Comparando Eficiência de Algoritmos

Nesse documento iremos comparar o tempo de execução dos algoritmos de busca linear, busca linear em ordem e busca binária.

1. O Que São?

Para fazer essa comparação, precisamos entender o que cada algoritmo faz:

1. 1. Busca Linear

A busca linear é a busca de um elemento em uma lista de forma que a comparação do número desejado com cada elemento da lista seja feita de um em um. É extremamente simples, mas pode ser ineficaz se o elemento a ser encontrado estiver no final de uma lista bem grande.

2. 2. Busca Linear Em Ordem

Segue o mesmo padrão da Busca Linear, mas como a lista está em ordem, a busca pode ser interrompida se um elemento da lista for maior que o elemento a ser encontrado. No caso, pode significar duas coisas: o elemento já foi achado ou o elemento não existe na lista, assim interrompendo o processo.

3. 3. Busca Binária

Para que a Busca Binária ocorra, a lista terá que estar ordenada. Em seguida, o elemento a ser achado é comparado ao elemento do meio da lista. Caso o valor não seja o mesmo, temos duas opções: o elemento a ser achado é maior ou é menor. Digamos que seja maior, então temos metade da lista para procurar, da posição 1 até a anterior do meio da lista. Com a lista reduzida, podemos repetir o processo de comparar a metade dessa metade com o elemento a ser achado e repetindo até o elemento ser encontrado ou dado como não existente na lista.

2. Calculando o Tempo de Execução

2. 1. Busca Linear

Usando o código em C abaixo podemos calcular os quatro casos a serem testados neste documento. Vejamos:

```c
int buscaLinear(int A[], int n, int x) {
    for (int i = 0; i < n; i++) {                // 1
        if (A[i] == x) {                         // 2
            return i;                            // 3
        }
    }
    return -1;                                   // 4
}
```

2.1.1. BL - x ∈ A

x E A (quando x está em A):

- **Iterações**: O número de iterações k é a posição de x em A.
- **Custo por iteração**:
    - Atribuição de valor a `i` (1 vez por iteração): t
    - Acesso ao array `A[i]` (1 vez por iteração): t
    - Comparação `A[i] == x` (1 vez por iteração): t
- **Tempo total**:
    - Dentro do loop: k × (t+t+t) = k × 3t
    - Após o loop (retorno): t
    - Total: k × 3t + t

2.1.2. BL - x = A[1]

x = A[1] (quando x está no primeiro elemento):

- **Iterações**: 1
- **Custo**:
    - Atribuição de valor a `i`: t
    - Acesso ao array `A[i]`: t
    - Comparação `A[i] == x`: t
- **Tempo total**:
    - Dentro do loop: 1 × (t+t+t)= 3t
    - Após o loop (retorno): t
    - Total: 4t

2.1.3. BL - x = A[n]

x = A[n] (quando n está no último elemento):

- **Iterações**: n
- **Custo**:
    - Atribuição de valor a `i`: n × t
    - Acesso ao array `A[i]`: n × t
    - Comparação `A[i] == x`: n × t
- **Tempo total**:

- Dentro do loop: n × (t+t+t) = n × 3t
- Após o loop (retorno): t
- Total: n × 3t + t = 3nt + t

2.1.4. BL - x ∉ A

x não pertence a A:

- **Iterações**: n
- **Custo**:
  - Atribuição de valor a `i`: n × t
  - Acesso ao array `A[i]`: n × t
  - Comparação `A[i] == x`: n × t
- **Tempo total**:
  - Dentro do loop: n × (t+t+t) = n × 3t
  - Após o loop (retorno): t
  - Total: n × 3t + t = 3nt + t

2.2. Busca Linear em Ordem

```
int buscaLinearEmOrdem(int A[], int n, int x) {
    for (int i = 0; i < n; i++) {                // 1
        if (A[i] > x) {                          // 2
            return -1;                           // 3
        }
        if (A[i] == x) {                         // 4
            return i;                            // 5
        }
    }
    return -1;                                   // 6
}
```

2.2.1. BLO - x ∈ A

x E A (quando x está em A):

- **Iterações**: k, onde k é a posição de x em A.
- **Custo por iteração**:
  - Atribuição de valor a `i` (1 vez por iteração): t
  - Acesso ao array `A[i]` (2 vezes por iteração): 2t
  - Comparações (`A[i] > x` e `A[i] == x`): 2t
- **Tempo total**:
  - Dentro do loop: k × (t+2t+2t)= k × 5t
  - Após o loop (retorno): t
  - Total: k × 5t + t

2.2.2. BLO - x = A[1]

x = A[1] (quando x está no primeiro elemento):

- **Iterações**: 1
- **Custo**:
  - Atribuição de valor a `i`: t
  - Acesso ao array `A[i]`: 2t
  - Comparações (`A[i] > x` e `A[i] == x`): 2t
- **Tempo total**:
  - Dentro do loop: 1 × (t+2t+2t) = 5t
  - Após o loop (retorno): t
  - Total: 5t + t = 6t

2.2.3. BLO - x = A[n]

x = A[n] (quando x está no último elemento):

- **Iterações**: n
- **Custo**:
  - Atribuição de valor a `i`: n × t
  - Acesso ao array `A[i]`: n × 2t
  - Comparações (`A[i] > x` e `A[i] == x`): n × 2t
- **Tempo total**:
  - Dentro do loop: n × (t+2t+2t) = n × 5t
  - Após o loop (retorno): t
  - Total: n × 5t + t = 5nt + t

2.2.4. BLO - x ∉ A

x não pertence a A:

- **Iterações**: n
- **Custo**:
  - Atribuição de valor a `i`: n × t
  - Acesso ao array `A[i]`: n × 2t
  - Comparações (`A[i] > x` e `A[i] == x`): n × 2t
- **Tempo total**:
  - Dentro do loop: n × (t+2t+2t) = n × 5t
  - Após o loop (retorno): t
  - Total: n × 5t + t =5nt + t

2.3. Busca Binária

```c
int buscaBinaria(int A[], int n, int x) {
    int esq = 0, dir = n - 1;                    // 1
    while (esq <= dir) {                         // 2
        int meio = esq + (dir - esq) / 2;        // 3
        if (A[meio] == x) {                      // 4
            return meio;                         // 5
        } else if (A[meio] < x) {                // 6
            esq = meio + 1;                      // 7
        } else {                                 // 8
            dir = meio - 1;                      // 9
        }
    }
    return -1;                                   // 10
}
```

2.3.1. BB - x ∈ A

x E A (quando x está em A):

- **Iterações**: log2n
- **Custo por iteração**:
  - Atribuições (para `esq`, `dir`, `meio`): 3 vezes por iteração: 3t
  - Acesso ao array `A[meio]` (1 vez por iteração): t
  - Comparações (`A[meio] == x`, `A[meio] < x`): 2 vezes por iteração: 2t
- **Tempo total**:
  - Dentro do loop: (log2n) × (3t + t + 2t) = (log2n) × 6t
  - Após o loop (retorno): t
  - Total: (log2n) × 6t + t = (log2n) × 6t + t

2.3.2. BB - x = A[1]

x = A[1] (quando x está no primeiro elemento):

- **Iterações**: log2n
- **Custo**:
  - Atribuições: 3t
  - Acesso ao array `A[meio]`: t
  - Comparações: 2t
- **Tempo total**:
  - Dentro do loop: (log2n) × (3t + t + 2t) = (log2n) × 6t
  - Após o loop (retorno): t
  - Total: (log2n) × 6t + t = (log2n) × 6t + t

2.3.3. BB - x = A[n]

x = A[n] (quando x está no último elemento):

- **Iterações**: log2n
- **Custo**:
    - Atribuições: 3t
    - Acesso ao array `A[meio]`: t
    - Comparações: 2t
- **Tempo total**:
    - Dentro do loop: (log2n) × (3t + t + 2t) = (log2n) × 6t
    - Após o loop (retorno): t
    - Total: (log2n) × 6t + t = (log2n) × 6t +t

2.3.4. BB - x ∉ A

x não pertence a A:

- **Iterações**: log2
- **Custo**:
    - Atribuições: 3t
    - Acesso ao array `A[meio]`: t
    - Comparações: 2t
- **Tempo total**:
    - Dentro do loop: (log2n) × (3t + t + 2t) = (log2n) × 6t
    - Após o loop (retorno): t
    - Total: (log2n) × 6t + t = (log2n) × 6t + t

3. Comparando resultados

Vimos que o tempo total de cada busca é:

XXXX	Busca Linear	Busca Linear em Ordem	Busca Binária
x ∈ A	k × 3t + t	k × 5t + t	(log2n)×6t+t
x = A[1]	4t	6t	(log2n)×6t+t
x = A[n]	3nt + t	5nt + t	(log2n)×6t+t
x ∉ A	3nt + t	5nt + t	(log2n)×6t+t

Busca Linear é simples, mas menos eficiente para grandes conjuntos de dados, especialmente quando o elemento não está presente.

Busca Linear em Ordem melhora um pouco a busca linear em casos onde os dados estão ordenados, mas ainda não é tão eficiente quanto a busca binária para grandes conjuntos de dados.

Busca Binária é a mais eficiente em termos de tempo, especialmente para grandes conjuntos de dados, desde que a lista esteja ordenada.

A escolha do algoritmo de busca deve considerar o tamanho dos dados e se a lista está ordenada. Para listas ordenadas e grandes, a busca binária é geralmente a melhor escolha.