Relatorio

Karine Piacentini Coelho da Costa[1]

March 21, 2019

[1]karinepcdc@ufrn.br

Contents

Chapter 1

Introdução

Esse relatório descreve uma análise de complexidade empírica de diferentes algoritmos de busca. São considerados os seguintes algoritmos de busca: busca linear; busca binária (versões iterativas e recursiva); busca ternária (versões iterativas e recursiva); *jump search*; e busca de Fibonacci.

O primeiro objetivo desse estudo é determinar qual dos dois algoritmos lineares são mais eficientes (a busca linear ou a *jump search*). O segundo objetivo é determinar qual implementação é mais eficiente, a recursiva ou iterativa. O terceiro, é determinar como o tamanho da partição influência nos algoritmos de busca não lineares. O quarto é determinar a partir de que momento algoritmos de classe de complexidade diferentes se diferenciam (comparando a busca linear com a binária). Por fim, o quinto objetivo procura determinar se existe diferentes categorias de cenários pior caso para o algoritmo de busca de Fibonacci.

Chapter 2

Metodologia

Nesta seção descrevemos os materiais e a metodologia utilizados para obteção dos resultados apresentados no capítulo 3.

2.1 Características técnicas

Os algorítmos de buscas foram implementados na linguagem C++ e o compilador utilizado foi o g++ (tipo e versão????). O computador onde as simulações foram realizadas possui as seguintes características:

- MacBook Pro (2014)
- Processador: 2.5 Ghz Intel Core i7
- memória: 16 GB 1600 MHz DDR3
- Placa mãe: ????
- Sistema operaciona: (tipo e versão????)

2.2 Algoritmos

Os algoritmos de busca utilizados nesse estudo estão apresentados aqui com uma breve descrição. Os códigos utilizados encontram-se no apêndice 1. ????

2.2.1 Busca linear

Algoritmo 1: Busca linear

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.
`/* Precondição:` $l \leq r$; $l, r \geq 0$; V `em ordem crescente. */`

```
1  Função buscaLin(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
     inteiro): inteiro
2      var i: inteiro
3      para i ← l até r faça
4          se V[i] == k então
5              retorna i
6          fim
7      fim
8      retorna −1
9  fim
```

2.2.2 Busca binária

Algoritmo 2: Busca binária iterativa

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.
`/* Precondição:` $l \leq r$; $l, r \geq 0$; V `em ordem crescente. */`

```
1  Função buscaBin_it(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
     inteiro): inteiro
2      var m: inteiro /* último valor da primeira metade do arranjo  */
3
4      enquanto r ≥ l faça
5          m ← (l + r)/2
6          se k == V[m] então
7              retorna m
8          senão se k < V[m] então
9              r ← m − 1
10         senão
11             l ← m + 1
12         fim
13     fim
14     retorna −1
15 fim
```

Algoritmo 3: Busca binária recursiva

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.
`/* Precondição:` $l \leq r$; $l, r \geq 0$; V `em ordem crescente. */`

```
1  Função buscaBin_rec(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
     inteiro): inteiro
2      var m: inteiro /* último valor da primeira metade do arranjo */
3
4      se r < l então
5          retorna −1
6      senão
7          m ← (l + r)/2
8          se k == V[m] então
9              retorna m
10         senão se k < V[m] então
11             retorna buscaBin_rec(V,l,m − 1,k)
12         senão
13             retorna buscaBin_rec(V,m + 1,r,k)
14         fim
15     fim
16 fim
```

2.2.3 Busca ternária

Algoritmo 4: Busca ternária iterativa

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.

```
/* Precondição:  l ≤ r; l, r ≥ 0; V em ordem crescente.          */
1  Função buscaTer_it(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
     inteiro): inteiro
2      var t1: inteiro /* último valor do primeiro terço do arranjo   */
3      var t2: inteiro /* último valor do segundo terço do arranjo    */
4
5      enquanto r ≥ l faça
6          t1 ← l + (r − l)/3
7          t2 ← r − (r − l)/3
8
9          se k == V[t1] então
10             retorna t1
11         senão se k == V[t2] então
12             retorna t2
13         senão se k < V[t1] então
14             r ← t1 − 1
15         senão se k < V[t2] então
16             l ← t1 + 1
17             r ← t2 − 1
18         senão
19             l ← t2 + 1
20         fim
21     fim
22     retorna −1
23 fim
```

Algoritmo 5: Busca ternária recursiva

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.

```
/* Precondição:  l ≤ r; l,r ≥ 0; V em ordem crescente.          */
1  Função buscaTer_rec(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
     inteiro): inteiro
2      var t1: inteiro /* último valor do primeiro terço do arranjo   */
3      var t2: inteiro /* último valor do segundo terço do arranjo    */
4
5      se r < l então
6          retorna −1
7      senão
8          t1 ← l + (r − l)/3
9          t2 ← r − (r − l)/3
10
11         se k == V[t1] então
12             retorna t1
13         senão se k == V[t2] então
14             retorna t2
15         senão se k < V[t1] então
16             retorna buscaTer_rec(V,l,t1 − 1,k)
17         senão se k < V[t2] então
18             retorna buscaTer_rec(V,t1 + 1,t2 − 1,k)
19         senão
20             retorna buscaTer_rec(V,t2 + 1,r,k)
21         fim
22     fim
23 fim
```

2.2.4 *Jump search*

Algoritmo 6: *Jump search*

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.

```
/* Precondição:  l ≤ r; l,r ≥ 0; V em ordem crescente.          */
1  Função buscaJump(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
     inteiro): inteiro
2      var m: inteiro
3      var p: inteiro /* tamanho do salto                        */
4
5      p ← √(r − l + 1)
6      m ← l + p
7      enquanto m ≤ r faça
8          se k == V[m] então
9              retorna m
10         senão se k < V[m] então
11             retorna buscaLin(V, m − p, m − 1, k)
12         fim
13         m ← m + p
14     fim
15     se m > r e V[r] > k então
16         retorna buscaLin(V, m − p, r, k)
17     fim
18     retorna −1
19 fim
```

2.2.5 Busca de Fibonacci

Algoritmo 7: Busca de Fibonacci

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.

```
/* Precondição:  l ≤ r; l,r ≥ 0; V em ordem crescente.          */
```

```
1  Função buscaBin_it(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
     inteiro): inteiro
2      var size: inteiro
3      var i: inteiro
4      var i_fib1: inteiro
5      var fib1: inteiro
6      var Fib: arranjo de inteiros
7      size ← r − l + 1
       /* Calcula a serie de Fibonacci até o i-ésimo termo,
       onde F(i) <= size                                          */
8      Fib[0] ← 0
9      Fib[1] ← 1
10     i ← 1
11     enquanto Fib[i] < size faça
12         i ← i + 1
13         Fib[i] ← Fib[i − 1] + Fib[i − 2]
14     fim
15
16     i_fib1 ← i − 2
17     enquanto l < r faça
18         fib1 ← l + F[i_fib1]
19         se k == V[fib1] então
20             retorna fib1
21         senão se k < V[fib1] então
               /* Novos tamanhos das partições à esquerda        */
22             r ← i_fib1 − 1
23             i_fib1 = i_fib1 − 2
24             se i_fib1 < 0 então
25                 i_fib1 ← 0
26             fim
27         senão
               /* Procure os novos tamanhos das partições à direita */
28             l ← i_fib1 + 1
29             i ← i_fib1 + 1
30             enquanto Fib[i] < r − l + 1 faça
31                 i ← i − 1
32             fim
33             i_fib1 = i − 1
34         fim
35     fim
36     retorna −1
37 fim
```

2.3 Cenários das simulações

As simulações foram feitas buscando um valor em um conjunto *ordenado crescente*. Consideramos o pior caso apenas, ou seja a busca de um valor que não pertence ao conjunto de busca, mas que é maior que o maior elemento neste conjunto.

2.4 metodologia

Um vetor de inteiros longos de tamanho 10^8 preenchido com números pares em ordem crescente foi utilizado para gerar as amostras.

Amostras de arranjo

2.4.1 Simulações de tempo de execução

média temporal progressiva...

2.4.2 Simulações do número de passos da operação dominante

Chapter 3

Resultados

3.1