Exercício 3

SCC0223 -
Estrutura de Dados I

Universidade de São Paulo (USP)

Aluno: Caio Assumpção Rezzadori

Nº USP: 11810481

Enunciado

Foi pedido a implementação de 4 programas:

1. Inversão de vetor;
2. Busca sequencial;
3. Busca binária iterativa;
4. Busca binária recursiva;

e a análise da contagem de operações/complexidade no pior caso de seus algoritmos. Além disso, foi pedido a medição do tempo para diferentes tamanhos do parâmetro vetor de cada programa pela média de 100 execuções para cada tamanho analisado.

Observações

- Os tamanhos dos vetores analisados em todos os algoritmos foram de 10, 100, 1.000, 5.000 e 10.000 elementos;
- Tais vetores foram disponibilizados em aula e no enunciado do exercício;
- Para a análise assintótica dos algoritmos no pior caso, foi utilizada a notação **Big O**;
- O código foi escrito em linguagem *C*;
- Os gráficos foram gerados pelo programa *gnuplot*;
- A compilação dos programas e a execução do *gnuplot* foram feitas no terminal do *Windows Subsystem for Linux (WSL)* com o auxílio de códigos passados em sala de aula;
- Todos os algoritmos utilizaram a estrutura (*struct*) definida abaixo, a qual foi declarada no arquivo *NumberVector.h*, também feita em aula. Ela recebe o tamanho do vetor, e o próprio vetor com seus elementos.

```
typedef struct _numberVector_t
{
    int* numbers;
    int vectorSize;
} numberVector_t;
```

1 Inversão de vetor

Algoritmo/Implementação

```
1  numberVector_t inverterVetor(numberVector_t numberVector){
2      numberVector_t vetorInvertido = numberVector;
3      int N = numberVector.vectorSize;
4      int aux;
5      for(int i = 0; i < N/2; ++i){
6          aux = vetorInvertido.numbers[i];
7          vetorInvertido.numbers[i] =
8          vetorInvertido.numbers[N - 1 - i];
9          vetorInvertido.numbers[N - 1 - i] = aux;
10     }
11
12     return vetorInvertido;
13 }
```

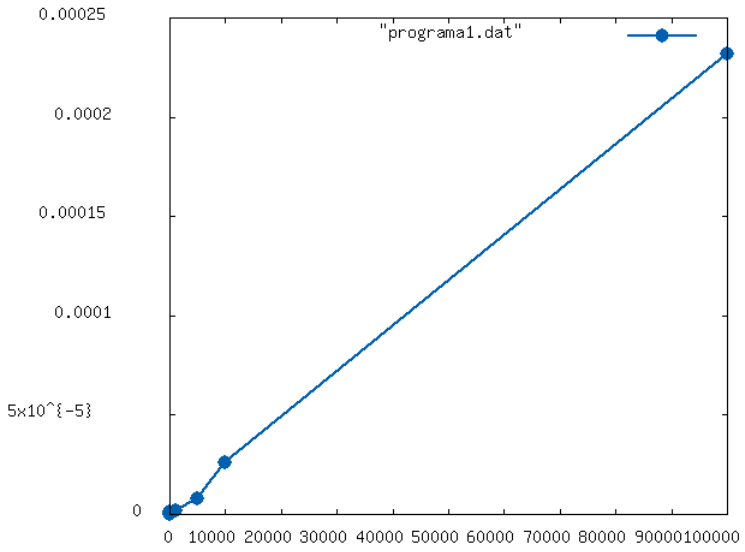
O código apresentado possui 2 atribuições (linhas 3 e 4) antes do *loop*, possui 1 atribuição, 1 comparação, e 1 divisão no início do *loop* (linha 5), e durante as n execuções, há 4 atribuições (linhas 5, 6, 7-8, 9). Portanto:

$$O(inverterVetor(n)) = O(2 + 1 + 1 + 1 + 4n) = O(5 + 4n) = O(n)$$

Tabela de valores

Tamanho vetor	Tempo
10	0.000000
100	0.000001
1000	0.000002
5000	0.000008
10000	0.000026
100000	0.000232

Gráfico



2 Busca sequencial

O pior caso foi estimado procurando o número -32.000, que normalmente é definido como saída de erro em códigos que retornam números inteiros. Aqui há a suposição que tal elemento não se encontra no vetor.

Algoritmo/Implementação

```
1 int buscaSequencial(numberVector_t numberVector, int
      elementoProcurado){
2     int N = numberVector.vectorSize;
3     for(int i = 0; i < N; i++){
4         if(numberVector.numbers[i] == elementoProcurado){
5             return i;
6         }
7     }
8     return -1; //Erro na busca: nada foi encontrado
9 }
```

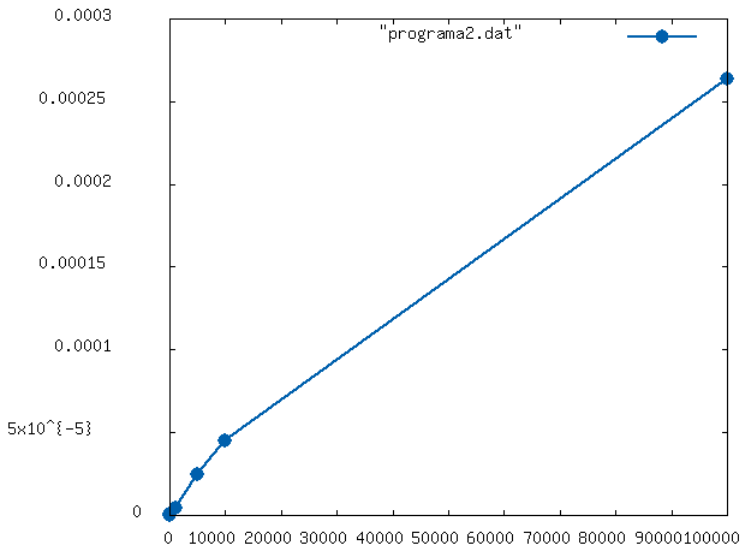
O código apresentado possui 1 atribuição antes do *loop* (linha 2), 1 atribuição e 1 comparação no início do *loop* (linha 2), e 1 comparação durante as n execuções do programa. Portanto:

$$O(buscaSequencial(n)) = O(1 + 1 + 1 + n) = O(3 + n) = O(n)$$

Tabela de valores

Tamanho vetor	Tempo
10	0.000000
100	0.000001
1000	0.000004
5000	0.000025
10000	0.000045
100000	0.000264

Gráfico



3 Busca binária

Os algoritmos de busca binária, como será visto nas próximas subseções do relatório, possuem custos que crescem muito devagar, pois são dominados pela função logarítmica. Dito isso, para ser possível medir adequadamente o tempo de execução, o tempo foi medido em micro segundos. Ou seja, multiplicou-se o tempo medido por 10^6.

Além disso, a média do tempo foi feita sobre 10.000 execuções, uma vez que para 100 e 1.000 execuções, tal medição ficou mais instável e não foi possível ter uma boa ideia do comportamento do custo do algoritmo. Isso é consequência também do custo computacional ser muito pequeno.

O pior caso do algoritmo foi estimado tomando o penúltimo elemento do vetor.

Por fim, foram utilizadas funções disponibilizadas durante as aulas para ordenar o vetor, uma condição necessária para aplicar a busca binária. Tal operação não foi contabilizada no tempo de execução, pois está fora do escopo da análise. Os códigos encontram-se logo abaixo.

```c
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```c
void bubbleSort(int *v, int n){
    if (n < 1)return;

    for (int i=0; i<n; i++)
        if (v[i] > v[i+1])
            swap(&v[i], &v[i+1]);
    bubbleSort(v, n-1);
}
```

3.1 Iterativa

Algoritmo/Implementação

```c
int buscaBinariaIterativa(numberVector_t vetorOrdenado, int
    elementoProcurado){
    int N = vetorOrdenado.vectorSize;
    int L = 0;
    int R = N - 1;
    int M;
    while(L <= R){
        M = (L + R)/2;
        if(vetorOrdenado.numbers[M] > elementoProcurado){
            R = M - 1;
        }
        else if(vetorOrdenado.numbers[M] < elementoProcurado){
            L = M + 1;
        }
        else{
            return M; //Elemento encontrado
        }
    }
    return -1; //Erro na busca: nada foi encontrado
}
```

Consideremos $L^{(k)}$ a k-ésima iteração do algoritmo acima e $N-1$ índice do último elemento do vetor.

Para $L^{(k)}$ chegar no penúltimo elemento do vetor ordenado, deve-se ter a sequência de iterandos dentro do *loop*:

$$L^{(0)} \leftarrow 0$$
$$R \leftarrow N - 1$$

Inicio do loop

$$L^{(1)} \leftarrow \frac{L^{(0)} + R}{2} = \frac{1}{2}(N-1)$$
$$L^{(2)} \leftarrow \frac{L^{(1)} + R}{2} = \frac{3}{4}(N-1)$$
$$L^{(3)} \leftarrow \frac{L^{(2)} + R}{2} = \frac{7}{8}(N-1)$$
$$\vdots$$
$$L^{(k)} \leftarrow \frac{L^{(k-1)} + R}{2} = \frac{2^k - 1}{2^k}(N-1)$$

Dito isso, devemos encontrar k tal que $L^{(k)} = N - 2$:

$$N - 2 = \frac{2^k - 1}{2^k}(N-1) = (1 - \frac{1}{2^k})(N-1)$$
$$\Rightarrow -\frac{1}{2^k}(N-1) = -1$$
$$\Rightarrow 2^k = N - 1 \Rightarrow k \log 2 = \log(N-1) < \log N$$

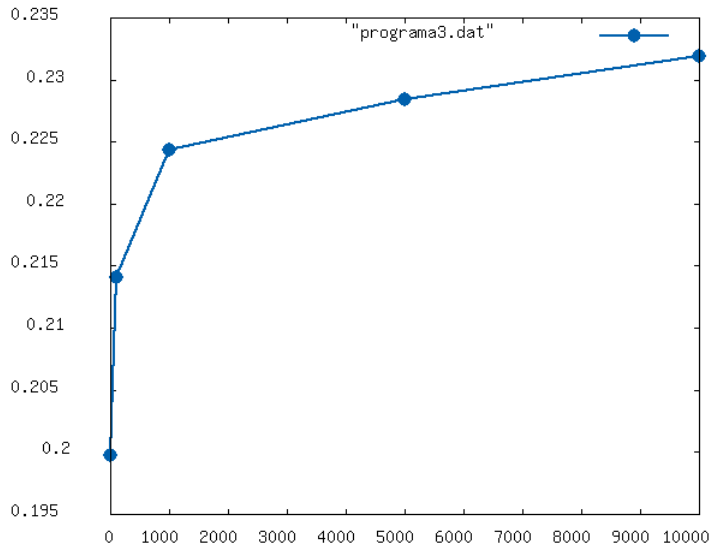
Como todas as outras operações no algoritmo são constantes (atribuição ou comparação), então tem-se que o algoritmo é dominado pelo número de iterações no *loop*. Portanto:

$$O(buscaBinariaIterativa(n)) = O(\log n)$$

Tabela de valores

Tamanho vetor	Tempo
10	0.199700
100	0.214100
1000	0.224300
5000	0.228400
10000	0.231900

Gráfico



3.2 Recursiva

Algoritmo/Implementação

```
int buscaBinariaRecursiva(numberVector_t vetorOrdenado, int
    elementoProcurado, int inicio, int fim){
    if(inicio <= fim){
        int meio =(inicio + fim)/2;

        if(vetorOrdenado.numbers[meio] > elementoProcurado){
          return buscaBinariaRecursiva(vetorOrdenado,
    elementoProcurado, inicio, meio - 1);
        }

        else if(vetorOrdenado.numbers[meio] < elementoProcurado){
            return buscaBinariaRecursiva(vetorOrdenado,
    elementoProcurado, meio + 1, fim);
        }
        else{
            return meio; //Elemento encontrado
        }
    }
    return -1; //Erro na busca: nada foi encontrado
}
```

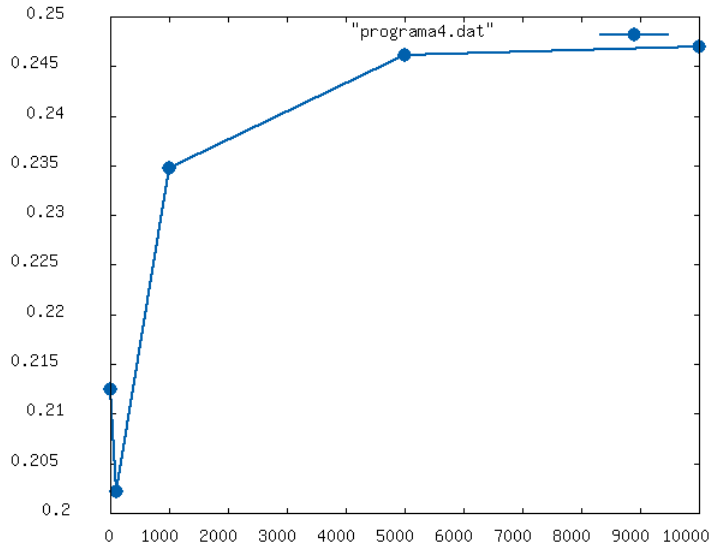
A complexidade do algoritmo acima pode ser demonstrada de forma análoga à busca binária iterativa, uma vez que a única coisa que muda é a forma de fazer o *loop* das divisões de índices. Todavia, a quantidade de vezes de divisões deve ser a mesma que a anterior, uma vez que o princípio/ideia na procura do elemento no vetor é o mesmo do algoritmo anterior. Portanto:

$$O(buscaBinariaRecursiva(n)) = O(\log n)$$

Tabela de valores

Tamanho vetor	Tempo
10	0.212500
100	0.202200
1000	0.234700
5000	0.246100
10000	0.247000

Gráfico



4 Considerações finais

Embora os gráficos dos algoritmos indique um comportamento semelhante ao que foi previsto pela análise assintótica, vale ressaltar que tais resultados alcançados foram decorrentes da geração de múltiplas medidas de tempo, até obter-se um conjunto de dados com comportamento semelhante ao esperado nas análises assintóticas dos algoritmos.

Não há problema fazer isso, uma vez que os algoritmos apenas são dominados pela função. Isso não significa que eles se comportarão sempre de forma semelhante à ela, mas sim, que não se espera que eles cresçam mais rápido que tais funções conforme o tamanho de operações aumenta.

O algoritmo não se comporta de forma idêntica à analise matemática pois ela exige simplificações, além do programa estar sujeito à erros numéricos de arredondamento ou erros aleatórios.