Algoritmos e Estrutura de Dados I

Contagem de operações e de tempo de execução.
A complexidade dos Algoritmos.

Prof. Leonardo Tortoro Pereira
Luís Roberto Piva
nº USP: 13687727

Instituto de Ciências Matemáticas e Computação -
ICMC-USP
USP-Campus São Carlos

INTRODUÇÃO

Nesta atividade nos foi disponibilizado um código base para a implementação de funções diferentes para que analisássemos o tempo de execução de cada uma, e com isso termos ideia sobre a complexidade de alguns algoritmos.

No código fonte um vetor é gerado com um tanto de posições que nos permita comparar a complexidade das funções implementadas. Este vetor é então submetido à análise de tempo baseado na complexidade de quatro funções e em número significativo de execuções.

Os números de execuções fixamos como sendo 100, e as quatro funções são:

1. Inversão do vetor;
2. Busca Sequencial no vetor;
3. Busca Binária no vetor;
4. Busca Binária recursiva no vetor.

Após a contagem do tempo, deveríamos obter os gráficos de tais operações e observar a complexidade dos algoritmos. Para tanto segue as análises feitas.

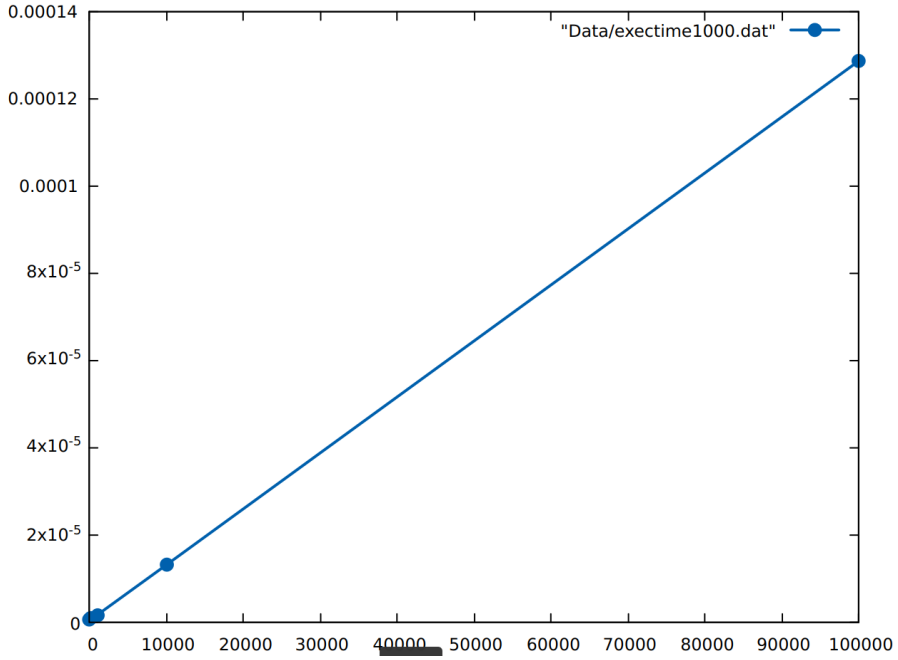
1. INVERSÃO DO VETOR

A função de inverter um vetor de n posições por inteiro foi necessário apenas n/2 operações, ou seja, se o tamanho do vetor era de 10 posições, por exemplo, apenas 5 operações foram necessárias para inverter o vetor completamente.

Por ser poucas operações, o tempo de execução é muito pequeno. Obviamente, para os casos em que os vetores eram colossais, como 100.000, o tempo era um pouco maior. Rodando os casos em que os vetores possuem tamanho, 10, 100, 1000, 10000, 100000 e 1000000, obtivemos os seguintes resultados:

```
1 10 0.000000810
2 100 0.000001630
3 1000 0.000007330
4 10000 0.000031770
5 100000 0.000159880
6 1000000 0.001308220
```

Desses resultados podemos criar o seguinte gráfico para a função definida para este caso:



A função definida para esse gráfico é uma função linear, logo uma função de complexidade **O(n)**, ou, como no caso, O(n/2).

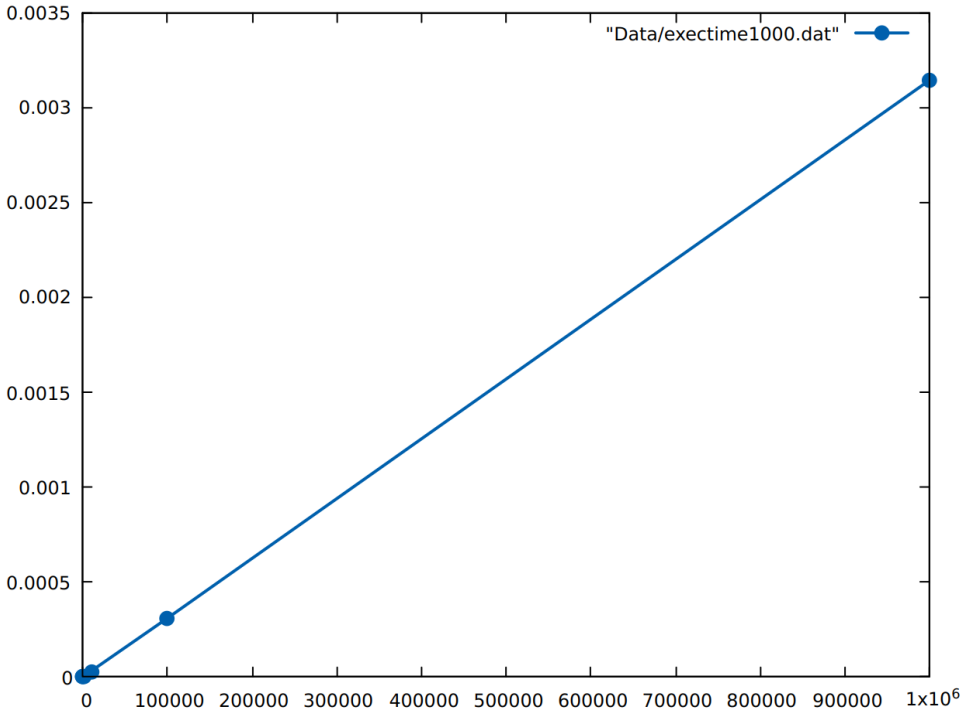
2. SEQUENCIAL

A pesquisa sequencial em um vetor consiste em percorrer o vetor desde a primeira posição até a última. Para cada posição i, comparamos o valor de vetor[i] com o valor que se pede.

Neste caso, para o pior caso aqui usaremos um valor que não exista no vetor para que ele pesquise até o final. Rodando os casos em que os vetores possuem tamanho, 10, 100, 1000, 10000 e 100000, obtivemos os seguintes resultados:

```
1 10 0.000000860
2 100 0.000001840
3 1000 0.000009430
4 10000 0.000059670
5 100000 0.000433120
```

Desses resultados podemos criar o seguinte gráfico para a função definida para este caso:



Analisando os dados do gráfico, da tabela e do código usado para a busca sequencial fica claro que a função é linear, uma vez que, no pior caso, é necessário apenas “n” comparações, onde “n” é o tamanho do vetor. Logo o problema da busca sequencial é linear e resolve o problema (achar a posição do vetor usado em que é igual ao número ou chave pesquisado) em **O(n),** no pior caso.

3. BINÁRIA

Se o nosso mesmo vetor for ordenado, é mais eficiente procurar um item com o auxílio da Busca Binária Iterativa. Esta consiste, em suma, em verificar se o item requerido é igual ao valor da posição do meio do vetor. Caso seja igual, devolvemos esse valor. Caso o valor seja maior ou menor, dividimos o vetor ao meio, e comparamos de novo, e de novo. Até acharmos o valor requerido, ou acabar as posições do vetor, o que significa que o valor não está no vetor.

Note que o melhor caso seria se o item procurado estivesse no meio do vetor. Mas como escolhemos o pior caso colocaremos o item procurado

como um que não pertença ao vetor (poderíamos ter colocado também algum de deus extremos, isto é, início ou fim).

Dessa maneira, matematicamente temos que esse problema é um problema baseado no logaritmo de 2, pois cada comparação reduz o número de possíveis posições por um fator igual a 2. Assim sendo, matematicamente temos que, no pior caso, o número de acesso é igual a **log2 n.**

Para o nosso problema, rodando os casos em que os vetores possuem tamanho, 10, 100, 1000, 10000 e 100000, obtivemos os seguintes resultados:

```
1 10 0.000000930
2 100 0.000000910
3 1000 0.000000930
4 10000 0.000001100
5 100000 0.000001030
6 1000000 0.000000450
```

Infelizmente, para este problema o gráfico não ficou “como deveria ficar”, eu tentei de tudo e engordei meu terminal de tanto “make plot”. Mas ficarei atento para a sua correção depois para ver onde errei.

4. RECURSIVA

A busca binária recursiva usa a mesma lógica de ação que a Binária Iterativa, o que muda é a função responsável pelo problema chamar a si própria, fazendo com que a cada iteração a recursividade reduz o espaço de busca através das variáveis de início e fim dos sub-vetor.

Para o nosso problema, rodando os casos em que os vetores possuem tamanho, 10, 100, 1000, 10000 e 100000, obtivemos os seguintes resultados:

```
1 10 0.000001050
2 100 0.000001020
3 1000 0.000001010
4 10000 0.000001070
5 100000 0.000001040
6 1000000 0.000001520
```

A abordagem da função de busca binária recursiva resulta em tempo de execução **O(log n)**, onde “n” é o tamanho do vetor. Assim como a busca binária iterativa, houve algum problema na geração do gráfico, e fico à espera da correção.