

Técnicas e Análise de Algoritmos Busca e Ordenação - Parte 01

Professor: Jeremias Moreira Gomes

E-mail: jeremias.gomes@idp.edu.br



Introdução





- Um algoritmo de busca consiste em uma função que identifica se um elemento x pertence ou não a um conjunto de elementos S
 - A função pode retornar um valor booleano (verdadeiro ou falso), caso o elemento pertença ou não ao conjunto S
 - Outra alternativa é retornar a posição (índice) do elemento no conjunto, caso este faça parte do mesmo, ou um valor sentinela, indicando que o elemento não pertence ao conjunto



- Caso o conjunto S seja um vetor, o algoritmo de busca mais simples é a busca sequencial, onde todos os elementos do vetor são comparados com o elemento que se deseja encontrar
- A ordem de complexidade do algoritmo é O(N), onde N é número de elementos do vetor
 - Embora existam algoritmos mais eficientes, este algoritmo
 funciona independentemente da ordenação dos elementos



Assinatura de uma função de busca:

```
int busca(int *vetor, int N, int chave);
```

- Onde:
 - chave é o elemento a ser buscado no vetor vetor de tamanho N.
 - Essa função retorna o índice do valor dentro do vetor, se ele existir ou -1, caso contrário



25								
0	1	2	3	4	5	6	7	8

busca(vetor, 9, 7);

• A chamada anterior deve retornar 4.



- A busca sequencial é o algoritmo mais simples de busca
 - Ele percorre todo o vetor comparando a chave com o valor de cada posição
 - Se for igual para alguma posição, ele retorna essa posição
 - Se o final do vetor for alcançado, ele devolve -1



```
int buscasequencial(int *vetor, int tamanho, int chave)
{
   int i;
   for (i = 0; i < tamanho; i++) {
      if (vetor[i] == chave) {
        return i;
      }
   }
   return -1;
}</pre>
```



Busca Sequencial em C++ (1/2)

- A biblioteca algorithm do C++ contém uma implementação da busca sequencial chamada find()
- A função find() recebe dois iteradores a e b, e um valor x, a ser procurado
- Caso x se encontre dentre os elementos que estão no intervalo
 [a, b), é retornado um iterador para a primeira ocorrência de x



Busca Sequencial em C++ (2/2)

- Caso x não esteja no intervalo, é retornado o valor b
- Esta função pode ser usada em qualquer contêiner que tenha iteradores que suportem a operação de incremento e que armazenem um tipo que suporte o operador == (de comparação)



Busca Sequencial em C++

```
vector<int> V {3, 2, 8, 1, 0, 7, 1, 5, 6};
int x = 7;

auto it = find(V.begin(), V.end(), x);

if (it == V.end()) {
    cout << "Valor nao encontrado" << endl;
} else {
    cout << "Valor encontrado na posicao " << it - V.begin() << endl;
}</pre>
```



- O processo de se visitar cada um dos elementos contidos em um contêiner é denominado travessia
- A busca sequencial usa uma travessia para confrontar cada um dos elementos do contêiner contra o valor que se deseja localizar



- Um padrão comum associado à travessia é o de se escolher um ou mais elementos do contêiner, de acordo com um predicado P
 - Um predicado P é uma função que recebe, dentre seus parâmetros, um elemento e do tipo T e retorna ou verdadeiro ou falso
 - Este padrão recebe o nome de filtro



 Uma busca sequencial pode ser interpretada como um filtro que seleciona um (ou mais) elemento do contêiner a partir do predicado

```
bool P(const T& e, const T& x) { return e == x; }
```



- A função copy_if() é um filtro genérico do C++ e resulta:
 - Todos os elementos "e" tais que P (e) é verdadeiro serão copiados no iterador de saída s
- Pode ser usada com a função back_inserter(), da biblioteca iterator, que gera um iterador de saída para o contêiner passado como parâmetro
 - O contêiner precisa ter push_back()



```
string vogais = "aeiou";
string mensagem = "Procurando vogais..";
string resultado;
auto P = [&vogais](const char& c) {
    return vogais.find(c) != string::npos;
};
copy_if(mensagem.begin(), mensagem.end(), back_inserter(resultado), P);
for (auto c : resultado) {
    cout << c << " ";
cout << "\n";
```



Busca - Transformações

- Outro padrão associado à travessia é a transformação
- Uma transformação visita cada um dos elementos x de S, e o substitui pelo resultado da transformação T (x)
- A biblioteca algorithm do C++ implementa transformações através da função transform()



Busca - Transformações

```
vector<double> dinheiros {1.37, 2.00, 3.14, 4.20};
vector<double> res(dinheiros.size());
auto dez porcento = [](const double& x) {
    return x * 1.1;
};
transform(dinheiros.begin(), dinheiros.end(), res.begin(), dez_porcento);
for (auto x : res) {
    cout << x << " ";
cout << endl;</pre>
```



Busca

- A busca sequencial é o método mais comum utilizado em vetores com poucos elementos e é necessário percorrer todos os elementos (no pior caso)
 - Principalmente se eles n\u00e3o estiverem ordenados
 - Mas e se os valores estiverem ordenados?
 - Aí conseguimos pensar de uma maneira mais "esperta"





- Pense no seguinte jogo:
 - O computador irá sortear um número aleatório entre 0 e 100
 - Seu objetivo é descobrir qual número foi sorteado pelo computador, fazendo chutes
 - Sempre que você chutar um número, o computador irá te informar se o seu palpite foi maior, menor ou igual ao sorteado
 - Qual a sua estratégia para achar esse número?



- Para o jogo sugerido:
 - Vale a pena buscar pelo valor correto sequencialmente?
 - Não, por causa da Lei de Murphy
 - Repare em qual é a nossa lista:
 - **1**, 2, 3, 4, 5, 6, 7, 8, ..., 97, 98, 99, 100
 - Como a lista já está ordenada, conseguimos fazer alguns saltos para facilitar encontrar o valor



- Esses saltos realizados podem e devem ser implementados em algoritmos sempre que possível
 - Esse "possível" é para quando os valores da lista encontram-se ordenados (que é o caso desse jogo de adivinhação)
- A técnica de busca envolvida aqui chama-se Busca Binária



- A Busca Binária é uma estratégia algorítmica "sofisticada"
- É mais eficiente que a Busca Sequencial.
 - No pior caso, percorremos todos os valores uma vez (O(n))
 - Porém, exige que o vetor esteja ordenado.
 - Dependendo da quantidade de buscas, pode valer a pena
 ordenar o vetor e em seguida realizar as buscas necessárias



- A ideia da Busca Binária é a seguinte:
 - Verifique se a chave de busca é igual ao valor da posição do meio do vetor
 - Se for igual, retorne essa posição
 - Caso o valor desta posição seja maior, repita a busca, mas agora na metade menor do vetor
 - Caso o valor desta posição seja menor, repita a busca, mas agora na metade maior do vetor



buscabinaria(vetor, 10, 9);

1									
0	1	2	3	4	5	6	7	8	9

- inicio = 0
- fim = 9
- \bullet meio = 4



buscabinaria(vetor, 10, 9);

<u></u>											
1	2	7	9	13	16	19	22	25	26		
_	_	_									
0	1	2	3	4	5	6	7	8	9		

- inicio = 0
- fim = 9
- meio = 4
- Na posição 4 (meio), o valor 13 é maior que o procurado (9).



buscabinaria(vetor, 10, 9);

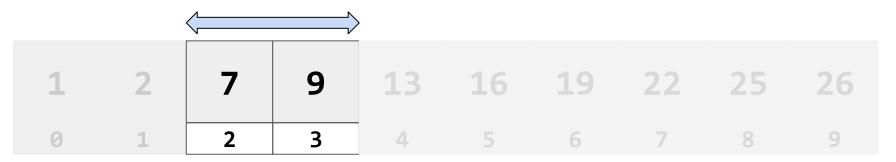
				•					
1	2	7	9	13	16	19	22	25	26
0	1	2	3	4	5	6	7		9

- inicio = 0
- fim = 3
- meio = 1

• Agora, na posição 1 (meio), o valor 2 é menor que o procurado (9).



buscabinaria(vetor, 10, 9);

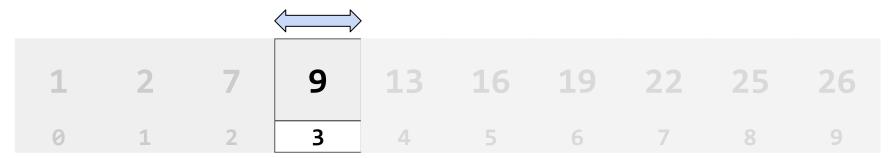


- inicio = 2
- fim = 3
- meio = 2

• Agora, na posição 2 (meio), o valor 7 é menor que o procurado (9).



buscabinaria(vetor, 10, 9);



- inicio = 3
- fim = 3
- meio = 3
- Agora, na posição 3 (meio), o valor 9 é igual ao procurado.



```
int busca_binaria(vector<int>& V, int x)
    int ini = 0, fim = V.size() - 1;
    while (ini <= fim) {</pre>
        int meio = ini + (fim - ini) / 2;
        if (V[meio] == x) {
            return meio;
        } else if (V[meio] > x) {
            fim = meio - 1;
        } else {
            ini = meio + 1;
    return -1;
```



Busca Binária em C++

- A biblioteca algorithm do C++ traz três funções associadas à busca binária
 - binary_search() retorna verdadeiro se encontrar
 - lower_bound() retorna um iterator para o primeiro encontrado mais a esquerda se encontrar, ou primeiro maior
 - upper_bound() retorna um iterador para o último elemento maior ou igual a x, ou estritamente maior



Eficiência da Busca Sequencial

- Na melhor das hipóteses, a chave de busca estará na posição 0
 - Um único acesso na lista
- Na pior das hipóteses, a chave é o último elemento ou não pertence à lista
 - Todos os elementos são acessados
- Se as chaves possuírem a mesma probabilidade de serem requisitadas, o número médio de acessos é:

$$\frac{n+1}{2}$$



Eficiência da Busca Binária

- Na melhor das hipóteses, a chave de busca estará na posição do meio da lista
 - Um acesso
- Na pior das hipóteses, dividimos a lista até a que ela fique com um único elemento
 - Cada acesso, o tamanho da lista é diminuído, pelo menos, pela metade
 - Quantas vezes um número pode ser dividido por dois antes dele se tornar igual a um?



Eficiência da Busca Binária

- Essas divisões sucessivas são exatamente a definição de logaritmo na base 2
- Se as chaves possuírem a mesma probabilidade de serem requisitadas, o número médio de acessos nas buscas cujas chaves encontram-se na lista será igual a

$$(\log_2 n) - 1 = \lg n - 1 = O(\lg n)$$



Conclusão