Análise Formal de Buscas

Luiz Fernando Rabelo – 11796893

1. Notas

- Em todas as contagens de operações, foram analisados e considerados os piores casos;
- Os gráficos apresentados representam o tempo de execução da busca pelo tamanho do array de inteiros em que a chave é buscada. Em todas as buscas simuladas, é procurada uma chave que não está no array.
- As operações consideradas como relevantes foram as comparações e as operações aritméticas, abreviadas como **comp** e **op**, respectivamente.

2. Busca Sequencial

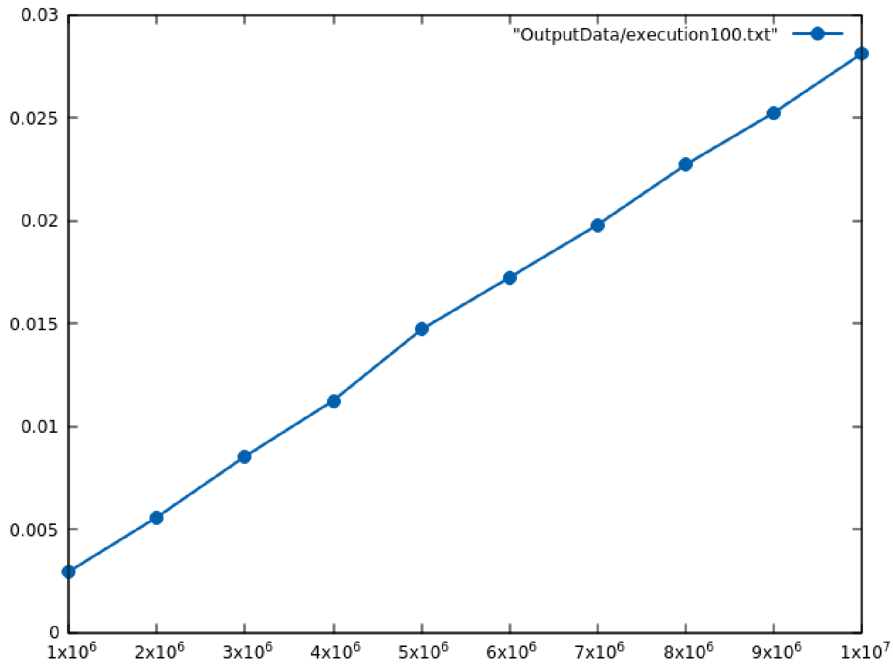
```
int sequentialSearch(NUMBER_ARRAY *numberArray, int key){
	for(int i = 0; i < numberArray->size; i++) // 1 comp e 1 op
		if(numberArray->elements[i] == key) return i; // 1 comp
	return -1;
}
```

Adotando **n** como numberArray->size, temos que a condição de parada i < numberArray->size do laço for é executada **n+1** vezes. Já o incremento i++ do laço for e o if dentro de seu escopo são executados **n** vezes, o que nos dá: $(n+1)c + n(a+c) = nc + c + na + nc$.

Se considerarmos que comparações e operações aritméticas têm o mesmo custo de processamento = 1, então $f(n) = 3n + 1$.

O gráfico abaixo corrobora a tendência de crescimento linear:

Figura 1: Gráfico Tempo de Execução x Tamanho de Array - Busca Sequencial



3. Busca Binária Iterativa

```
int iterativeBinarySearch(NUMBER_ARRAY *numberArray, int key){
	int half = 0, max = numberArray->size -1, min = 0;
	while(min <= max){  // 1 comp
		half = (min + max) / 2;  // 2 op
		if(numberArray->elements[half] == key) // 1 comp
			return half;
		else if(numberArray->elements[half] > key) // 1 comp
			max = half - 1; //1 op
		else
			min = half + 1; //1 op
	}
   return -1;
}
```

Completada a atribuição de valores iniciais às variáveis half, max e min, é feita uma comparação na condição de parada do laço while. Após isso, o escopo do mesmo e sua condição de parada são executados até que a chave buscada seja encontrada ou até que o índice de início do vetor seja maior que o índice de seu fim.

Assim, tomando **n** como numberArray->size, para o caso em que a chave buscada não esteja no vetor, o processo é realizado $n\ /\ 2\ /\ 2\ /\ 2\ /\ 2\ \ ...\ /\ 2$ vezes, até que min passe a ser maior que max, o que corresponde à $\log_2 n$ repetições.

Dessa forma, temos $1 + \log_2 n\ (c + 2o + c + c + o)$ operações.

Considerando que comparações e operações aritméticas tenham o mesmo custo = 1, $f\,(n) = 1 + 6\log_2\ n$.

4. Busca Binária Recursiva

```
int recursiveBinarySearch(NUMBER_ARRAY *numberArray, int key, int min, int max){
	if(min > max)  // 1 comp
		return -1;
	int half = (min + max) / 2;  // 2 op
	if(numberArray->elements[half] == key)  // 1 comp
		return half;
	else if(numberArray->elements[half] > key)  // 1 comp
		return recursiveBinarySearch(numberArray, key, min, half-1);  // 1 op
	else
		return recursiveBinarySearch(numberArray, key, half+1, max);  // 1 op
}
```

A chamada recursiva da função é feita até se chegar ao caso base dessa recursão, que ocorre quando o vetor não precisa mais ser "dividido" e sua chave buscada é retornada, ou seja, $f(1) = 1$.

Considerando $f(n) = 1c + 2o + 1c + 1c + 1o + f(\frac{n}{2})$, temos que:

$$f(n) = 3c + 3o + f\left(\frac{n}{2}\right)$$

$$f\left(\frac{n}{2}\right) = 3c + 3o + f\left(\frac{n}{4}\right)$$

$$f\left(\frac{n}{4}\right) = 3c + 3o + f\left(\frac{n}{8}\right)$$

Assim, de forma genérica, na k-ésima chamada da função teremos $k(3c + 3o) + f\left(\frac{n}{2^k}\right)$.

Igualando essa expressão ao caso base, chegamos em: $n = 2^k$, e em $k = \log_2 n$. Portanto, $f(n) = \log_2 n(3c + 3o) + 1$.

Considerando que comparações e operações aritméticas tenham o mesmo custo = 1, $f(n) = 6\log_2 n + 1$.

Não são notáveis, pelos gráficos, desempenhos diferentes entre a busca binária iterativa e a recursiva:

Figura 2: Gráfico Tempo de Execução x Tamanho de Array - Busca Binária Iterativa

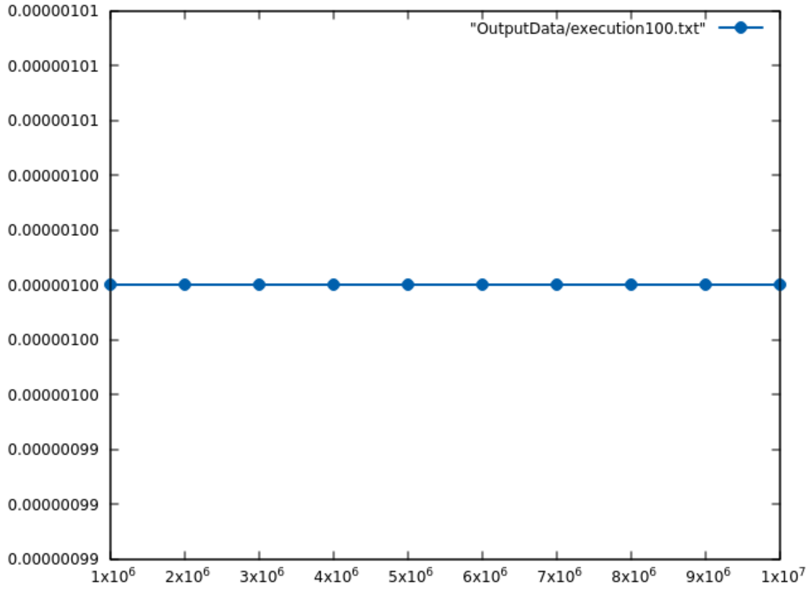


Figura 3: Gráfico Tempo de Execução x Tamanho de Array - Busca Binária Recursiva