# DOCUMENTAÇÃO TP2

Ramon Gonçalves Gonze

## INTRODUÇÃO

O objetivo deste trabalho prático é analisar o comportamento dos algoritmos de ordenação apresentados em sala de aula. São utilizados oito algoritmos de ordenação, sendo sete que ordenam por *comparação* e um que ordena por *distribuição*. O trabalho foi desenvolvido para testar diferentes tipos de entrada, ou seja, vetores de diferentes tipos que precisam ser ordenados. Para gerar o executável do arquivo, deve-se abrir o prompt de comando (Windows) ou o terminal (Linux), acessar a pasta que contém os 3 arquivos do tp, *ordena.c, algoritmos.c* e *algoritmos.h*, e executar o comando **make**, que gerará um arquivo executável através de um *Makefile*.

## IMPLEMENTAÇÃO

### ❑ Estrutura de dados

Para a implementação do trabalho, foram utilizados os tipos de dados *TipoIndice* e *TipoChave*, ambos do tipo *long,* que representam, respectivamente, os indices do vetor que será ordenado na execução, e as chaves de cada item do vetor.

O único Tibo Abstrato de Dado (TAD) criado é o *TipoItem*, que possui um elemento do tipo *TipoChave*, ou seja, cada elemento *TipoItem* possui uma chave de identificação, a variável **chave**. O vetor de elementos que será utilizado como teste para os algoritmos de ordenação, é um vetor do tipo *TipoItem*.

### ❑ Funções e procedimentos

Os algoritmos de ordenação utilizados foram:

- Seleção
- Inserção
- Bolha
- Shellsort
- Quicksort
- Heapsort

- Mergesort
- Radixsort

Todos os algoritmos, com exceção do *Radixsort*, são algoritmos que ordenam por comparações. O *Radixsort* é um algoritmo que ordena por distribuição, e seu desempenho e calculado não pela quantidade de comparações, mas sim pela quantidade de dígitos que a maior chave do elemento de um vetor possui.

A execução do programa é feita via linha de comando, na qual será indicado qual o algoritmo de ordenação a ser testado, o tamanho do vetor, a situação inicial dos elementos do vetor e por último, a opção de visualizar o vetor antes e depois de seu ordenamento. Estrutura de execução:

**./ordena <algoritmo> <quantidade_de_elementos> <situação_inicial_do_vetor> <p>**

As tabelas abaixo indicam os possíveis comandos para a execução:

| Algoritmo | Comando |
|---|---|
| Seleção | sel |
| Inserção | ins |
| Bolha | bol |
| Shellsort | she |
| Quicksort | qui |
| Heapsort | hea |
| Mergesort | mer |
| Radixsort | rad |

| Situação inicial do vetor | Comando |
|---|---|
| Ordenado de forma crescente | c |
| Ordenado de forma decrescente | d |
| Quase ordenado | q |
| Aleatório | a |

Exemplo de comando:

- Ordenação de um vetor de 20 elementos, ordenado inicialmente de forma decrescente e utilizando o algoritmo *Quicksort*: **./ordena qui 20 d**

O último parâmetro da linha de comando – o comando **p** – é opcional. Ele deve ser utilizado caso seja necessário imprimir o vetor antes e depois de seu ordenamento. A impressão dos elementos é feita na saída padrão (stdout). Para execuções com vetores aleatórios, são gerados 10 vetores, e é calculado a média do tempo de execução, quantidade de comparações e movimentações destes, e caso haja o comando **p**, é impresso somente o primeiro vetor criado.

Exemplo de comando:

- Ordenação de um vetor de 1000 elementos, ordenado inicialmente de forma aleatória, utilizando o algoritmo de *Inserção* e que seja impresso o vetor: **./ordena ins 1000 a p**

❑ **Análise de complexidade**

A análise será feita sob o número de elementos do vetor, e terá ênfase no número de comparações e movimentações.

**void criaVetor(TipoItem *vetor, TipoIndice tam_vetor, char tipo_ordenacao)**

Essa função recebe o terceiro parâmetro passado na linha de comando. Há um *switch* que tem 4 casos. Em cada um, há um laço for que vai de *1* até *tam_vetor*, com exceção do caso *'q'*. Neste, há dois laços de repetição, um que vai de *1* até *(9*tam_vetor)/10* e o outro de *(9*tam_vetor)/10* até *tam_vetor*, o que, ambos somados no final, percorrem também *tam_vetor* vezes. Portanto, a ordem de complexidade é $O(n)$.

**void Selecao(TipoItem *A, TipoIndice n, long *comparacoes, long *movimentacoes)**

- Comparações:
  $$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = O(n^2)$$

- Movimentações:
  $$3(n-1) = O(n)$$

**void Insercao(TipoItem *A, TipoIndice n, long *comparacoes, long *movimentacoes)**

- Comparações:
  Anel interno (while):
  - ➔ Melhor caso: É executado somente uma vez.
  - ➔ Pior caso: É executado $i$ vezes.
  - ➔ Caso médio:
    $$\frac{1}{i}\sum_{j=1}^{i} j = \frac{1}{i}\left(\frac{i(i+1)}{2}\right) = \frac{i+1}{2}$$

  Anel externo (for):
  - ➔ Melhor caso:
    $$\sum_{i=2}^{n} 1 = n-1 = O(n)$$
  - ➔ Pior caso:
    $$\sum_{i=2}^{n} i = \frac{n^2}{2} + \frac{n}{2} - 1 = O(n^2)$$
  - ➔ Caso médio:

$$\sum_{i=2}^{n} \frac{i+1}{2} = \frac{n^2}{4} + \frac{3n}{4} - 1 = O(n^2)$$

- Movimentações:
  - ➔ Melhor caso:
    $$\sum_{i=2}^{n} 3 = 3(n-1) = O(n)$$
  - ➔ Pior caso:
    $$\sum_{i=2}^{n} i + 2 = \frac{n^2}{2} + \frac{5n}{2} - 3 = O(n^2)$$
  - ➔ Caso médio:
    $$\sum_{i=2}^{n} (\frac{i+1}{2} + 2) = \frac{n^2}{4} + \frac{11n}{4} - 3 = O(n^2)$$

**void Bolha(TipoItem *A, TipoIndice n, long *comparacoes, long *movimentacoes)**

- Comparações:
  $$\sum_{i=1}^{n-1} i = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

- Movimentações:
  - ➔ Pior caso: $3 \times \frac{n^2}{2} - \frac{n}{2} = O(n^2)$ (sempre entra no if):
  - ➔ Melhor caso: $0$ movimentações (nunca entra no if):

**void Shellsort(TipoItem *A, TipoIndice n, long *comparacoes, long *movimentacoes)**

Para esse algoritmo de ordenação, ainda não foi possível calcular a sua complexidade, por conta de problemas matemáticos complexos. Há duas conjecturas de que sua ordem de complexidade seja $O(n^{1,25})$ ou $O(n(ln\ n)^2)$.

**void Quicksort(TipoItem *A, TipoIndice n, long *comparacoes, long *movimentacoes)**
**void Ordena(TipoIndice Esq, TipoIndice Dir, TipoItem *A, long *comparacoes, long *movimentacoes)**
**void Particao(TipoIndice Esq, TipoIndice Dir, TipoIndice *i, TipoIndice *j, TipoItem *A, long *comparacoes, long *movimentacoes)**

- Comparações:
  - ➔ Melhor caso: Acontece quando o pivô escolhido é o elemento central do vetor analisado.
    Particao(Esq, Dir, &i, &j, A, comparacoes, movimentacoes); → **n comparações**
    if (Esq < j) Ordena(Esq, j, A, comparacoes, movimentacoes); → **n/2 comparações**
    if (i < Dir) Ordena(i, Dir, A, comparacoes, movimentacoes); → **n/2 comparações**
    Complexidade $C(n) = 2C(\frac{n}{2}) + n = n\ logn\ -\ n + 1 =\ O(n\ logn)$
  - ➔ Pior caso: Acontece quando o pivô escolhido é o maior ou menor elemento do vetor analisado.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} = O(n^2)$$

➔ Caso médio: Acontece quando o pivô escolhido está em qualquer posição do vetor analisado.

Complexidade $C(n) = n + 1 + \frac{1}{n}\sum_{i=1}^{n} C(i-1) + C(n-i)$

$$= n + 1 + \frac{2}{n}\sum_{i=1}^{n} C(i-1) \approx 2n\ ln\ n\ \approx 1,39n\ logn$$

$$= O(n\ logn)$$

- Movimentações:

$$\frac{1}{3} n\ ln\ n \approx 0,23n\ logn = O(n\ logn)$$

**void Heapsort(TipoItem *A, TipoIndice n, long *comparacoes, long *movimentacoes)**
**void Constroi(TipoItem *A, TipoIndice n, long *comparacoes, long *movimentacoes)**
**void Refaz(TipoIndice Esq, TipoIndice Dir, TipoItem *A, long *comparacoes, long *movimentacoes)**

Esse algoritmo de ordenação possui a complexidade $O(n\ logn)$ para todos os casos possíveis. A função *Refaz* possui complexidade $O(logn)$. A função *Heapsort* possui complexidade $O(n\ logn)$. Já a complexidade da função *Constroi*, pode ser calculada como $O(n) \times Refaz$.

**void Mergesort(TipoItem *A, TipoIndice inicio, TipoIndice fim, long *comparacoes, long *movimentacoes)**

Para analisarmos sua complexidade, vejamos a equação de recursividade da função *Heapsort*:

Para $n = 1$, $C(1) = 1$

Para $n > 1$, executa-se recursivamente:

Uma vez $piso(\frac{n}{2})$ com elementos;

Outra vez com os $teto(\frac{n}{2})$ elementos restantes;

Merge, que executa $n$ operações $C(n) = 2C(\frac{n}{2}) + n$

Complexidade: $O(log\ n)$

Função *Merge*:

Cada chamada une um total de no máximo $n$ elementos (tamanho do vetor inteiro)

Uma comparação a cada iteração

Total de passos do primeiro while: não mais que **n**

do outro, no pior caso, **n/2**

A cópia de A para B (vetor menor para o vetor maior): **n** passos

Complexidade final: $O(n)$

Portanto, a complexidade de todo o algoritmo será $O(n\ logn)$ .

**void Radixsort(TipoItem * A, TipoIndice tamanho, long *comparacoes, long *movimentacoes)**
**TipoIndice MaiorNumero(TipoItem *A, TipoIndice tamanho)**

Não é feita nenhumacomparação nesse algoritmo. Sua complexidade depende do número de dígitos da maior chave de um elemento do vetor. A função *MaiorNumero* possui um laço que vai de $i = 1$ até $tamanho$, que é o total de elementos do vetor, logo sua complexidade é $O(n)$. Já a função *Radixsort* dependerá da quantidade de dígitos do número retornado pela função *MaiorNumero*. Temos que dados

$n$ números com $d$ dígitos, e sendo $k$ o valor máximo para os números, o algoritmo ordenará os elemento em $\Theta(d(n+k))$. Se $k = O(n)$ e $d$ for pequeno, sua complexidade final será $O(n)$.

**Programa principal**

A função main só executará um laço de repetição caso o estado inicial do vetor escolhido seja aleatório. Para este, ela irá executar o laço **for(i = 0; i < aux; i++)**, seja qual o for o algoritmo de ordenação escolhido. Este laço é utilizado para calcular a média do tempo de execução, número de comparações e movimentações de 10 vetores aleatórios diferentes. Como **aux** só assumirá os valores 1 (caso o estado inicial não seja aleatório) ou 10 (caso seja aleatório), a complexidade final será de $O(1)$.

## TESTES

Foram feitos testes utilizando vetores de 100, 1000, 10000 e 100000 elementos e ordenados de formas diferentes. Os testes foram feitos em um core i3, com 6GB de memória. Abaixo seguem tabelas que fazem a comparação do tempo de execução de cada algoritmo:

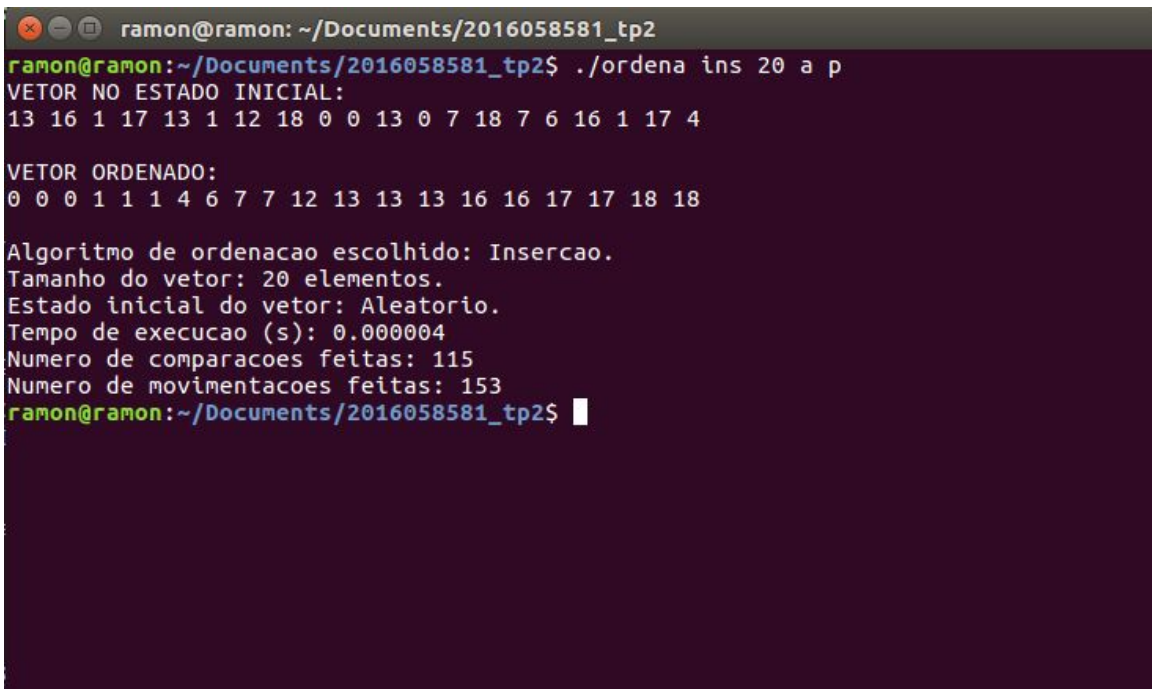
| Tempo de execução (segundos) – 100 elementos | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Estado inicial do vetor | Seleção | Inserção | Bolha | Shellsort | Quicksort | Heapsort | Mergesort | Radixsort |
| Ordenado | 0.000046 | 0.000002 | 0.000053 | 0.000005 | 0.000008 | 0.000033 | 0.000033 | 0.000042 |
| Inversamente ordenado | 0.000059 | 0.000063 | 0.000114 | 0.000017 | 0.000019 | 0.000030 | 0.000055 | 0.000052 |
| Quase ordenado | 0.000057 | 0.000008 | 0.000054 | 0.000020 | 0.000031 | 0.000033 | 0.000052 | 0.000033 |
| Aleatório | 0.000070 | 0.000043 | 0.000130 | 0.000030 | 0.000039 | 0.000035 | 0.000062 | 0.000026 |

| Tempo de execução (segundos) – 1000 elementos | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Estado inicial do vetor | Seleção | Inserção | Bolha | Shellsort | Quicksort | Heapsort | Mergesort | Radixsort |
| Ordenado | 0.005398 | 0.000020 | 0.002530 | 0.000095 | 0.000100 | 0.000202 | 0.000499 | 0.000588 |
| Inversamente ordenado | 0.005402 | 0.005765 | 0.005679 | 0.000172 | 0.000063 | 0.000482 | 0.000435 | 0.000225 |
| Quase ordenado | 0.005591 | 0.000327 | 0.005294 | 0.000338 | 0.000324 | 0.000431 | 0.000696 | 0.000507 |
| Aleatório | 0.004559 | 0.003603 | 0.008039 | 0.000525 | 0.000501 | 0.000510 | 0.000798 | 0.000485 |

| Tempo de execução em segundos – 10000 elementos | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Estado inicial do vetor | Seleção | Inserção | Bolha | Shellsort | Quicksort | Heapsort | Mergesort | Radixsort |
| Ordenado | 0.204388 | 0.000114 | 0.214155 | 0.001231 | 0.001802 | 0.004828 | 0.004563 | 0.005418 |
| Inversamente ordenado | 0.243553 | 0.273925 | 0.415654 | 0.002199 | 0.002131 | 0.002380 | 0.004097 | 0.003591 |
| Quase ordenado | 0.233027 | 0.042765 | 0.251852 | 0.004856 | 0.004352 | 0.002464 | 0.004874 | 0.005982 |
| Aleatório | 0.200165 | 0.159825 | 0.494481 | 0.005830 | 0.004778 | 0.005500 | 0.007094 | 0.005286 |

| Tempo de execução em segundos – 100000 elementos | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Estado inicial do vetor | Seleção | Inserção | Bolha | Shellsort | Quicksort | Heapsort | Mergesort | Radixsort |
| Ordenado | 21.987824 | 0.001219 | 18.498937 | 0.011778 | 0.020254 | 0.045071 | 0.052136 | 0.065400 |
| Inversamente ordenado | 21.275939 | 22.894718 | 40.715057 | 0.025565 | 0.020794 | 0.027547 | 0.047272 | 0.051930 |
| Quase ordenado | 20.481872 | 1.850133 | 33.620581 | 0.062201 | 0.026013 | 0.050307 | 0.053938 | 0.034386 |
| Aleatório | 21.226889 | 19.000693 | 66.816116 | 0.051205 | 0.031398 | 0.037267 | 0.047471 | 0.034735 |

Exemplo de execução:



## CONCLUSÃO

O trabalho foi feito baseado no conteúdo discutido em sala e nos slides fornecidos pelo professor. A aplicação de técnicas e métodos foram utilizadas sem grandes problemas. Foi possível notar que diferentes algoritmos de ordenação são mais efetivos para diferentes situações. O objetivo proposto pelo enunciado foi atingido.

## BIBLIOGRAFIA


UFF. **Algoritmos de ordenação**. Disponível em: <http://www2.ic.uff.br/~boeres/slides_ed/ed4.pdf>. Acesso em: 08 dez. 2016.

WALTERS, G. Austin. **Radix Sort in C**. Disponível em: <http://austingwalters.com/radix-sort-in-c/>. Acesso em: 08 dez. 2016.

FURTADO, Kevi. **Tempo de Operação em C**. Disponível em: <https://www.youtube.com/watch?v=AQm9gYERUhk>. Acesso em: 07 dez. 2016