Programação Estruturada

Aula 14 - Recursão: Arrays, Matrizes e Exercícios

Videoaula 03: Buscando elementos em Arrays

Videoaula 03: Buscando elementos em Arrays

Além de ajudar a reduzir a complexidade de um problema, como já foi mencionado, a recursão tem outra capacidade importante, que é a de voltar atrás. Em inglês esse conceito é chamado de *backtracking*. Os problemas que requerem *backtracking*, geralmente, estão relacionados a árvores e grafos, estruturas de dados que, assim como arrays e matrizes, são muito importantes na computação, com inúmeras aplicações práticas, mas que você conhecerá apenas mais na frente na sua formação em TI, ao estudar Algoritmos e Estruturas de dados.

Para exemplificar a importância de algoritmos eficientes, conheça um algoritmo que faz de maneira bastante eficaz uma busca em um array **ordenado**. Em aulas anteriores, você conheceu o método `indexOf`, que retorna o índice do valor passado no array. Caso o elemento não exista no array, esse método retorna -1. No entanto, outras linguagens de programação podem não possuir essa facilidade. Então, como posso encontrar esse índice nessas outras linguagens? Para aprender como fazer isso, considere uma solução que não use o método `indexOf`.

Uma possível solução é simplesmente passar por todos os elementos do array em busca do elemento.

Figura 1 - Exemplo de Array

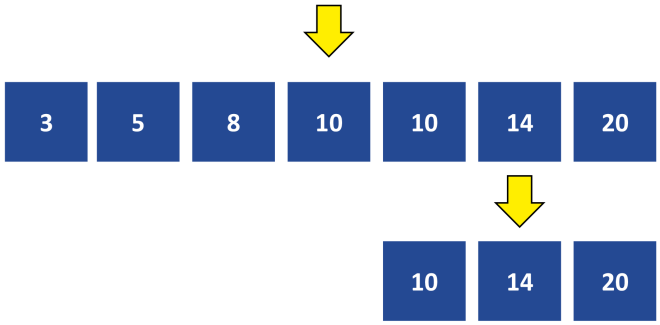
Ao encontrar o elemento, você terá o retorno daquele índice. Porém, caso chegue ao final do array, terá que retornar -1. No exemplo do slide (Figura 1), se procurar o elemento 14, você passará por cinco elementos até encontrá-lo, na sexta posição. Ou seja, teve que fazer seis comparações.

Caso procure o elemento 25, terá que comparar o elemento com todos os sete elementos do array antes de dar o retorno -1.

No entanto, existem maneiras mais eficientes de fazer essa busca. Uma delas chama-se **Busca Binária**. Esse algoritmo de busca em arrays baseia-se no fato de que os arrays estão ordenados para, utilizando o conceito de "dividir para conquistar", já visto nesta disciplina, efetuar sucessivas divisões no array comparando o elemento buscado com o elemento no meio do array. Se o elemento do meio do array for igual ao procurado, terminamos a busca com sucesso. Caso contrário, se o elemento do meio do array for maior que o procurado, sabemos que o elemento procurado se encontra na primeira metade do array e continuamos a busca nessa parte do array. Por fim, se o elemento do meio do array for menor que o procurado, sabemos que o elemento procurado se encontra na segunda metade do array e continuamos a busca nessa parte do array.

Por exemplo, voltando ao slide, suponha que queiramos encontrar o valor 14. Ao comparar o elemento do meio, ou seja, o 10, com 14, nota-se que o elemento do meio é menor que 14. Portanto, como o array está ordenado, sabe-se que o 14 está na segunda parte do array. Continuamos a busca nessa metade do array.

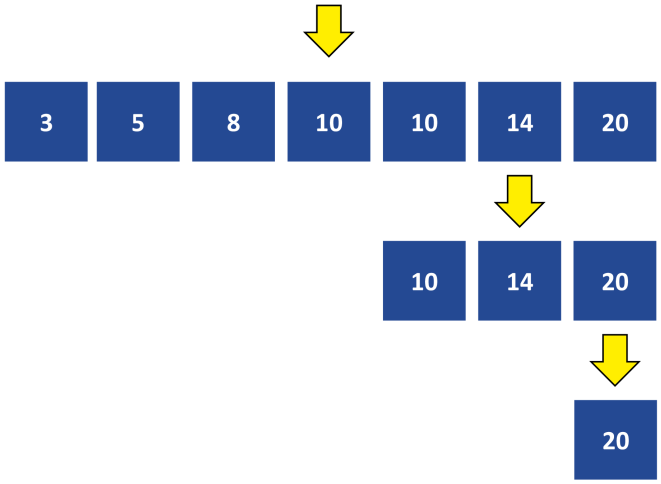
Figura 2 - Buscando na Segunda Metade do Array



Imediatamente, note que agora o elemento do meio do array é exatamente o 14, terminando assim a busca. Você percebeu que foram feitas apenas duas comparações ao invés das cinco feitas na busca linear?

Vejamos com quantas comparações pode-se concluir que o elemento 25 não está no array (Figura 3).

Figura 3 - Busca Binária do Número 25



Ao comparar o elemento do meio, ou seja, o 10, com o 25, nota-se que ele é menor que 25. Como o array está ordenado, sabe-se que o 25 está na segunda parte do array. Continuamos a busca nessa metade do array. Ao comparar o elemento do meio, ou seja, o 14, com o 25, nota-se que ele é menor que 25. Sabe-se que o 25 está na segunda parte do array. Por fim, como temos um array com apenas um elemento, compara-se o 25 com esse elemento e retorna -1, pois não foi encontrado o elemento. Notou que foram feitas apenas três comparações ao invés das sete na busca linear?

A **Busca Binária** pode ser implementada de maneira iterativa, usando laços, ou recursiva. Nesta aula, apresentarei a solução recursiva, mas você pode encontrar uma explicação da solução iterativa acessando o link do QR Code.

Figura 4 - Link para Busca Binária Iterativa

Vamos agora conhecer a implementação recursiva, em JavaScript, da busca linear e da busca binária.

Código 1 - 14_6 Busca Binária.html

```
<html >
   <head>
      <meta charset="UTF-8" />
      <title>Programação Estruturada - Aula 14</title>
   </head>
   <body>
      <noscript>Seu navegador não suporta JavaScript ou ele está desabilitado.</noscript>

      <h1>Busca Binária</h1>

      Entrada: <input type="number" id="entrada" value="">
      <button onclick="inserir()">INSERIR</button>
      <button onclick="limpar()">LIMPAR</button>
      <button onclick="buscar_elemento()">BUSCAR</button>
      <p id="resultado"></p>

      <script src="script.js"></script>
   </body>
</html>

```

```
var numeros = [];
var elemento;

function inserir() {
   var x = Number(document.getElementById("entrada").value);
```

```
    numeros.push(x);
    numeros.sort(function(a, b){return a-b});
    imprimir();
}

function limpar() {
    numeros = [];
    imprimir();
}

function buscar_elemento() {
    do {
        elemento = prompt("Digite um número:");
    } while (elemento == "" || isNaN(elemento));
    elemento = Number(elemento);
    imprimir();
}

function imprimir() {
    var resposta = "Array  = [" + numeros + "] <br>";
    if (!isNaN(elemento)) {
        var indice = busca_linear(numeros, elemento);
        //var indice = busca_binaria(numeros, elemento);
        if (indice >=0) {
            resposta = resposta + "Elemento " + elemento + " ocorre no índice " + indice + "<br>";
        } else {
            resposta = resposta + "Elemento " + elemento + " não ocorre no array " + "<br>";
        }
    }
    document.getElementById("resultado").innerHTML = resposta;
}

function busca_linear(a,e) {
    var indice;
    if (a.length == 0) {
        indice = -1;
    } else {
        if (e == a[0]) {
            indice = 0;
        } else {
            indice_do_resto = busca_linear(a.slice(1), e);

            if (indice_do_resto < 0) {
                indice = -1;
            } else {
                indice = 1 + indice_do_resto;
            }
        }
```

```
    }
    return indice;
}

function busca_binaria(a,e) {
    var indice;

    // Caso Base: Array Vazio
    if (a.length == 0) {
        indice = -1;
    // Caso Base: Array com Apenas Um Elemento
    } else if (a.length == 1) {
        if (a[0] == e) {
            indice = 0;
        } else {
            indice = -1;
        }
    } else {
        var metade = parseInt(a.length / 2);
        // Caso Base: Encontrei o Elemento
        if (e == a[metade]) {
            indice = metade;
         } else if (e > a[metade]) {
            var indice_do_resto = busca_binaria(a.slice(metade, a.length), e);
            if (indice_do_resto < 0) {
              indice = -1;
            } else {
              indice = metade + indice_do_resto;
            }
         } else {
            indice = busca_binaria(a.slice(0, metade), e);
        }
    }

    return indice;
}

```

Para esse exemplo, em nossa página temos um campo de entrada onde a gente pode inserir valores. Tem os botões de INSERIR, LIMPAR e BUSCAR, para que eu possa buscar no array. Ao tentar buscar o elemento 3 no array vazio clicando em OK, ele informa: "Elemento 3 não ocorre no array", como mostra a Figura 5.

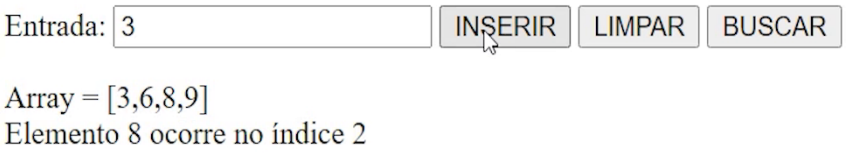
Figura 5 - Buscando 3 no Array Vazio

Ao inserir um elemento, o 8, por exemplo, e tentar buscar o elemento 3 novamente, a mesma mensagem aparecerá, ele não ocorre no array. Entretanto, se eu buscar o 8, ele informa em que índice que ele ocorre, nesse caso, índice 0, como mostra a Figura 6.

Figura 6 - Buscando 8 no Array `[8]`

Vamos inserir o 6, o 9, o 3, mas tem uma coisa interessante que está acontecendo, observe que à medida que os números vão sendo inseridos, eles não estão indo para a calda, na verdade, estão sendo inseridos e ordenados ao mesmo tempo no array, como mostra a Figura 7.

Figura 7 - Buscando 8 no Array `[3,6,8,9]`



Lembre-se que a busca binária considera que o array está sempre ordenado, então toda vez que insere um elemento, ele segue a ordem do array para que a busca fique eficiente. Por exemplo: ao inserir o elemento 7, ele foi parar no meio do array, ou seja, [3, 6, 7, 8, 9]. Isso é o diferencial na nossa inserção.

Figura 8 - Inserindo o 7

Na nossa solução, o HTML é praticamente o mesmo, mas a gente tem a função `buscar_elemento` na linha 14.

No JavaScript, temos novamente a variável `numeros` e a variável `elemento` que a gente vai procurar. A função `inserir`, parecida com a que já usamos, porém depois de colocar o elemento no array na linha 6, vemos na linha 7 a ordenação e já vimos como ordenar arrays numéricos anteriormente. Chamamos o método `sort`, passando a função de ordenação, que pega `a` e `b` e retorna `a-b` para ordenar numericamente o array `numeros`, ok?

A função `limpar` é exatamente a mesma e a função `buscar_elemento` é praticamente igual à que vimos antes também, porém vou converter o elemento que peguei, que é uma *string*, em um número, pois podemos fazer isso, e fica até melhor para que a pessoa enxergue que essa conversão está acontecendo.

A função `imprimir` imprime o array e, para isso, a nossa resposta contém o array e o índice é a busca naquele array daquele elemento. Então, vou apresentar duas soluções: a `busca_linear` e a `busca_binaria`.

Na `busca_linear` (Linha 38), se o índice retornado for maior ou igual a 0 é porque ele encontrou o elemento no array, e vou dizer qual índice em que o elemento ocorre. Caso contrário, ele vai retornar `-1` e, ao invés de dizer qual foi o índice que ele ocorreu, vou dizer que o elemento não ocorre no array, para ficar uma mensagem mais amigável. Assim, a partir da linha 38, podemos ver como funciona a `busca_linear`. Quando o array não tem elementos, ou seja, se `a.length==0`, chegamos no final do array e não encontramos o elemento. Então, temos que `indice = -1`, está certo?

Caso contrário, a gente vai ter que fazer o seguinte: se o `e` (elemento procurado) foi encontrado, dizemos que o índice é 0 porque foi no índice 0 que a gente encontrou o elemento, ou seja, `a[0]`. Ou a gente vai procurar ele no resto, então qual é o índice dele no resto do array? Vai ser a busca linear, no resto do array, desse elemento, ou seja, `busca_linear(a.slice(1)), e)`. Portanto, a gente não encontrou ele no primeiro elemento, fomos buscar no resto do array e vemos o que retornou. Se retornou um número negativo é porque não encontrou no resto do array, então eu retorno o -1, tá certo?

Entretanto, vindo um número positivo, significa que encontrei nesse array, mas na verdade o índice que eu encontrei está defasado em 1. Por quê? Porque eu estava procurando na calda do array, então, se eu encontrei no índice 3, por exemplo, do resto do array, vou ter que retornar 4 porque, na verdade, eu tirei um elemento para procurar no resto. Então, quando eu retornar o que encontrei, adiciono mais 1, e é o que fazemos na linha 51, `indice = 1 + indice_do_resto`. Percebam que estou procurando de maneira recursiva em todos os elementos do array, está bom?

Agora, vamos usar a solução binária e, para isso, vamos apagar a linha 27 e inserir a chamada à `busca_binaria`. E como ela está definida? Na linha 57, vou fazer uma busca binária no array do elemento `e`. Primeiro caso base: o array está vazio e, se está vazio, sabemos que o índice é `-1`.

Outro caso base que podemos trabalhar é se a gente tiver um array com apenas um elemento. Caso isso aconteça, preciso olhar se esse elemento é o que eu estou procurando ou não. Se ele for o mesmo, encontrei no índice 0 e informo que `indice = 0`; caso contrário, `indice = -1`, como consta nas linhas 65 a 68.

Porém, se eu não acabei o array e se ele não tem tamanho 1, significa que ele tem tamanho pelo menos 2. E o que vou fazer? Vou encontrar qual é o índice da metade. Para isso, pego `a.length` e divido por 2, e faço um `parseInt` (linha 71).

O `parseInt` transforma um número que não é inteiro, por exemplo 1.5, em um número inteiro. Mas por que tenho que fazer isso? Porque se eu tiver um array com três elementos, por exemplo, e dividir por 2, vou ter como resultado 1.5, já que o JavaScript vai converter esse número, e não tenho índice que seja 1.5, então, como eu faço? Uso o `parseInt` para transformar esse 1.5 em um número inteiro, já que não tenho índice 1.5, e chamo ele de `metade`, vejo se o array naquele elemento da metade é igual ao elemento que eu estou procurando. Se ele for igual, então o meu `indice` é `metade`. Caso contrário, o que a gente faz? Vemos se meu elemento é maior ou menor que o elemento do array no `indice metade`. Se ele for maior, está na segunda parte e se for menor, está na primeira parte.

Se ele for maior (linha 75), então o `indice_do_resto` (o resto é a segunda parte) é uma `busca_binaria` no `a.slice` da metade para o final, lembrando que ele vai fazer até `a.length(-1)`. Se o `indice_do_resto < 0`, ele é -1, o que significa que não encontrei no resto, por isso é -1. Caso contrário, ele está na metade mais o índice que eu encontrei, mais o `indice_do_resto`, como descrito na linha 80.

Esse `else` da linha 82, é se o elemento não for maior do que a metade, então ele não passou por essa condição, ou seja, ele não é nem igual, nem maior, mas sim menor que o elemento que está na metade. Assim, o que tenho que fazer é bem mais simples, eu faço uma `busca_binaria` na primeira parte do array, que é o slice do primeiro elemento até a `metade - 1`, ou seja, até antes da metade. Então, não preciso somar nada porque estou olhando já na primeira parte, então o que eu encontrar aqui é o "índice real" do elemento que é igual ao que estou procurando.

Perceba que esse algoritmo é bem mais eficiente porque ele faz menos comparações, logo, ele resolve o problema de maneira bem mais rápida. Por fim, para conferirmos, vou recarregar a página HTML, insiro os elementos 3, 6, 8, 5 e 6 e, ao buscar, por exemplo, o elemento 6, ele informa que ocorre no índice 2, então veja que está funcionando como mostra a Figura 9.

Figura 9 - Resultado da Busca Binária

Não dá para sentir a eficiência desse algoritmo, mas quando se tem muitos dados, ou seja, quando o nosso array é muito grande, essa eficiência pode ser visualizada de maneira considerável, está certo?

São soluções mais eficientes como essa que você acabou de conhecer que aprenderá quando estudar Algoritmos e Estruturas de Dados em Ciência da Computação. Massa, não acha? Continue estudando... 😉 Tchau, tchau!