

ALGORITMOS E PROGRAMAÇÃO II

Dados Internacionais de Catalogação na Publicação (CIP)
(Jeane Passos de Souza - CRB 8ª/6189)

Masiero, Andrey Araujo

Algoritmos e programação II / Andrey Araujo Masiero. – São Paulo :
Editora Senac São Paulo, 2020. (Série Universitária)

Bibliografia.

e-ISBN 978-65-5536-123-0 (ePub/2020)

e-ISBN 978-65-5536-124-7 (PDF/2020)

1. Desenvolvimento de sistemas 2. Linguagem de programação
3. Algoritmos : Programação 4. Programação recursiva I. Título.
II. Série

20-1131t

CDD – 005.13

003

BISAC COM051300

COM051230

Índice para catálogo sistemático

1. Linguagem de programação : Algoritmos 005.13

2. Desenvolvimento de sistemas 003

ALGORITMOS E PROGRAMAÇÃO II

Andrey Araujo Masiero





Administração Regional do Senac no Estado de São Paulo

Presidente do Conselho Regional

Abram Szajman

Diretor do Departamento Regional

Luiz Francisco de A. Salgado

Superintendente Universitário e de Desenvolvimento

Luiz Carlos Dourado

Editora Senac São Paulo

Conselho Editorial

Luiz Francisco de A. Salgado

Luiz Carlos Dourado

Darcio Sayad Maia

Lucila Mara Sbrana Sciotti

Jeane Passos de Souza

Gerente/Publisher

Jeane Passos de Souza (jpassos@sp.senac.br)

Coordenação Editorial/Prospecção

Luís Américo Tousi Botelho (luis.tbotelho@sp.senac.br)

Márcia Cavalheiro Rodrigues de Almeida (mcavalhe@sp.senac.br)

Administrativo

João Almeida Santos (joao.santos@sp.senac.br)

Comercial

Marcos Telmo da Costa (mtcosta@sp.senac.br)

Acompanhamento Pedagógico

Ariádney Carolina Brasileiro

Designer Educacional

Sueli Brianezi Carvalho

Revisão Técnica

Gustavo Moreira Calixto

Coordenação de Preparação e Revisão de Texto

Luiza Elena Luchini

Preparação de Texto

Ana Luiza Candido

Revisão de Texto

Ana Luiza Candido

Projeto Gráfico

Alexandre Lemes da Silva

Emília Corrêa Abreu

Capa

Antonio Carlos De Angelis

Editoração Eletrônica

Michel Iuiti Navarro Moreno

Ilustrações

Michel Iuiti Navarro Moreno

Imagens

iStock Photos

E-pub

Ricardo Diana

Proibida a reprodução sem autorização expressa.

Todos os direitos desta edição reservados à

Editora Senac São Paulo

Rua 24 de Maio, 208 – 3º andar

Centro – CEP 01041-000 – São Paulo – SP

Caixa Postal 1120 – CEP 01032-970 – São Paulo – SP

Tel. (11) 2187-4450 – Fax (11) 2187-4486

E-mail: editora@sp.senac.br

Home page: <http://www.livrariasenac.com.br>

© Editora Senac São Paulo, 2020

Sumário

Capítulo 1 **Métodos de busca e** **classificação de dados em** **memória, 7**

- 1 Busca sequencial, 8
- 2 Busca binária: a mais rápida entre as buscas, 19
- 3 Exercícios de fixação, 22
- Considerações finais, 23
- Referências, 23

Capítulo 2 **Técnicas de programação** **recursiva, 25**

- 1 Definição da recursão, 26
- 2 Traga a matemática à vida, 28
- 3 Tipos de recursão, 36
- 4 Iteração ou recursão?
Eis a questão, 42
- 5 Exercícios de fixação, 43
- Considerações finais, 43
- Referências, 44

Capítulo 3 **Algoritmos de ordenação** **simples, 45**

- 1 Bubble sort, 47
- 2 Insertion sort, 53
- 3 Selection sort, 56
- 4 Exercícios de fixação, 60
- Considerações finais, 60
- Referências, 61

Capítulo 4 **Algoritmos de ordenação** **sofisticados, 63**

- 1 Merge sort, 66
- 2 Quicksort, 70
- 3 Exercícios de fixação, 75
- Considerações finais, 75
- Referências, 76

Capítulo 5 **Eficiência de algoritmos, 77**

- 1 Matemática como ferramenta para análise de algoritmos, 79
- 2 Comparativo das taxas de crescimento, 85
- 3 Identificando a função do algoritmo, 88
- 4 Notação Big-O, 91
- 5 Exercícios de fixação, 93
- Considerações finais, 94
- Referências, 95

Capítulo 6 **Filas, 97**

- 1 Implementando a fila com vetores, 100
- 2 Andando em círculo, 108
- 3 Ligando os elementos, 112
- 4 Exercícios de fixação, 116
- Considerações finais, 117
- Referências, 117

Capítulo 7 **Pilha, 119**

- 1 Implementando a pilha com o uso de vetores, 122
- 2 Utilizando uma lista ligada para empilhar objetos, 130
- 3 Exercícios de fixação, 134
- Considerações finais, 135
- Referências, 135

Capítulo 8 **Fila de prioridade e heap, 137**

- 1 Implementando a fila de prioridades com vetores, 140
- 2 Heap, 149
- 3 Exercícios de fixação, 159
- Considerações finais, 159
- Referências, 160

Sobre o autor, 163

Métodos de busca e classificação de dados em memória

Uma das operações que mais realizamos no dia a dia é a busca por informações. Desde o momento em que acordamos até a hora de dormir, é difícil passarmos o dia sem fazermos uma busca. Nomes na agenda, cereal no armário, comida na geladeira, lápis no estojo, entre muitas outras. Nas aplicações computacionais isso não é diferente: a busca é a tarefa mais realizada. Sempre é necessário encontrar algum dado na memória ou em qualquer dispositivo de armazenamento.

Para realizar um processo de busca, é necessário um conjunto de dados que geralmente é estruturado com um vetor (WIRTH, 1989). Um vetor de objetos v é definido pela equação a seguir:

$$v: \text{VETOR}[0 \dots n - 1] \text{ de } \text{OBJETOS}$$

Em que v é um vetor que possui n objetos, classificados com índices de 0 até $n - 1$.

Essa definição é dada considerando que esse é um conjunto de dados de tamanho fixo, para facilitar a compreensão de todo o processo. Todo objeto existente no vetor v é inserido junto com uma chave de busca c (WIRTH, 1989). Um objeto o pode conter quantos atributos forem necessários para definir o problema a ser tratado. Aconselha-se que sua chave de busca seja um valor numérico, inteiro e positivo. Assim, as comparações para uma busca são mais eficientes devido ao tipo de dado.

Graficamente, é natural definir o vetor como uma barra retangular dividida em pequenos quadrados, os quais representam os objetos armazenados no vetor. A figura 1 apresenta a ilustração do vetor descrito, no qual os objetos estão armazenados a partir da posição de chave 0 aumentando até o valor final de chave $n - 1$. É importante salientar que n é o número de objetos armazenados no vetor.

Figura 1 – Ilustração de um vetor com 10 posições

0(0)	0(1)	0(2)	0(3)	0($n - 2$)	0($n - 1$)
------	------	------	------	-----	-----	-----	-----	--------------	--------------

Existem dois principais métodos para realizar buscas em um vetor de dados. A busca sequencial e a busca binária são as duas formas mais utilizadas. Além delas, algumas pequenas variações desses algoritmos serão apresentadas ao longo do capítulo. As implementações aqui expostas utilizaram a linguagem de programação Java, a qual pode ser facilmente traduzida para outras linguagens como C, C++, JavaScript, Python, entre outras.

1 Busca sequencial

Entre os métodos de busca, a busca sequencial, ou linear, é a que tem a implementação mais intuitiva. Como o nome já descreve, a busca

é realizada em cada uma das posições existentes no conjunto de dados, estruturados com vetor, até que se encontre a posição desejada do valor procurado. As condições de parada para esse algoritmo são determinadas por dois critérios básicos (WIRTH, 1989):

- o elemento procurado é encontrado;
- o vetor é completamente analisado, ou seja, o elemento procurado não é encontrado.

Geralmente, dois tipos abstratos de dados (TAD) são empregados nesse tipo de algoritmo. Ambos os tipos podem ser denominados como uma tabela de dados organizada. O primeiro TAD é o vetor no qual o uso de memória de armazenamento é realizado de forma estática (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). Isso é o espaço utilizado por um vetor na memória, determinado antes da compilação do sistema para uso.

A segunda opção é utilizar o TAD lista ligada, que possui um tipo de armazenamento de informação dinâmico (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). Este funciona com uma estrutura de ligação de nós (objetos), sendo assim, o consumo de memória é mais efetivo, uma vez que os objetos são criados e conectados de acordo com a necessidade do sistema. Nos exemplos que serão apresentados ao longo deste capítulo, o TAD escolhido será o vetor, pois, por possuir uma sintaxe mais simples e prática do que a da lista ligada, é mais fácil compreendê-lo.

Vamos apresentar um vetor contendo cinco elementos armazenados. Em seguida, será realizada a busca por um elemento existente e por outro não existente no vetor. Cada posição do vetor possui um índice que é a chave de acesso às informações armazenadas. Veja a ilustração gráfica do vetor e suas chaves de acesso na figura 2: existem cinco quadrados para armazenar as informações do vetor, que no exemplo em questão são números. Embaixo de cada quadrado existente há um número chamado de índice, o qual determina o valor de acesso à informação.

Figura 2 – Representação gráfica de um vetor com cinco elementos

35	15	29	83	10
0	1	2	3	4

Para realizar o processo de percorrer um vetor, utiliza-se uma estrutura de laço de repetição. Esse tipo de estrutura é encontrado em todas as linguagens de programação, mesmo que com diferentes nomenclaturas e padrões de declarações. Em linguagens derivadas da linguagem C, é comum encontrar três tipos de laços de repetição. São eles: *for*, *while* e *do-while*. É possível obter o mesmo resultado com qualquer um desses laços de repetição. Por uma questão didática, ao longo deste capítulo serão utilizados apenas os laços de repetição *while* e *for*.

Para implementar a busca sequencial, primeiro é necessário criar o vetor em memória. O código será implementado na linguagem Java e utilizará como exemplo o mesmo vetor demonstrado na figura 2. O vetor é declarado a seguir:

```
int vetor[] = {35, 15, 29, 83, 10};
```

Agora, vamos declarar o valor a ser procurado no vetor:

```
int valorProcurado = 83;
```

O próximo passo é construir o método para a busca sequencial. O método recebe como parâmetro um vetor do mesmo tipo do declarado e o valor procurado, que, no exemplo, é um número inteiro. O método deve retornar um valor inteiro correspondente à posição que o valor procurado se encontra no vetor. Se o valor procurado não existir no vetor, o

valor retornado deve ser -1 , isso porque não existe um índice ou posição negativa em um vetor.

```
int buscaSequencial(int[] vetor, int valorProcurado) {  
    for (int i = 0; i < vetor.length; i++) {  
        if (vetor[i] == valorProcurado) {  
            return i;  
        }  
    }  
    return -1;  
}
```

A instrução *for* é utilizada por facilitar a implementação da busca pelo valor. Nela a variável *i* é declarada e inicializada com o valor 0, que representa a primeira posição do vetor. O critério de parada é determinado pela condição *i* menor que o tamanho do vetor. Enquanto essa condição for verdadeira, será executado o bloco de código determinado entre chaves. A cada iteração o valor de *i* é incrementado em 1, fazendo com que o vetor seja percorrido posição a posição.

Quando o valor armazenado na posição *i* do vetor for igual ao valor procurado, condição da instrução *if*, o valor atual de *i* é retornado pelo método *buscaSequencial*. Se em nenhum momento a condição for verdadeira, a busca do dado no vetor será finalizada e, com a finalização do *for*, o valor -1 é retornado para o programa. O passo a passo da procura pelo valor 83 é apresentado através da forma gráfica, iniciando pela figura 3, na qual são apresentados o vetor de busca, o valor procurado, o valor de *i* e também o resultado da comparação existente dentro da condição do *if* no algoritmo:

Figura 3 – Passo 1 no processo da busca sequencial

35	15	29	83	10
0	1	2	3	4

Comparação:
35 == 83 -- Falso

83	Valor Procurado
0	<i>i</i>

Figura 4 – Passo 2 no processo da busca sequencial

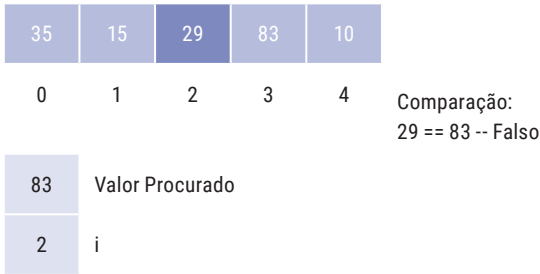
35	15	29	83	10
0	1	2	3	4

Comparação:
15 == 83 -- Falso

83	Valor Procurado
1	<i>i</i>

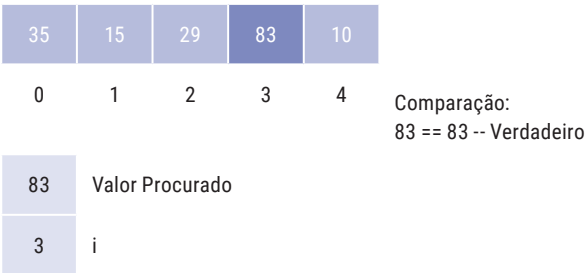
Agora, na figura 4, na qual são apresentados o vetor de busca, o valor procurado, o valor de *i* e também o resultado da comparação existente dentro da condição do *if* no algoritmo, a posição de número 1 é acionada para comparação (quadrado mais escuro). O espaço de memória *i* recebe o valor de 1 (posição atual do vetor no algoritmo). Neste passo, a comparação continua com o resultado falso, portanto uma nova iteração é necessária. A figura 5 apresenta o próximo passo do algoritmo, no qual são apresentados o vetor de busca, o valor procurado, o valor de *i* e também o resultado da comparação existente dentro da condição do *if* no algoritmo:

Figura 5 – Passo 3 no processo da busca sequencial



O valor armazenado nesta posição é 29, que continua diferente de 83, o valor procurado, portanto a condição é falsa. É necessário incrementar mais uma vez a variável *i* para seguir para a próxima posição do vetor. Acompanhe o passo 4 na figura 6, na qual são apresentados o vetor de busca, o valor procurado, o valor de *i* e também o resultado da comparação existente dentro da condição do *if* no algoritmo:

Figura 6 – Passo 4 no processo da busca sequencial



Após incrementar a variável *i* para o valor 3 e, por consequência, acessar a posição 3 do vetor, o algoritmo irá comparar o valor armazenado com o valor procurado. Neste momento, a comparação será verdadeira e o método deve retornar o valor de *i* para o programa que o chamou.

Apesar de ser um método intuitivo e de baixa complexidade de implementação, nos cenários com casos mais extremos de busca, como a não existência do objeto ou o posicionamento dele na última posição do vetor, será necessária uma quantidade *n* de operações, em que *n* é a quantidade de elementos existentes no vetor. Quando esse número

atingir uma quantidade considerada grande, esse algoritmo torna-se ineficiente, devido ao tempo de processamento. Sendo assim, na próxima seção, serão apresentadas algumas técnicas para otimizar o algoritmo de busca sequencial.

1.1 Otimizando a busca sequencial

Otimizar a busca sequencial é uma tarefa almejada em diversos estudos em ciências da computação. Uma forma de realizá-la é determinar a probabilidade de cada elemento ser pesquisado. Com base nessa probabilidade, deve-se realizar uma ordenação, na qual o elemento com maior probabilidade de ser pesquisado no conjunto seja posicionado entre as primeiras posições do vetor (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). A probabilidade do elemento na posição i é definida pela notação $p(i)$, em que:

$$p(0) \geq p(1) \geq p(2) \geq \dots \geq p(n - 1)$$

Sendo essa a definição das probabilidades de todos os objetos contidos no vetor, a soma de todas as probabilidades deve ser igual a 1, conforme a equação a seguir:

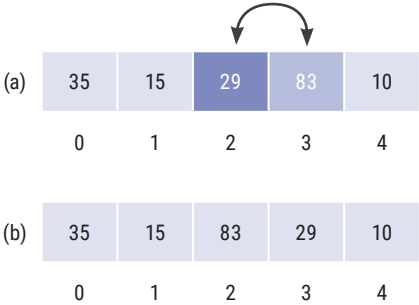
$$\sum_{i=0}^{n-1} p(i) = 1$$

Para armazenar o valor da probabilidade de cada elemento armazenado, é recomendado utilizar um atributo extra ou que seja calculado a partir de alguma regra estabelecida de acordo com um conhecimento prévio sobre os dados armazenados (TENENBAUM; LANGSAM; AUGENSTEIN, 2004).

Entretanto, estabelecer uma regra de probabilidade para operações cotidianas não é uma tarefa simples e pode onerar mais o algoritmo em

vez de otimizá-lo. Para contornar a situação, pode-se utilizar uma regra mais simples que o cálculo de probabilidade. Quando um elemento for encontrado durante a busca, faça a troca dele com o elemento na posição de menor índice (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). Para ilustrar, voltemos ao passo 4 da busca sequencial apresentado na figura 6. O valor 83 foi encontrado durante a realização da busca sequencial. Imediatamente, ele trocaria de posição com o valor anterior, no caso 29, se aproximando mais do início do vetor. O processo de troca é apresentado na figura 7, a seguir.

Figura 7 – Processo de priorização dos valores buscados no vetor



A ideia apresentada com o processo de priorização do valor encontrado na busca pode diminuir o tempo de busca de um elemento específico. Contudo, é um procedimento que pode prejudicar os elementos menos buscados (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). Se pensarmos em um cenário mais extremo, buscar o elemento que é acessado esporadicamente levará ao mesmo número de comparações que a busca sequencial original, ou seja, n elementos.

Trabalhar com o vetor sem uma ordenação prévia, aparentemente, não torna o algoritmo de busca significativamente mais eficiente que o sequencial tradicional. Algoritmos mais eficientes na operação de busca estabelecem a premissa de que o vetor precisa estar ordenado. Por exemplo, a busca sequencial tradicional já possui um pequeno ganho no caso de um vetor ordenado. No critério de parada da instrução *for*,

é possível adicionar uma condição para verificar se o elemento atual é menor que o procurado. Quando essa condição for falsa, a busca pode terminar, não sendo necessário percorrer o vetor completo. Veja o algoritmo modificado para atender a esse critério:

```
int buscaSequencial(int[] vetor, int valorProcurado) {  
    for (int i = 0; i < vetor.length  
        && vetor[i - 1] < vetor[i]; i++) {  
        if (vetor[i] == valorProcurado) {  
            return i;  
        }  
    }  
    return -1;  
}
```

Essa alteração não elimina todos os casos de percorrer o vetor completo quando os valores buscados não se encontram no vetor. Porém, é possível eliminar processamentos exaustivos já sabendo o resultado final (ASCENCIO; ARAÚJO, 2010). Pode parecer simples a ideia de ordenar o vetor antes de realizar o processo de busca, mas um vetor ordenado sofre em média $n/2$ operações para encontrar um valor, enquanto o vetor sem ordenação sofre n operações, lembrando que n é a quantidade de elementos armazenados no vetor.



IMPORTANTE

Dado um vetor que possui n elementos, a quantidade média de operações para encontrar um valor é de:

- n comparações para um vetor sem ordenação; e
- $n/2$ comparações para um vetor ordenado.

Um algoritmo que otimiza a busca sequencial tradicional é o da busca sequencial indexada, que será apresentado na seção seguinte.

1.2 Busca sequencial indexada

O algoritmo de busca sequencial indexada utiliza uma tabela auxiliar para otimizar o processo de busca. Essa tabela geralmente é chamada de índice. A tabela índice armazena a posição onde um determinado elemento está contido na tabela geral. Dessa maneira, é possível reduzir o espaço da busca sequencial, tornando-a mais eficiente em sua velocidade (TENENBAUM; LANGSAM; AUGENSTEIN, 2004).



IMPORTANTE

Na busca sequencial indexada, o uso do TAD lista ligada não é recomendado em hipótese alguma. Seu uso pode gerar uma sobrecarga nos ponteiros de memórias e isso provoca instabilidades no sistema operacional.

Na figura 8, é ilustrado um exemplo de busca sequencial indexada, com um trecho da tabela índice e outro da tabela geral.

Figura 8 – Ilustração das tabelas índice e geral utilizadas na busca sequencial indexada

Tabela índice		Tabela geral	
Elemento	Índice	Índice	Elemento
31	6	0	5
62	14	1	8
		2	12
		3	18
		4	23
		5	27
		6	31
		7	35
		8	38
		9	42
		10	46
		11	49
		12	51
		13	55
		14	62

Imagine que o valor procurado no vetor seja 12. O algoritmo deve ir primeiro à tabela índice e verificar se 12 é menor ou igual ao valor da chave de busca armazenada. No exemplo da figura 8, logo na primeira comparação a condição é verdadeira, uma vez que a comparação realizada é “12 é menor ou igual a 31”. Nesse momento, é estabelecido o intervalo de busca na tabela geral, no qual o índice referente ao valor 31 delimitará o último elemento de comparação do vetor.

Parece que não fará diferença nenhuma, afinal, o exemplo ilustrado possui apenas 15 elementos e foi reduzido para 7. Mas se o cenário ultrapassar 1 milhão de elementos, será que o ganho pode ser considerado, justificando a implementação do algoritmo de busca sequencial indexada? Nessa situação, a busca indexada é bem interessante para diminuir o tempo e o espaço de procura no processo. Contudo, vale uma ressalva importante de que a busca sequencial indexada é mais rápida, mas consome mais memória de armazenamento. Isso ocorre pois são criadas novas tabelas menores para o processo anterior à busca na tabela geral. Algoritmos sempre devem ser balanceados entre memória de processamento e memória de armazenamento e nem sempre é possível otimizar os dois. Uma das memórias deve ser sacrificada em benefício da outra.

Durante a implementação de uma tabela índice o uso do vetor não é uma boa escolha, já que a chave de busca é o mesmo elemento armazenado na tabela geral. Sendo assim, uma melhor escolha de implementação é o uso do dicionário de dados, no qual os tipos da chave e o valor são determinados de acordo com a necessidade do problema. Outro ponto importante é estar ciente de que o uso das tabelas índice não tem limites. Podem ser criadas quantas tabelas índice forem necessárias para diminuir o espaço de busca, atentando-se apenas ao uso da memória de armazenamento para não sobrecarregar o sistema.

Apesar de a busca sequencial indexada otimizar a velocidade na busca por um elemento armazenado, ela ainda não é tão eficiente em relação à constância do tempo gasto na tarefa. Para isso, existe um algoritmo que realiza uma busca mais eficiente. Esse algoritmo é o da busca binária, que será discutido em detalhes no próximo tópico.

2 Busca binária: a mais rápida entre as buscas

A maneira de acelerar um processo de busca é obtendo um conhecimento prévio sobre o dado armazenado no vetor ou qualquer outra estrutura de dados. No entanto, é possível acelerar a busca sem esse conhecimento prévio específico sobre cada tipo de dado. Como apresentado anteriormente, a forma mais simples é manter o vetor totalmente ordenado. Esse tipo de cuidado com o armazenamento auxilia em muitos aspectos. Imagine uma lista de nomes fora da ordem alfabética: ela é praticamente inútil (WIRTH, 1989).

Um vetor ordenado tem a seguinte definição formal:

$$V[n]: 1 \leq k < n : a_{k-1} \leq a_k$$

A partir do vetor ordenado, pode-se aplicar uma técnica chamada de busca binária, que consiste em dividir o vetor de dados em dois subvetores. Os objetos que ficam à esquerda do ponto de divisão devem ser menores que ele, e os que estão à direita, maiores. Quando o algoritmo comparar o objeto no ponto de divisão e este for igual ao elemento procurado, ele retorna sua posição no vetor.

Esse ponto de divisão pode ser determinado por quaisquer posições existentes no vetor, sendo sua escolha aleatória ou arbitrária. Essa decisão específica não influenciará o resultado final da busca (WIRTH, 1989). O importante em cada passo é eliminar o maior número de objetos, atingindo o objetivo de maneira mais rápida. Existe uma solução ótima para esse algoritmo: a escolha do ponto de divisão deve ser baseada no ponto central do conjunto de dados.

A estratégia de cortar o conjunto de dados sempre ao meio faz com que o espaço de busca por um objeto saia de n elementos para $\log n$. É um ganho significativo entre os algoritmos existentes, melhor do que esse resultado apenas uma busca em tempo constante. Sendo assim,

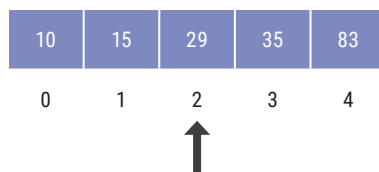
conforme o tamanho dos dados aumenta, o ganho de velocidade da busca binária é computacionalmente expressiva para todo o processo (WIRTH, 1989).

A condição de parada do algoritmo é dada através da seguinte equação:

$$l > F : l \leq m < F$$

Em que l é a posição de início do vetor, portanto menor que m , que representa a posição central do vetor. F é a posição final do vetor, sendo ela maior que m e l . Os passos do algoritmo da busca binária serão apresentados a seguir, iniciando pela figura 9, que representa o passo 1 da busca pelo elemento 83, agora no vetor ordenado.

Figura 9 – Passo 1 do processo do algoritmo da busca binária, onde o centro do vetor é encontrado



O cálculo para encontrar a posição central é repetido novamente. O valor 4 é somado ao 3 e o resultado é dividido por 2. Nesse momento, o resultado obtido é 3,5, porém apenas a parte inteira do valor obtido é considerada, ou seja, 3. O elemento escolhido no passo 2 é apresentado na figura 10, a seguir.

Figura 10 – Passo 2 do processo do algoritmo da busca binária, onde o centro do subvetor é encontrado

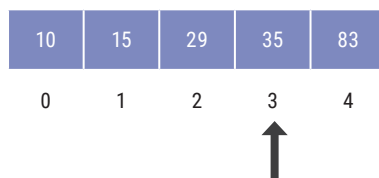



Figura 11 – Passo 3 do processo do algoritmo de busca binária, escolha do elemento no momento em que o vetor dividido possui apenas um elemento

10	15	29	35	83
0	1	2	3	4



Nesse momento, a comparação realizada tem o resultado como verdadeiro. Assim como o algoritmo de busca sequencial, a busca binária retorna agora a posição referente ao elemento procurado, no caso do exemplo, o valor 4. Perceba que, em comparação ao mesmo vetor sem a aplicação da ordenação, o algoritmo fez a busca em uma comparação a menos que a sequencial. Para o vetor ordenado, são exatamente duas comparações a menos. Isso parece pouco no começo, mas com o aumento do volume de dados nota-se um ganho muito grande de desempenho computacional. Veja a seguir a implementação do algoritmo em Java:

```
int buscaBinaria(int[] vetor, int valorProcurado) {
    int inicio = 0;
    int fim = vetor.length - 1;

    while (inicio < fim) {
        int meio = (inicio + fim) / 2;
        if (vetor[meio] > valorProcurado)
            fim = meio - 1;
        else if (vetor[meio] <
valorProcurado)
            inicio = meio + 1;
        else
            return meio;
    }
    return -1;
}
```

É uma implementação relativamente simples, porém o mais importante neste algoritmo refere-se à sua entrada de dados. Todo vetor no processo da busca binária deve ser ordenado. Caso ele não esteja ordenado, o algoritmo poderá finalizar sem realizar a pesquisa de maneira esperada.

3 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

1. No vetor [2, 5, 7, 9, 10, 8, 92] é possível aplicar a busca binária.
2. Crie um vetor ordenado com aproximadamente mil números inteiros gerados aleatoriamente. Agora, utilize a técnica da busca sequencial indexada para otimizar o processo de busca. Sugestão, utilize uma tabela índice com um intervalo de cem chaves aproximadamente.
3. Com base no exercício 2, crie mais uma tabela índice e veja o comportamento do algoritmo. Qual deles foi o mais eficiente?
4. Qual a quantidade de tabelas índice ideal para obter o melhor desempenho computacional?
5. Qual a condição essencial para que o algoritmo da busca binária seja aplicado no vetor? Por quê?
6. Dados os vetores abaixo, qual o método de busca mais adequado para cada um deles:
 - a. [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
 - b. [30, 28, 25, 22, 19, 16, 12, 8, 6, 1]
 - c. [45, 12, 23, 74, 83, 29, 10, 45]

7. Implemente o algoritmo de busca sequencial, reordenando-o com o elemento encontrado vindo uma posição para a frente.

Considerações finais

Neste capítulo, vimos o quão importante é o processo de busca em atividades cotidianas e também nas principais atividades computacionais. A busca sequencial, apesar de intuitiva, não possui um bom desempenho em conjuntos de dados relativamente grandes. Ela pode ser otimizada quando se cria uma tabela de indexação de alguns elementos existentes no vetor. Isso delimita o espaço de busca, tornando a busca sequencial um pouco mais rápida.

No entanto, esse método só é bem-sucedido com o vetor ordenado. A regra da ordenação é essencial para o funcionamento do algoritmo de busca mais eficiente até aqui: a busca binária. Ela consegue reduzir o espaço de pesquisa de um algoritmo de n comparações para $\log n$. Contudo, quando o vetor possui um tamanho reduzido, a diferença de desempenho entre a busca sequencial e a binária é tão baixa que não faz diferença qual método utilizar. A decisão fica a critério da equipe de desenvolvimento e de acordo com os requisitos do sistema em construção.

Referências

ASCENCIO, Ana Fernanda Gomes; ARAÚJO, Graziela Santos de. **Estruturas de dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Hall, 2010.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. **Estruturas de dados usando C**. São Paulo: MAKRON Books, 2004.

WIRTH, Niklaus. **Algoritmos e estruturas de dados**. Rio de Janeiro: Prentice Hall do Brasil, 1989.

