

UNIP – UNIVERSIDADE PAULISTA

Curso de Engenharia da Computação

**ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS**

**AS TECNICAS DE ORDENAÇÃO DE ALGORITMO, CONCEITOS, USOS E APLICAÇÕES**

Igor Gabriel Mariano Mendonça - RA – D71386-8

Yan Gabriel Furlan - RA – N30662-9

São José dos Campos, 10 de outubro de 2020.

UNIP – UNIVERSIDADE PAULISTA

Curso de Engenharia da Computação

**ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS**

**AS TECNICAS DE ORDENAÇÃO DE ALGORITMO, CONCEITOS, USOS E APLICAÇÕES**

Atividades Práticas Supervisionadas do 5º/6º Semestres do Curso de Engenharia da Computação da **Universidade Paulista – UNIP**.

Coordenador: Prof. Élcio H. Shiguemori

Prof Responsável: Luiz Gustavo Miranda Pinto

São José dos Campos, 10 de outubro 2020.

Resumo

O uso do algoritmo de ordenação em programação é um método que coloca os elementos de uma dada sequência em determinada ordem. Efetuando sua ordenação completa ou parcial. Ordens mais usadas são as numéricas e as lexicográficas (Conversão de uma sequência de caracteres em uma sequência de tokens” strings”). Existem diversas razões para se ordenador uma sequência. Uma delas é a possibilidade de acessar seus dados de maneira mais eficiente. No desenvolvimento de tais ordenações existem diversos modos, como: Selection Sort, Insertion Sort, MergeSort. Classificado como métodos simples SelectSort e InsertSort; e um dos métodos sofisticados: MergeSort.

Palavras-chave: Ordenação, algoritmo e sequência.

ABSTRACT

The use of the ordering algorithm in programming is a method that places the elements of a data sequence in the order provided. Performing your full or partial ordering. Most used orders are numeric and lexicographic (Conversion of a sequence of characters into a sequence of tokens "strings"). There are several reasons for ordering a sequence. One is the ability to access your data more efficiently. In the development of such sorting there are several ways, such as: Sort Sort, Insertion Sort, MergeSort. Stop as simple methods SelectSort and InsertSort; and one of the sophisticated methods: MergeSort.

Keywords: Algorithm, ordering and sequence.

**SUMÁRIO**

1. iNTRODUÇÃO8
2. desenvolvimento9

2.1 ORDENAÇÃO DE ALGORÍTMO 9

2.1.1 ORDENAÇÃO INTERNA 9

2.1.2 ORDENAÇÃO EXTERNA10

2.2 SELECTION SORT 10

2.3 INSERTION SORT11

* 1. MARGE SORT12

1. TÉCNICAS ORDENAÇÃO DE ALGORITMo 14

3.1 SELECTION SORT 14

* 1. INSERTION SORT 15
  2. MARGE SORT16

1. projeto 17
2. RELATÓRIO COM LINHAS DE CÓDIGO 18
3. Introdução

Para a realização da APS (Atividade Pratica Supervisionada), na Universidade Paulista – UNIP, o aprimoramento e compreensão de estilos de programação são essenciais. Colocou-se em prática os elementos até então adquiridos durante as aulas em conjunto com a busca de um maior aprofundamento em fontes externas, para conseguir desta forma desenvolver técnicas básicas de ordenação de algoritmo.

A ordenação de algoritmo tem como função básica a ideia de colocar os elementos de dada sequência em uma certa ordem e é de competência de engenheiros a realização de programas para realizar tais ordenações de algoritmo, compreendendo qual seria o melhor para determinada situação. Tal noção levou os membros a empenharem-se na formação de vários protótipos de ordenação de algoritmo em diversas linguagens para buscar a melhor que pudesse adequar-se as necessidades do projeto atual.

Tendo tais aspectos como base, as necessidades de um programa simples, mas eficaz tornaram as buscas e aprimoramentos o pilar principal para a realização de um trabalho voltado aos aspectos essenciais dos programas atuais. Foram necessárias várias horas de pesquisa e testes para atingir o resultado atual, necessariamente desta forma adquiriu-se uma carga de conhecimento notável referente a ordenação de algoritmos, sendo dissertada neste trabalho.

A metodologia nesse programa foi de ordenação de algoritmo que auxiliam na elaboração do material, além de orientação de professores. Neste trabalho apresentamos formas de ordenação de algoritmo, buscamos informações, pesquisas e realizamos o experimento, para garantir o melhor desempenho na escolha de um código para ordenar.

1. DESENVOLVIMENTO

2.1 Ordenação de algoritmo

Algoritmo de ordenação é um algoritmo que coloca os elementos de um conjunto de informações em uma certa ordem, crescente ou decrescente. Tem por definição o ato de reorganizar listas de dados partindo de uma relação de ordem predefinida. A ordenação tem como objetivo facilitar as buscas e pesquisas de ocorrências de determinado elemento em um conjunto ordenado.

Determinados tipos de dados trazem consigo uma maior simplicidade em sua ordenação, podendo ser destacadas: ordem numérica e ordem alfabética. Segundo (CORMEN, 2009) ordenação é: “É a operação de rearranjar os dados em uma determinada ordem.”; A ordenação por mais que não pareça é antiga tendo o bubble sort sido analisado já em 1965.

A importância da ordenação é analisada desta forma por Neto (2016, p.1):

Às vezes, a necessidade de ordenar informações é inerente a uma aplicação. Por exemplo: Para preparar os extratos dos clientes, o banco precisa ordenar os cheques pelo número. Outros algoritmos usam frequentemente a ordenação como uma sub-rotina chave. Exemplo: Pesquisa binária.

Na computação existem diversos algoritmos que utilizam variadas técnicas de ordenação para organizar um conjunto de dados, sendo então conhecidos como Métodos de Ordenação ou Algoritmos de Ordenação. Tais métodos podem ser classificados em dois tipos sendo eles:

* Ordenação Interna
* Ordenação Externa

**2.1.1 Ordenação Interna**

Tem como principal característica que, todos os elementos a serem ordenados cabem na memória principal, sendo assim qualquer registro pode vir a ser acessado imediatamente, trazendo uma maior velocidade e versatilidade em determinadas necessidades.

**2.1.2 Ordenação Externa**

Difere em simples pontos da ordenação interna sendo o mais relevante e exponencialmente observável a diferença no armazenamento. Tem-se que na ordenação externa os elementos a serem ordenados não cabem na memória principal, sendo necessário que os registros sejam acessados em grandiosos blocos ou sequencialmente de maneira externa como por exemplo usando banco de dados externo.

Entre os diversos algoritmos de ordenação existentes, pode-se destacar os seguintes: bubble sort, heap sort, insert sort e merge sort. Neste projeto deu-se ênfase nos seguintes processos de ordenação: Selection sort, Insertion sort e Merge sort.

**2.2 Selection Sort**

A ordenação por seleção ou selection sort tem como base selecionar a menor informação e colocar na primeira posição, selecionar a segunda menor informação e colocar na segunda posição, segue estes passos até que reste um único elemento, sendo este o último da lista e o maior. Observa-se também que tal método pode-se ser usado para ordenação decrescente e crescente então os dados podem vir a serem invertidos.

Sua fórmula geral de observação é:

C(n) = O(n²)

Sendo assim tal algoritimo não é estável, podendo apresentar variações como no tempo de execução e tendo melhor aproveitando com listas já pré-ordenadas. Exemplo:

vetor = 9 - 7 - 8 - 1 - 2 - 0 - 4

0 - 7 - 8 - 1 - 2 - 9 - 4

0 - 1 - 2 -7 - 8 - 9 - 4

0 - 1 - 2 - 4 - 8 - 9 - 7

0 - 1 - 2 - 4 - 7 - 9 - 8

0 - 1 - 2 - 4 - 7 - 8 – 9

Mesmo com tais aspectos deve-se sempre observar por um todo as qualidades e defeitos deste método de ordenação, podendo-se destacar por vantagens a simplicidade de implementação, a ocupação de pouca memória já que não utiliza de vetores auxiliares e tem uma das maiores velocidades na ordenação. Como desvantagens além da sua instabilidade nota-se uma lentidão maior com vetores grandes e sempre utiliza da mesma forma de comparação o vetor estando ordenado ou não.

**2.3 Insertion Sort**

Insertion sort é um algoritmo de ordenação que pode ser aplicado em Arrays e Listas dinâmicas. Ele percorre o vetor da esquerda para a direita e subsequentemente os elementos à esquerda vão sendo ordenados.

O insertion sort é um algoritmo de classificação simples que funciona de forma semelhante à maneira como classifica-se as cartas de baralho. O array é virtualmente dividido em uma parte ordenada e outra não ordenada. Os valores da parte não ordenada são selecionados e colocados na posição correta na parte ordenada.

Este método de ordenação utiliza da mesma fórmula do selection sort sendo ela:

C(n) = O(n²)

Ao analisar tal método também deve-se levar em consideração seus pontos positivos e negativos, sendo assim pode-se dar ênfase em tais características positivas: Diferentemente do selection sort ele é um método estável e é um método recomendável quando a poucos elementos a serem ordenados em um arquivo.

Como pontos negativos deve-se levar em consideração a alta movimentação de elementos dentro do vetor, para poder desta forma ordenar os dados necessários.

**2.4 Merge Sort**

O merge sort, ou ordenação por mistura, é um tipo de algoritmo de ordenação por comparação do tipo dividir-para-conquistar. Por definição geral ele tem como base a divisão da matriz de entrada em duas metades, chama a si mesmo para as duas metades e, em seguida, mescla as duas metades classificadas, ou seja: Dividir o problema original em subproblemas menores. Conquista: Resolver cada subproblema recursivamente. Combinação: Combinar as soluções encontradas, compondo uma solução para o problema original.

É o mais complexo algoritmo de ordenação dos apresentados e utilizados no trabalho, e tem como ponto forte principal que, em algoritmos grandes ele utiliza menos tempo para a realização das operações.

Sua função é a mais complexa e a que utiliza mais espaços na memória dada a criação de uma cópia do vetor em cada nível de criação na chamada recursiva. Fórmula de funcionamento:

Ο (n log n)

Suas vantagens como já citado são a alta velocidade na resolução e ordenação de uma grande quantidade de dados, e tem por problema principal o alto uso da memória devido a criação de cópias do vetor. Segundo o professor Tulio Toffolo merge sort é indicado para aplicações que tem restrição de tempo.

3 TÉCNICAS ORDENAÇÃO DE ALGORITMo

**3.1 Selection Sort**

Para a realização do projeto utilizou-se um conjunto de valores previamente fornecidos sendo eles na quantidade de: 1000, 5000 e 10000. Através desses valores foram realizados 5 testes de tempo de execução em milissegundos.

Testes realizados com 1000 valores:

|  |  |
| --- | --- |
| TESTES | TEMPO (ms) |
| 1 | 1.531 |
| 2 | 1.460 |
| 3 | 0.938 |
| 4 | 1.009 |
| 5 | 1.033 |
| Média | 1.194 |

Testes realizados com 5000 valores:

|  |  |
| --- | --- |
| TESTES | TEMPO (ms) |
| 1 | 32.355 |
| 2 | 34.885 |
| 3 | 32.928 |
| 4 | 39.336 |
| 5 | 41.718 |
| Média | 36,244 |

Testes realizados com 10000 valores:

|  |  |
| --- | --- |
| TESTES | TEMPO (ms) |
| 1 | 151.819 |
| 2 | 160.401 |
| 3 | 147,261 |
| 4 | 161.224 |
| 5 | 155.590 |
| Média | 155.259 |

**3.2 Insertion Sort**

Para a realização do projeto utilizou-se um conjunto de valores previamente fornecidos sendo eles na quantidade de: 1000, 5000 e 10000. Através desses valores foram realizados 5 testes de tempo de execução em milissegundos.

Testes realizados com 1000 valores:

|  |  |
| --- | --- |
| TESTES | TEMPO (ms) |
| 1 | 2.852 |
| 2 | 2.744 |
| 3 | 2.743 |
| 4 | 2.653 |
| 5 | 2.704 |
| Média | 2,739 |

Testes realizados com 5000 valores:

|  |  |
| --- | --- |
| TESTES | TEMPO (ms) |
| 1 | 2.686 |
| 2 | 2.934 |
| 3 | 2.541 |
| 4 | 2.936 |
| 5 | 2.925 |
| Média | 2.804 |

Testes realizados com 10000 valores:

|  |  |
| --- | --- |
| TESTES | TEMPO (ms) |
| 1 | 3.051 |
| 2 | 2.966 |
| 3 | 2.523 |
| 4 | 2.834 |
| 5 | 3.054 |
| Média | 2.885 |

**3.3 Merge Sort**

Para a realização do projeto utilizou-se um conjunto de valores previamente fornecidos sendo eles na quantidade de: 1000, 5000 e 10000. Através desses valores foram realizados 5 testes de tempo de execução em milissegundos.

Testes realizados com 1000 valores:

|  |  |
| --- | --- |
| TESTES | TEMPO (ms) |
| 1 | 0.002 |
| 2 | 0.002 |
| 3 | 0.003 |
| 4 | 0.002 |
| 5 | 0.002 |
| Média | 0.002 |

Testes realizados com 5000 valores:

|  |  |
| --- | --- |
| TESTES | TEMPO (ms) |
| 1 | 0.003 |
| 2 | 0.002 |
| 3 | 0.002 |
| 4 | 0.003 |
| 5 | 0.003 |
| Média | 0.002 |

Testes realizados com 10000 valores:

|  |  |
| --- | --- |
| TESTES | TEMPO (ms) |
| 1 | 0.003 |
| 2 | 0.003 |
| 3 | 0.003 |
| 4 | 0.002 |
| 5 | 0.002 |
| Média | 0.002 |

4 projeto

Com intuito de apreender a funcionalidade da ordenação de algoritmo tanto na base de programação quanto em sua essência, quando começamos a elaborar este trabalho tínhamos diversas duvidas perante quais das formas de ordenação seria mais eficiente em determinados casos.

Durante a execução dividiu-se o grupo em dois para testar e entender cada ordenação de algoritmo proposta. Apresentando os aspectos positivos e negativos de cada ordenação. Mediu-se o tempo de execução de cada método proposto e analisou-se as diferenças de cada um. Encontrou-se a maior diferença quando se analisou a complexidade dos códigos usados para a ordenação dos valores fornecidos, tendo assim o Merge Sort sido o mais rápido.

Observando por outro ângulo notou-se uma facilidade muito grande e maior na implementação do selection sort sendo ele o mais prático, mesmo não apresentando valores tão expressivos quanto o merge sort. O grande problema encontrado no selection sort é o tempo de execução quando os dados fornecidos são muito altos, tendo uma diferença gigantesca em comparação aos outros métodos de ordenação testados.

Centralizando-se como a opção mais viável, tendo como base uma noção de implementação por tempo o insertion sort manteve uma facilidade de implementação e tempos muito mais baixos comparados ao selection sort quando se usou vetores com mais de 1000 números.

Em uma análise conjunta e levando em conta o fator simplicidade com tempo o método de ordenação mais apropriado para a resolução do problema proposto é o selection sort. Contudo é necessário destacar a complexidade e qualidade do merge sort sendo um método muito mais sofisticado e necessário para operações complexas de vários dados.

5 RELATÓRIO COM LINHAS DE CÓDIGO

**5.1 Selection Sort**

#include <stdio.h>

#include <time.h>

void selectSort(int \*pVetor)

{

int vMenor;

int vAux;

int vTemp;

int vTroca;

for(vAux=0; vAux < n-1; vAux++)

{

vMenor = vAux;

for (vTemp=vAux+1; vTemp < n; vTemp++) até o final;

{

if (pVetor[vTemp] < pVetor[vMenor])

{

vMenor = vTemp;

}

}

if (vMenor != vAux)

{

vTroca = pVetor[vAux];

pVetor[vAux] = pVetor[vMenor];

pVetor[vMenor] = vTroca;

}

}

}

int main()

{

clock\_t t;

int vetor[ n ] = {787,

869,

246,

[...]

755,

644};

int p, r, a;

p = 0;

r = n;

t = clock();

selectSort(vetor);

t = clock() - t;

printf("Tempo de execucao: %lf", ((double)t)/((CLOCKS\_PER\_SEC/1000)));

}

**5.2 Insertion Sort**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void fInsertion\_Sort(int \*pVetor);

int main()

{

Int vVetor[ n ] = {};

};

int vAux;

clock\_t t;

t = clock();

fInsertion\_Sort(vVetor);

t = clock() - t;

printf("Tempo de execucao: %lf", ((double)t)/((CLOCKS\_PER\_SEC/1000)));

fInsertion\_Sort(vVetor);

return 0;

}

void fInsertion\_Sort(int \*pVetor)

{

int vAux;

int vTemp;

int vTroca;

for (vAux=1; vAux < 1000; vAux++)

{

vTemp = vAux;

while (pVetor[vTemp] < pVetor[vTemp-1])

{

vTroca = pVetor[vTemp];

pVetor[vTemp] = pVetor[vTemp-1];

pVetor[vTemp-1] = vTroca;

vTemp--;

if (vTemp == 0)

break;

}

}

}

**5.3 Merge Sort**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

void merge(int vetor[], int comeco, int meio, int fim)

{

int com1 = comeco, com2 = meio+1, comAux = 0, tam = fim-comeco+1;

int \*vetAux;

vetAux = (int\*)malloc(tam \* sizeof(int));

while(com1 <= meio && com2 <= fim)

{

if(vetor[com1] < vetor[com2])

{

vetAux[comAux] = vetor[com1];

com1++;

} else {

vetAux[comAux] = vetor[com2];

com2++;

}

comAux++;

}

while(com1 <= meio)

{

vetAux[comAux] = vetor[com1];

comAux++;

com1++;

}

while(com2 <= fim)

{

vetAux[comAux] = vetor[com2];

comAux++;

com2++;

}

for(comAux = comeco; comAux <= fim; comAux++)

{

vetor[comAux] = vetAux[comAux-comeco];

}

free(vetAux);

}

void mergeSort(int vetor[], int comeco, int fim)

{

if (comeco < fim)

{

int meio = (fim+comeco)/2;

mergeSort(vetor, comeco, meio);

mergeSort(vetor, meio+1, fim);

merge(vetor, comeco, meio, fim);

}

}

int main (void)

{

int vetor[ n ] =

{

};

clock\_t t;

int comeco, fim;

t = clock();

mergeSort(vetor, comeco, fim);

//a = 10000 \* log ;

t = clock() - t;

printf("Tempo de execucao: %lf", ((double)t)/((CLOCKS\_PER\_SEC/1000)));

return 0;

}

REFERÊNCIAS

Algorítmos de ordenação história em: http://www.ic.uff.br/~boeres/slides\_ed/ed4.pdf

Algoritmos de Ordenação: Insertion Sort. Disponível em:https://www.geeksforgeeks.org/insertion-sort/

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. e STEIN, C.

Estrutura de Dados e Algoritmos. 2 Semestre de 2014. De Professor Ricardo Farias. Disponível em https://www.cos.ufrj.br/~rfarias/cos121/aula\_07.html

Introduction to Algorithms, 3a edição, MIT Press, 2009.

Merge Sort em python disponível em: https://panda.ime.usp.br/pythonds/static/pythonds\_pt/05-OrdenacaoBusca/OMergeSort.html

Métodos de Ordenação: Selection, Insetion, Bubble, Merge (Sort) por(Hebert Coelho e Nádia Félix) disponível em: http://ww2.inf.ufg.br/~hebert/disc/aed1/AED1\_04\_ordenacao1.pdf

Ordenação: Merge Sort (Prof: Tulio Toffolo) disponível em: http://www3.decom.ufop.br/toffolo/site\_media/uploads/2013-1/bcc202/slides/14.\_mergesort.pdf

Principais algoritmos de ordenação: https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao/#:~:text=A%20ordena%C3%A7%C3%A3o%20ou%20classifica%C3%A7%C3%A3o%20de,elemento%20em%20um%20conjunto%20ordenado.