todos os elementos da nossa intercalação. Os oito itens estarão ordenados.

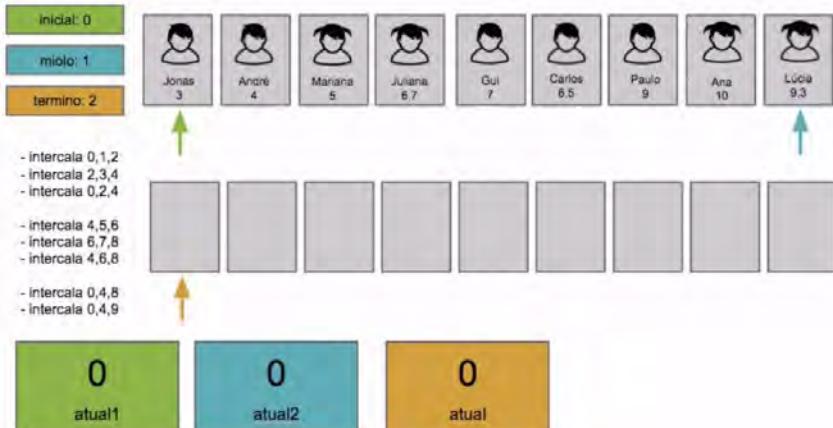


Figura 10.48: Intercalando diversas vezes seguidas — passo 47

Faltou um elemento na ordenação. Vamos ordená-lo juntamente com os outros. Primeiro desceremos o Jonas.

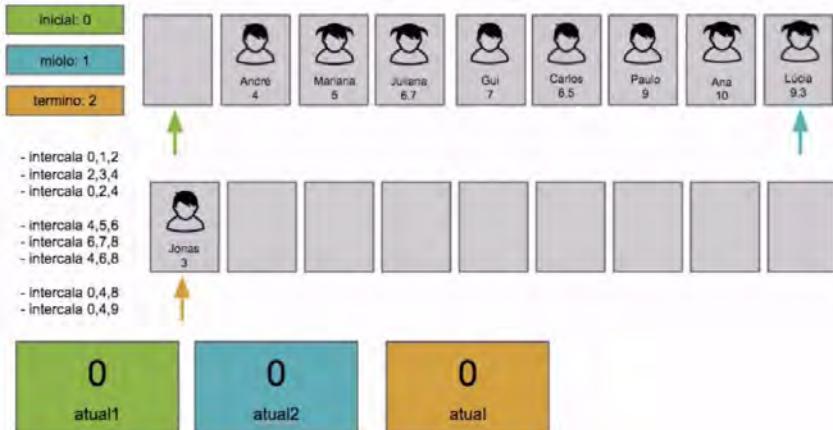


Figura 10.49: Intercalando diversas vezes seguidas — passo 48

O André.

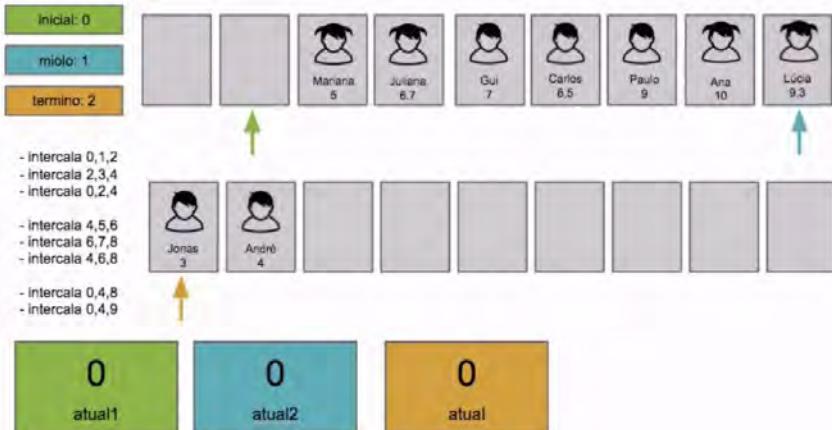


Figura 10.50: Intercalando diversas vezes seguidas — passo 49

A Mariana.

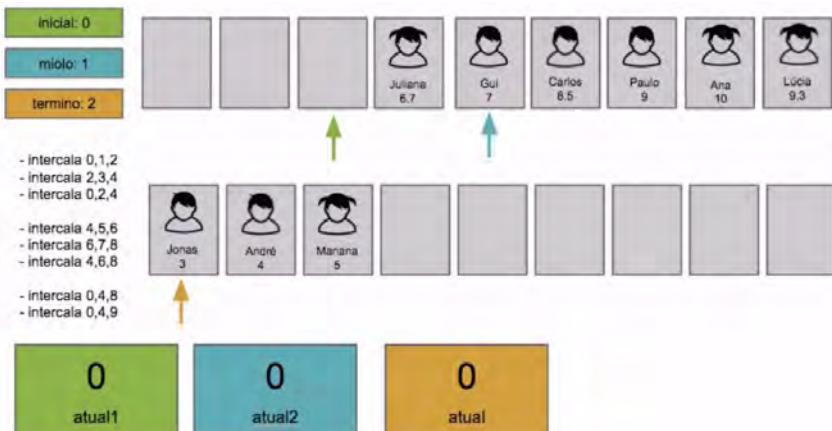


Figura 10.51: Intercalando diversas vezes seguidas — passo 50

Entre a Juliana e Lúcia, qual é a menor? A Juliana.

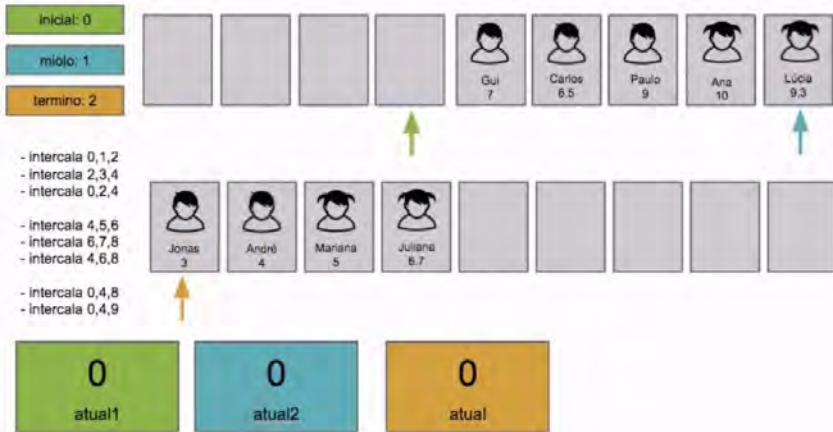


Figura 10.52: Intercalando diversas vezes seguidas — passo 51

Gui ou Lúcia? O Gui.

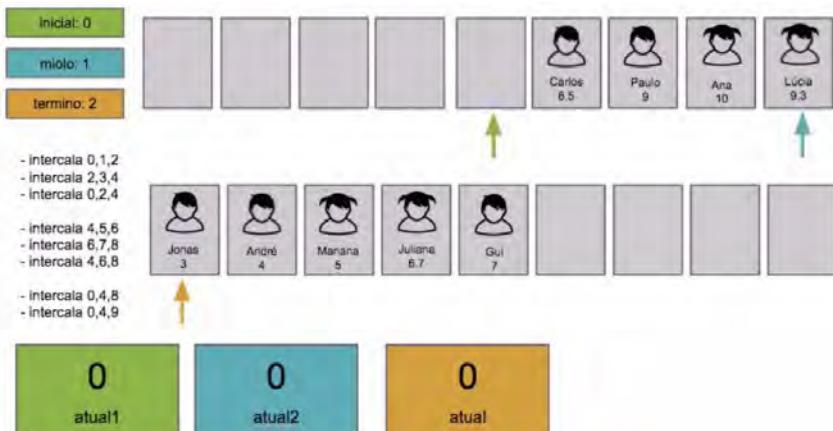


Figura 10.53: Intercalando diversas vezes seguidas — passo 52

Carlos ou Lúcia? O Carlos.

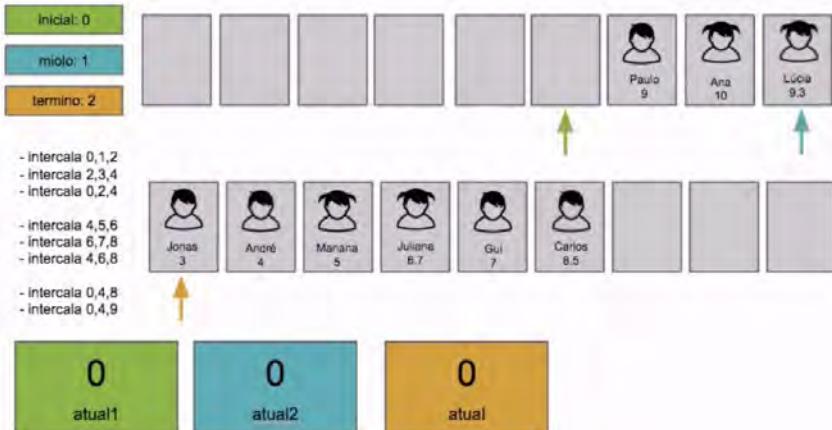


Figura 10.54: Intercalando diversas vezes seguidas — passo 53

Paulo ou Lúcia? O Paulo.

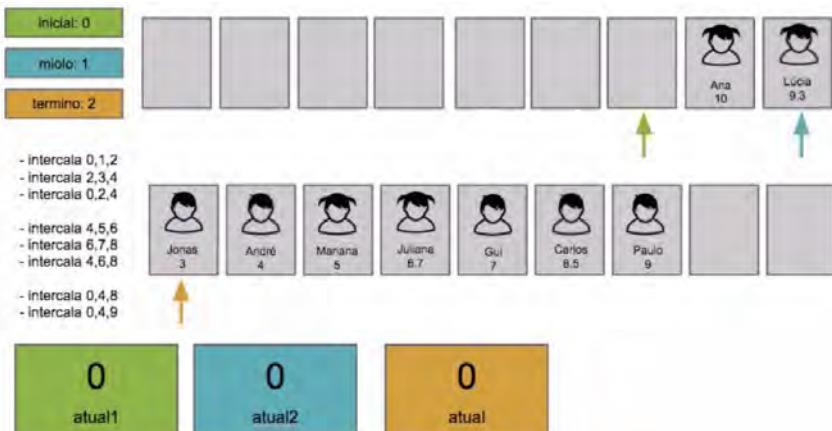


Figura 10.55: Intercalando diversas vezes seguidas — passo 54

Ana ou Lúcia? A Lúcia.

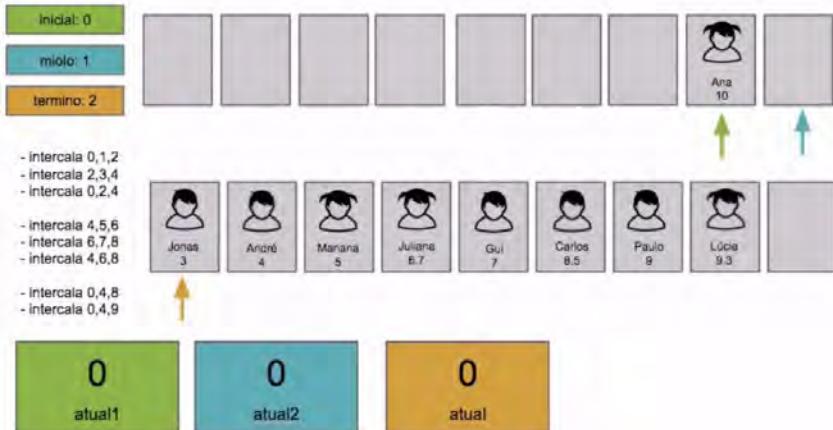


Figura 10.56: Intercalando diversas vezes seguidas — passo 55

Sobrou a Ana, vamos copiá-la para o novo array.

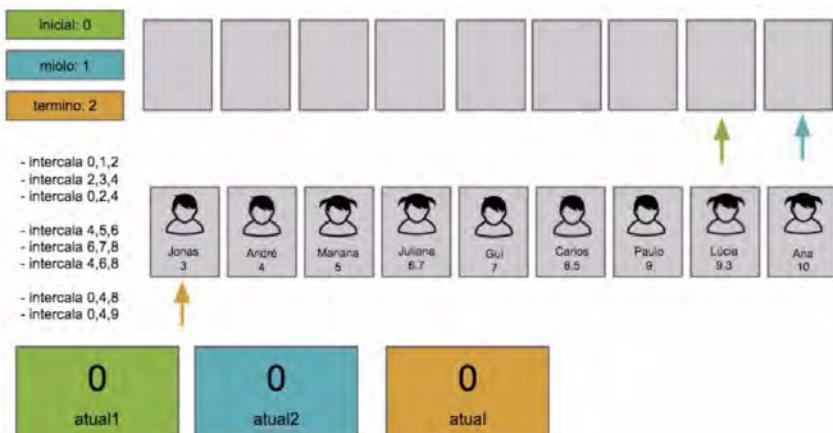


Figura 10.57: Intercalando diversas vezes seguidas — passo 56

Agora todos os elementos voltarão para o array de origem.

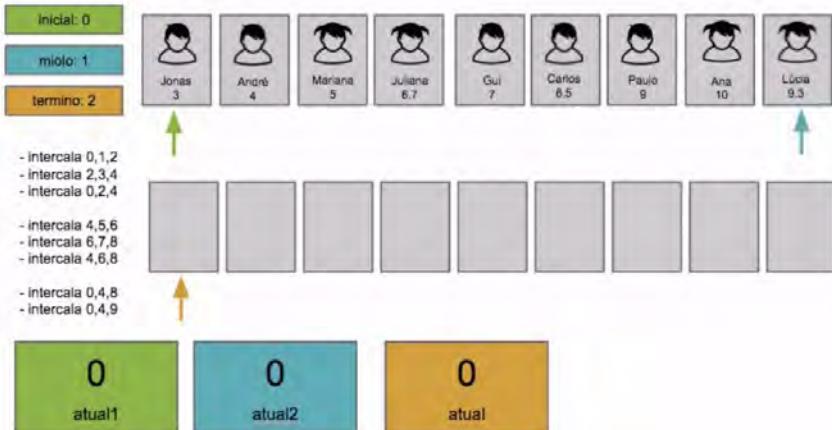


Figura 10.58: Intercalando diversas vezes seguidas — passo 57

Responda: como ficou o nosso array após executarmos o algoritmo assim?

O que foi feito? Dividimos nosso problema em pequenas intercalações, para depois seguirmos para as grandes. O resultado foi o array ordenado.

10.1 INTERCALANDO ATÉ O FIM

Nós vimos que se chamarmos uma série de intercalações, das menores para as maiores, conseguimos ordenar o nosso array. Então, como poderíamos implementar um algoritmo de ordenação? Caso me peçam para ordenar do 0 até 9, eu vou responder: *"É melhor ordenar primeiro do 0 até 4, e depois do 5 ao 9. Assim poderemos intercalar os elementos".*

Mas quando tivermos de intercalar de 0 até 4, o que precisaremos fazer? Podemos dividir o problema, ir do 0 até 2, de 2 até 4, e então intercalaremos. Quando formos ordenar do 0 até 2, vamos ordenar um primeiro elemento, depois outro, e então, intercalamos os dois.

Ordenar um ou dois itens é fácil. O algoritmo consegue intercalar os primeiros elementos. Após intercalarmos os dois primeiros, intercalaremos os dois seguintes. E continuaremos com o processo de ordenação.

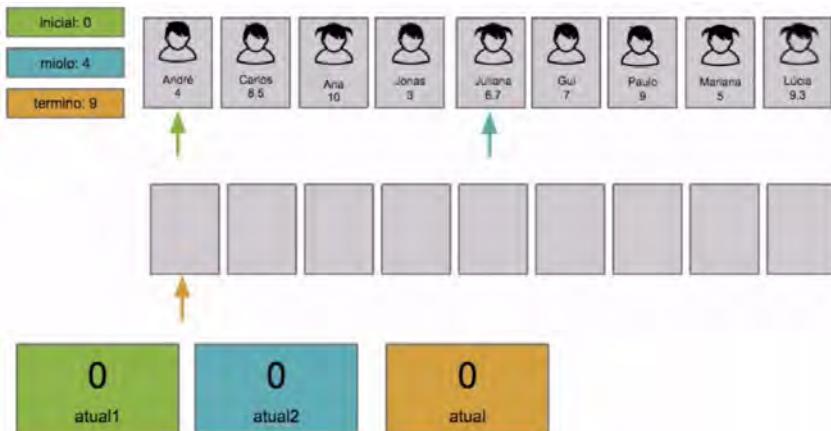


Figura 10.59: Intercala um trecho — passo 1

Nós vamos dividir em duas partes o array. O fato de termos um número ímpar de elementos não é um problema.

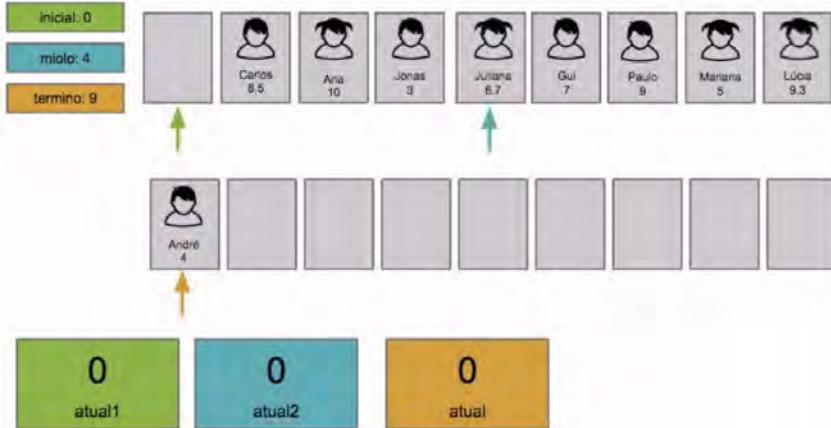


Figura 10.60: Intercala um trecho — passo 2

Vamos ordenar as partes da esquerda e da direita, para então intercalarmos todos os elementos. Como ordenar cada parte? Também dividiremos o problema.

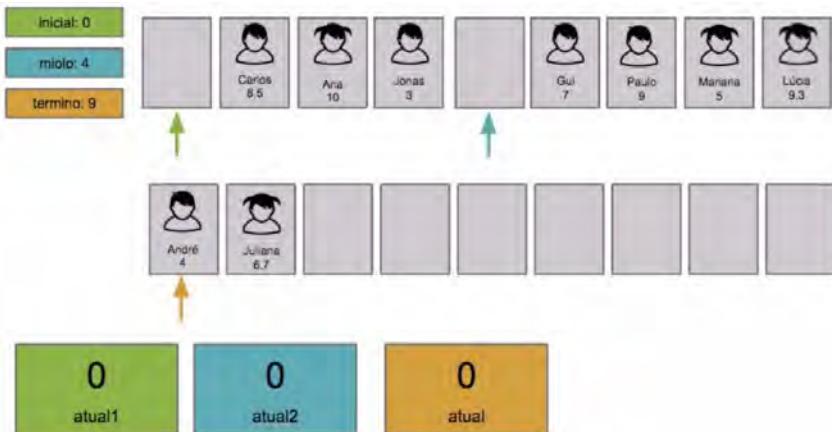


Figura 10.61: Intercala um trecho — passo 3

Depois vamos intercalar os elementos. Isto significar que o próprio **ordenar** está chamando o **ordenar**. Até ficar uma ordenação bem pequena. Vamos ordenar o primeiro elemento, depois o segundo e, então, intercalamos os dois — algo que já sabemos fazer.

Precisamos seguir ordenando os outros elementos, vamos intercalar os dois seguintes.

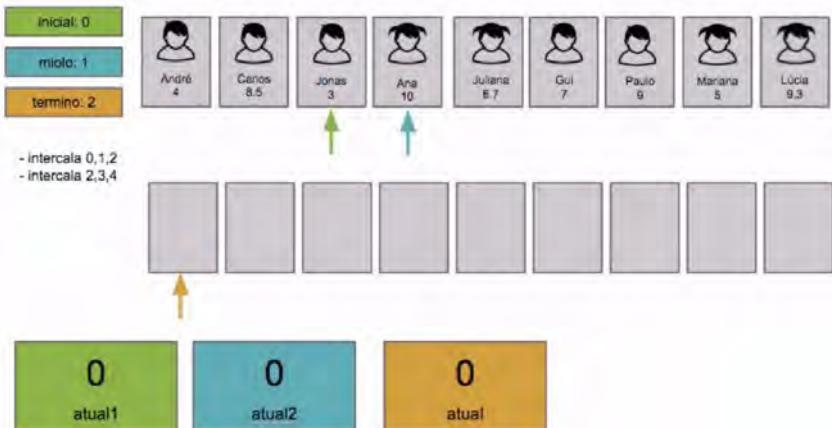


Figura 10.62: Intercalando diversas vezes seguidas

Agora que temos os dois primeiros e os dois últimos ordenados, vamos intercalar os quatro? Então, a minha função de ordenação pede que eu divida no meio a lista com os elementos. Depois chame a **ordenação** para os elementos da esquerda, e depois para a direita. Agora que temos os dois trechos ordenados, intercalaremos todos. Vamos ver o que acontece?

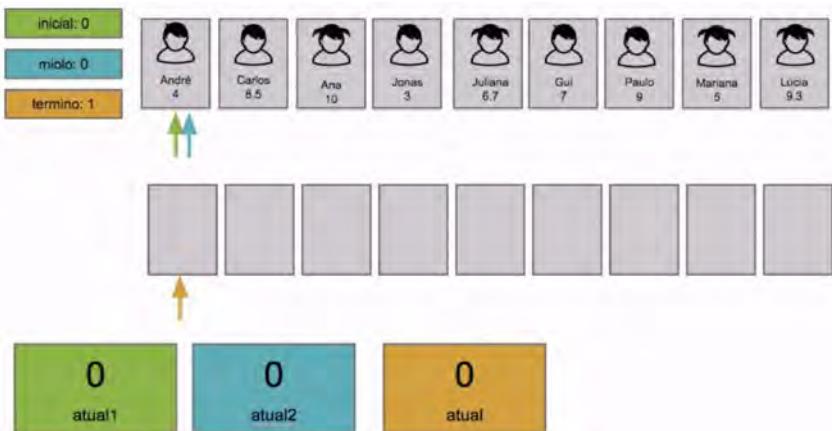


Figura 10.63: Intercalando até o fim — passo 1

Temos o nosso código. No começo, vamos chamar a minha função de ordenação do 0 até 9.

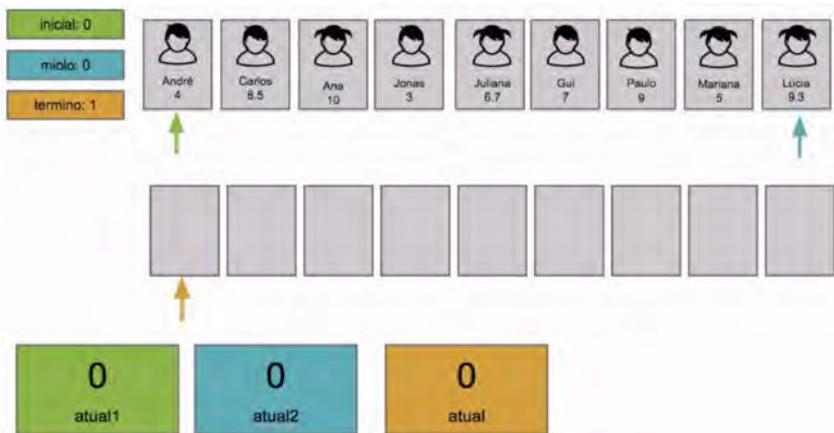


Figura 10.64: Intercalando até o fim — passo 2

Se dividir nove (a quantidade total de elementos), precisaremos de um resultado inteiro. No caso, será o quatro. Então, ordenaremos do 0 até 4.

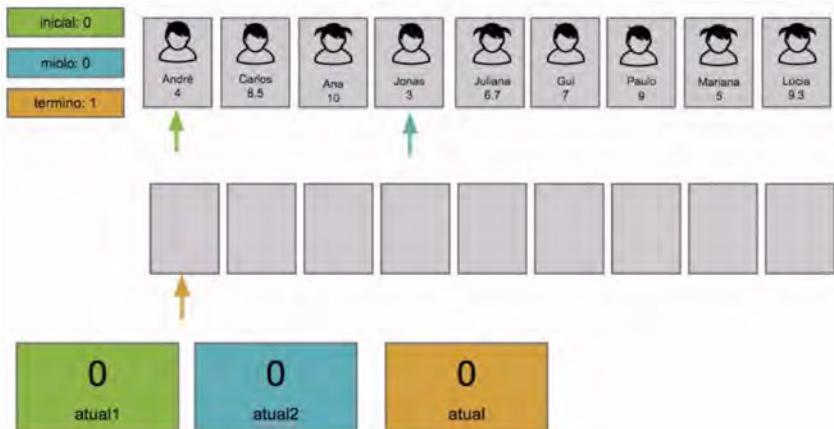


Figura 10.65: Intercalando até o fim — passo 3

Depois, o código vai pedir para dividirmos novamente os

elementos. Se temos quatro elementos, dividiremos do 0 até 2.

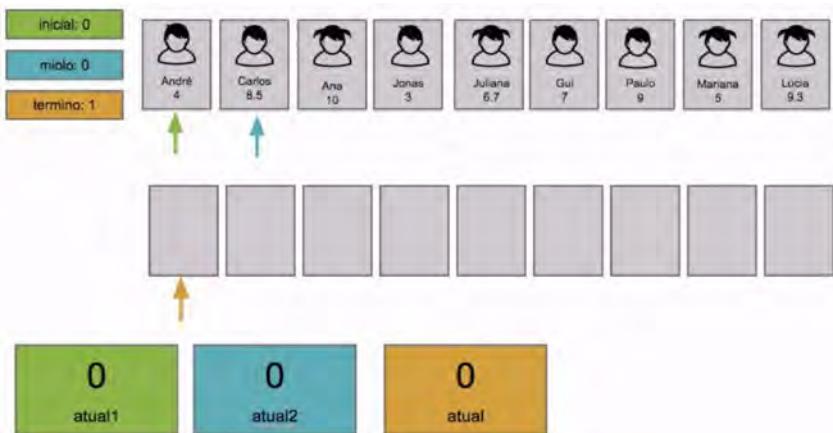


Figura 10.66: Intercalando até o fim — passo 4

Quando tivermos dois elementos, teremos de dividir por dois.

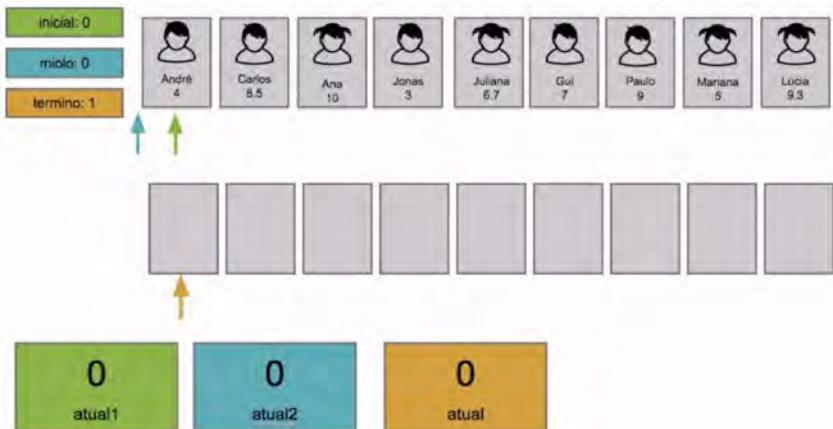


Figura 10.67: Intercalando até o fim — passo 5

É possível repetir o processo várias vezes. Mas se os valores forem muito pequenos, em algum momentos teremos de parar. Este algoritmo de ordenação só vai dividir no meio se tiver mais de um elemento.

Se pedirmos para ordenar **um** elemento, o resultado será imediato. E ordenar **dois**? Mandaremos ordenar um e depois o outro, então intercalaremos. Para ordenar **três** elementos, mandaremos ordenar os dois, depois o item restante, logo intercalaremos os três.

Para organizar os dois primeiros, precisaremos chamar o `ordena` para dois elementos — sabemos que isto funciona. Como faremos para ordenar **quatro** elementos? Chamaremos o `ordena` para os dois primeiros, e depois para os dois últimos, então, intercalaremos.

Como faremos com **cinco** elementos? Chamaremos o `ordena` para dois, depois para três. Logo intercalaremos as duas partes. Como ordenaremos **seis** elementos? Chamaremos o `ordena` para os três primeiros, depois para os três últimos. No fim, intercalaremos todos. E para ordenar **sete**? Chamaremos o `ordena` para três, em seguida para quatro — sabemos que funciona — e intercalaremos.

Basta sabermos que o `intercala` funciona para um e dois elementos, que ele funcionará para três, quatro, cinco ou para qualquer número. E então, teremos nossa ordenação.

O que a função de ordenação faz? Se trabalhamos com número único, ele já está ordenado. Se temos dois, ordenamos o da esquerda e o da direita, em seguida, intercalamos. Na verdade, só vamos intercalar os dois. Se temos três elementos, quebramos em uma parte com dois números, e outra com apenas um.

Fazendo as divisões, tudo passa a funcionar. Esta é a nossa função de **ordenação** e o código que vamos implementar.

10.2 INTERCALANDO VALORES INVÁLIDOS

EM JAVA

Intercalar é um processo interessante, nós trabalhamos com dois arrays já ordenados e os intercalamos na ordem correta. Porém, para intercalarmos todos os resultados de provas do Enem, primeiro eles precisam estar ordenados em partes menores. Isto significa que eles precisam ter sido distribuídos entre diversas pessoas que ordenaram partes pequenas.

Enquanto as partes menores não estiverem ordenadas, não temos o que intercalar. Se temos um array com 1 milhão de provas, vamos dividi-las em duas partes com 500 mil, e duas pessoas vão ordená-las. Só após esta ação, poderemos intercalar. Temos um problema grande.

Se quiséssemos ordenar as provas sem ter nada ordenado, isto é, se quisermos ordenar um array e não termos nada ordenado, apenas chamar o `intercala` não vai resolver. Vamos testar?

Vamos copiar o código do `TestaIntercalaEmUmArray` e daremos o nome de `TestaOrdenacaoAoIntercalar`. Teremos um array que era igual ao que já usamos, mas misturaremos os elementos.

```
public static void main(String[] args) {
    Nota[] notas = {
        new Nota("andre", 4),
        new Nota("carlos", 8.5),
        new Nota("ana", 10),
        new Nota("jonas", 3),
        new Nota("juliana", 6.7),
        new Nota("guilherme", 7),
        new Nota("paulo", 9),
        new Nota("mariana", 5),
        new Nota("lucia", 9.3)
    };
    Nota[] rank = intercala(notas, 0, 4, notas.length);
    for(Nota nota : rank) {
```

```
        System.out.println(nota.getAluno() + " " + nota.getValor())
    ;
}

}
```

Misturamos todos os elementos. Se chamarmos o `intercala`, o resultado provavelmente não será o correto.

```
andre 4.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
mariana 5.0
lucia 9.3
ana 10.0
jonas 3.0
```

A ordem não ficou totalmente correta. Para intercalar, precisávamos ter dois arrays já intercalados. Eles poderiam ser de tamanho 1, ou um ser de tamanho 0 e o outro 1. Ambos poderiam ser de tamanho 3. Mas era preciso que cada pedaço do array já estivesse ordenado, para podermos intercalar.

Como faremos para intercalar já ordenando? O que poderemos fazer?

Usando o próprio array

O que queremos fazer agora é tentar ordenar um array inteiro, chamando o `intercala`. Já vimos que, se o array é grande e os dois trechos não estão ordenados, o `intercala` não vai funcionar. Observe o resultado do nosso array com os elementos misturados.

```
andre 4.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
mariana 5.0
lucia 9.3
```

```
ana 10.0
jonas 3.0
```

O algoritmo me devolve os elementos desordenados. Então, ele precisa ter dois trechos ordenados. Só assim ele consegue intercalar o array inteiro e manter ordenado. Se não for assim, não funcionará.

Vamos dar uma olhada na função `intercala()`. Ela intercala os valores e devolve o próprio array.

```
while(atual2 < termino) {
    resultado[atual] = notas[atual2];
    atual2++;
    atual++;

}

return notas;
```

Se o `intercala` devolve o próprio array, mandaremos que o programa não retorne mais nada (`private static void`).

```
private static void intercala(Nota[] notas, int inicial, int m10
o, int termino) {
    Nota[] resultado = new Nota[termino - inicial];
```

E depois, chamaremos o `intercala`:

```
intercala(notas, 0, 4, notas.length);
for(Nota nota : rank) {
    System.out.println(nota.getAluno() + " " + nota.getValor());
}

}
```

No momento de imprimir, apresentaremos as `notas`. Vamos escrever isto no `for`:

```
for(Nota nota : notas) {
```

Apenas refatoramos porque não tínhamos razão para ficar criando e devolvendo o próprio array. Vamos usar o próprio array e

ficará mais simples o código. No entanto, se rodarmos, veremos que ele continua não funcionando.

```
andre 4.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
mariana 5.0
lucia 9.3
ana 10.0
jonas 3.0
```

Ele deixou de devolver o array, mas continuamos usando o mesmo. E por isso, ele continua não funcionando e os elementos não ficam ordenados. `intercala` de 0 a 4, com estas notas, não é o suficiente.

Como podemos usar o `intercala` para ordenar uma lista que ainda não está ordenada?

10.3 INTERCALANDO DIVERSAS VEZES

Nós vimos que chamar o `intercala` para ordenar um array que não está ordenado não funciona. Porém, vimos também outras ideias. Em vez de chamar o `intercala` do 0 até o 4, pediremos para intercalar apenas dois elementos.

```
intercala(notas, 0, 4, notas.length);
for(Nota nota : notas) {
    System.out.println(nota.getAluno() + " " + nota.getValor());
```

Isto funciona, assim como sabemos que intercalar apenas um único elemento também funciona (uma parte tem um elemento, a outra nenhum, já está ordenado). Mas e se mandarmos ordenar dois elementos?

Cada um já estará ordenado, pois, ou vamos trocá-los de posição, ou eles permaneceram como estão. `intercala` com dois

elementos também funciona.

Então, não vamos ordenar todos, como se fosse uma grande ordenação. Pediremos para intercalar somente o primeiro e o segundo elemento.

```
intercala(notas, 0, 1, 2);
```

Isto é, queremos que intercale entre 0 e 2 (excluindo o 2), e o miolo será igual a 1. Queremos intercalar só os dois itens. Qual será o resultado?

```
andre 4.0
carlos 8.5
ana 10.0
jonas 3.0
juliana 6.7
guilherme 7.0
paulo 9.0
mariana 5.0
lucia 9.3
```

Ele intercalou os dois primeiros e deixou o restante nas posições em que estavam. Nós já tínhamos testado este código. Ao intercalar dois elementos que já estavam em ordem, ele mantinha os dois como estavam.

Vamos tentar intercalar os próximos dois elementos: a Ana e o Jonas. Vou especificar que intercalaremos a partir da posição 2, o miolo é igual a 3, vamos até a posição 4.

```
intercala(notas, 0, 1, 2);
intercala(notas, 2, 3, 4);
```

Ao intercalarmos os elementos, o algoritmo trocou a posição do Jonas com a da Ana.

```
andre 4.0
carlos 8.5
jonas 3.0
ana 10.0
juliana 6.7
```

```
guilherme 7.0  
paulo 9.0  
mariana 5.0  
lucia 9.3
```

Antes o Jonas e a Ana estavam fora de ordem. Agora, tanto do 0 até 2 como do 2 até 4 já estão ordenados. Isto significa que já podemos intercalar do 0, usando o 2 como `miolo`, até o 4.

```
intercala(notas, 0, 1, 2);  
intercala(notas, 2, 3, 4);  
intercala(notas, 0, 2, 4);
```

Se rodarmos novamente, o resultado será:

```
jonas 3.0  
andre 4.0  
carlos 8.5  
ana 10.0  
juliana 6.7  
guilherme 7.0  
paulo 9.0  
mariana 5.0  
lucia 9.3
```

Os quatros elementos foram ordenados: Jonas, André, Carlos e Ana. Por quê? Porque intercalamos os dois primeiros e os dois seguintes. Como eles já tinham sido ordenados, era possível intercalar os quatro. O resultado devolveu os elementos ordenados, fazendo primeiro uma intercalação pequena e, depois, seguindo para a intercalação maior. E assim, parte do meu array foi ordenado.

Se ao intercalarmos uma parte funcionou, vamos continuar com os dois próximos, 5 e 6. Vamos intercalar a partir do 4, porque ele ainda não foi intercalado.

```
intercala(notas, 0, 1, 2);  
intercala(notas, 2, 3, 4);  
intercala(notas, 0, 2, 4);  
intercala(notas, 4, 5, 6);
```

Depois, repetiremos o processo para 6, 7 e 8.

```
intercala(notas, 6, 8);
```

Ao testar, o resultado será:

```
jonas 3.0
andre 4.0
carlos 8.5
ana 10.0
juliana 6.7
guilherme 7.0
mariana 5.0
paulo 9.0
lucia 9.3
```

Observe os quatro elementos: Juliana, Guilherme, Mariana e Paulo. O que podemos fazer agora? Intercalar. Então vamos intercalar as notas do 4 parando no 6, e depois seguimos até o 8.

```
intercala(notas, 4, 6, 8);
```

O resultado será:

```
jonas 3.0
andre 4.0
carlos 8.5
ana 10.0
mariana 5.0
juliana 6.7
guilherme 7.0
paulo 9.0
lucia 9.3
```

Os quatro elementos foram ordenados corretamente: Mariana, Juliana, Guilherme e Paulo. Nossa array está ficando organizado.

Agora os quatro primeiros itens já estão ordenados, assim como os quatro seguintes. Isto significa que podemos intercalar os elementos. É o que faremos.

```
intercala(notas, 0, 1, 2);
intercala(notas, 2, 3, 4);
intercala(notas, 0, 2, 4);
intercala(notas, 4, 5, 6);
intercala(notas, 6, 7, 8);
intercala(notas, 4, 6, 8);
```

```
intercala(notas, 0, 8);
```

Vamos intercalar do 0 até 4, que estavam ordenados, e seguiremos até 8, uma parte que também já foi ordenada. Estamos intercalando os blocos de elementos que estão organizados. Vamos testar se está funcionando bem?

```
jonas 3.0
andre 4.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
ana 10.0
lucia 9.3
```

Do Jonas até a Ana, os elementos estão ordenados. Faltou apenas o último. Como restou apenas um, basta intercalarmos a Lúcia com os demais, porque ela está também ordenada. Então, intercalaremos do 0 até 8, em que todos estão ordenados, e depois até 9, que igualmente está ordenado.

```
intercala(notas, 0, 8, 9);
```

Vamos testar novamente?

```
jonas 3.0
andre 4.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
lucia 9.3
ana 10.0
```

Agora todos estão ordenados! Observe que conseguimos fazer diversas invocações ao `intercala` e, com isto, ordenar o nosso array inteiro. Ele é capaz de tal ação.

10.4 INTERCALANDO PASSO A PASSO

Nós vimos que, se chamarmos o `intercala` apenas uma vez, não ordenaremos o array. Mas se formos chamando de uma maneira inteligente, conseguiremos quebrar o array grande em pedaços menores, e começaremos a ordenar pelas partes pequenas. Ao iniciar assim, o `intercala` seguirá ordenando até todos os elementos estarem organizados.

```
intercala(notas, 0, 1, 2);
intercala(notas, 2, 3, 4);
intercala(notas, 0, 2, 4);
```

O que fizemos aqui? Nós primeiro ordenamos do 0 até 2 — foram dois elementos. Depois do 2 até 4, mais dois elementos. Então, ordenamos do 0 até o 4, quatro elementos.

```
intercala(notas, 4, 5, 6);
intercala(notas, 6, 7, 8);
intercala(notas, 4, 6, 8);
```

Seguimos ordenando mais dois elementos. E dois seguintes. Depois ordenamos de 4 até 8, os quatro elementos.

```
intercala(notas, 0, 4, 8);
```

Ordenamos os oito elementos, de 0 até 8.

```
intercala(notas, 0, 8, 9);
```

Depois, ordenamos o último que sobrou com os restantes.

Nós fomos intercalando cada pedaço. Para intercalarmos um array grande, precisamos dividir em dois pedaços menores. Depois, intercalamos as partes ordenadas. Mas para fazer isto, as divisões precisam ser pequenas o suficiente. Enquanto não for assim, é preciso quebrar o array até que o `intercala` funcione.

Em vez de mandarmos intercalar todos elementos de uma vez, precisaremos dividi-los em blocos menores e, então, poderemos intercalá-los.

10.5 O MERGE SORT

Nós vimos que não podemos intercalar todos os elementos de um array grande. Antes, precisamos quebrar a quantidade de itens em partes menores, e depois intercalar os elementos. Então, se quero criar daqui o método de ordenação, clico em *Extract Method* e nomeio o método que vai ordenar o array como `ordena`.

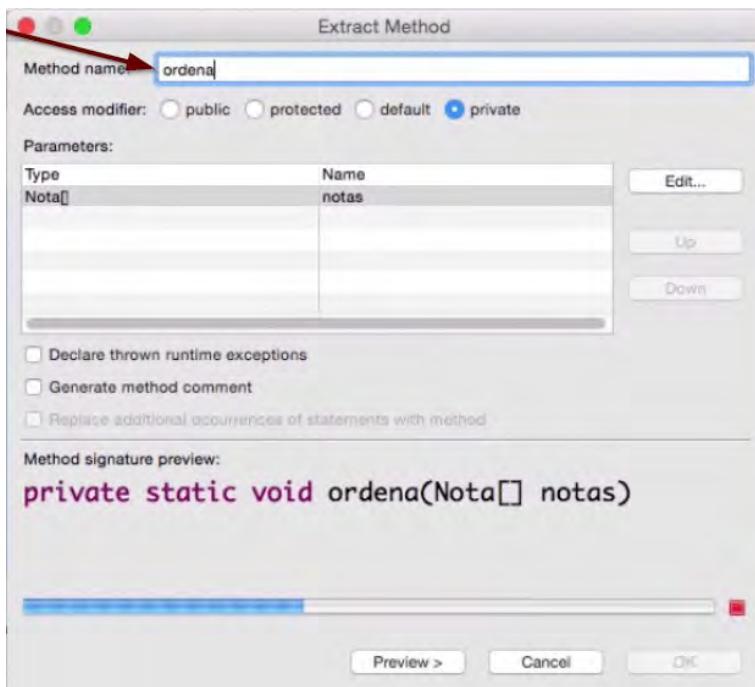


Figura 10.68: Método que ordena o meu array

Se queremos ordenar o array em vez o `intercala` dessa maneira:

```
intercala(notas, 0, 1, 2);
intercala(notas, 2, 3, 4);
intercala(notas, 0, 2, 4);

intercala(notas, 4, 5, 6);
intercala(notas, 6, 7, 8);
```

```
intercala(notas, 4, 8);
intercala(notas, 0, 4, 8);
intercala(notas, 0, 8, 9);
```

Intercalando todos os elementos, temos de intercalar partes pequenas do array. Quando vamos ordenar um array, precisamos indicar a posição inicial (`inicial`) e a final (`termino`). Vamos escrever isto no código:

```
private static void ordena(Nota[] notas, int inicial, int termino)
{
    intercala(notas, 0, 8, 9);
}
```

Vamos especificar o `inicial` e o `termino`.

```
intercala(notas, inicial, 8, termino);
```

E quando mandamos ordenar, precisamos indicar do 0 até `notas.length`.

```
ordena(notas, 0, notas.length);
```

Queremos ordenar do `inicial` até o `termino`. Não podemos fazer assim. Temos de chamar para parte menores. Isto significa que vamos dividir no meio as partes maiores (`int inicial, int termino`).

Por exemplo, se temos muitas provas para corrigir, dividimos o total em duas partes e pedimos para outra pessoa corrigir metade, e depois as juntamos em um único monte. O que vamos fazer? Encontrar o `meio` do array, que será o `inicial` somado ao `termino` dividido por 2.

```
private static void ordena(Nota[] notas, int inicial, int termino)
{
    int meio = (inicial + termino) / 2;
}
```

Ao dividirmos no meio, teremos dois arrays menores, que vai do `inicial` até o `meio`, e do `meio` até o `termino`. Nós ordenaremos estes dois trechos. No código, determinaremos para que ele ordene do `inicial` até o `meio`. Depois, ordenaremos do `meio` até o `termino`. Após termos os dois trechos ordenados, vamos intercalar os elementos começando pelo `inicial`, passando pelo `meio` e indo até o `termino`.

```
ordena(notas, inicia, meio);
ordena(notas, meio, termino);
intercala(notas, inicial, meio, termino);
```

O que estamos fazendo? Se temos de ordenar as dez provas, vamos quebrar o total no meio, para ordenar primeiro a metade da esquerda e depois da direita. Quando as duas partes estiverem ordenadas, intercalamos os elementos e todos estarão em ordem.

Quando quebramos no meio, mandamos ordenar a parte da esquerda e da direita, e depois intercalamos os elementos. Chamaremos então o `ordena` para organizar a primeira metade, que chama o `ordena` para organizar a metade desta primeira metade, que chama o `ordena` para organizar a metade da metade da primeira metade, infinitamente.

Ao testarmos, veremos que o programa dará mensagem de erro *Stack Overflow*. Nossa pilha de execução estoura, porque o `ordena` chama o `ordena`, e isto se repetirá infinitamente. Um método que se invoca infinitamente de maneira recursiva por padrão em algum momento estourará a pilha de execução, que é o que aconteceu. Ou seja, não funciona.

O número `inicial`, do `meio` e do `termino` vão diminuindo constantemente até que se repetem. Vamos ver quais são estes números com um `System.out`. Veremos o `inicial`, o `termino` e o `meio` aparecer.

```
System.out.println(inicial + " " + termino + " " + meio);
```

```
ordena(notas, inicial, meio);
```

Ao rodarmos, aparecerá uma mensagem de erro, mas no alto da tela teremos os números aparecendo:

```
0 9 4  
0 4 2  
0 2 1  
0 1 0  
0 0 0  
0 0 0  
0 0 0  
0 0 0  
0 0 0  
0 0 0  
0 0 0  
0 0 0  
0 0 0  
0 0 0  
0 0 0
```

Do 0 até 9, a metade é 4 (porque ele arredonda o resultado), e do 0 até 4, a metade é 2. Do 0 até 2, a metade é 1. Do 0 até 1, a metade é 0 (outra aproximação). Então começam os 0. Metade de 0 é o próprio 0. Em algum momento, temos de parar de executar o código.

Quando trabalhamos com dois elementos, fará sentido ordenar. No entanto, se temos apenas um elemento, não precisaremos ordenar. Então a quantidade de elementos é o `termo` menos o `inicial`. Se (`if`) a quantidade de elementos for menor do que 1, fará sentido ordenar. Se a quantidade de elemento for 1, já estará ordenado. Esta é a sacada que estava faltando.

Vamos testar? Ao rodarmos o programa, ele vai imprimir o seguinte resultado:

```
6 9 7  
7 9 8  
jonas 3.0  
andre 4.0  
mariana 5.0  
juliana 6.7  
guilherme 7.0  
carlos 8.5
```

```
paulo 9.0  
lucia 9.3  
ana 10.0
```

Os elementos estão ordenados. Vamos revisar? Determinamos que vamos ordenar do 0 até o fim (`notas.length`).

```
ordena(notas, 0, notas.length)
```

O que isso significa? Ordenar do **0 até 9** será o mesmo que ordenar do **0 até 4**, ou do **4 até 9**, e intercalar. Está especificado no código que escrevemos. Ordene uma parte, depois a outra e intercalamos. Porém, pedimos para quebrar a quantidade de elementos na metade, depois na metade, repetindo o processo infinitamente, até que, em um determinado momento, a pilha de execução estourou.

Quando não podemos mais dividir no meio, por exemplo, quando só temos um elemento, devemos parar a execução. Ele já está ordenado e não é preciso fazer mais nada. Mesmo com um array de apenas um elemento, o código vai funcionar, porque a ordenação é desnecessária.

Ao pedirmos para o programa ordenar o array inteiro, ele quebrará a quantidade total na metade e ordenará os elementos. A ação será repetida até termos apenas um elemento. Quando isto acontece, ele vai voltar para o elemento anterior e intercalará com o que sobrou. O processo vai se repetir até que o array esteja totalmente ordenado.

Isto significa que ele fez todas as chamadas de `intercala` , mas que ele próprio resolve o problema. Ele tem um array, divide em duas partes, ordena cada uma e, no fim, aplica uma solução que junta o resultado. Ao dividir o problema em dois pedaços menores, ele se torna mais simples. Então, o programa manda intercalar todos os elementos.

10.6 O PRÓXIMO DESAFIO: OUTRA MANEIRA DE ENTENDER O QUE É ORDEM

Vimos como implementar um algoritmo que divide nosso problema em dois e resolve para cada um dos dois, finalizando o trabalho de juntá-los em um só. Isso foi possível por lançarmos um olhar diferente na maneira de ordenar: dividimos o problema e conquistamos.

Se olharmos o que é uma ordenação de outros pontos de vista, chegaremos em outros algoritmos, e é o que faremos em breve. Veremos como encontrar a posição absoluta de um único elemento e, a partir daí, criaremos um dos algoritmos mais rápidos de ordenação no mundo real.

CAPÍTULO 11

ENCONTRANDO A POSIÇÃO RELATIVA DE UM ELEMENTO

Há diversas perguntas que podemos fazer para um array, como:

- Qual é o menor dos elementos?
- O maior dos elementos?
- Qual é o menor?
- Qual está no meio?

Para isto, vimos diversas técnicas para responder estas perguntas através da ordenação. Mas existem outras perguntas importantes que podemos querer fazer para um array. Por exemplo, se eu fiz uma prova e não fui tão bem, tirando uma nota 7 e levando em consideração que essas são as notas de todos os alunos:



Figura 11.1: Encontrando a posição de um elemento

A nota 7 é boa ou ruim? Preciso observar as notas dos outros e saber quais foram os resultados. Se todos tiraram uma nota abaixo de 7, a minha nota será um bom resultado. Porém, se tiverem tirado notas acima de 7, não será um bom resultado. É importante

comparar a minha nota com as dos demais. Não para poder dizer "eu sou o melhor" ou "eu sou o pior", mas para saber se estou indo bem e se posso focar em outros temas.

Como posso saber isto? Se eu tenho nove alunos na minha sala, em qual colocação fiquei? Ao analisar as demais notas, vou descobrir que quatro alunos tiveram uma nota menor do que a minha. Isto significa que eu fiquei no meio. Fiquei em quinto lugar, tanto entre os melhores e os piores.

Mas como eu fiz esta conta? Foi apenas observação, ou contagem com os dedos? Como fiquei em quinto, era possível fazer o cálculo mentalmente, mas o que exatamente eu fiz para identificar qual a posição que fiquei?

11.1 SIMULANDO QUANTOS SÃO MENORES

Qual algoritmo nós usamos para descobrir em que posição da sala eu fiquei? Vamos tentar calcular da mesma maneira que fiz anteriormente, mas usando o computador. Uma setinha indicará a minha posição:

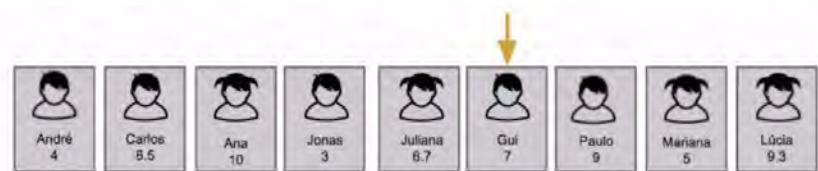


Figura 11.2: Simulando quantos são menores — passo 1

Agora observaremos as notas dos demais alunos e compararemos com a minha. Temos o André:

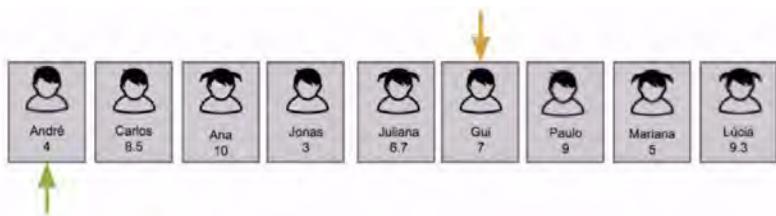


Figura 11.3: Simulando quantos são menores — passo 2

Ele tirou nota 4 e teve um resultado pior do que o meu. Uma pessoa tirou uma nota menor. Já o Carlos tirou 8.5 e se saiu melhor do que eu.

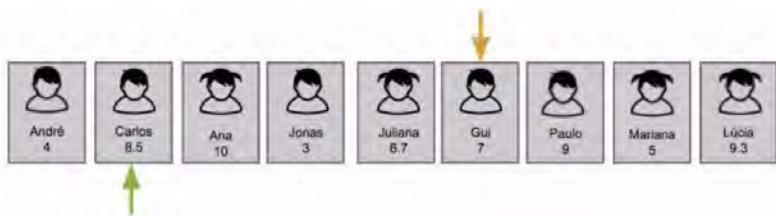


Figura 11.4: Simulando quantos são menores — passo 3

A Ana tirou 10 e foi melhor do que eu.

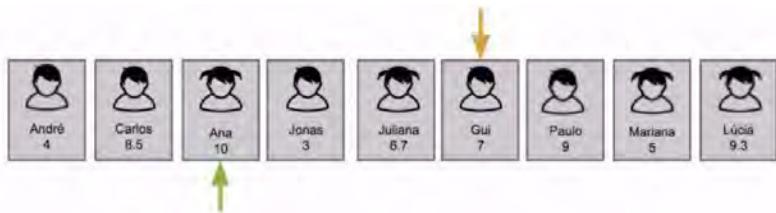


Figura 11.5: Simulando quantos são menores — passo 4

O Jonas tirou 3 e foi pior do que eu. Por enquanto, duas pessoas tiraram notas menores do que a minha.

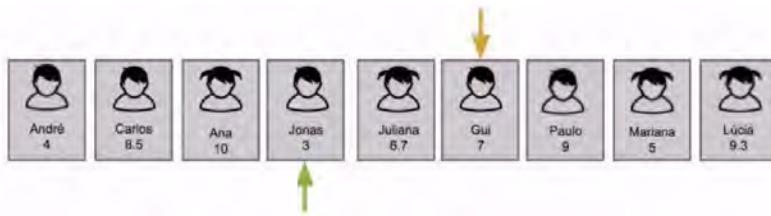


Figura 11.6: Simulando quantos são menores — passo 5

A Juliana tirou 6.7 e foi pior do que eu. Três pessoas tiraram notas menores.

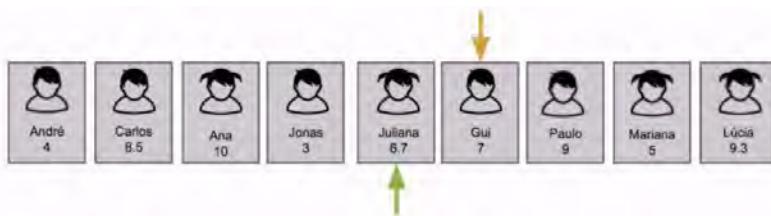


Figura 11.7: Simulando quantos são menores — passo 6

Quando chegamos à minha posição, ignoramos. O Paulo tirou 9 e se saiu melhor do que eu.

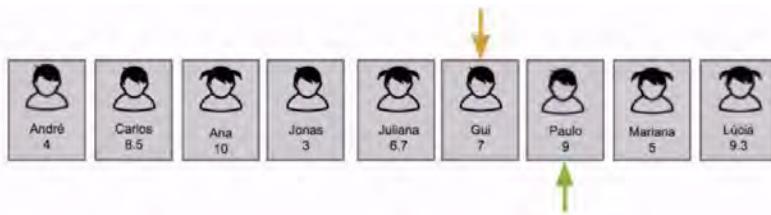


Figura 11.8: Simulando quantos são menores — passo 7

A Mariana tirou 5 e se saiu pior do que eu. Temos quatro pessoas com notas menores do que a minha. A Lúcia tirou 9.3 e teve um melhor resultado.

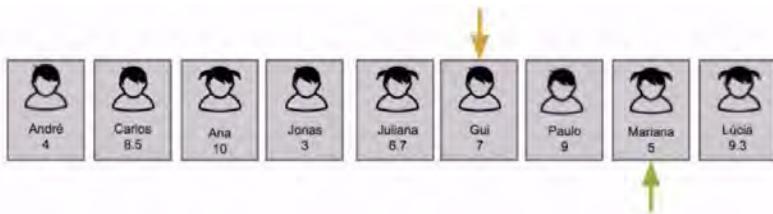


Figura 11.9: Simulando quantos são menores — passo 8

Se formos contando o número de pessoas com notas menores do que a minha, ficaremos sabendo o quanto baixa minha nota está. Se contarmos quantas pessoas foram melhores do que eu, saberemos o quanto alto minha nota está. Basta contar quantas pessoas foram melhores e piores.

Vamos agora anotar quantas notas são menores do que a minha usando a variável `menores`.

Vamos simular o algoritmo? Entre Gui e o André, qual elemento é menor? O André. Seguiremos para a direita e somaremos +1 na variável `menores`.

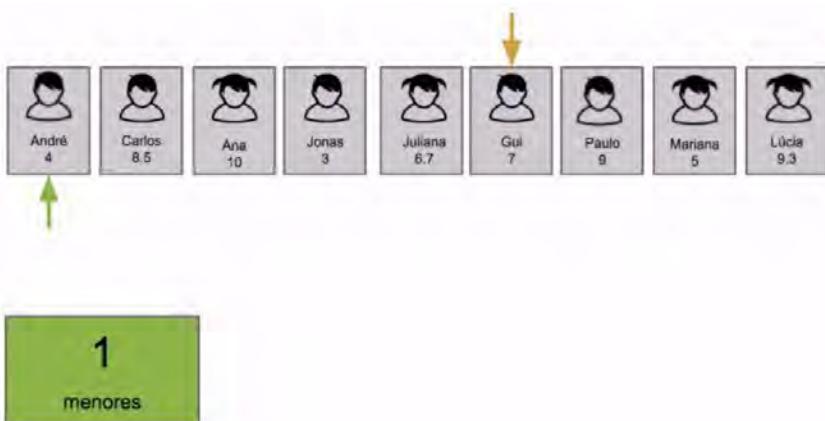


Figura 11.10: Simulando quantos são menores — passo 9

Iremos comparar com a nota do Carlos. Ela é menor? Não.

Seguiremos para a Ana, ela se saiu pior? Não. E Jonas, ele tirou uma nota pior? Sim. Vamos somar +1 na variável `menores`.

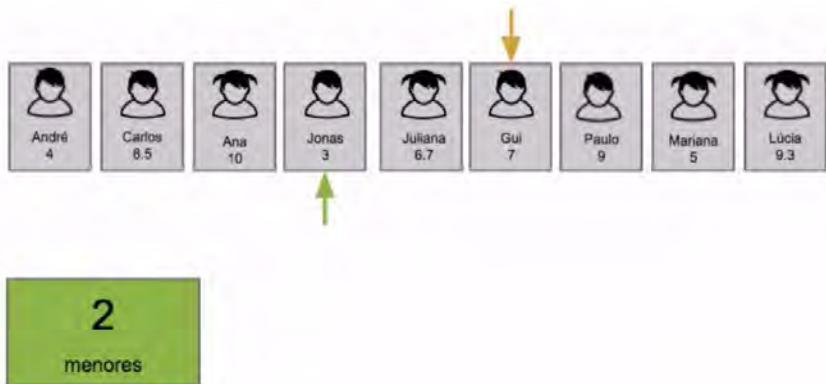


Figura 11.11: Simulando quantos são menores — passo 10

A Juliana tirou uma nota menor? Sim. Somamos +1 na variável `menores`.



Figura 11.12: Simulando quantos são menores — passo 11

Chegamos na minha posição, eu tirei uma nota menor? Além de não ter um resultado pior, devemos ignorar minha própria nota.

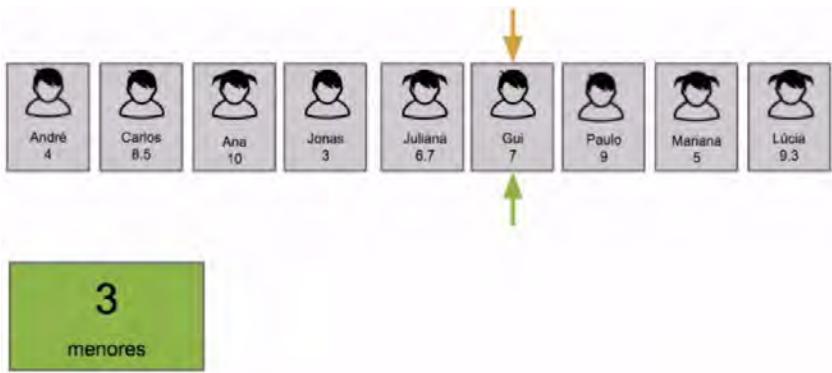


Figura 11.13: Simulando quantos são menores — passo 12

O Paulo tirou uma nota menor? Não.



Figura 11.14: Simulando quantos são menores — passo 13

A Mariana tirou uma nota menor? Sim. Vamos somar +1 na variável `menores`.



Figura 11.15: Simulando quantos são menores — passo 14

Seguimos para a Lúcia, ela é menor? Não. Então, terminamos.

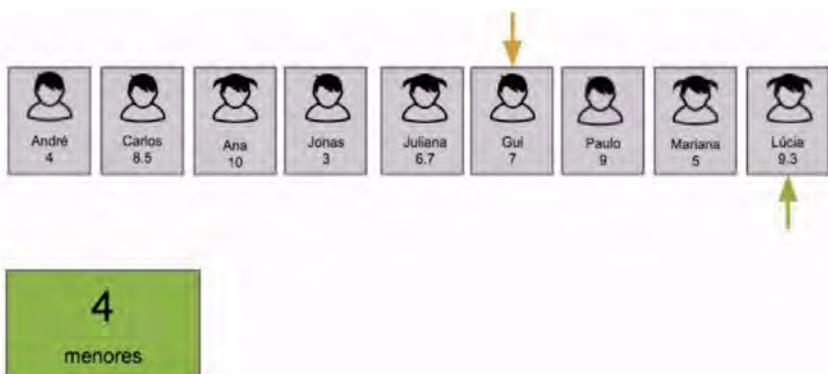


Figura 11.16: Simulando quantos são menores — passo 15

Quantas pessoas tiraram uma nota menor do que a minha?
Quatro pessoas. Então, quando chegar em casa, poderei dizer aos meus pais que tirei uma nota 7, mas quatro pessoas se saíram pior do que eu. Posso usar como argumento a quantidade de pessoas que tiraram notas maiores ou menores.

Como fizemos isto? Fomos comparando cada um dos elementos com a nossa nota e verificando qual era a menor. À medida que fomos passando por cada um, identificamos quantas pessoas

tiveram resultados piores. Em seguida, implementaremos o algoritmo que identifica quantas pessoas no array tiraram notas piores do que a nossa.

11.2 IMPLEMENTANDO O ENCONTRA MENORES

Dentro do nosso projeto, nós vamos criar um novo teste, e o chamaremos de `TestaEncontraMenores`.

Nós queremos descobrir quantas pessoas tiraram uma nota menor do que a minha. Primeiro, vamos criar o método `main()`. Depois vou abrir o `TestaOrdenacaoAoIntercalar` e copiar as notas, pois trabalharemos com elas.

```
public static void main(String[] args) {
    Nota[] notas = {
        new Nota("andre", 4),
        new Nota("carlos", 8.5),
        new Nota("ana", 10),
        new Nota("jonas", 3),
        new Nota("juliana", 6.7),
        new Nota("guilherme", 7),
        new Nota("paulo", 9),
        new Nota("mariana", 5),
        new Nota("lucia", 9.3),
    };
}
```

Estas são as notas que tínhamos no nosso sistema. Agora queremos encontrar todas as notas que foram menor do que a do Guilherme. Vamos extrair o Guilherme, considerando-o como uma única variável.

```
public static void main(String[] args) {
    Nota guilherme = new Nota("guilherme", 7);
    Nota[] notas = {
        new Nota("andre", 4),
        new Nota("carlos", 8.5),
```

```
        new Nota("ana", 10),
        new Nota("jonas", 3),
        new Nota("juliana", 6.7),
        guilherme,
        new Nota("paulo", 9),
        new Nota("mariana", 5),
        new Nota("lucia", 9.3),
    };

}
```

Especificaremos que queremos encontrar as pessoas com notas menores do que o Guilherme (`encontraMenores(guilherme)`).

```
encontraMenores(guilherme);
```

Para isto, teremos de incluir também o array.

```
encontraMenores(guilherme, notas);
```

Vamos criar a função?

```
private static void encontraMenores(guilherme, Nota[] notas) {
}
```

Agora começou o nosso trabalho: procuraremos quem tem a nota menor do que o Guilherme dentro do array. Qual foi o algoritmo que usamos mentalmente? Quando observamos todas as notas, analisamos cada uma por vez. Por isto, no nosso código precisamos criar um `for`.

```
for(int atual = 0; atual < notas.length; atual++) {
```



```
}
```

Analisaremos as notas. Se elas forem menores do que a do Guilherme, teremos encontrado o elemento!

```
Nota nota = notas[atual];
```

Se (`if`) a nota (`nota.getValor`) for menor do que a do Guilherme (`guilherme.getValor`), sabemos que encontramos um

valor menor. Então, ficamos felizes, achamos!

```
if(nota.getValor() < guilherme.getValor()) {  
    // joinha, achei!  
}
```

Quando calculamos mentalmente, o que faríamos em seguida? Somaríamos +1 na variável. Então, precisávamos em alguma parte, começar do 0. Vamos começar fora do laço.

```
int menores = 0;
```

E o trecho do código ficará assim:

```
private static void encontraMenores(Nota guilherme, Nota[] notas)  
{  
    int menores = 0;  
    for(int atual = 0; atual < notas.length; atual++) {  
        Nota nota = notas[atual];
```

Cada vez que encontrarmos uma nota menor, somaremos +1 na variável `menores`.

```
if(nota.getValor() < guilherme.getValor()) {  
    menores = menores + 1;  
}
```

Que podemos traduzir para `menores ++`.

```
if(nota.getValor() < guilherme.getValor()) {  
    menores = menores++;  
}
```

Ao somarmos um ao número de notas menores, o algoritmo vai passar para a próxima nota continuamente. No fim, pediremos para ele retornar (`return`) o número de notas menores.

```
return menores;
```

Precisamos que o método retorne `int`:

```
private static int encontraMenores(guilherme, Nota[] notas) {
```

```
}
```

Vamos encontrar a quantidade de menores. Então, acrescentaremos isto no código.

```
int menores = encontraMenores(Nota guilherme, notas);
```

E depois, imprimir o Número de menores e o valor de menores .

```
int menores = encontraMenores(guilherme, notas);
System.out.println("Número de menores: " + menores);
```

Será que o programa imprimirá 4? Rodaremos o programa, clicamos no botão à direita, depois em *Run As*, e em *Java Application*. O resultado será:

```
Número de menores: 4
```

O que o algoritmo fez? Ele passou por cada elemento, verificando quantas notas menores do que a minha tinha dentro do array.

Podemos alterar o nome da variável `guilherme` para `busca` , porque nem sempre estou encontrando as menores baseadas no `Guilherme`. Logo, vamos alterar a seguinte linha:

```
private static void encontraMenores(Nota guilherme, Nota[] notas)
{
```

Ela receberá o novo nome da variável, `busca` .

```
private static void encontraMenores(Nota busca, Nota[] notas) {
```

Faremos o mesmo com a linha:

```
if(nota.getValor() < guilherme.getValor()) {
```

A alteração será:

```
if(nota.getValor() < busca.getValor()) {
```

Após as modificações, nosso código ficará assim:

```
private static void encontraMenores(Nota busca, Nota[] notas) {  
    int menores = 0;  
    for(int atual = 0; atual < notas.length; atual++) {  
        Nota nota = notas[atual];  
        if(nota.getValor() < busca.getValor()) {  
            menores = menores ++;  
        }  
    }  
    return menores;  
}
```

A variável é a nota que estamos buscando, verificamos o valor dela e se é maior ou menor. Veremos que o nosso código está funcionando bem. Mas ainda é possível melhorá-lo: podemos fazer um `for` com Java 8 ou outras opções.

Para nós, é interessante ver a variável passando por cada nota, porque em diversas linguagens precisaríamos escrever desta forma. Por isso, optamos por este tipo de `for`.

11.3 O PRÓXIMO DESAFIO: COLOCANDO UM ELEMENTO EM SEU LUGAR

Então, vamos rever o que fizemos: tínhamos diversas notas e queríamos encontrar as que eram menores do que a do Guilherme. Sabemos que, quando realizamos este tipo de ação, analisaremos todos os itens. À medida que olhamos todos, contamos quatro notas menores. O algoritmo que implementamos determina quantas notas são menores do que a minha.

Mas se sabemos que somente 1 pessoa tirou nota menor do que a minha, então quer dizer que eu fiquei em segundo lugar, de baixo para cima. Se tem 5 pessoas que tiraram nota menor que a minha, eu estou em sexto lugar, de baixo para cima.

Então, existe um padrão aqui: se tem **n** elementos menores do

que o meu, o meu lugar parece ser $n+1$. Será? É o que veremos no próximo capítulo.

CAPÍTULO 12

COLOCANDO UM ELEMENTO NO SEU LUGAR: O PIVÔ

Nós já temos uma maneira de descobrir quantas pessoas se saíram piores do que eu em uma prova. Se eu sei quantas pessoas foram melhores ou piores, é natural imaginar com quem vou estudar para o próximo teste.



Figura 12.1: Separar maiores e menores — passo 1

Separarei na lista os alunos que se saíram bem ou mal na prova. Quem foi pior, deixarei de lado. Para a próxima prova, vou tentar estudar com quem se saiu melhor. Seria interessante se pudéssemos separar aqueles que tiraram notas maiores daqueles que tiraram notas menores.

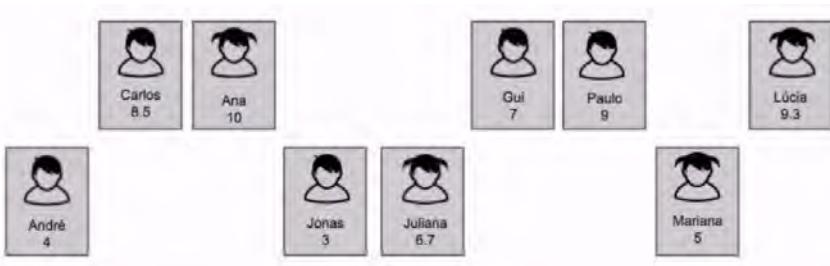


Figura 12.2: Separar maiores e menores — passo 2

E colocar cada grupo de um lado.

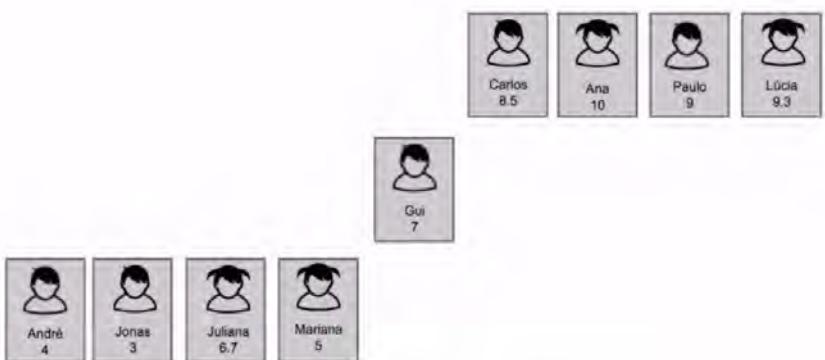


Figura 12.3: Separar maiores e menores — passo 3

Porque se eu ficar exatamente no meio, saberei que as pessoas na minha esquerda serão pessoas que tiveram resultados piores. Mesmo que elas não estejam ordenadas, isto é irrelevante.



Figura 12.4: Separar maiores e menores — passo 4

Porém, os alunos com quais quero estudar serão os posicionados à direita, porque foram os que se saíram melhor.

Sabemos contar os que se saíram melhor. O que nos falta aprender é como separar os dois grupos.

Temos a lista original, que está desordenada, e queremos colocar quem se saiu melhor para esquerda e quem se saiu melhor para a direita.



Figura 12.5: Separar maiores e menores — passo 5

Eu permanecerei no meio, porque é o lugar que corresponde a mim. Se existem quatro pessoas que foram melhores e outras quatro que foram piores, ficarei na posição que mereço. Está tudo bem, fico feliz em permanecer na posição que estou. Se os elementos posicionados na esquerda estão desordenados, não é minha preocupação. Estou mais interessado em saber aonde mereço ficar.

O objetivo é apenas descobrir quem teve resultados melhores ou piores. Então, seria bom encontrar um algoritmo que, dada a minha posição, eu consiga encontrar quem tirou notas maiores e menores. Assim saberei com quem eu quero estudar.

Observe que os elementos não ficaram ordenados, somente a minha posição está adequada, e quebra a tabela em duas. Eu sou o *pivot que separa quem foi melhor daqueles que foram pior. E nós só

queremos saber quem teve resultados melhores ou piores do que o meu na prova. Como podemos fazer isto?

Agora que temos nosso array de elementos, em que estou posicionado no meio, queremos fazer a ordenação. Por isso, quando eu for para a posição adequada, todos os demais que estiverem na minha esquerda terão notas menores, e todos que estiverem à direita terão notas maiores, mesmo que cada parte não fique ordenada.

O importante é que eu divida estes dois pedaços. Os menores vão para esquerda, e os maiores para a direita. Isto é, me deslocarei para indicar quais são os menores e os maiores. Quem é maior ou menor, é irrelevante para mim. O importante é que eu esteja na posição que mereço.

12.1 O PIVÔ DA SEPARAÇÃO

Para tentar criar um algoritmo que seja capaz de dividir um array em dois, com os elementos maiores e menores separados pelo Guilherme posicionado no meio, receberemos um array e o elemento que particiona, que divide.



Figura 12.6: O pivô da separação — passo 1

Quando você tem um grupo e um integrante que divide o grupo no meio, causando uma grande quebra, costumamos chamá-lo de **pivô**. Ele é quem causa a separação. Queremos saber quem será o pivô que separará o grupo.

Se escolhermos um pivô do meio da lista, teremos problemas,

porque cada vez precisaremos escolher um elemento diferente. Será mais complexo do que se usássemos como critério escolher sempre o primeiro ou o último.

Vamos facilitar um pouco. Em vez de escolhermos um pivô aleatório — o que é válido —, vamos simplificar. Vamos trocar de lugar o **Guilherme** com a **Lúcia**.



Figura 12.7: O pivô da separação — passo 2

Vamos trabalhar com o pivô sempre ocupando a última posição. Assim será mais fácil para o algoritmo particionar — dividir o array em dois pedaços. Sendo que o pivô (no nosso caso, o Guilherme) sempre será o último elemento. Dada a nossa lista, como poderemos encontrar a posição adequada do último item?

Além de posicionar corretamente o pivô, também queremos colocar na esquerda os menores e na direita, os maiores. O importante é saber que o pivô é o último elemento.

12.2 VARIÁVEIS PARA PARTICIONAR

Como toda função, em todo problema computacional, nós recebemos uma entrada e precisamos devolver uma saída. A entrada é o array. Mas quais elementos vamos analisar? Como de costume, precisaremos saber qual é o começo (`inicio`) e qual é o fim (`termino`).



Figura 12.8: Variáveis para particionar — passo 1

A nossa saída será o array já particionado, com o pivô na posição adequada. Todos os elementos da esquerda podem ser menores ou iguais, e os demais da direita podem ser maiores. Isto é o que importa, independente da posição em que ele caia.

Quem é o nosso pivô? Considerando os dados, ele é o `termínio` menos 1. Meu pivô é o elemento da posição 8, o Guilherme, que tirou nota 7.

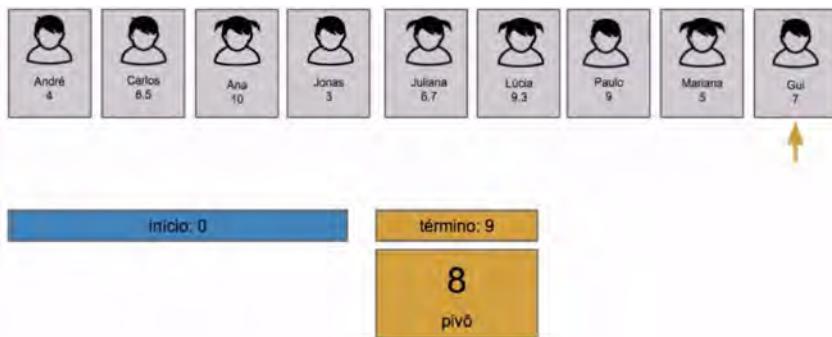


Figura 12.9: Variáveis para particionar — passo 2

Onde o Guilherme merece estar? É a grande questão. Vamos colocá-lo no lugar, assim ficaremos satisfeitos se todos que estiverem posicionados à sua esquerda forem menores, e à sua direita forem maiores.

Não importa se ele vai cair na segunda posição ou se a Lúcia deveria ficar na penúltima posição. As posições são irrelevantes. O

importante é que, dado o pivô, os elementos menores fiquem à esquerda e os maiores, à direita. Vamos testar?

Teremos de varrer o array, porque precisamos saber quem é menor que o Guilherme. Se existirem três notas menores, ele precisará ocupar a quarta posição. Se existirem duas notas menores, ele precisará ocupar a terceira posição. Se existirem 17 notas, ele precisará ocupar 18^a posição.

Vamos varrer o array, contando quantos são os menores. Usaremos indicadores para andar no array, indicando quais elementos serão analisados. A variável analisando começará do `inicio` e vai até o `termino`.

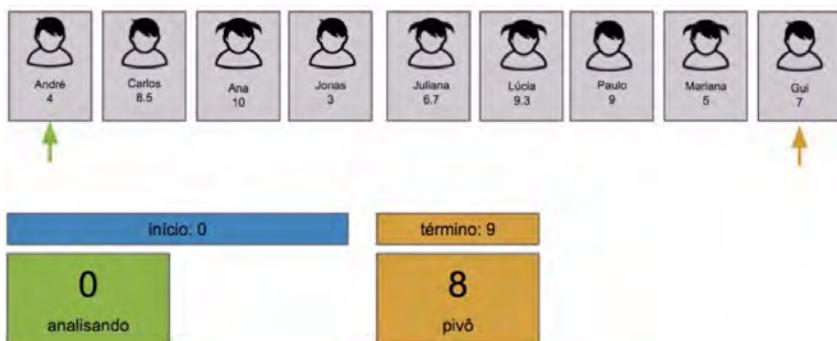


Figura 12.10: Variáveis para particionar — passo 3

Ao analisarmos o André, ele é menor do que o Gui? Sim. Vamos somar +1 ao `analizando`. Então, quantos já temos menores? Um elemento.

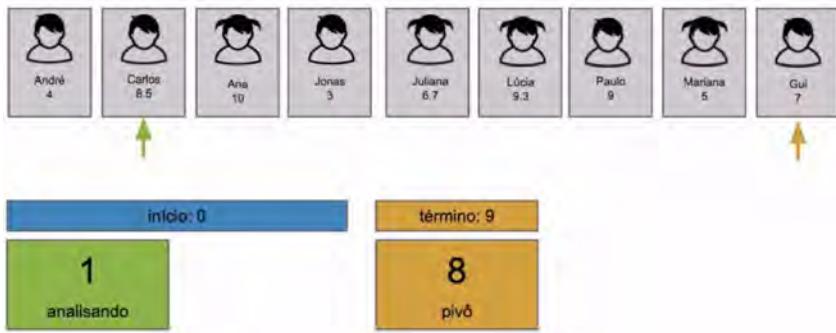


Figura 12.11: Variáveis para particionar — passo 4

Agora, o Carlos. Ele é maior ou menor? Maior. Seguiremos para o próximo.

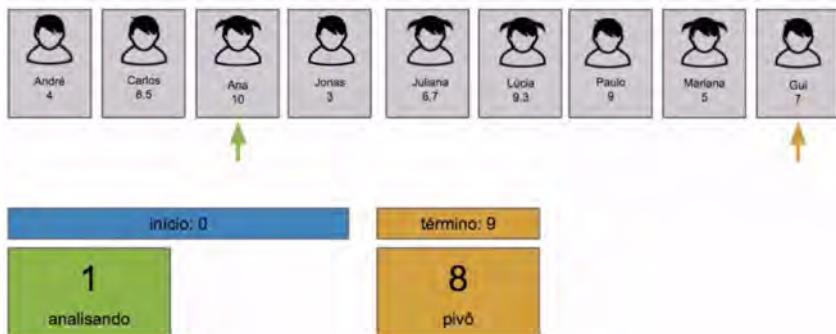


Figura 12.12: Variáveis para particionar — passo 5

A Ana é menor ou maior? Maior.

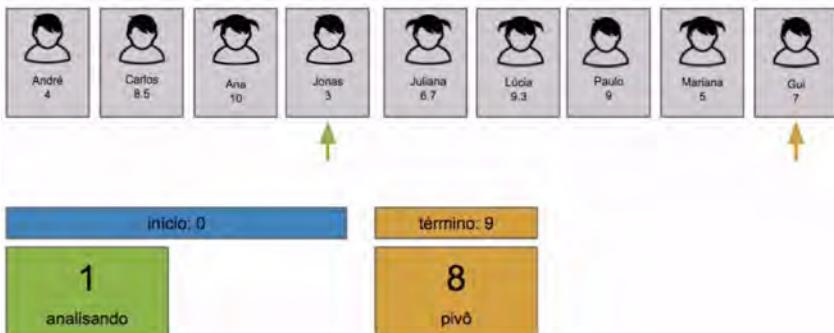


Figura 12.13: Variáveis para particionar — passo 6

O Jonas é menor ou maior? Menor. Temos **duas** pessoas com notas menores.

Existe uma forma de anotarmos no computador as pessoas que tiraram notas menores? Sim. Voltaremos ao início do processo para criar a variável `menoresAtéAgora`, que armazenará o número de elementos menores que o pivô até agora.

No começo, o valor será igual a 0, porque ninguém será menor do que o pivô. Vamos testar novamente o nosso algoritmo. Analisaremos cada um dos elementos, da esquerda para direita, tentando encontrar quem tirou notas menores do que a do Guilherme.

12.3 COLOCANDO UM ELEMENTO NO SEU LUGAR DO ARRAY

Agora, queremos rodar algum processo que seja capaz de indicar quais elementos são menores que o Guilherme. Então, analisaremos todos os itens. Para isto, vamos usar a variável `analisando`. Contamos as pessoas com menores notas no `menoresAtéAgora`. Vamos fazer isto?



Figura 12.14: Colocando-me no meu lugar — passo 1

Primeiro passo, o André é menor do que o Guilherme? Sim. Então `menoresAtéAgora` será igual a 1.



Figura 12.15: Colocando-me no meu lugar — passo 2

O próximo é o Carlos. A variável `analisando` será igual a 1.



Figura 12.16: Colocando-me no meu lugar — passo 3

Ele é menor do que o Guilherme? Não. Somaremos +1 na variável analisando .



Figura 12.17: Colocando-me no meu lugar — passo 4

A Ana é melhor do que o Guilherme? Não. O analisando será igual a 3.



Figura 12.18: Me colocando no meu lugar — passo 5

Jonas é menor do que o Guilherme? Sim. Quantos são menores até agora? **Duas** pessoas. A variável `analizando` será igual a 4 e seguiremos para a Juliana.

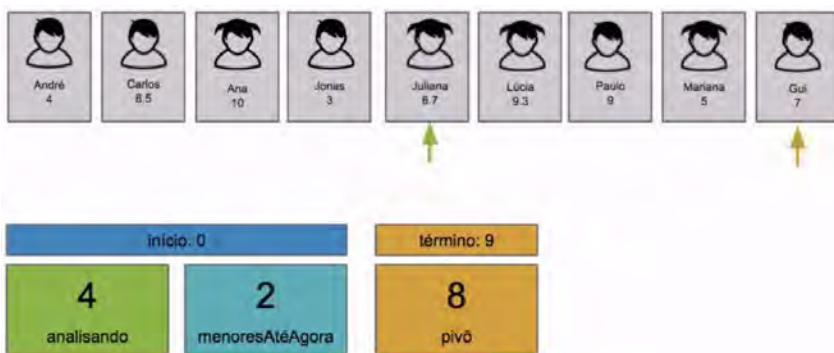


Figura 12.19: Colocando-me no meu lugar — passo 6

Ela é menor do que o Guilherme? Sim. Vamos analisar o próximo item e somaremos +1 nas variáveis `analizando`, que será igual a 5, e `menoresAtéAgora`, que será igual a 3.



Figura 12.20: Colocando-me no meu lugar — passo 7

A Lúcia é menor do que o Guilherme? Não. Então, vamos para o próximo e analisando será igual a 6.



Figura 12.21: Colocando-me no meu lugar — passo 8

O Paulo é menor do que o Guilherme? Não. Vamos para o próximo elemento e analisando será igual a 7.



Figura 12.22: Colocando-me no meu lugar — passo 9

A Mariana é menor do que o Guilherme? Sim. Então somaremos +1 nas variáveis `analisando` e `menoresAtéAgora`. Quatro pessoas tiraram notas menores.



Figura 12.23: Colocando-me no meu lugar — passo 10

Ao passarmos para o próximo, chegamos ao fim. Nós analisamos todos os itens e descobrimos quantos são menores. Era o que desejávamos saber, porque assim poderemos indicar a posição em que o Guilherme deve ficar. Agora poderemos colocar o Guilherme na **posição 4**.



Figura 12.24: Colocando-me no meu lugar — passo 11

Se tinha quatro pessoas com notas menores do que a dele, esta será a posição adequada. Se existissem duas pessoas com notas menores, ele deveria ocupar a posição 2. Se ninguém fosse menor que o Guilherme, ele deveria ocupar a posição 0, que é a primeira do array.

Logo, quando terminamos de varrer todos os elementos, o pivô deve ficar na posição da variável `menoresAtéAgora`. Nós já encontramos a posição adequada. Porém, não basta apenas trocá-lo de lugar com a Juliana.



Figura 12.25: Colocando-me no meu lugar — passo 12

Por quê? Porque nem todos os demais elementos posicionados à esquerda são menores do que o Guilherme. Existem maiores e menores. Enquanto analisávamos, precisávamos organizar a ordem dos alunos para garantir que os menores ficassem à esquerda e expulsando os que fossem maiores desta parte.

Como podemos fazer isto? Veremos em seguida como é possível encontrar a posição adequada do Guilherme e expulsar todos que tiraram notas maiores para o lado direito do array. Temos de fazer isto de alguma maneira.

12.4 IMPLEMENTANDO O CÓDIGO DE POSICIONAMENTO

Queremos escrever um algoritmo que não apenas encontre o número de elementos menores, mas que também posicione o pivô (o último elemento) adequadamente. Assim poderemos separar as notas menores das maiores.

Criaremos uma nova classe, que se chamará `TestaPivota`, porque nós vamos colocar o pivô em seu lugar, "pivotaremos" nosso array. Dentro da classe, criaremos o método `main()`, no qual colocaremos as notas dos alunos.

```
public static void main(String[] args) {
    Nota guilherme = new Nota("guilherme", 7);
    Nota[] notas = {
        new Nota("andre", 4),
        new Nota("carlos", 8.5),
        new Nota("ana", 10),
        new Nota("jonas", 3),
        new Nota("juliana", 6.7),
        guilherme,
        new Nota("paulo", 9),
        new Nota("mariana", 5),
        new Nota("lucia", 9.3)
    };
}
```

Nossa lista terá uma única diferença: o Guilherme não ficará mais no meio, porque nós definimos que o pivô sempre será o último elemento. Então ele trocou de lugar com a Lúcia. Vamos trocar os elementos de posição:

```
public static void main(String[] args) {  
    Nota guilherme = new Nota("guilherme", 7);  
    Nota[] notas = {  
        new Nota("andre", 4),  
        new Nota("carlos", 8.5),  
        new Nota("ana", 10),  
        new Nota("jonas", 3),  
        new Nota("juliana", 6.7),  
        new Nota("paulo", 9),  
        new Nota("mariana", 5),  
        new Nota("lucia", 9.3)  
        guilherme,  
    };  
}
```

A nova ordem será: André, Carlos, Ana, Jonas, Juliana, Paulo, Mariana, Lúcia e Guilherme. Agora queremos que o nosso algoritmo quebre o array em duas partes (`quebraNoPivo`) e depois faça outras ações. Após dividir os elementos, as notas estarão disponíveis. Vamos especificar isso no código:

```
Nota[] notas = {  
    new Nota("andre", 4),  
    new Nota("carlos", 8.5),  
    new Nota("ana", 10),  
    new Nota("jonas", 3),  
    new Nota("juliana", 6.7),  
    new Nota("paulo", 9),  
    new Nota("mariana", 5),  
    new Nota("lucia", 9.3)  
    guilherme,  
};  
  
quebraNoPivo(notas);  
}
```

Em seguida, vamos implementar o método `quebraNoPivo`. Como ele funciona? Nós já tínhamos encontrado uma forma de

descobrir o menor elemento da lista. Primeiro, especificávamos quais elementos seriam analisados. Para isto, precisávamos saber o valor do `inicial` e do `termino`.

```
private static void quebraNoPivo(Nota[] notas, int inicial, int termino) {  
}
```

Isto significa que vamos do 0 até `notas.length`.

```
quebraNoPivo(notas, 0, notas.length);
```

Este era o problema que queríamos resolver. E quem é o pivô? O elemento que está na última posição que analisaremos. Como temos nove elementos, isto significa que ele estará na posição 8. Logo, ele será o item da posição `termino - 1`. Caso tivéssemos cinco elementos, o pivô estaria na posição 4. Se fossem dezessete elementos, ele estaria na posição 16. O último elemento sempre será o pivô.

```
private static void quebraNoPivo(Nota[] notas, int inicial, int termino) {  
    Nota pivo = notas[termino - 1];  
}
```

Agora queremos varrer todo o array. Por isso, escreveremos o `for`:

```
for(int analisando = 0; analisando < termino; analisando++) {  
}
```

Nosso código ficará assim:

```
private static void quebraNoPivo(Nota[] notas, int inicial, int termino) {  
    Nota pivo = notas[termino - 1];  
    for(int analisando = 0; analisando < termino; analisando++) {  
    }  
}
```

Vamos varrer o array inteiro, porém não será preciso passar por

todos os itens, afinal o Guilherme será o último elemento. Por isso, podemos ignorá-lo e não o analisaremos. Vamos alterar o nosso `for` e especificar que devemos analisar até o `termíno -1`.

```
for(int analisando = 0; analisando < termíno - 1; analisando++) {  
}
```

Não precisamos comparar as demais notas com o pivô. O que faremos em seguida? Analisaremos a nota atual (`notas[analisando]`).

```
for(int analisando = 0; analisando < termíno - 1; analisando++) {  
    Nota atual = notas[analisando];  
}
```

Será que a nota atual é mais baixa? Se (`if`) a `atual.getValor()` for menor ou igual ao `pivo.getValor()`, ficará posicionado à esquerda. Para isto, usaremos a variável `menoresEncontrados++`.

```
for(int analisando = 0; analisando < termíno - 1; analisando++) {  
    Nota atual = notas[analisando];  
    if(atual.getValor() <= pivo.getValor()) {  
        menoresEncontrados++;  
    }  
}
```

Agora precisaremos calcular os `menoresEncontrados`.

```
private static void quebraNoPivo(Nota[] notas, int inicial, int te  
rmino) {  
    int menoresEncontrados = 0;  
}
```

Vamos revisar o que estamos fazendo: `menoresEncontrado` começa com 0, depois vamos analisar todos os elementos (com exceção do pivô que será ignorado), e contaremos quantos são menores do que o Guilherme. No fim, o que faremos? Já sabemos que o pivô ficará na posição `menoresEncontrados`. Isso significa que vamos trocar no array o elemento que estiver na posição

```
termino -1 com o que estiver na posição menoresEncontrados .  
troca(notas, termino -1, menoresEncontrados);
```

Nosso objetivo é trocar os dois itens de posição no fim. Vamos implementar a função `troca()`? Nós queremos receber o `de` e o `para`. Então, `nota1` será o `notas[de]` e o `nota2` será o `notas[para]`.

```
private static void troca(Nota[] notas, int de, int para) {  
    Nota nota1 = notas[de];  
    Nota nota2 = notas[para];  
    notas[para] = nota1;  
    notas[de] = nota2;  
}
```

Trocamos de lugar os elementos que estavam no `de` para a posição dos que estavam no `para`, e vice-versa.

Depois que rodarmos a `quebraNoPivo`, temos de verificar se Guilherme caiu na casa correta, na posição 4. Faremos um `for` para cada uma das notas. Teremos a nota atual (`notas[atual]`), e vamos imprimir o aluno (`nota.getAluno`) e o valor (`nota.getValor`):

```
quebraNoPivo(notas,0, notas.length);  
  
for(int atual = 0; atual < notas.length; atual++) {  
    Nota nota = notas[atual];  
    System.out.println(nota.getAluno() + " " + nota.getValor());  
}
```

Vamos testar e ver se o código funciona? Clicamos em *Run As*, depois em *Java Application*, e o resultado será:

```
andre 4.0  
carlos 8.5  
ana 10.0  
jonas 3.0  
guilherme 7.0  
lucia 9.3  
paulo 9.0  
mariana 5.0
```

O Guilherme ficou na posição adequada, porque nós contamos quantas pessoas eram menores do que o pivô e o colocamos na posição 4. Lembrando de que o array começa com a posição 0. Assim, conseguimos colocar o Guilherme na posição certa.

12.5 O PRÓXIMO DESAFIO: E OS OUTROS ELEMENTOS?

Já sabemos colocar um elemento em sua posição. Se eu fiquei em quinto lugar em uma competição, devo ficar na posição onde o quinto lugar deve ficar, claro. Se eu fiquei em terceiro lugar, devo ficar na posição do terceiro lugar. São tarefas possíveis de implementar com o nosso algoritmo atual, basta rodá-lo para um ou outro elemento.

Mas e se eu pegar um aluno e colocar no lugar dele, como ficariam os outros alunos? Gostaria que os piores que ele ficassem em um lado, e os melhores em outro lado, algo que faremos no próximo capítulo.

CAPÍTULO 13

PIVOTANDO UM ARRAY POR COMPLETO

13.1 VERIFICANDO A MUDANÇA DE POSIÇÃO

No capítulo anterior, conseguimos encontrar a posição exata em que o pivô deveria ficar. Mas faltou separar os elementos menores para a esquerda do array. Nós vimos como fazer isto.

Quando estamos dentro do laço e encontramos alguém menor, qual é o próximo passo? Devemos trocar o elemento que está na posição analisando para a posição menoresEncontrados .

```
troca(notas, analisando, menoresEncontrados);
```

O laço ficará assim:

```
Nota pivo = notas[termino - 1];
for(int analisando = 0; analisando < termino - 1: analisando++) {
    Nota atual = notas[analisando];
    if(atual.getValor() <= pivo.getValor()) {
        troca(notas, analisando, menoresEncontrados);
        menoresEncontrados++;
    }
}
troca(notas, analisando, menoresEncontrados);
```

Se temos três notas menores até agora, então as posições 0, 1 e 2 são para elementos menores. Mas a casinha 3 será maior. Então, trocarei o elemento que estou analisando de posição. Se temos 17 elementos que são menores até agora, da posição 0 até 16, todos os

itens serão menores. O que farei é colocar o pivô na casinha 17. Em seguida, somarei +1 na variável `menoresEncontrados`.

Testaremos novamente e veremos o que acontecerá. Clicamos em *Run As*, depois em *Java Application*, e o resultado será:

```
andre 4.0
jonas 3.0
juliana 6.7
mariana 5.0
guilherme 7.0
lucia 9.3
paulo 9.0
carlos 8.5
ana 10.0
```

Os quatro primeiros elementos da lista (André, Jonas, Juliana e Mariana) têm notas menores do que o pivô (Guilherme). Todos que eram menores foram particionados para a esquerda e todos maiores para a direita. Agora, nós efetivamente quebramos o array no pivô.

Também queremos ser capazes de informar em qual posição o Guilherme ficou. Então vamos retornar no fim `menoresEncontrados`.

```
Nota pivo = notas[termino - 1];
for(int analisando = 0; analisando < termino - 1; analisando++) {
    Nota atual = notas[analisando];
    if(atual.getValor() <= pivo.getValor()) {
        troca(notas, analisando, menoresEncontrados);
        menoresEncontrados++;
    }
}
troca(notas, analisando, menoresEncontrados);
return menoresEncontrados;
```

O código vai nos informar algo como "o Guilherme foi para essa posição". O nosso método devolverá um `int` e também teremos a nova posição:

```
int novaPosição = quebraNoPivo(notas, 0, notas.length);

for(int atual = 0; atual < notas.length; atual++) {
```

```
    Nota nota = notas[atual];
    System.out.println(nota.getAluno() + " " + nota.getValor());
}
```

Se quisermos, podemos imprimir: "O pivô foi parar em " + novaPosição .

```
int novaPosição = quebraNoPivo(notas, 0, notas.length);

System.out.println("O pivô foi parar em " + novaPosição);
```

Ao rodarmos novamente, o programa informará que o pivô foi colocado na posição 4.

```
O pivô foi parar em 4
andre 4.0
jonas 3.0
juliana 6.7
mariana 5.0
guilherme 7.0
lucia 9.3
paulo 9.0
carlos 8.5
ana 10.0
```

A função quebraNoPivo() funciona. Ela encontra a posição adequada para o pivô e depois quebra o array em duas partes: os menores e os maiores.

13.2 SIMULANDO A PARTIÇÃO

Queremos varrer o nosso array, contando os menoresAtéAgora , porque assim poderemos encontrar a posição do pivô. Ao descobrirmos a posição adequada, trocaremos o elemento de lugar e ficaremos satisfeitos.

Mas apenas isto não é o suficiente. Queremos também saber quem foi melhor ou pior, e que os elementos menores fiquem posicionados à esquerda. À medida que vamos varrendo e analisando os elementos do array, nós queremos de alguma maneira

informar que: "se o elemento é menor do que o Guilherme, ele deve ficar à esquerda". Analisando cada elemento, identificamos se ele deve ser posicionado na parte esquerda do array. Caso contrário, não é preciso movê-lo.

Começaremos a trabalhar com o algoritmo. As variáveis analisando e menoresAtéAgora serão iguais a 0. E o pivô estará na posição 8. O elemento analisado será o André.



Figura 13.1: Simulando a partição — passo 1

A nota 4 é menor do que 7? Sim. Sendo assim, vou manter o elemento à esquerda, na posição 0. menoresAtéAgora e analisando serão igual a 1. Por enquanto, funciona o algoritmo. Seguiremos para o Carlos.

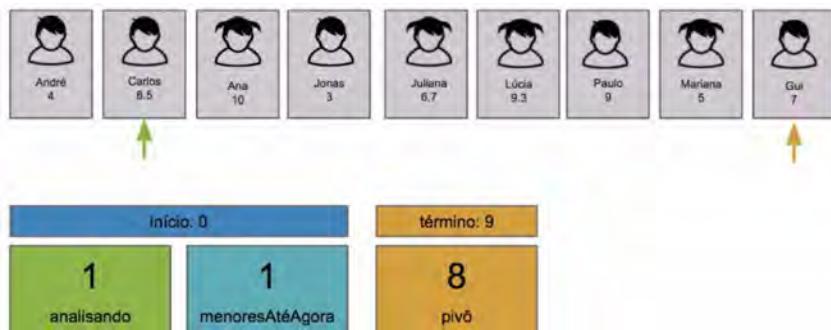


Figura 13.2: Simulando a partição — passo 2

A nota 8.5 é menor ou maior do que 7? É maior. Não faremos alterações e seguiremos para o próximo. O analisando será igual a 2.



Figura 13.3: Simulando a partição — passo 3

A Ana tirou nota maior ou menor do que o Guilherme? Ela tirou 10, uma nota explicitamente maior. Vamos para o elemento seguinte. Somaremos +1 na variável analisando .



Figura 13.4: Simulando a partição — passo 4

Agora aumenta a dificuldade. O Jonas é menor do que o Guilherme? Sim. Se ele é menor, o que precisamos fazer? Colocá-lo mais à esquerda no array. Vamos trocá-lo de posição. Com qual elemento? Não precisamos tirar o André da posição 0, porque ele também é menor do que o pivô. E se trocar os dois elementos de

posição o André não ficará em um lugar adequado.

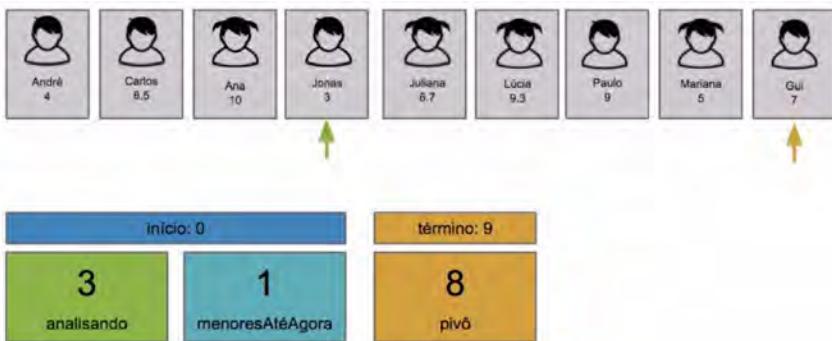


Figura 13.5: Simulando a partição — passo 5

Queremos trocar o Jonas de lugar com o primeiro elemento que não seja menor que o Guilherme. Quantos são menores até agora? Apenas **um**. Isto significa que devemos colocar o Jonas na posição do `menoresAtéAgora`. O Carlos, que ocupa a posição 1, não é menor que o pivô. Então podemos trocar os elementos de posição.

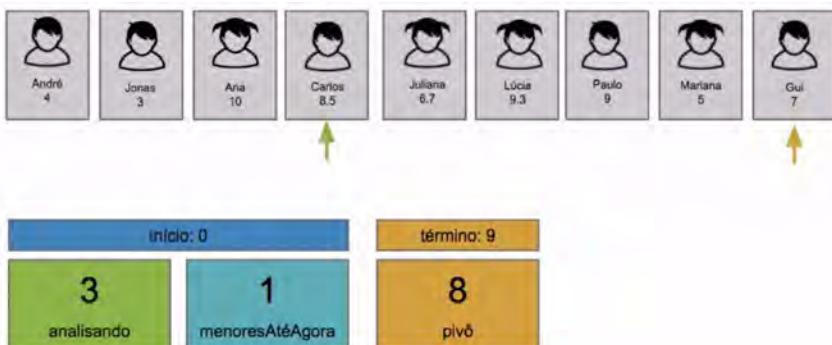


Figura 13.6: Simulando a partição — passo 6

Seguiremos com o processo de sempre. Somaremos +1 nas variáveis `menoresAtéAgora` e `analizando`. Vamos para o próximo.



Figura 13.7: Simulando a partição — passo 7

A Juliana é maior ou menor? Menor. Como ela tirou uma nota menor, vamos trocá-la de lugar. Mas para qual posição? Não será nas posições 0 e 1, porque atrapalharia o processo. A posição correta será a 2, afinal é o número de `menoresAtéAgora`. Vamos trocá-la de lugar com a Ana.

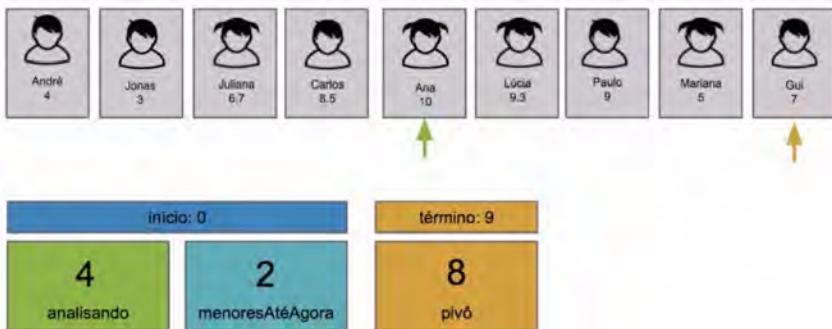


Figura 13.8: Simulando a partição — passo 8

Agora que encontramos outro elemento menor, somaremos +1 na variável `menoresAtéAgora`. Aumentaremos também o valor de `analisando` que será igual a 5. Vamos para o próximo.

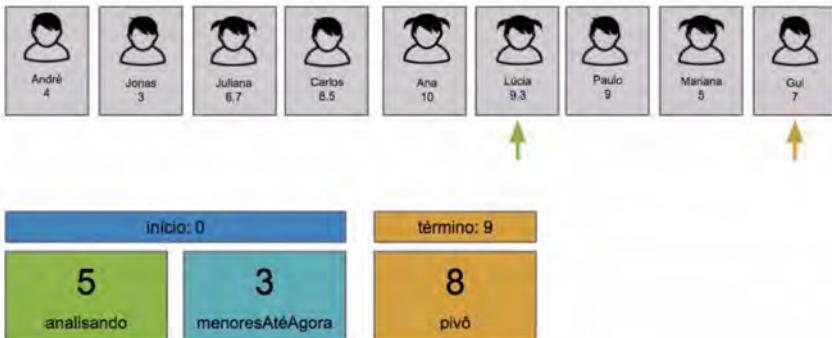


Figura 13.9: Simulando a partição — passo 9

A Lúcia é maior do que o Guilherme? Sim. Então apenas seguiremos para o próximo. analisando será igual a 6.



Figura 13.10: Simulando a partição — passo 10

O Paulo é maior? Sim. Então vamos para o próximo. Somaremos +1 no analisando .



Figura 13.11: Simulando a partição — passo 11

E a Mariana é menor ou maior? Menor. Então vamos ter de trocá-la de posição. Não podemos movê-la para as posições 0, 1 e 2; caso contrário, algum elemento menor ficaria posicionado à direita. Temos de colocá-la na posição 3. Por quê? Porque temos **três** no menoresAtéAgora . Então trocaremos a Mariana com que estiver na posição 3, no caso será com o Carlos.



Figura 13.12: Simulando a partição — passo 12

Aumentaremos +1 nas variáveis analisando e menoresAtéAgora , e chegaremos ao último elemento, o Guilherme.



Figura 13.13: Simulando a partição — passo 13

Quando chegamos a mim mesmo, já sabemos quantos elementos são menores no array e em qual posição devo ficar, que corresponde à do `menoresAtéAgora`. Então, trocaremos o pivô com o elemento na posição 4, a Ana.



Figura 13.14: Simulando a partição — passo 14

Durante o processo, colocamos todos os elementos menores para a esquerda. À direita, ficaram os demais itens. Isto significa que particionamos o array com o pivô, que ficou dividido em duas partes: os elementos que ficaram à esquerda são menores e à direita, são maiores.

Qual algoritmo nós usamos? Nós analisamos todo o array e os elementos que eram maiores permaneciam na mesma posição até

que identificávamos os menores, que eram colocados na esquerda e trocados com os elementos na posição do `menoresAtéAgora`. Em seguida, somávamos +1 na mesma variável. Terminada a análise de cada item, também movíamos o pivô para a posição do `menoresAtéAgora`.

Ou seja, após observarmos todos os elementos, todos que eram menores foram colocados na parte da esquerda. Depois, movemos o Guilherme para a posição 4 e os demais itens ficaram à direita. Assim nosso array ficou particionado entre os menores e os maiores, e eu fiquei posicionado no meio. Lembrando de que o pivô era o último elemento.

13.3 PIVOTA COLOCA NA POSIÇÃO

Nós já sabemos fazer diversas coisas com o nosso array: encontrar quem é maior ou menor, quantos menores temos, qual posição o elemento deveria ficar, colocar os itens menores à esquerda e os maiores, à direita. Também particionamos o array em duas partes.

É possível responder muitos problemas do mundo real. Como por exemplo, se queremos ajudar os alunos que estão com notas abaixo de 4, que foi a nota que tirei. Então, eu posso ajudar aqueles que se saíram pior. Separo estas pessoas e ficam separadas também as pessoas que tiraram notas maiores do que a minha.

Em uma única varrida dos elementos, foi possível descobrir tudo isto. Podemos fazer muitas coisas com os algoritmos que aprendemos até agora.

Quando rodamos o particiona e temos um array com as notas de todos os alunos, sempre indicaremos o `inicio` e o `termino`, quais elementos serão analisados.

Então, colocaremos o último elemento (o pivô), no lugar adequado:

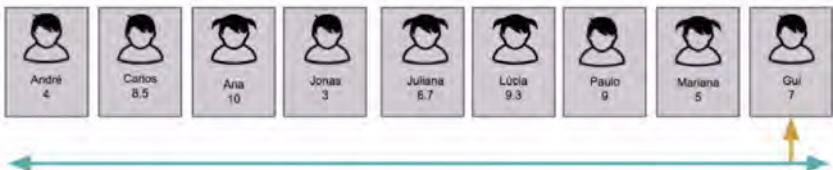


Figura 13.15: Como último elemento



Figura 13.16: Colocando no lugar adequado

Ao trocarmos os elementos de lugar, os demais também serão reposicionados. Desta forma, todos que estarão à esquerda serão menores e os à direita, serão maiores. Assim rodamos o pivô , o algoritmo que partitiona o array em duas partes com o pivô. Este foi o resultado.

13.4 O PRÓXIMO DESAFIO: PIVOTANDO MAIS VEZES

Já sabemos que o elemento Guilherme está na posição adequada, quem está a esquerda é menor, a direita é maior. O que é possível fazer com o array agora que sabemos que uma execução deste código pode colocar um elemento no lugar, além de organizar os outros?

O que acontece se rodarmos o algoritmo para mais elementos? Veremos em breve!

CAPÍTULO 14

O QUICK SORT

O nosso array foi pivotado uma vez. Se ele foi particionado, o elemento que anteriormente ocupava a última posição já foi colocado na posição correta, que será a definitiva.

O Guilherme está na posição que merece, os demais ainda não. Tanto os elementos que estão posicionados à esquerda como os que estão à direita do pivô não estão adequados. Talvez, por sorte, algum esteja ocupando a posição correta — como a Ana, por exemplo. Mas não sabemos com segurança. O único de que temos certeza é o que usamos como parâmetro para o `particiona`.

Como garantir que todos os outros estejam em seu lugar? Veremos o algoritmo que fará isso acontecer.

14.1 PARTICIONAR APÓS PARTICIONAR, E DEPOIS PARTICIONAR NOVAMENTE

Se o Guilherme está na posição correta e os itens da esquerda não estão, podemos posicionar a Mariana adequadamente? Sabemos como fazer isto. Também sei como reposicionar corretamente a Ana. Se mandarmos particionar os quatro primeiros elementos, onde a Mariana vai terminar? Na posição certa dela.

O mesmo acontecerá se mandarmos particionar os quatro últimos elementos do array: a Ana ficará na posição certa. Isto é, logo depois que particionamos o array na primeira vez, rodamos o

mesmo algoritmo para cada uma das duas partes menores.

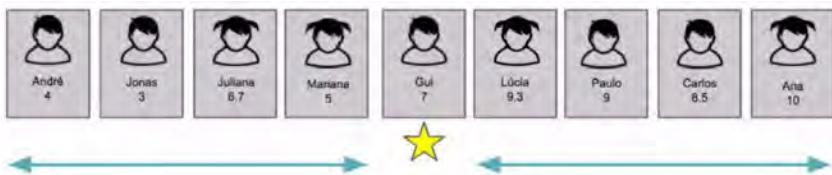


Figura 14.1: Partitiona, partitiona, partitiona — passo 1

Vamos começar a partitionar a parte da esquerda? O pivô será a Mariana. Ela ficará na casinha 2, a posição correta.



Figura 14.2: Partitiona, partitiona, partitiona — passo 2

Porém, os elementos que ficaram à sua esquerda estão em posições adequadas? E quem ficou à direita? Não sabemos. Precisaremos rodar o algoritmo de partitionar também para quem está posicionado tanto à direita como à esquerda.



Figura 14.3: Partitiona, partitiona, partitiona — passo 3

Se rodarmos para estes elementos também, o que acontecerá? Veremos. Vamos partitionar o André e o Jonas. Quem será o pivô? O Jonas.

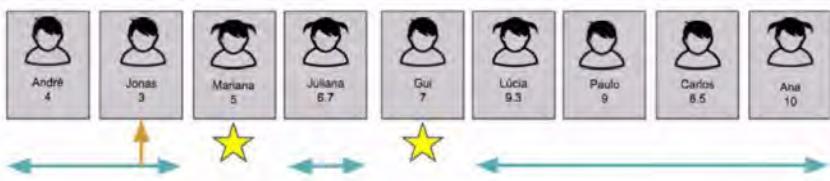


Figura 14.4: Partitiona, partitiona, partitiona — passo 4

Logo, colocaremos o Jonas no lugar correto.



Figura 14.5: Partitiona, partitiona, partitiona — passo 5

Agora precisamos pivotar os elementos que estão à esquerda e à direita do Jonas.



Figura 14.6: Partitiona, partitiona, partitiona — passo 6

Porém, no lado esquerdo do Jonas, não temos elementos. Se o número de elementos é igual a 0, não fará sentido particionar.



Figura 14.7: Partitiona, partitiona, partitiona — passo 7

No lado direito, temos apenas um elemento. Se temos de ordenar um elemento, também não será preciso reposicioná-lo.



Figura 14.8: Particiona, particiona, particiona — passo 8

Será o mesmo com a Juliana. Também teremos apenas um elemento e, por isso, não vamos reposicioná-la.



Figura 14.9: Particiona, particiona, particiona — passo 9

Agora particionaremos os quatro elementos posicionadas à direita. Quem será o pivô? A Ana.



Figura 14.10: Particiona, particiona, particiona — passo 10

Vamos colocá-la no seu lugar! Ela já ocupa a posição adequada.



Figura 14.11: Particiona, particiona, particiona — passo 11

A Ana está no lugar certo. Em seguida, teremos de particionar à esquerda e à direita do pivô.



Figura 14.12: Particiona, particiona, particiona — passo 12

Começaremos pelo lado esquerdo. Quem será o novo pivô? O Carlos.



Figura 14.13: Particiona, particiona, particiona — passo 13

Ao pivotarmos o Carlos, precisamos repetir a ação com os elementos posicionados à sua esquerda e à sua direita.



Figura 14.14: Particiona, particiona, particiona — passo 14

No entanto, à esquerda do Carlos está vazia e só nos resta elementos à direita. Quem será o pivô dos itens restantes? A Lúcia.



Figura 14.15: Partitiona, particiona, particiona — passo 15

A Lúcia está na posição correta. Temos agora de pivotar à sua esquerda e direita.



Figura 14.16: Partitiona, particiona, particiona - passo 16

Vamos pivotar o lado esquerdo. Teremos apenas um elemento, logo ele permanecerá no mesmo lugar.



Figura 14.17: Partitiona, particiona, particiona — passo 17

À direita, não haverá elementos, por isso, não precisaremos fazer nada. Faltou um último pivotamento, novamente de tamanho 0.



Figura 14.18: Particiona, particiona, particiona — passo 18

O que faremos? Nada, novamente.



Figura 14.19: Particiona, particiona, particiona — passo 19

O array está completamente ordenado. Nós usamos a sacada de, ao pivotar um elemento, colocá-lo em seguida na posição certa. Quando mandávamos ordenar os elementos posicionados à direita e à esquerda do novo pivô, pivotávamos e ordenávamos os pedaços menores que se formavam.

Repetimos o processo até que precisávamos pivotar 0 ou 1, e não era preciso fazer nada. Seguimos por todos elementos, até que array estava ordenado e cada um ocupava a sua posição certa.

Vamos revisar o que fizemos. Para ordenar **de** uma posição **até** a outra, precisamos considerar o número de elementos. Se ele for de 0 até 1, não é preciso fazer nada. Caso ele seja maior, mandamos particionar o pedaço e colocar o pivô na posição adequada. Ordenamos os elementos posicionados à esquerda e à direita do pivô. Em seguida, o algoritmo vai ordenando os trechos menores que vão se formando. Ele seguirá pivotando, até que todos os elementos estejam ordenados. Vamos tentar implementar isto?

14.2 ORDENANDO ATRAVÉS DAS PARTIÇÕES

Vamos usar a função de pivotar, o `quebraNoPivo`, que na verdade pode ser renomeada como `particiona`, afinal já está bem claro o que ela faz. Ela particionará o array usando como pivô o último elemento, com o objetivo de ordenar a lista completa. Então, criaremos uma classe nova de teste que se chamará `TestaOrdenacaoRapida`.

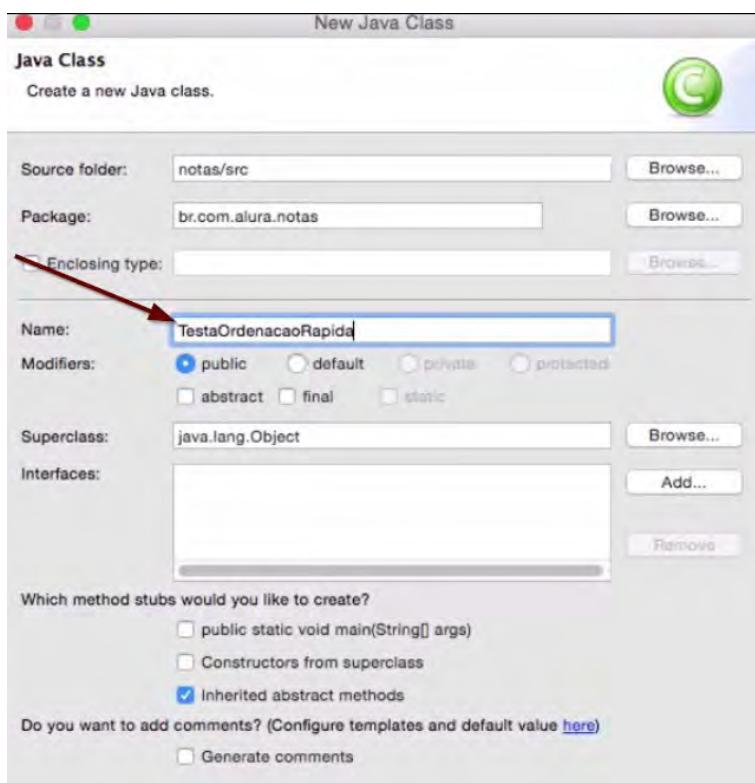


Figura 14.20: Testando a ordenação rápida

Dentro do `TestaOrdenacaoRapida`, criaremos o método `main()`.

```
package br.com.alura.notas;
```

```
public class TestaOrdenacaoRapida {  
    public static void main(String[] args) {  
        }  
}
```

Vamos copiar as notas do método `main` do `TestaPivota`. Nós utilizaremos as mesmas notas.

```
Nota guilherme = new Nota("guilherme", 7);  
Nota[] notas = {  
    new Nota("andre", 4),  
    new Nota("carlos", 8.5),  
    new Nota("ana", 10),  
    new Nota("jonas", 3)  
    new Nota("juliana", 6.7),  
    new Nota("lucia", 9.3),  
    new Nota("paulo", 9),  
    new Nota("marianna", 5),  
    guilherme  
  
};
```

No fim, queremos imprimir também o resultado da ordenação. Copiaremos da outra classe também o `for`:

```
for(int atual = 0; atual < notas.length; atual++) {  
    Nota nota = notas[atual];  
    System.out.println(nota.getAluno() + " " + nota.getValor());  
}
```

Nosso código ficará assim:

```
Nota guilherme = new Nota("guilherme", 7);  
Nota[] notas = {  
    new Nota("andre", 4),  
    new Nota("carlos", 8.5),  
    new Nota("ana", 10),  
    new Nota("jonas", 3)  
    new Nota("juliana", 6.7),  
    new Nota("lucia", 9.3),  
    new Nota("paulo", 9),  
    new Nota("marianna", 5),  
    guilherme  
  
};
```

```
    for(int atual = 0; atual < notas.length; atual++) {  
        Nota nota = notas[atual];  
        System.out.println(nota.getAluno() + " " + nota.getValor());  
    }
```

Nós precisaremos ordenar os elementos, por isso, teremos de pivotar. Vamos ordenar do 0 até o `notas.length`. Colocaremos a seguinte linha acima do `for` :

```
ordenar(notas, 0, notas.length);
```

Mais abaixo, vamos criar a função de ordenação e teremos o de e o ate :

```
private static void ordena(Nota[] notas, int de, int ate) {  
}
```

Nós vamos particionar as notas do de ao ate .

```
private static void ordena(Nota[] notas, int de, int ate) {  
}
```

Isto vai nos devolver a posição do pivô (`posicaoDoPivo`). O algoritmo nos informará a posição adequada para o elemento.

```
private static void ordena(Nota[] notas, int de, int ate)  
    int posicaoDoPivo = particiona(notas, de, ate);  
{  
}
```

Se o pivô estiver na posição correta e todos os elementos posicionados à esquerda forem menores, assim como os localizados à direita forem maiores, basta ordenarmos estas duas partes. Então, mandaremos ordenar as notas da posição de até a `posicaoDoPivo` .

```
ordena(notas, de, posicaoDoPivo);
```

Depois, mandaremos ordenar os elementos da direita. Ordenaremos as notas da `posicaoDoPivo + 1` até o fim.

```
ordena(notas, posicaoDoPivo + 1, ate);
```

A parte da direita será ordenada também. Mas esta será a maneira de ordenar para sempre? Não, se temos 0 elementos, não será preciso fazer nada. Logo, o número de elementos será: até menos o de .

```
int elementos = ate - de;
```

Se (if) o número de elementos for maior do que 1, ou seja, superior a dois elementos, teremos de ordená-los. Quando temos um número de elementos igual ou menor do que 1, não precisaremos nos preocupar, porque já estará ordenado. Vamos ver como ficou o código com as novas linhas:

```
private static void ordena(Nota[] notas, int de, int ate) {  
    int elementos = ate - de;  
    if(elementos > 1) {  
        int posicaoDoPivo = particiona(notas, de, ate);  
        ordena(notas, de, posicaoDoPivo);  
        ordena(notas, posicaoDoPivo + 1, ate);  
    }  
}
```

Em seguida, copiaremos da classe `TestaPivota` as funções `particiona()` e `troca()`. Ambas serão usadas após o `ordena`.

```
private static void ordena(Nota[] notas, int de, int ate) {  
    int elementos = ate - de;  
    if(elementos > 1) {  
        int posicaoDoPivo = particiona(notas, de, ate);  
        ordena(notas, de, posicaoDoPivo);  
        ordena(notas, posicaoDoPivo + 1, ate);  
    }  
}  
  
private static int particiona(Nota[] notas, int inicial, int termino) {  
    int menoresEncontrados = 0;  
  
    Nota pivo = notas[termino - 1];  
    for(int analisando = 0; analisando < termino - 1; analisando++) {  
        Nota atual = notas[analisando];  
        if(atual.getValor() <= pivo.getValor()) {  
            troca(notas, analisando, menoresEncontrados);  
            menoresEncontrados++;  
        }  
    }  
}
```

```

        menoresEncontrados++;
    }
}
troca(notas, termino -1, menoresEncontrados);
return menoresEncontrados;
}

private static void troca(Nota[] notas, int de, int para) {
    Nota nota1 = notas[de];
    Nota nota2 = notas[para];
    notas[para] = nota1;
    notas[de] = nota2;
}

```

Após adicionarmos as duas funções, testaremos nossa ordenação. Quando rodamos o programa, o resultado será:

```

jonas 3.0
andre 4.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
lucia 9.3
ana 10.0

```

Todos os elementos estão ordenados.

14.3 QUEM É ESTE ALGORITMO ESPERTO?

Repare que o algoritmo foi muito esperto em implementar essa divisão em partes para resolver o problema, de forma similar ao que fizemos com o Merge Sort. Na prática, quando pensado, esse algoritmo era visto como tão rápido que seu nome é Quick Sort.

14.4 O PRÓXIMO DESAFIO: A BUSCA

Nós conseguimos organizar o array apenas com a ação de particionar e encontrar a posição de cada item. Nós dividimos o número de elementos em dois e colocamos todos nas posições

certas. Repetimos o processo diversas vezes. Isto nos lembra a nossa estratégia de dividir o trabalho entre diversas pessoas.

Neste caso, o trabalho que cada pessoa teria de executar seria identificar a posição adequada para cada elemento. Dividiríamos a ação em diversas partes até que a quantidade de itens se tornasse muito pequena. Assim conseguimos terminar o trabalho. Então, precisamos dividir o número de elementos e reposicioná-los em seguida. À medida que o nosso algoritmo separa os elementos, ele posiciona um item novo. No fim, o array estará ordenado.

Agora que já vimos algoritmos mais avançados de ordenação, entraremos em outra pergunta importante: será que existe um elemento dentro de um array?

CAPÍTULO 15

A BUSCA LINEAR

Nós sabemos fazer diversas coisas com um array de elementos: podemos dizer quem é menor, maior ou igual. Por exemplo, a nota 3 é menor do que a nota 10. Da mesma forma que a carta 3 é menor do que a 10. Continuando com as comparações, na nossa lista de alunos, o Guilherme está posicionado antes do que o Paulo. Nós trabalhamos com elementos maiores e menores.



Figura 15.1: Buscando

Alguns itens são iguais também: o Guilherme é igual ao Guilherme, o Paulo é igual ao Paulo, as cartas 3, 7 e Valete são equivalentes a elas mesmas. Podemos executar várias ações com estes tipos de array.

No entanto, o que faremos se quisermos fazer tarefas diferentes das quais realizamos até agora? Por exemplo, se eu fiz a prova do Enem e foi divulgado o resultado, quero saber se fui aprovado. Precisarei encontrar a minha nota na lista com todos os resultados. Logo, terei de procurar o meu nome.

Vamos imaginar esta situação: temos a lista com nove dos alunos que fizeram uma prova — depois deles existem mais 1 milhão de notas. O que acontece em seguida? A Mariana vai conferir o seu

resultado e fará a pergunta: "Será que eu passei? Qual será a minha nota?". Para responder, precisamos buscá-la na lista.

Como fazemos isto? Nós conferimos se o nome dela está na lista. E como a procuramos dentro da lista? Se observarmos os alunos, poderemos encontrar a Mariana.

Mas como foi o processo de busca? Por onde passamos para encontrá-la? O que foi feito para decidirmos se o elemento está incluso ou não? Como descobrimos se o elemento está ou não dentro do array? Mentalmente, realizamos uma tarefa quando foi feita a pergunta: "A Mariana está na lista?".

Modificaremos a pergunta: "Alguém tirou a nota 6.7?". Em que você pensou ao ver a pergunta? Como você descobriu que o elemento com nota 6.7 está dentro do array? Usaremos o mesmo algoritmo utilizado recentemente para resolver: como descobrir se um elemento faz parte do array? Uma pergunta extremamente comum, que pode ser feita para diversas áreas.

Por exemplo, as perguntas: "Será que fui ou não classificado? Temos a nota 6.7 na nossa lista? E a nota 3.5? A Mariana está na lista dos aprovados? O João está entre os alunos?". Todas estas questões envolvem descobrir se um elemento integra um array.

Você consegue descobrir esta informação? É provável que consiga. Qual foi o processo? Pense no algoritmo que você usou mentalmente e tente descrevê-lo no papel.

15.1 A IDEIA DA BUSCA LINEAR

Vamos pensar em como resolveríamos o seguinte problema: existe alguém com a nota 5 dentro da lista? O elemento 5 faz parte do meu array?



Figura 15.2: A busca linear — passo 1

Observando a lista, podemos encontrar o elemento com nota 5. Qual foi o processo feito por nós: analisamos o André e percebemos que ele não tirou 5. Seguimos observando os demais elementos. O Carlos não tirou a nota 5, e a Ana também não. Analisei o Jonas, e ele não tirou 5. Continuei com a Juliana, o Gui e o Paulo, e nenhum deles tirou essa nota. Então, ao analisar a Mariana, encontrei uma pessoa que tirou a nota 5.

Temos alguém que tirou a nota 5 na lista. Nossa pergunta foi respondida. Foi fácil resolver a questão: passamos por todos os elementos e observamos se algum aluno havia tirado 5.

Daremos mais um exemplo. Será que alguém tirou a nota 3.7? Como faremos? Vamos analisar o André, a Ana, o Jonas, a Juliana. Seguiremos observando o Gui, o Paulo, a Mariana e a Lúcia. Não encontramos algum elemento que tenha a nota 3.7.



Figura 15.3: A busca linear — passo 2

Se dermos sorte, encontraremos a resposta da pergunta que fizermos já na primeira pessoa analisada. Resolver o problema analisando o primeiro elemento é muita sorte! Mas se não for o nosso caso, e o nosso array não tiver o elemento que estamos

buscando, precisaremos passar pelo array inteiro para resolver a questão. Por mais que ele possa ser ruim no pior caso, essa é a busca tradicional: simplesmente olhar em todos os elementos, buscando aquilo que procuramos.

Imagine um placar com uma lista das mil pessoas aprovadas na faculdade e, depois de procurar o meu nome até o fim, descubro que não passei. Este tipo de busca não parece ser inteligente. Quando você procura o seu nome em uma lista, é assim que realiza a busca? Você precisa passar por cada um dos nomes? Reflita, se temos uma lista de nomes e buscamos alguém específico, o que costumamos fazer?

15.2 IMPLEMENTAÇÃO DA BUSCA LINEAR

O nosso array já está ordenado, porque nós mandamos organizar as notas com o algoritmo de ordenação usado no `TestaOrdenacaoRapido`.

```
Nota[] notas = {
    new Nota("andre", 4),
    new Nota("carlos", 8.5),
    new Nota("ana", 10),
    new Nota("jonas", 3),
    new Nota("juliana", 6.7),
    new Nota("lucia", 9.3),
    new Nota("paulo", 9),
    new Nota("marianna", 5),
    guilherme
};

ordena(notas, 0, notas.length);

for(int atual = 0; atual < notas.length; atual++) {
    Nota nota = notas[atual];
    System.out.println(nota.getAluno() + " " + nota.getValor());
}
```

Agora quero saber se temos, dentro da lista, a pessoa com uma

nota específica: 9.3. O que faremos é buscar entre as notas alguém com essa nota. Vamos acrescentar a linha de `busca` no código:

```
ordena(notas, 0, notas.length);
busca(notas, 9.3);
```

Vamos procurar? Porém, quando fazemos uma busca da mesma forma quando ordenamos, precisamos especificar quais elementos serão analisados. No nosso caso, será de 0 até `notas.length`.

```
busca(notas, 0, notas.length, 9.3);
```

Será que existe algum elemento com esta nota dentro do array? Para isto, criaremos a função `busca`. Então, buscaremos na lista de notas de `a` até o valor específico que estamos buscando.

```
private static void busca(Nota[] notas, int de, int ate, double buscando) {
}
```

Será que a nota existe dentro do array? Podemos buscá-la, varrendo todos os elementos. Para isto, vamos criar um `for`.

```
private static void busca(Nota[] notas, int de, int ate, double buscando) {
    for(int atual = de; atual < ate; atual++) {
        }
}
```

Se (`if`) o `notas[atual].getValor()` for igual à nota que estamos buscando, ela existirá no array. Ela estará na posição `atual`.

```
for(int atual = de; atual < ate; atual++) {
    if(notas[atual].getValor() == buscando) {
        return atual;
    }
}
```

Se a nota não existir, vamos retornar -1.

```
if(notas[atual].getValor() == buscando) {  
    return atual;  
}  
}  
return -1;
```

Mudaremos o tipo de retorno de `void` para `int`.

```
private static int busca(Nota[] notas, int de, int ate, double buscando) {  
  
}
```

Vamos especificar que `int` será igual ao do `busca`.

```
ordena(notas, 0, notas.length);  
int encontrei = busca(notas, 0, notas.length, 9.3);
```

Em seguida, adicionaremos o `System.out`:

```
ordena(notas, 0, notas.length);  
int encontrei = busca(notas, 0, notas.length, 9.3);  
  
System.out.println("Encontrei a nota em " + encontrei + ".");
```

Agora podemos testar o programa. Vamos rodá-lo. Clicaremos em *Run As*, depois em *Java Application*, e o resultado será:

```
Encontrei a nota em 7.  
jonas 3.0  
andre 4.0  
mariana 5.0  
juliana 6.7  
guilherme 7.0  
carlos 8.5  
paulo 9.0  
lucia 9.3  
ana 10.0
```

Encontramos! Nós procuramos pelo array completo e encontramos a nota especificada na posição 7.

15.3 O PRÓXIMO DESAFIO: UMA BUSCA MAIS RÁPIDA

Mas repare que foi preciso criar um `for` "muito grande", que, no pior caso, tem de passar por todos os elementos para dizer que um elemento não está contido em nosso array:

```
private static int busca(Nota[] notas, int de, int de, int ate, double buscando) {
    for(int atual = de; atual < ate; atual++) {
        if(notas[atual].getValor() == buscando) {
            return atual;
        }
    }
    return -1;
```

Será que esta é a melhor solução?

CAPÍTULO 16

A BUSCA BINÁRIA

O que fazer quando eu tenho uma lista de nomes e quero saber se estou listado? Ou se tenho diversos números, e quero saber se um número específico integra a lista? Ou se tenho uma lista de produtos, e quero saber se um produto determinado faz parte dela? Como nós costumamos resolver problemas assim?

Outro exemplo seria a lista de chamada que os professores costumam usar. Temos uma lista assim:



Figura 16.1: Ordenar antes de buscar — passo 1

Esta é a forma que são feitas as listas de chamada? Precisamos procurar nosso nome passando por cada uma das pessoas da lista. Quando temos uma lista de alunos aprovados com as notas que cada um deles, os nomes aparecem desordenados ou ordenados?



Figura 16.2: Ordenar antes de buscar — passo 2

No nosso caso, os elementos foram ordenados considerando as

notas. Em uma lista, os elementos costumam aparecer ordenados. E quando procuramos um item, geralmente buscamos pela ordem em que ele foi colocado. Isto é, eles primeiros foram ordenados, e então procuramos um elemento listado.

No exemplo, ordenamos pelas notas. Por isso, repetirei uma pergunta: alguém tirou a nota 9.5? Sim ou não? Como você fará para descobrir se alguém tirou a nota 9.5? Observe que o array está ordenado pela pontuação dos alunos.

Farei outra pergunta: alguém tirou a nota 5? Como você fará para encontrar a nota 5? Alguém tirou a nota 7.3? Qual é o processo para descobrir se alguém tirou essa nota? Alguém tirou a nota 13 ou 2? Ou 5.5?

Como você está fazendo para encontrar as notas dentro de um array ordenado pela pontuação? Você fará o mesmo processo quando for procurar o seu nome em uma lista de presença, em que os nomes também estão ordenados? Como você encontraria o seu nome na lista? O que fazer para encontrar o nome do seu amigo em uma agenda telefônica?

Pense no processo que você faz para encontrar a nota no meio da lista ordenada pela pontuação.

16.1 BUSCANDO EM UM ARRAY ORDENADO

Temos uma lista de presença com todos os nomes ordenados. Como farei para procurar o meu nome entre os demais? Meu nome é Guilherme. Começo a procurar os nomes pelas letras iniciais: A, B, C, D, F, G. O meu nome estará na letra G! É uma maneira de pesquisá-lo. O mesmo pode ser feito com nossa lista:



Figura 16.3: Buscando em um array ordenado

Os alunos foram ordenados pela nota. Se eu perguntar: alguém tirou a nota 5? Vamos verificar: o Jonas tirou 3, o André tirou 4, e a Mariana tirou 5. Temos a nota 5 na lista!

Nova pergunta: tem alguém com a nota 5.7? Vamos verificar. O Jonas tirou 3, o André tirou 4, a Mariana tirou 5 e a Juliana tirou 6.7. Então, não temos um elemento com a nota 5.7.

Em seguida, vamos verificar se tem alguém com a nota 9.9? O Jonas tirou 3, o André tirou 4, a Mariana tirou 5 e a Juliana tirou 6.7, o Gui tirou 7, o Carlos tirou 8.5, o Paulo tirou 9, a Lúcia tirou 9.3 e a Ana tirou 10. Não temos a nota 9.9 na lista. Mas esta é forma como você vai procurar se eu perguntar sobre a nota 9.9?

Observe novamente a relação das notas dos alunos. Se eu fizer a pergunta: alguém tirou a nota 8.6? Em que direção começaremos a olhar? Vou perguntar por outra pontuação: tem alguém com a nota 3.3? Por onde você começaria a analisar na lista?

Perceba que nós não começamos a analisar no sentido da esquerda para a direita. Por onde começamos a analisar quando procuramos um número dentro de um array? Da mesma forma, por onde começamos a analisar quando procuramos alguém em uma lista de nomes ordenados?

16.2 DIVIDINDO O PROBLEMA EM DOIS PARA DEPOIS BUSCAR

Temos a lista com todas as notas ordenadas. Poderia ser uma

lista de presença com todos os nomes ordenados, ou um catálogo telefônico com todos os nomes em ordem. Queremos procurar algo em específico dentro de uma lista.

Se eu pedir que você procure alguém com a nota 3, em que direção você olhará na lista? Se eu pedir que você procure alguém com a nota 7, você vai começar a analisar a partir do fim do array. Se eu perguntar sobre alguém com a nota 5, você analisará já a partir do meio do array.

Por que direcionamos o nosso olhar desta maneira? Por que, automaticamente, o nosso olhar se dirige para um determinado lado ou para o meio, dependendo do elemento que estivermos procurando? Porque sabemos que, se procurarmos do menor para o maior da lista, e o item que estivermos buscando tiver um valor grande, demoraremos para encontrá-lo. Se procurarmos um número pequeno, e começarmos a analisar a lista do maior elemento para o menor, vamos demorar para achá-lo.

Nós tentaremos otimizar o processo e tornaremos a busca mais rápida do que se passássemos por todos os elementos. Nós só passaremos pelos elementos com uma localização aproximada de onde pode estar o que estamos procurando.

Então, quando recebemos uma lista de presença, não começaremos a procurar o Guilherme pelo fim da relação de nomes. Também não vamos começar pelo início. Analisaremos a lista a partir do meio. Ao encontrarmos a posição da letra I, saberemos que o Guilherme estará em uma posição anterior. Se olharmos para a esquerda, veremos os nomes com a letra G, como Gabriel. Se encontrarmos o Gabriel, o elemento que buscamos estará posicionado à direita. Logo, encontraremos o Guilherme.

Vamos detectar aproximadamente onde está posicionado o elemento. Em uma lista com 100 mil nomes, não analisaremos

todos os elementos se o nome que buscamos começa Z. Não somos loucos de fazer isso, nem devemos ensinar esta forma de busca para o computador. Nós o ensinaremos a fazer da mesma maneira como fazemos, uma maneira de procurar mais inteligente.

Porém, não conseguiremos que o computador fique analisando a todo o momento se um número é pequeno. Na verdade, seria uma tarefa possível. Mas se ele tivesse de saber identificar "este é um número pequeno, devemos começar pela esquerda, ou este é um número grande, analisaremos a partir da direita" seria um processo mais complexo. Nós queremos encontrar uma forma mais simples de especificar para o programa "em vez de procurar por todos os elementos, busque em apenas uma parte".

Nós já trabalhamos com pedaços de um array diversas vezes. Por exemplo, dividimos uma pilha de baralho em dois, para que duas pessoas organizassem cada parte. Nós também já particionamos um array em dois, e mandávamos executar uma ordenação em cada um dos pedaços. Se estamos procurando um elemento que esteja entre o primeiro e o último item, o que poderemos fazer? Nós podemos dividir.



Figura 16.4: Dividir para buscar — passo 1

Tente dividir a nossa lista e encontrar o elemento 8.5.

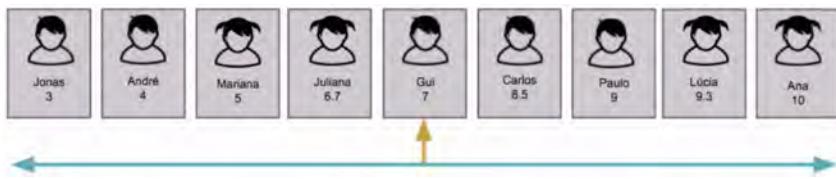


Figura 16.5: Dividir para buscar — passo 2

Depois que você fizer a divisão, qual será o processo para encontrar a nota? Em que direção você vai olhar?

E se estivéssemos procurando o elemento 3.4? Após dividir o total de elementos, por onde você começaria analisar? Se procurássemos a nota 7, em que direção você olharia na lista? Se procurássemos o elemento 1, qual seria a direção? Você percebeu alguma regra? Qual foi?

16.3 DIVIDINDO, DIVIDINDO NOVAMENTE E DIVIDINDO MAIS UMA VEZ

Dado um array com diversos itens já ordenados, quero saber se um elemento específico está dentro dele. Independentemente se procuro o meu nome ou a nota de um aluno, como realizaremos a busca? Podemos procurar entre todos os elementos da esquerda para direita — o que demoraria bastante —, ou podemos quebrar a lista em duas partes.

Vamos procurar a nota 9.3. Será que ela está na lista? Por isso, quebraremos a lista em dois.

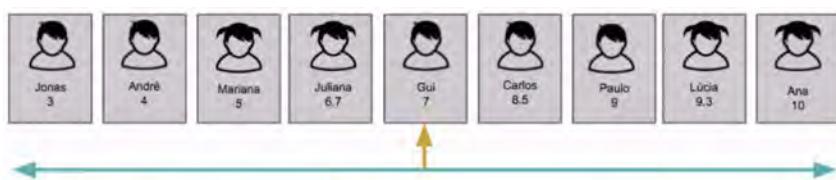


Figura 16.6: Dividindo, dividindo, dividindo — passo 1

O elemento que ficará no meio será o Gui. Ele tirou 9.3? Não. Inclusive, o Guilherme tirou uma nota menor do que 9.3. Se ele é menor, adiantaria procurar entre os elementos posicionados à esquerda? Não. O elemento procurado está posicionado à direita. Podemos ignorar todos os itens à esquerda e o Gui também.

Por que buscaríamos entre os elementos da esquerda? Agora, sobraram apenas os elementos da direita.



Figura 16.7: Dividindo, dividindo, dividindo — passo 2

Existe o elemento 9.3 entre os que estão posicionados à direita? Sim. Vamos quebrar os elementos que sobraram no meio.



Figura 16.8: Dividindo, dividindo, dividindo — passo 3

O Paulo é o novo pivô. Ele tirou 9.3? Não. Ele é menor? Sim. Então, o elemento que estamos buscando estará à direita. Vamos ignorar também os que ficaram à esquerda.



Figura 16.9: Dividindo, dividindo, dividindo — passo 4

Nós buscaremos novamente.



Figura 16.10: Dividindo, dividindo, dividindo — passo 5

Qual elemento no meio? A Lúcia. Ela tirou 9.3? Sim. Encontramos o elemento! Quantas comparações nós fizemos? Comparamos com a nota 7, 9 e 9.3. Fizemos três comparações praticamente em vez de fazermos oito comparações. Parece ter sido um processo mais rápido. Por quê? Porque a cada vez que observamos o array, nós eliminávamos metade dos elementos.

Não precisamos mais analisar cada elemento e eliminar os menores. Começamos a análise pelo meio. Nós compararmos com um elemento que era menor, depois eliminamos o que não servia. Repetimos o processo e fomos eliminando metade dos elementos em cada parte. Assim encontramos o resultado rapidamente.

Se tivermos **dezesseis** elementos e conseguirmos eliminar a metade com uma comparação, sobrarão apenas **oito**. Com duas comparações, sobrarão **quatro**. Com três, sobrarão **dois**. Com quatro, sobrará **um**. Se tivermos dezesseis elementos, com quatro comparações chegaremos ao resultado.

Se tivermos 1024 elementos, no pior caso, teremos de passar por cada um deles. Precisaremos de 1024 comparações. E se usarmos o algoritmo que divide no meio? Com **uma** comparação, sobrarão 512 elementos, que estarão à direita ou à esquerda. Com **duas**, ficam 256; e com **três**, 128, que poderão ser da direita ou da esquerda. Desta forma, com **quatro**, restam 64; com **cinco**, 32; com **seis**, 16; com **sete**, oito; com **oito**, quatro; com **nove**, dois; e com apenas mais

uma comparação, sobrará apenas **um**.

Em **10** comparações, conseguimos percorrer todos os elementos que nos interessam em um array de 1024 elementos, e encontrar (ou não) o item que procurávamos. Dez comparado com 1024, existe uma grande diferença.

O algoritmo de busca que dividimos o array em duas partes, e seguimos procurando apenas no pedaço que nos interessa, é mais rápido do que o que estamos acostumados a utilizar. Nós estamos interessados em implementar este algoritmo agora.

16.4 IMPLEMENTANDO A BUSCA PELA METADE

Nós implementamos a busca, mas vimos que varrer o array inteiro para procurar um elemento em um que tenha 1024 itens, por exemplo, é um processo muito lento. Precisaríamos passar por todos os elementos, sendo que podemos estimar se o que estamos buscando está posicionado na direita ou na esquerda.

Podemos ir cortando pedaços do array e buscar apenas onde temos interesse. Vamos alterar o nosso código para que ele faça isto?

```
private static int busca(Nota[] notas, int de, int de, int ate, double buscando) {
    for(int atual = de; atual < ate; atual++) {
        if(notas[atual].getValor() == buscando) {
            return atual;
        }
    }
    return -1;
}
```

Então, vamos tirar o `for` e tentaremos descobrir onde está o elemento. Mas precisamos ter uma base. Será que ele está à esquerda ou à direita do meio? Então, podemos usar o **meio** como base.

meio será o de mais o ate dividido por 2.

```
private static int busca(Nota[] notas, int de, int de, int ate, double buscando) {
    int meio = (de + ate) / 2;
}
```

Qual é a nota que está no meio? Será o notas[meio] . Adicionaremos mais uma linha:

```
private static int busca(Nota[] notas, int de, int de, int ate, double buscando) {
    int meio = (de + ate) / 2;
    Nota nota = notas[meio];
}
```

Nós vamos querer verificar se a nota do meio é a que estamos procurando. Vamos criar um if que verifique se o buscando é igual a nota.getValor .

```
if(buscando == nota.getValor()) {  
}
```

Caso seja, ficaremos satisfeitos em saber que encontramos o resultado. O return está na posição do meio .

```
if(buscando == nota.getValor()) {  
    return meio;  
}
```

Mas se o elemento não estiver na posição do meio , teremos de seguir procurando. Onde será que ele está? No lado direito ou esquerdo? Se a nota que estamos buscando for menor do que a do meio (nota.getValor), então ela estará na esquerda.

```
if(buscando < nota.getValor()) {  
    ta na esquerda  
}
```

Se buscando não for igual ou menor, a nota estará na direita.

```
if(buscando < nota.getValor()) {  
    ta na esquerda  
  
}  
ta na direita
```

Veremos como ficou o nosso código:

```
private static int busca(Nota[] notas, int de, int de, int ate, double buscando) {  
    int meio = (de + ate) / 2;  
    Nota nota = notas[meio];  
    if(buscando == nota.getValor()) {  
        return meio;  
  
    }  
    if(buscando < nota.getValor()) {  
        ta na esquerda  
    }  
    ta na direita  
}
```

Então, temos algumas possibilidades para estimar a posição do elemento. A nota que buscamos pode estar exatamente no meio. Caso não esteja, ela pode ser menor e estar à esquerda, ou pode ser maior do que o `meio` e estar à direita. Temos estes três casos.

Como faremos para buscar na esquerda?

```
if(buscando < nota.getValor()) {  
    ta na esquerda  
}
```

Pediremos que o algoritmo retorne (`return`) e busque nas minhas notas, da posição inicial `de` até o `meio`. Porém, sabemos que, neste caso, o `meio` não é a posição certa, então vamos diminuir 1 e eliminar a posição do `meio`. Nossa objetivo é descobrir o `buscando`.

```
if(buscando < nota.getValor()) {  
    return busca(notas, de, meio -1, buscando)  
}
```

E como faremos para buscar apenas na metade da direita? Para

buscar na parte da direita, adicionaremos a seguinte linha:

```
return busca(notas, meio + 1, ate, buscando);
```

O `if` ficará assim:

```
if(buscando == nota.getValor()) {  
    return meio;  
  
}  
if(buscando < nota.getValor()) {  
    return busca(notas, de, meio -1, buscando)  
}  
return busca(notas, meio + 1, ate, buscando);
```

Vale lembrar de que temos certeza de que o elemento não está no meio. Se estivesse, teríamos interrompido a busca mais acima.

Vamos testar o código e ver se ele vai encontrar a nota 9.3? Ao rodarmos o programa, ele encontrará o seguinte resultado:

```
Encontrei a nota em 7.  
jonas 3.0  
andre 4.0  
mariana 5.0  
juliana 6.7  
guilherme 7.0  
carlos 8.5  
paulo 9.0  
lucia 9.3  
ana 10.0
```

Ele encontrou a nota na posição 7. A Lúcia, que tirou 9.3, está na posição 7.

16.5 DESEMPENHO AO DIVIDIR E BUSCAR

Vamos ver passo a passo como o nosso algoritmo se comportou? Veremos como ele dividiu o array.

Daremos um `System.out` e, no momento em que formos buscar, ele falará: "Buscando" + buscando + " entre " + de +

```

" e " + ate); .

private static int busca(Nota[] notas, int de, int ate, double buscando) {
    System.out.println("Buscando" + buscando + "entre" + de + " e "
+ ate);
    int meio = (de + ate) / 2;
    Nota nota = notas[meio];
    if(buscando == nota.getValor()) {
        return meio;
    }
    if(buscando < nota.getValor()) {
        return busca(notas, de, meio -1, buscando)
    }
    return busca(notas, meio + 1, ate, buscando);
}

```

Vamos rodar o algoritmo, e o resultado será:

```

Buscando 9.3 entre 0 e 9
Buscando 9.3 entre 5 e 9
Encontrei a nota em 7.
jonas 3.0
andre 4.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
lucia 9.3
ana 10.0

```

Ele buscou o 9.3 entre 0 e 9. Logo, ele percebeu que o elemento estava para a direita, por isso, ele buscou entre 5 e 9. Depois, ele encontrou a nota na posição 7. Foi um processo muito rápido. Ele fez duas comparações e, na segunda, ele encontrou o elemento.

Agora tentaremos encontrar outro valor: **6.7**.

```

ordena(notas, 0, notas.length);
int encontrei = busca(notas, 0, notas.length, 6.7);

```

Ao rodarmos o algoritmo novamente, este será o resultado:

```
Buscando 6.7 entre 0 e 9
```

```
Buscando 6.7 entre 0 e 3
Buscando 6.7 entre 2 e 3
Buscando 6.7 entre 3 e 3
Encontrei a nota em 3
jonas 3.0
andre 4.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
lucia 9.3
```

Neste caso, ele buscou a nota 6.7 entre 0 e 9. Ele percebeu que o elemento era menor, então eliminou a parte da direita e buscou apenas na primeira metade menos o elemento do meio, de 0 até 3. Da parte restante, ele observou que era maior do que o meio. Então, ele ignorou o elemento 1. Depois, ele viu que estava mais para a direita que o elemento 2, e buscou entre o 3 e o 3. Então, ele encontrou! A nota estava na posição 3.

16.6 QUANDO NÃO ENCONTRAMOS UM ELEMENTO

É o momento de fazermos mais alguns testes com o nosso código. Nós já testamos e imprimimos o que ele estava buscando, com dois valores que existiam no array. Agora vamos testar dois valores que não existem.

Primeiro, testaremos com o valor **3.7**.

```
ordena(notas, 0, notas.length);
int encontrei = busca(notas, 0, notas.length, 3.7);
```

Ao rodarmos o meu programa, ele vai apresentar uma mensagem de erro. O programa chamou o `busca`, depois o chamou novamente, e repetiu infinitamente até que explodiu. Por que ele chamou o `busca` infinitas vezes? Veremos o que aconteceu.

Ele buscou entre 0 e 9, mas não encontrou e dividiu pela metade. Depois, buscou entre 0 e 3, mas não encontrou novamente. Ele dividiu pela metade e buscou entre 0 e 0. Também não encontrou. Então ele ficou confuso, porque precisou dividir 0 na metade. Não fazia mais sentido continuar.

Quando temos apenas um elemento, só existem duas opções: ele é o que buscamos ou não é. Não adianta continuar a dividir por dois. Então, se o número que temos entre de e ate for igual ou menor do que 1, não adianta continuar com a busca. No entanto, é o que continuamos fazendo.

Quando tínhamos 1 elemento, seguimos com a procura. Mas se temos apenas um, ou ele é ou não o que buscamos. Se chegamos a 0 elementos, ele não é. Então, nós não nos preocupamos com o caso de que o elemento não fosse encontrado no nosso array — o caso do `return -1`.

Como você alteraria essa parte do algoritmo para verificar se a nota não foi encontrada?

```
private static int busca(Nota[] notas, int de, int ate, double buscando) {
    System.out.println("Buscando" + buscando + "entre" + de + " e "
+ ate);
    int meio = (de + ate) / 2;
    Nota nota = notas[meio];
    if(buscando == nota.getValor()) {
        return meio;
    }
    if(buscando < nota.getValor()) {
        return busca(notas, de, meio -1, buscando)
    }
    return busca(notas, meio + 1, ate, buscando);
}
```

O array ficou muito pequeno, mas ele não encontrou o elemento.

Como podemos corrigir o nosso algoritmo para que, nos casos em

que sobre um elemento ou nos quais de seja maior do que ate , ele funcione?

No exemplo, nós testamos a nota 3.7, e o resultado foi:

```
Buscando 3.7 entre 0 e 9
Buscando 3.7 entre 0 e 3
Buscando 3.7 entre 0 e 0
Buscando 3.7 entre 1 e 0
```

Ele buscou de 1 até 0, mas não encontrou a nota e ficou louco, entrando em loop infinito. Então, se (if) de for maior do que ate , significa que não encontramos nada e o código vai retornar -1.

```
private static int busca(Nota[] notas, int de, int ate, double buscando) {
    System.out.println("Buscando " + buscando + " entre " + de + " e " + ate);
    if(de > ate) {
        Nota nota = nota[meio];
        if(buscando == nota.getValor()) {
            return -1;
        }
    }
}
```

Vamos salvar o arquivo, e depois rodamos o programa. O resultado será:

```
Buscando 3.7 entre 0 e 9
Buscando 3.7 entre 0 e 3
Buscando 3.7 entre 0 e 0
Buscando 3.7 entre 1 e 0
Encontrei a nota em -1.
jonas 3.0
andre 4.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
```

```
lucia 9.3  
ana 10.0
```

Ele buscou 3.7 entre 0 e 9, em seguida entre 0 e 3, e depois, entre 0 e 0. Quando ele buscou entre 1 e 0, ele ficou confuso e concluiu que não havia encontrado o item. Por isso, ele devolveu que encontrou a nota na posição -1. Ou seja, precisamos adicionar um `if` ao nosso código: se (`if`) `encontrei` menor ou igual a 0, então teremos descoberto a nota.

```
ordena(notas, 0, notas.length);  
int encontrei = busca(notas, 0, notas.length, 3.7);  
  
if(encontrei >= 0) {  
    System.out.println("Encontrei a nota em " + encontrei + " . ")  
};  
}
```

Senão (`else`), um `System.out` vai dizer: "Não encontrei a nota".

```
if(encontrei >= 0) {  
    System.out.println("Encontrei a nota em " + encontrei + " . ")  
};  
} else {  
    System.out.println("Encontrei a nota");  
}
```

Ao rodarmos novamente, o programa imprimirá:

```
Buscando 3.7 entre 1 e 0  
Não encontrei a nota  
jonas 3.0  
andre 4.0  
mariana 5.0  
juliana 6.7  
guilherme 7.0  
carlos 8.5  
paulo 9.0  
lucia 9.3  
ana 10.0
```

Ele informa que não encontrou a nota 3.7. Se testarmos com uma nota que existe no array, como a nota 9, o programa vai

imprimir o seguinte resultado:

```
Buscando 3.7 entre 1 e 0
Encontrei a nota em 6.
jonas 3.0
andre 4.0
mariana 5.0
juliana 6.7
guilherme 7.0
carlos 8.5
paulo 9.0
lucia 9.3
ana 10.0
```

Ele dirá que encontrou a nota na posição 6.

16.7 DEFININDO A BUSCA BINÁRIA

Nós implementamos uma busca mais inteligente do que a busca tradicional — que varre todo o nosso array. Isto significa que, quando tínhamos 100 elementos, a busca tradicional passava com um `for` pelos 1000 elementos. Se tivéssemos 1000 elementos, o `for` passava por todos eles. Se tivéssemos n elementos, o `for` passava por n elementos.

Analisando o algoritmo da busca tradicional, ele era $O(n)$ e passava por todos os elementos. Era uma busca do tipo linear.

Este algoritmo será rápido se trabalharmos com 100 ou 1000. Mas se forem 1 milhão de elementos, ele começará a ser lento e, por isso, buscamos outra maneira de fazer a busca. Nós procuramos uma forma em que o array já estivesse ordenado.

Quando temos uma lista com os elementos ordenados, podemos direcionar nosso olhar para uma parte dela. Se temos um array organizado, podemos quebrá-lo em duas partes, observar cada uma delas e, depois, realizar a busca em apenas uma.

Essa é a **busca binária**: ela buscará em um dos dois pedaços do

array.

16.8 O PRÓXIMO DESAFIO: COMPARANDO AS BUSCAS

Falta analisar a rapidez da busca binária. Temos a sensação de que ela é mais rápida. Mas quão mais rápida ela é de verdade comparada a outras buscas existentes?

CAPÍTULO 17

ANÁLISE ASSINTÓTICA DAS BUSCAS

Vamos analisar o quanto mais rápida é a busca tradicional, a busca linear, que passa por cada um dos elementos procurando um específico.

Usaremos exemplos, e o primeiro será com a Mariana. Será que ela faz parte do array?



Figura 17.1: Analisando a busca tradicional — passo 1

Caso ela faça parte da lista, como funcionava o nosso algoritmo? Ele passava por cada um dos elementos, e então verificávamos se o item procurado estava dentro. Vamos analisar o primeiro elemento:



Figura 17.2: Analisando a busca tradicional — passo 2

O Jonas é a Mariana? Não. Seguimos para a posição 1.



Figura 17.3: Analisando a busca tradicional — passo 3

O André é a Mariana? Não. Vamos para a posição 2.

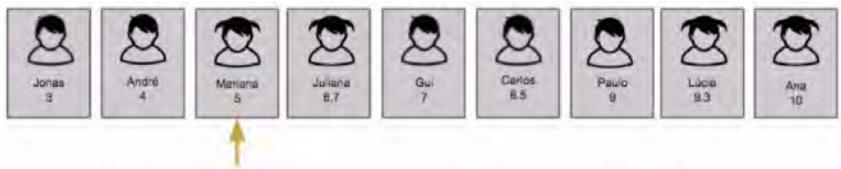


Figura 17.4: Analisando a busca tradicional — passo 4

A Mariana está nesta posição? Sim! Ela não estava na 0, também não estava na 1, mas estava na posição 2. Assim nós passamos por cada um dos elementos, até encontrarmos o que buscávamos.

Em uma busca tradicional, nós varreremos todos os elementos do array. Se tivermos sorte, o elemento que buscamos estará na primeira posição. Se dermos azar, estaremos buscando o último elemento e teremos de passar pelo Jonas, o André, a Mariana, a Juliana, o Gui, o Carlos, o Paulo, a Lúcia e a Ana. Em uma situação ainda pior, procuraríamos alguém que não está no array, porém, foi preciso passar por todos os elementos para termos certeza de que ele não fazia parte.

Levando em consideração que precisamos passar por todos os elementos, qual será o número de operações que teremos de fazer? Se temos **cinco** elementos, faremos **cinco** operações. Se temos **1000** elementos, faremos **1000** operações. Basicamente, precisamos fazer uma operação para cada elemento. Isto significa que, se temos n elementos, teremos de fazer n operações.

O algoritmo que busca todos os elementos cresce de acordo com uma linha ascendente que indica: "Se tenho um elemento, farei uma operação. Se tenho dois, farei duas operações. Se tenho cinco, farei cinco". Ele é um algoritmo linear e segue uma linha.

Mesmo que trabalhássemos com $2n$ ou $3n$, não faria diferença. O gráfico seria um pouco diferente, mas não de maneira significativa. Nós veremos isto em seguida, quando o compararmos com outros tipos de buscas que nós conhecemos.

Nós chamamos a busca tradicional de **busca linear**, porque ela vai passar por todos os elementos, tentando encontrar o elemento procurado. Ela é linear e passará pelos n elementos de um array de tamanho n .

17.1 O DESEMPENHO DE BUSCA BINÁRIA

Analisaremos a **busca binária**. Em vez de fazermos um busca tradicional, com um `for` que varre da esquerda para a direita, nós optamos por arriscar um elemento do meio.

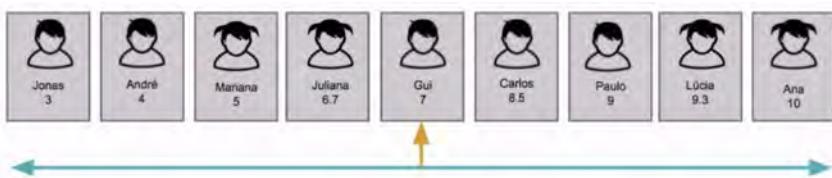


Figura 17.5: O desempenho da busca binária — passo 1

Depois, qual será o próximo passo? Verificaremos se o elemento está para esquerda ou para direita? Por exemplo, se vamos procurar a nota 9.3, ela está posicionada à direita. Então, descartaremos todos os elementos do meio para a esquerda, e analisaremos apenas o pedaço que sobrou à direita.

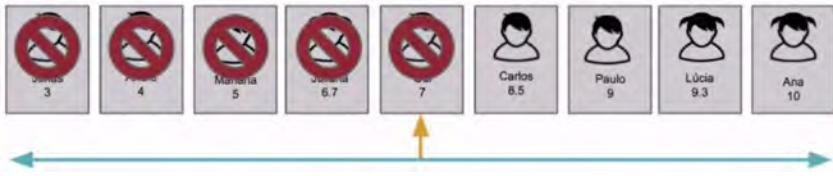


Figura 17.6: O desempenho da busca binária — passo 2

Agora vamos observar o pedaço e dividiremos no meio novamente.



Figura 17.7: O desempenho da busca binária — passo 3

Ao compararmos o elemento, faremos a pergunta: ele é o que procuramos? Não. A nota está para esquerda ou para direita? Está na direita. O faremos? Descartaremos os que estão na esquerda.



Figura 17.8: O desempenho da busca binária — passo 4

Dividiremos os elementos que sobraram no meio.



Figura 17.9: O desempenho da busca binária — passo 5

E então, teremos encontrado.



Figura 17.10: O desempenho da busca binária — passo 6

Como funciona a busca binária? Nós dividimos o array no meio, verificamos as duas partes e ficamos com apenas uma delas. Seguimos repetindo o processo de dividir e selecionar uma das partes.

A cada nova operação, o número de elementos que descartamos não será **um** como na busca antiga, em que descartávamos um elemento por vez. Em cada operação, nós descartaremos metade do array e, ao fazermos outra operação, eliminaremos a metade dos que sobraram.

Quando fazemos uma operação, ficamos com $1/2$, depois com $1/4$, em seguida com $1/8$, e seguimos descartando sempre. Assim o número de elementos diminuirá mais rapidamente. Porém, o quanto mais rápido?

Vamos ver o que acontece quando buscamos em um array com a busca binária. Se temos um único elemento, não precisaremos dividi-lo.

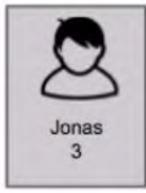


Figura 17.11: O desempenho da busca binária — passo 7

Fazemos a comparação e acabou a busca. Praticamente não faremos nada. No entanto, se tivermos dois elementos, precisaremos fazer uma operação de divisão.

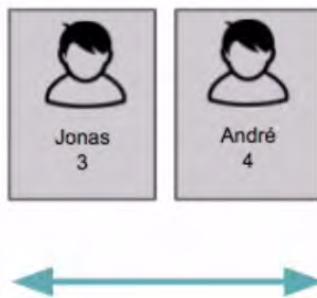


Figura 17.12: O desempenho da busca binária — passo 8

Vamos analisar o item do meio, que será o da posição 0, veremos se ele é maior ou menor e se continuaremos para direita ou para a esquerda. Neste caso, faremos **uma** operação de divisão. Não temos como evitá-la, quando trabalhamos com dois elementos.

E se tivermos três ou quatro elementos, o que faremos? Primeiro, uma operação vai dividir o array no meio.



Figura 17.13: O desempenho da busca binária — passo 9

Então, analisaremos em qual metade está o elemento. Ficaremos com uma delas, e em seguida dividiremos novamente no meio. Ou seja, precisaremos de **duas** operações para ficar com um elemento. Na primeira operação, ficaremos com dois elementos, com a segunda, restará apenas um.

E se tivermos mais elementos, como nove, dez ou até 16 elementos? Como faremos? Com uma operação, quebraremos o número na metade e ficarão no máximo oito. Com duas operações, sobrarão quatro. Com três operações, sobrarão dois. E no máximo, com quatro operações, sobrará um.

Está surgindo um padrão no processo. A cada novo passo, nós dividimos o número de elemento por 2. Isto significa que a cada operação de divisão, analisávamos 2^1 (elementos), ou seja, dois elevado a primeira é 2, dois elementos.

Com duas operações, nós somos capazes de analisar um número de elementos igual a 2^2 , ou seja, quatro elementos. Com três operações, somos capazes de analisar um número de elementos igual a 2^3 , ou seja, oito elementos. Com quatro operações, somos capazes de analisar um número de elementos igual a 2 elevado 4, ou seja, dezesseis elementos.

Com dez operações, serão 2 elevado a 10, ou seja, 1024 elementos. Com dez operações, conseguiremos comparar 1024

elementos. É um número bem elevado de itens em relação ao número de operações. Então, se tivermos o valor 1024, como conseguiremos descobrir que o número será elevado à potência dez? Como conseguiremos descobrir que 2 elevado ao número de operações será igual a 1024? Ele é o **log** na **base 2**, que significa: qual número que exponenciado resultará no valor total.

O número 2 elevado a qual potência será igual a 1024? O número 2 elevado a qual potência será igual a 32? Ou o número 2 elevado a qual potência será igual a 8? Se temos 2048 elementos, 2 elevado a qual potência resultará neste valor? A resposta será onze. Então, precisaremos de **onze** operações.

O número de operações crescerá de acordo com o log do número de elementos. Quando faço uma busca binária, ele é O de $\log n$ na *base 2* — em geral, usamos *base 2* na computação.

17.2 ANALISANDO A BUSCA BINÁRIA

Agora que sabemos que uma busca tradicional é **linear**, se temos n elementos, ela executará n operações ou algo na grandeza de n , como $O(n)$, $3n$, $3n + 15$, $5n - 17$. Ela sempre estará em função de n solto (e não exponenciado ao quadrado ou ao cubo). A busca **binária** é igual a $\log n$ na *base 2*, o que significa que ela crescerá de acordo com o $\log n$.

Se a busca cresce de maneira logarítmica, como poderemos compará-la com a linear? Criaremos um gráfico para compararmos o algoritmo **linear (n^*) e o quadrático* (n^2*)**.

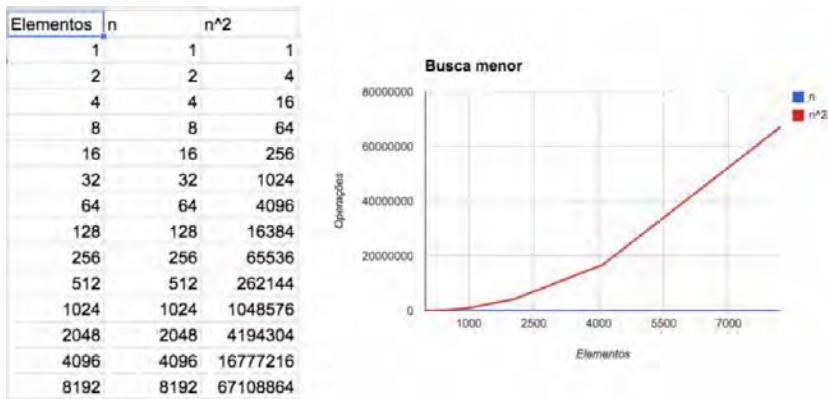


Figura 17.14: Analisando a busca binária — passo 1

Observe como a linha representando n^2 do **quadrático** cresce no gráfico, enquanto a linha do **linear** permanece próxima ao eixo inferior, porque executa o número de operações exatamente igual a n .

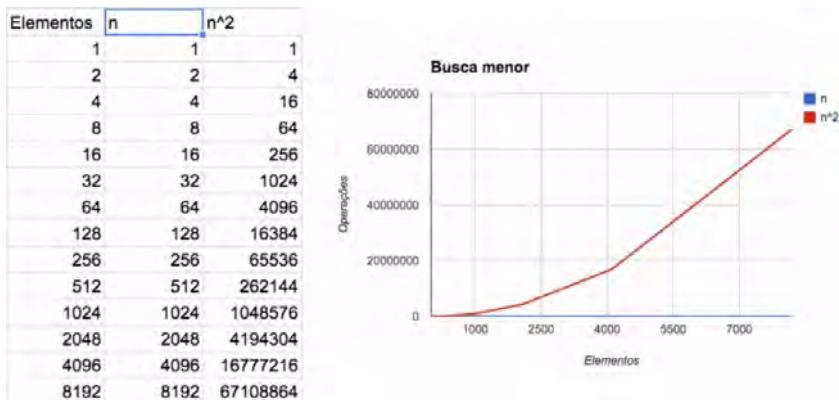


Figura 17.15: Analisando a busca binária — passo 2

A busca binária é logarítmica e buscará $\log n$ na base 2 — escreveremos apenas $\log n$.

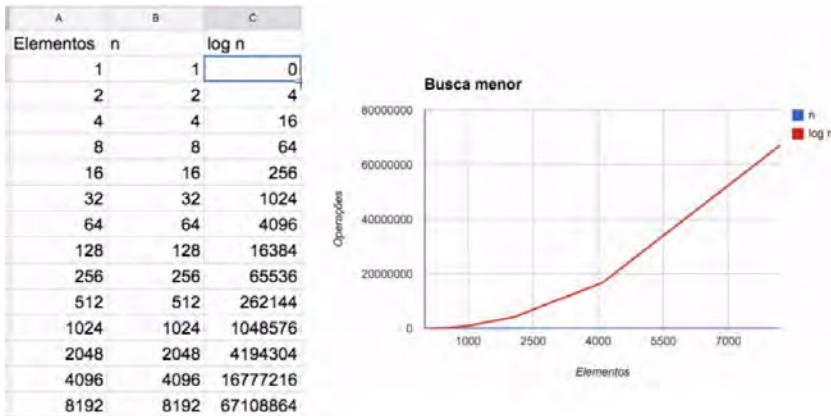


Figura 17.16: Analisando a busca binária — passo 3

Quando temos 1 elemento, o resultado será igual a 0, ou seja, ele fará **zero** divisões. Faremos uma comparação simples, porém, ignoraremos o +1, -1, ou +15. O importante é a maneira como o gráfico crescerá.

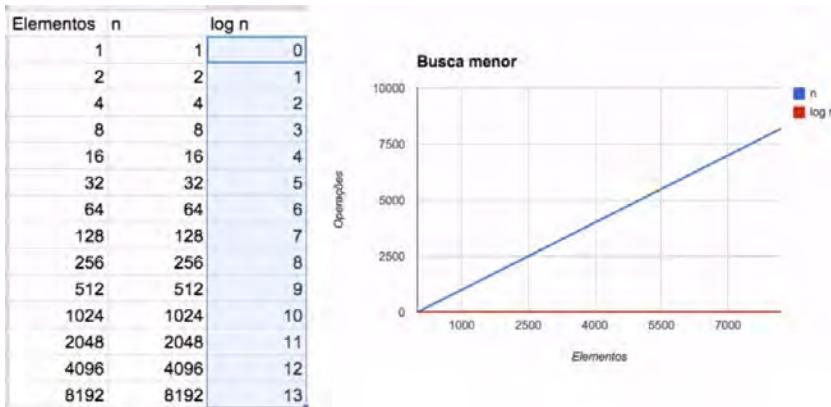


Figura 17.17: Analisando a busca binária — passo 4

Quando tivermos um elemento, precisaremos dividir zero vezes. Se forem dois elementos, faremos **uma** divisão. Se forem quatro elementos, faremos **duas** divisões. Com oito elementos, faremos **três** divisões. O número de operações estará relacionado com a

potência em que o número 2 será exponenciado. Por exemplo, 2 elevado à quarta potência será igual a 16.

Elementos	n	$\log n$
1	1	0
2	2	1
4	4	2
8	8	3
16	16	4

Figura 17.18: Analisando a busca binária — passo 5

Com 32 elementos, faremos **cinco** comparações. Com 64, faremos **seis** comparações. Com 128, faremos **sete** comparações. O número de divisões que faremos será os valores que estão na coluna $\log n$.

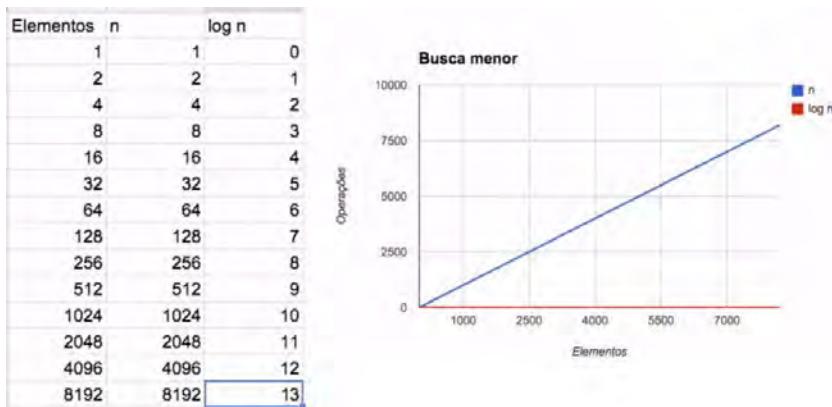


Figura 17.19: Analisando a busca binária — passo 6

Vamos comparar com os resultados da outra busca (coluna n). Se temos um array com 8192, em uma busca linear tradicional, faremos 8192 comparações. Em uma busca logarítmica (binária), nós faremos quantas comparações? No máximo, treze operações.

Qual é a sacada? Nós dividimos o array exatamente no meio e analisamos. Depois repetimos o processo enquanto for preciso; em vez de fazermos 8192 operações, faremos apenas 13. É por isso que, por exemplo, quando verificamos uma lista de presença, nós direcionamos nosso olhar para uma posição aproximada, onde esperamos que o elemento esteja localizado.

Se olhássemos um item por vez, teríamos de fazer 8192 comparações. Em uma sala de aula com esta quantidade de alunos, seria preciso fazer um número muito elevado de comparações.

As pessoas em geral, naturalmente, fazem uma busca mais inteligente quando trabalham com um array ordenado. Nós começamos a procura a partir de uma posição aproximadamente correta. Fazemos este processo intuitivamente em vez de varrermos o array inteiro. Por isso, a busca binária é extremamente mais rápida do que a sequencial normal, que passará por todos os elementos. Mas para isto, precisamos que o array já esteja ordenado.

17.3 O PRÓXIMO DESAFIO: COMPARAR AS ORDENAÇÕES

Agora que já entendemos que o processo de dividir um array em dois e analisá-lo por partes pode gerar um algoritmo logarítmico, vamos analisar os algoritmos de ordenação implementados que fazem um trabalho similar.

ANÁLISE ASSINTÓTICA DAS ORDENAÇÕES

E os outros algoritmos de ordenação que nós já vimos? O quanto rápido eles são e o que fazem exatamente? É o que vamos descobrir.

O primeiro algoritmo que conhecemos tentava intercalar os elementos que tínhamos. O que fazia a função `intercala`? Vamos analisar o código:

```
private static void intercala(Nota[] notas, int inicial, int miolo  
, int termino) {  
    Nota[] resultado = new Nota(termino - inicial);  
  
    int atual = 0;  
    int atual1 = inicial;  
    int atual2 = miolo;  
    while(atual1 < miolo &&  
        atual2 < termino) {  
        Nota nota1 = notas[atual1];  
        Nota nota2 = notas[atual2];  
        if(nota1.getValor() < nota2.getValor()) {  
            resultado[atual] = nota1  
            atual++;  
        } else {  
            resultado[atual] = nota2;  
            atual2++;  
        }  
        atual++;  
    }  
}
```

Se o array tivesse n elementos, o `intercala` passava por cada

um dos itens. Isso significa que ele fará n operações. Talvez, ele não faça todas as operações no trecho do código anterior, mas ele terminará no `while` que está logo abaixo desta parte.

```
while(atual1 < miolo &&
      atual2 < termino) {
    Nota nota1 = notas[atual1];
    Nota nota2 = notas[atual2];
    if(nota1.getValor() < nota2.getValor()) {
        resultado[atual] = nota1
        atual++;
    } else {
        resultado[atual] = nota2;
        atual2++;
    }
    atual++;
}
while(atual1 < miolo) {
    resultado[atual] = notas[atual1];
    atual1++;
    atual++;
}
while(atual2 < termino) {
    resultado[atual] = notas[atual2];
    atual2++;
    atual++;
}
```

Para intercalarmos dois trechos de um array, ele precisará fazer n operações. Depois, com o `for`, faremos mais n operações novamente.

```
for(int contador = 0; contador < atual, contador++) {
    notas[inicial + contador] = resultado[contador];
}
```

Porém, $2n$ não fará diferença na nossa análise. Da maneira como estamos analisando, $2n$, $2n + 17$, $2n - 35$, todos se comportam igualmente a n . Por quê? Porque todos crescerão de forma linear no gráfico. Eles não crescerão exponencialmente ou quadraticamente. Ele crescerá linearmente. Para nós, é um fator interessante, considerando que buscamos algoritmos mais rápidos

do que o linear.

Então, com o `intercala`, teremos de fazer n operações. O algoritmo que intercala dois trechos de um array é linear.

18.1 O DESEMPENHO DE MERGE SORT

Sabemos que, para intercalar o trecho de um array, executaremos uma operação linear. Isso significa que precisaremos realizar n operações. A ordenação que fizemos anteriormente usava o `intercala` diversas vezes, ou seja, executava várias vezes a quantidade n de operações. Se n for um número fixo como três ou quatro vezes, tudo bem. O problema é que não trabalharemos com números fixos. Como funcionará o algoritmo de ordenar neste caso?

```
private static void ordena(Nota[] notas, int inicial, int termino)
{
    int quantidade = termino - inicial;
    if(quantidade > 1) {
        int meio = (inicial + termino) / 2;
        System.out.println(inicial + " " + termino + " " + meio);

        ordena(notas, inicial, meio);
        ordena(notas, meio, termino);
        intercala(notas, inicial, meio, termino);

    }
}
```

Ele separava metade do array, ordenava um trecho e, depois, intercalava todos os elementos. Ou seja, ele dividia por 2, então dividia por 2 novamente e seguia repetindo a divisão várias vezes. Em cada uma delas, ele intercalava. Quantas vezes ele executará a divisão por 2, até chegar a quantidade de um elemento?

Da mesma forma que na busca binária nós repetimos diversas vezes a divisão por 2, nós queremos descobrir qual a potência de 2 que resultará no número total de elementos. Nós já conhecemos este

algoritmo! Assim como na busca binária, nós repetiremos o processo de divisão até restar um elemento. Quantas vezes precisaremos repetir a operação? A resposta é: **log** do número na **base 2**.

Se usarmos o `ordena`, ele executará o `intercala` diversas vezes, o que será a quantidade de n operações. Então, nosso algoritmo será n multiplicado pelo número de vezes que a operação será executada ($\log n$). Ou seja, ele será $n \log n$. Em seguida, vamos compará-lo com os outros algoritmos de ordenação que conhecemos.

Como funciona o algoritmo $n \log n$? Ele quebra o array e intercala as partes menores. Assim, temos um algoritmo que executa $n \log n$.

18.2 COMPARANDO O MERGE SORT COM OUTROS SORTS

Veremos a comparação do algoritmo do tipo *Selection sort* e *Insertion sort* que são quadráticos, ou seja, são **n^2** (n multiplicado por ele mesmo).

Elementos	n^2	$\log n$
1	1	0
2	4	1
4	16	2
8	64	3
16	256	4
32	1024	5
64	4096	6
128	16384	7
256	65536	8
512	262144	9
1024	1048576	10
2048	4194304	11
4096	16777216	12
8192	67108864	13

Figura 18.1: Comparando o mergesort com outros sorts — passo 1

Nós vamos comparar o número de operações de um **quadrático** com um algoritmo que é $n \times \log n$.

Elementos	n^2	$n \times \log n$
1	1	0
2	4	2
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384
128	16384	896
256	65536	2048
512	262144	4608
1024	1048576	10240
2048	4194304	22528
4096	16777216	49152
8192	67108864	106496

Figura 18.2: Comparando o mergesort com outros sorts — passo 2

O *sort* novo divide e faz merge (intercala). Se observarmos a tabela com 64 elementos, a ordenação **quadrática** fará 4.096 operações, enquanto a ordenação $n \times \log n$ fará 384.

Elementos	n^2	$n \times \log n$
1	1	0
2	4	2
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384

Figura 18.3: Comparando o mergesort com outros sorts — passo 3

Talvez, uma operação dez vezes mais rápida não pareça suficiente boa. Mas se compararmos os algoritmos utilizados em um array com 8.192 de elementos, o **quadrático** fará 67.108.864 operações para ordenar o *Selection Sort* e o *Insertion Sort*. Não parece um desempenho bom.

A	B	C
Elementos	n^2	$n * \log n$
1	1	0
2	4	2
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384
128	16384	896
256	65536	2048
512	262144	4608
1024	1048576	10240
2048	4194304	22528
4096	16777216	49152
8192	67108864	106496

Figura 18.4: Comparando o mergesort com outros sorts — passo 4

Se tivéssemos um baralho com 8.192 cartas e precisássemos ordená-las, o que faríamos? Dividiríamos o monte com outras pessoas, porque o número de operações que faríamos seria muito menor. Nós dividiríamos e intercalaríamos. Por exemplo, para a mesma quantidade de elementos, com o algoritmo novo, faríamos 106.496 operação. Uma diferença grande na quantidade de operações.

Elementos	n^2	$n * \log n$
1	1	0
2	4	2
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384
128	16384	896
256	65536	2048
512	262144	4608
1024	1048576	10240
2048	4194304	22528
4096	16777216	49152
8192	67108864	106496

Figura 18.5: Comparando o mergesort com outros sorts — passo 5

Podemos ver no gráfico a diferença de crescimento dos algoritmos.



Figura 18.6: Comparando o mergesort com outros sorts — passo 6

Desejo boa sorte para quem quiser ordenar um array gigante com um algoritmo quadrático.

18.3 ANALISANDO O MERGE SORT

Nós vimos que o algoritmo novo de ordenação é n^2 , que primeiro divide e depois intercala. Como ele é baseado em **intercalar** os elementos, o nome do algoritmo será relacionado com fundir dois trechos intercalando: **Merge Sort**.

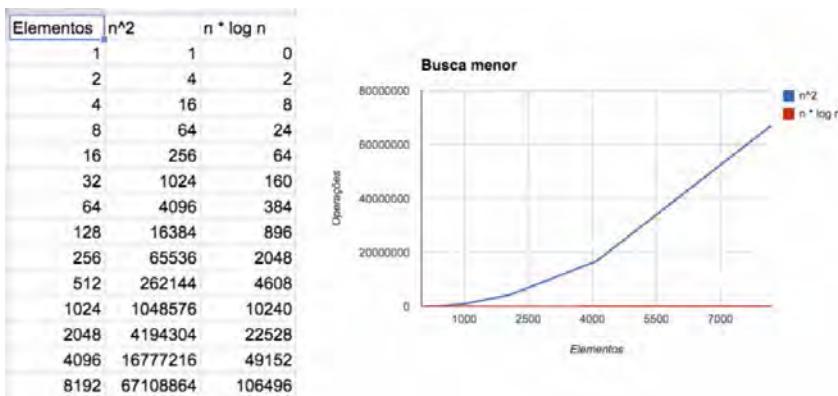


Figura 18.7: Analisando o mergesort — passo 1

Trabalhar com *Merge Sort* é mais rápido do que com o *Selection Sort* ou *Insertion Sort*. Quando trabalhamos com uma quantidade pequena de elementos, não fará muita diferença para o computador.

Elementos	n^2	$n * \log n$
1	1	0
2	4	2
4	16	8
8	64	24

Figura 18.8: Analisando o mergesort — passo 2

De 64 para 24 operações, a diferença é irrelevante. Mesmo que um algoritmo seja dez vezes mais lento para 64 elementos, a diferença também é irrelevante.

Elementos	n^2	$n * \log n$
1	1	0
2	4	2
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384

Figura 18.9: Analisando o mergesort — passo 3

No cotidiano, se você tiver de fazer 4 mil ou 400 operações, o computador executará em um piscar de olhos. Porém, se aumentarmos significantemente o número de elementos, você terá dificuldades em conseguir executar todas.

Se aumentar o número de usuários acessando a mesma máquina simultaneamente, compartilhando o mesmo processador, não vamos conseguir. Poderíamos usar uma ordenação quadrática para um número baixo de elementos. No entanto, com um número de elementos elevado, a diferença será grotesca.

Elementos	n^2	$n * \log n$
1	1	0
2	4	2
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384

Figura 18.10: Analisando o mergesort — passo 4

Se aumentarmos o número de pessoas querendo acessar simultaneamente o processador, vamos sobrecarregá-lo. A ordem de grandeza de um algoritmo vai influenciar bastante na rapidez. Quanto maior for o número de elementos, maior será a distância das linhas no gráfico.

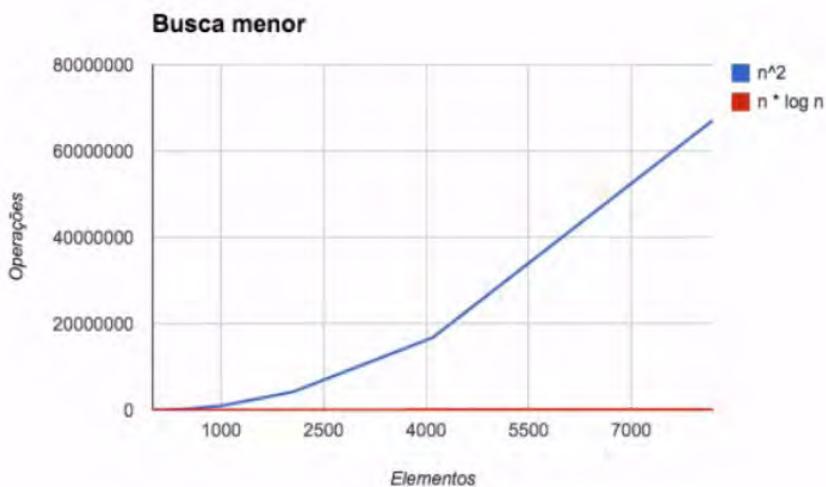


Figura 18.11: Analisando o mergesort — passo 5

A análise que fazemos considera o número de *sorts* no *Selection Sort* e no *Merge Sort*, verificando o crescimento no longo prazo. A linha n^2 cresce quadraticamente e ficará inviável. A linha do *Merge Sort* crescerá de acordo com $n * \log n$, bem mais suave, tanto que ela aparece "encostada" ao eixo inferior.

18.4 ANALISANDO O PARTICIONA

Da mesma maneira que analisamos o algoritmo do Merge Sort, vamos analisar também o outro algoritmo de ordenação que implementamos. Analisaremos sem nos focarmos no "melhor" ou o "pior" caso, mas como o algoritmo cresce em geral.

Nós fizemos uma função da classe `TestaPivota`, que usava a função `particiona`. Como ela funcionava?

```
private static int particiona(Nota[] notas, int inicial, int termino) {  
    int menoresEncontrados = 0;  
  
    Nota pivo = notas[termino - 1];  
    for(int analisando = 0; analisando < termino - 1; analisando++) {  
        Nota atual = notas[analisando];  
        if(atual.getValor() <= pivo.getValor()) {  
            troca(notas, analisando, menoresEncontrados);  
            menoresEncontrados++;  
        }  
    }  
    troca(notas, termino - 1, menoresEncontrados);  
    return menoresEncontrados;  
}
```

Primeiro executa-se um `for` que passava por todos os elementos entre o `inicio` e o `termino`.

```
for(int analisando = 0; analisando < termino - 1; analisando++) {  
    Nota atual = notas[analisando];  
    if(atual.getValor() <= pivo.getValor()) {  
        troca(notas, analisando, menoresEncontrados);  
        menoresEncontrados++;  
    }  
}
```

Isto significa que o nosso `for` realiza quantas operações? A resposta é **n**. Poderia ser também $2n$, $5n$, $5n - 3$, porém, nós estamos interessados na grandeza, na maneira como o algoritmo cresce. Neste caso, ele vai crescer de forma **linear**.

Para nós particionarmos o array com o pivô, a quantidade de operações realizadas crescerá de acordo com o número de elementos analisados. Se temos dez elementos, teremos de passar por cada um deles para encontrar a posição correta do pivô. O mesmo acontecerá se tivermos vinte elementos: igualmente precisaremos passar por todos os itens, antes de descobrirmos a

posição do pivô.

Isto acontece porque teremos de identificar todos os elementos menores do que o pivô. Então, passamos pelo array inteiro, do `início` ao `término` do trecho analisado. Logo, o `particiona` crescerá de acordo com o número de elementos. Ele é linear.

18.5 DESEMPENHO DO QUICK SORT

Depois de entender como cresce o `particiona`, veremos a ordenação baseada nesta função: o `TestaOrdenacaoRapida`. O `TestaOrdenacaoRapida` chama o método `ordena`.

```
private static void ordena(Nota[] notas, int de, int ate) {  
    int elementos = ate - de;  
    if(elementos > 1) {  
        int posicaoDoPivo = particiona(notas, de, ate);  
        ordena(notas, de, posicaoDoPivo);  
        ordena(notas, posicaoDoPivo + 1, ate);  
    }  
}
```

O que ele fazia? Ele primeiro particionava o array inteiro, ou seja, é linear e realizava n operações. Logo, ele chamava o próprio algoritmo para uma das metades, depois para outra. Cada vez que ele chamava o algoritmo, ele também executava o `particiona`. Ou seja, é o linear (n) multiplicado pelo número de vezes que chamamos o `ordena`.

E quantas vezes chamamos o `ordena`? Como calculamos o número de repetições do processo em que dividimos o array em dois trechos e selecionamos apenas uma parte?

Nós já vimos um cálculo parecido anteriormente duas vezes: uma na busca binária, uma no processo do Merge Sort. O que podemos imaginar é que o número de operações da ordenação vai crescer de acordo com que realizarmos n operações — devido ao `particiona` — multiplicado pela quantidade de vezes que chamamos o `ordena` ($\log n$).

Este algoritmo crescerá como um *Merge Sort*, também será $n \times \log(n)$, ou seja, $\Theta(n \log(n))$.

18.6 COMPARANDO O QUICK SORT COM O MERGE SORT

Vamos comparar o nosso *Merge Sort* com a implementação de ordenação baseada no `particiona`, em posicionamos o pivô e particionamos. Um deles é o $n \times \log n$ (o *Merge Sort*).

Elementos	n^2	$n * \log n$
1	1	0
2	4	2
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384
128	16384	896
256	65536	2048
512	262144	4608
1024	1048576	10240
2048	4194304	22528
4096	16777216	49152
8192	67108864	106496

Figura 18.12: Comparando o sort rápido com o Merge Sort — passo 1

O outro algoritmo será $n \times \log n$, que será o nosso *sort* novo.

Elementos	$n * \log n$	$n * \log n$
1	0	0
2	2	2
4	8	8
8	24	24
16	64	64
32	160	160
64	384	384
128	896	896
256	2048	2048
512	4608	4608
1024	10240	10240
2048	22528	22528
4096	49152	49152
8192	106496	106496

Figura 18.13: Comparando o sort rápido com o Merge Sort — passo 2

Então, um algoritmo crescerá $n * \log n$ e o outro crescerá $n * \log n$.

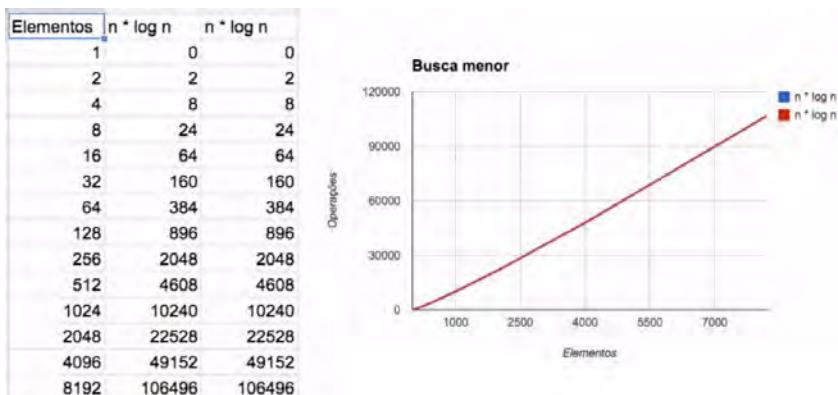


Figura 18.14: Comparando o sort rápido com o Merge Sort — passo 3

Vamos comparar os dois algoritmos no gráfico. As linhas de

ambos estão emparelhadas. Os dois crescem da mesma maneira! Como poderemos comparar o *Merge Sort* e o novo algoritmo, se eles crescem igualmente?

18.7 QUICKSORT

Temos dois algoritmos de ordenação novos: o *Merge Sort* e o novo *Sort*. Os dois crescem de maneira $n \log(n)$. Queremos descobrir qual é o melhor para ser usado.

Na prática, costumamos comparar quantas operações os algoritmos fazem em média. Porém, como eles crescem igualmente, precisaremos analisar outros detalhes, por exemplo, quantas trocas eles fazem? Então, ele é $2(n \log(n))$ ou $3(n \log(n))$? Ele é $2(n \log(n) + 5)$ ou $2(n \log(n) + 17)$?

Em média, trabalhando com dados reais, o que acontecerá com o *Merge Sort* e o novo *sort*? Quais dos dois terá um melhor comportamento? Este é o tipo de análise feita nesta situação pelos cientistas da computação.

A resposta dos especialistas é que, em média, o novo algoritmo é mais rápido do que o algoritmo de intercalação, o *Merge Sort*. Como vimos antes, o novo algoritmo mais rápido de ordenação nós chamamos de **Quicksort**. Ele usa o pivô para particionar e ordenar. Na prática, ele executa um número menor de operações do que o *Merge Sort* — ainda que ambos cresçam de forma parecida.

CAPÍTULO 19

CONCLUSÃO

Conhecemos diversos algoritmos em que trabalhávamos com uma quantidade de elementos dentro do array, depois o dividíamos em partes menores, para atacá-las separadamente.

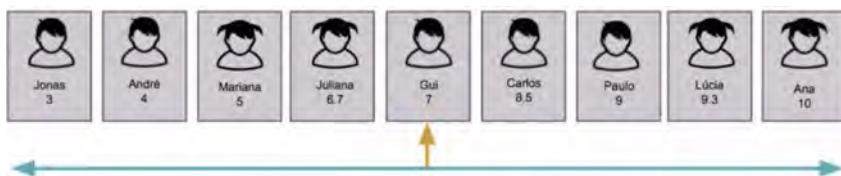


Figura 19.1: Array dividido em duas partes

Tanto no **Merge Sort** quanto no **Quicksort** usamos uma técnica de quebrar o problema em dois, para então resolvê-los.

No **busca binária**, o que fazíamos? Chamávamos o algoritmo para uma parte, depois para a outra. Opa, a base do que estamos fazendo nos três casos é: nós dividimos um problema, e depois juntamos as partes, para assim termos um resultado final.

No fim do *Quicksort*, como resultado nós tínhamos o array ordenado. No *Merge Sort*, nós intercalávamos os dois trechos separados, resultando no array ordenado.

No *busca binária*, separávamos o resultado do lado em que continuávamos buscando. Ou seja, nós sempre dividimos o nosso problema e, em seguida, juntávamos o resultado das divisões. Logo, chegávamos a um resultado final. Este tipo de solução ou técnica é

chamada de **divisão e conquista**.

Se não conseguimos conquistar diretamente o problema, porque tornará a conquista demorada (como o *Selection Sort* e o *Insertion Sort*), então o que fazemos é dividir o problema em partes. A estratégia pode ser melhor como foi nos três casos, em que dividimos e conquistamos. Com isto, nós criamos algoritmos que são **logarítmicos** — $n \log(n)$ ou $\log n$ — que se saíram melhor do que os anteriores.

No mundo real, sempre vamos questionar qual algoritmo vale a pena implementar, quando ele ainda não foi implementado. Mas a base do conteúdo apresentado aqui é compreender que existem diversos algoritmos para resolver os mesmos tipos de problemas. E em cada situação, cada um será o mais apropriado para ser usado.

Alguns funcionarão melhor em determinados cenários e outros funcionarão em qualquer um. Uns podem ser mais lentos, outros mais rápidos; alguns consumirão mais ou menos memória. Mas todos estes algoritmos são usados para resolver problemas computacionais, como por exemplo:

- Quero ordenar;
- Quero saber quem é o menor e o maior;
- Quero saber se um elemento está dentro da lista;
- Quero saber em qual posição um determinado elemento ficou.

Problemas como estes podem ser resolvidos usando diversos tipos de algoritmos. Aqui nós analisamos e entendemos para que servem os algoritmos e como podemos compará-los. A partir de agora, quando conhecermos novos algoritmos, sempre poderemos observar se eles são lentos ou rápidos, se são apropriados para uma situação, ou se são complicados de serem aplicados em outra.

19.1 COMO CONTINUAR OS ESTUDOS

Assim que se sentir confortável com a implementação dos algoritmos apresentados neste livro, você estará mais preparado a resolver problemas de estrutura lógica que aparecem no dia a dia de um programador. Claro que não é nossa rotina implementar uma busca binária, mas o conceito de divisão e conquista, assim como os outros apresentados aqui, servem de base para permitir que você implemente soluções para diversos algoritmos presentes no mundo real, que as empresas pedem para serem transferidos para um computador.

Para quem deseja entender a matemática por trás desses algoritmos e não tem medo dela, o livro *Introduction to Algorithms*, de Charles Eric Leiserson, Ronald Rivest e Thomas H. Cormen, é um ótimo ponto de partida.

Se você se interessou nos algoritmos e quer entender como funcionam outros similares e mais avançados, recomendo meus cursos de Machine Learning, feito em Python. Como o curso está na Alura, para quem se interessar, minha sugestão é a assinatura. Ela dá direito aos dois cursos, além dos cursos de Python:

<http://bit.ly/analise-de-dados>

Se você quer entender como funcionam estruturas de dados em Java, também recomendo o curso da Alura que aborda o mesmo conteúdo:

<http://bit.ly/estrutura-de-dados>

Caso se aventure em novos algoritmos e tenha dificuldades, a comunidade do <http://www.guj.com.br> é o melhor lugar para conversar sobre as dificuldades de implementação que esteja encontrando.

Caso tenha dificuldades ou erros nos algoritmos relacionados ao livro, temos o fórum da Casa do Código: <http://forum.casadocodigo.com.br>.

Por fim, para praticar, temos o site do <http://www.spoj.com>. Nele há uma infinidade de problemas a serem resolvidos, desde os mais clássicos (<http://www.spoj.com/problems/classical/>) até outros extremamente complexos, que aparecem nas finais mundiais de computação.

Para você que venceu esse desafio, desejo que continue seus estudos e práticas com novos algoritmos. Assim, você exerce seu pensamento lógico e tem cada vez mais facilidade na hora de encontrar e implementar a solução de problemas que encontramos no nosso cotidiano.

Bons estudos!

CÓDIGO FINAL

Você pode encontrar o código final de todo o conteúdo apresentado até aqui no GitHub em <http://bit.ly/codigo-algoritmos2>. Lá você pode baixá-lo e rodar a vontade. Apesar de que minha sugestão é que você crie o seu código do zero a partir do que vimos neste livro, para que você sinta as dificuldades e possa superá-las. Vale lembrar que a prática é fundamental para o processo do aprendizado.