

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 0.

Sumario

Pag 0 – Sumario.

Pag 1 – Como compilar, estrutura da make e introdução

Pag 2- Apresentação das structs e funções do trabalho.

Pag 3- Algoritmo de ordenação por seleção.

Pag 4- Algoritmo de ordenação por inserção.

Pag 6 -Algoritmo de ordenação shellsort.

Pag 7-Algoritmo de ordenação quicksort.

Pag 9 -Algoritmo de Ordenação heapsort.

Pag 12 -Algoritmo de Ordenação Mergesort.

Pag 14 – Início dos testes.

Pag 16- testes por seleção.

Pag 18 –teste por inserção.

Pag 21 – teste Shellsorte.

Pag 24 – teste quicksort.

Pag 26 – teste heapsort.

Pag 29- teste mergesort.

Pag 32 – conclusões gerais.

Pag 33 – Tempos médio em todos os testes, Dificuldades na produção do trabalho e Bibliografias.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 1.

Vitor Ferreira França 19/05/2023

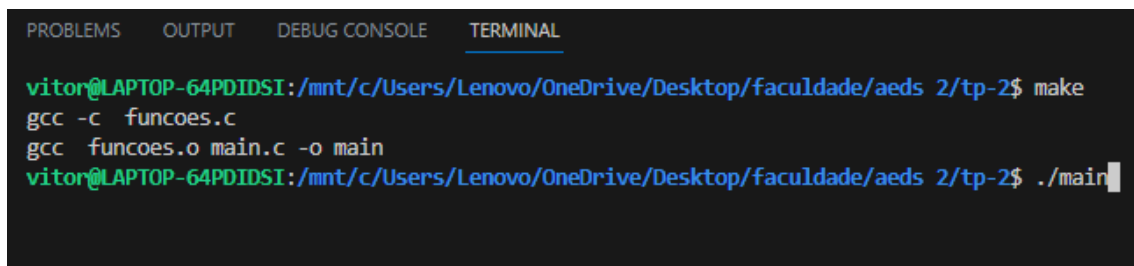
Documentação TP-2

Para compilar:

Desenvolvido no VSC utilizando a linguagem C num terminal ubuntu.

Utilize make.

Imagem

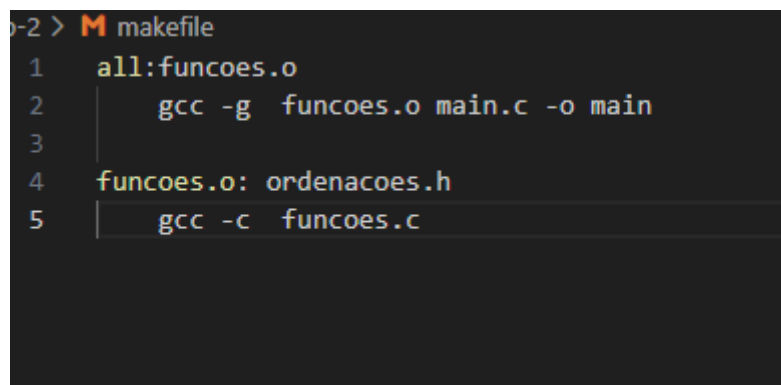
A screenshot of a terminal window with a dark background. The terminal shows the following commands and output:

```
vitor@LAPTOP-64PDIDSI:/mnt/c/Users/Lenovo/OneDrive/Desktop/faculdade/aeds 2/tp-2$ make
gcc -c funcoes.c
gcc funcoes.o main.c -o main
vitor@LAPTOP-64PDIDSI:/mnt/c/Users/Lenovo/OneDrive/Desktop/faculdade/aeds 2/tp-2$ ./main
```

 The terminal window has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL, with TERMINAL being the active tab.

O MakeFile feito:

imagem 2

A screenshot of a code editor showing the content of a Makefile. The text is as follows:

```
1 all:funcoes.o
2     gcc -g funcoes.o main.c -o main
3
4 funcoes.o: ordenacoes.h
5     gcc -c funcoes.c
```

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Introdução:

Pag 2.

O seguinte algoritmo tem como objetivo a criação e implementação das estruturas de ordenações por seleção, por inserção, Shellsort, Quicksort, heapsort e Mergesort , para com vetores de estruturas apenas com a chave e outra estrutura com vetor de caracter de 50 por 50, também testar cada uma das estruturas e obter os dados das mesmas para comparação .

funções utilizadas :

imagem 3

```
19 void selecao_int( apenas_chave *vet,int n);
20 void insercao_int( apenas_chave *vet, int n);
21 void shellsort_int(apenas_chave *vet , int n);
22 void quicksort_int(apenas_chave *vet, int esq, int dir);
23 void construir_int( apenas_chave *vet , int inicio , int final);
24 void heapsort_int( apenas_chave *vet , int n);
25 void merge_sort_int(apenas_chave *vet, int inicio, int fim);
26 void merge_int(apenas_chave *vet ,int p ,int q ,int r);
27
28 //funcoes para teste com string -----
29 void encher_vetor_complexa(teste *vet, int n);
30 void printar_vetor_complexa(teste *vet, int n);
31 void selecao_complexa( teste *vet,int n);
32 void insercao_complexa( teste *vet, int n);
33 void shellsort_complexa(teste *vet , int n);
34 void quicksort_complexa(teste *vet, int esq, int dir);
35 void construir_complexa(teste *vet, int raiz, int fundo);
36 void heapsort_complexa(teste *vet,int n );
37 void merge_sort_complexa(teste *vet, int inicio, int fim);
38 void merge_complexa(teste *vet, int inicio, int meio, int fim);
39 #endif
```

Structs utilizadas

```
// estruturas utilizadas para os
typedef struct teste{
    int chave;
    char T[50][50];
}teste;

typedef struct apenas_chave{
    int chave;
}apenas_chave;
```

imagem 4

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 3.

Funções:

Por seleção:

Imagem 5

```
30
31 void insercao_int( apenas_chave *vet, int n){
32     int i , j ;
33     apenas_chave aux;
34
35     for( i = 0 ; i < n - 1 ;i++){
36
37         if(vet[i].chave > vet[i+1].chave){
38             aux = vet[i + 1];
39             vet[i+1] = vet[i];
40             vet[i] = aux;
41
42             j = i - 1 ;
43
44             while(j>=0){
45
46                 if(aux < vet[j].chave){
47                     vet[j+1] = vet[j];
48                     vet[j] = aux;
49                 }else { break ;}
50                 j = j - 1 ;
51             }
52         }
53     }
54 }
55
56
```

< 8 >-A função deve receber um vetor e seu tamanho.

< 10 >-Logo após é criado o “auxiliar” e o “menor” no qual, o “menor” guardara a menor chave nas comparações , e o “auxiliar” é usado Para guardar a chave do item que será sobre posto para o dado não se perder, assim a troca ocorrendo corretamente.

< 12 – 19> Nestas linhas ocorre o percorrer do vetor e a comparação das chaves para denotar o menor item a ser posto na primeira posição.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

<22-24> Nesta etapa é guardado o item que será sobre posto e inserido o menor nele, depois o item guardado é posto no lugar do outro item. assim a troca sendo concluída

Pag 4.

Aplicada a outra struct mais complexa (porém com o mesmo funcionamento):

Imagem 6

```
7 // Ordenações Simples
8 void selecao_int( apenas_chave *vet, int n){
9
10     int menor;
11     apenas_chave auxiliar;
12     for(int i = 0 ; i <= n ; i++){
13         menor = i ;
14
15         for (int j = i+1 ; j <= n ; j++){
16
17             if(vet[j].chave < vet[menor].chave ){ menor = j ;}
18
19         }
20
21         auxiliar = vet[i];
22         vet[i] = vet[menor];
23         vet[menor] = auxiliar;
24
25     }
26
27 }
28
29 }
30
```

-A diferença é que esse recebe um vetor de struct “teste”.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Por inserção:

```
30
31 void insercao_int( apenas_chave *vet, int n){
32     int i , j ,aux;
33
34     for( i = 0 ; i < n - 1 ;i++ ){
35
36         if(vet[i].chave > vet[i+1].chave){
37             aux = vet[i + 1].chave;
38             vet[i+1].chave = vet[i].chave;
39             vet[i].chave = aux;
40
41             j = i - 1 ;
42
43             while(j>=0){
44
45                 if(aux < vet[j].chave){
46                     vet[j+1].chave = vet[j].chave;
47                     vet[j].chave = aux;
48                 }else { break ;}
49                 j = j - 1 ;
50             }
51         }
52     }
53
54 }
55
```

imagem 7

pag 5.

<31> A função deve receber um vetor da struct “apenas_chave” e o tamanho desse vetor.

<34-52> “For” utilizado para percorrer o vetor até o penúltimo item ,pressupondo que a esquerda no vetor os itens estão corretamente alocados.

<36-51> Nesta etapa é feita comparação para saber se é preciso que a troca seja feita.

<43-50> Se os dados forem compatíveis com o “if “ ,será percorrido para “esquerda” do item onde esta o “for” ate o 0 ,comparando os itens , para se necessário ocorrer a troca.

Aplicada a outra struct mais complexa (porém com o mesmo funcionamento):

Imagem 8

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

```
}
void insercao_complexa( teste *vet, int n){
    int i , j ;
    teste aux;

    for( i = 0 ; i < n - 1 ;i++ ){

        if(vet[i].chave > vet[i+1].chave){
            aux = vet[i + 1];
            vet[i+1] = vet[i];
            vet[i] = aux;

            j = i - 1 ;

            while(j>=0){

                if(aux.chave < vet[j].chave){
                    vet[j+1] = vet[j];
                    vet[j] = aux;
                }else { break ;}
                j = j - 1 ;
            }
        }
    }
}
```

-A diferença é que esse recebe um vetor de struct “teste”.

pag 6.

Shellsort:

Imagem 9

```
57 void shellsort_int(apenas_chave *vet , int n){
58     int h = 1;
59     int j;
60     apenas_chave aux;
61     while(h < n){
62         h = 3 * h + 1 ;
63     }
64     while( h > 1 ){
65         h /= 3 ;
66         for( int i = h ; i < n ; i ++ ){
67             aux = vet[i];
68             j = i - h;
69             while(j >= 0 && aux.chave < vet[j].chave){
70                 vet[j+h] = vet[j];
71                 j-= h;
72             }
73             vet[j+h] = aux;
74         }
75     }
76 }
77 }
```

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

<57> A função recebe um vetor o tamanho do mesmo.

Pag 7:

<58-60> São declaradas as variáveis de apoio para a implementação do shellsorte.

<61-63> “While” utilizado para calcular a variável “h” até ela chegar ao número de itens do vetor para as comparações dos itens serem feitas.

<64-75> Outro “while” para ser feito o processo enquanto o h for maior que 1 .

<66-74> “For” utilizado para percorrer os itens do vetor para ser feito as comparações e as atribuições.

Pag 7

Aplicada a outra struct mais complexa (porém com o mesmo funcionamento):

Pag 7:

Imagem 10

```
void shellsort_complexa(teste *vet , int n){
    int h = 1;
    int j;
    teste aux;

    while(h < n){
        h = 3 * h + 1 ;
    }
    while( h > 1 ){
        h /= 3 ;
        for( int i = h ; i < n ; i ++ ){
            aux = vet[i];
            j = i - h;
            while(j >= 0 && aux.chave < vet[j].chave){
                vet[j+h] = vet[j];
                j -= h;
            }
            vet[j+h] = aux;
        }
    }
}
```

-A diferença é que esse recebe um vetor de struct “teste”.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Quicksort:

Imagem 11

pag 8.

```
79 void quicksort_int(apenas_chave *vet, int esq, int dir) {
80     int i, j, x;
81     apenas_chave y ;
82     i = esq;
83     j = dir;
84     x = vet[(esq + dir) / 2].chave;
85
86     while(i <= j) {
87         while(vet[i].chave < x && i < dir) {
88             i++;
89         }
90         while(vet[j].chave > x && j > esq) {
91             j--;
92         }
93         if(i <= j) {
94             y = vet[i];
95             vet[i] = vet[j];
96             vet[j] = y;
97             i++;
98             j--;
99         }
100     }
101
102     if(j > esq) {
103         quicksort_int(vet, esq, j);
104     }
105     if(i < dir) {
106         quicksort_int(vet, i, dir);
107     }
108 }
```

<79> A função precisa de um vetor , e um inteiro para representar o primeiro item do vetor , e outro inteiro para representar o ultimo item do vetor.

<80-81> É declarado as variáveis utilizadas onde o x é o pivô , o y é o auxiliar utilizado para as trocas , o i e o j representaram o a esquerda e a direita utilizada para os subvetores.

<86-100> “while” utilizado para partição e o percorrer do vetor.

<87-89> ” While” utilizado para percorrer do inicio do vetor ate o pivô.

<90-92> “While” utilizado para percorrer do final do vetor ate o pivô.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

<93-99> É feito a comparação para chegar se é necessário ocorrer as trocas.

<102-107>É checado se é preciso o processo ser repetido ou se já foi ordenado corretamente.

Pag 9

Aplicada a outra struct mais complexa (porém com o mesmo funcionamento):

Imagem 12

```
void quicksort_complexa(teste *vet, int esq, int dir){
    int i, j, x;
    teste y;

    i = esq;
    j = dir;
    x = vet[(esq + dir) / 2].chave;

    while(i <= j) {
        while(vet[i].chave < x && i < dir) {
            i++;
        }
        while(vet[j].chave > x && j > esq) {
            j--;
        }
        if(i <= j) {
            y = vet[i];
            vet[i] = vet[j];
            vet[j] = y;
            i++;
            j--;
        }
    }

    if(j > esq) {
        quicksort_complexa(vet, esq, j);
    }
    if(i < dir) {
        quicksort_complexa(vet, i, dir);
    }
}
```

-A diferença é que esse recebe um vetor de struct “teste”.

Heapsort:

A heapsort é composta por duas funções a “constroi” designada para a criação da árvore binária e comparação dos itens “filhos” com os “pais e a

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

“heapsort” que chama o processo e ver quando o processo esta satisfatoriamente ordenado.

Função de construção e processo:

Imagem 13

Pag 10.

```
208
209 void construir_int( apenas_chave *vet, int raiz , int fundo){
210     int pronto, filhoMax;
211     apenas_chave tmp;
212
213     pronto = 0;
214     while ((raiz*2 <= fundo) && pronto!=1) {
215         if (raiz*2 == fundo) {
216             filhoMax = raiz * 2;
217         }
218         else if (vet[raiz * 2].chave > vet[raiz * 2 + 1].chave) {
219             filhoMax = raiz * 2;
220         }
221         else {
222             filhoMax = raiz * 2 + 1;
223         }
224
225         if (vet[raiz].chave < vet[filhoMax].chave) {
226             tmp = vet[raiz];
227             vet[raiz] = vet[filhoMax];
228             vet[filhoMax] = tmp;
229             raiz = filhoMax;
230         }
231         else {
232             pronto = 1;
233         }
234     }
235 }
```

<210-213> É declarado as variáveis para a construção da “arvore” ,trocas e para denotar quando o “while” deve se finalizar .

<214-234>Compara se os “filhos” são maiores que os pais ,se forem é efetuada a troca ,também percorrendo a “arvore” para comparar todos itens.

<231-233>Denota o fim do “while” após o processo terminar.

A função a ser chamada pelo usuario:

Imagem 14

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

```
234     }
235 }
236 void heapsort_int( apenas_chave *vet , int n){
237     int i;
238     apenas_chave aux;
239
240     for (i = (n / 2); i >= 0; i--) {
241         construir_int(vet, i, n - 1);
242     }
243
244     for (i = n-1; i >= 1; i--) {
245         aux = vet[0];
246         vet[0] = vet[i];
247         vet[i] = aux;
248         construir_int(vet, 0, i-1);
249     }
250 }
```

Pag 11.

<237-238> É declarado as necessidades para rodar a função o “i” percorrer o vetor e o “aux” para poder efetuar as trocas necessárias.

<240-242> Este “for” percorre da metade ate o inicio do vetor, criando a estrutura de “arvore”.

<244-249>Este “for” percorre do penúltimo item ate o fim do vetor criando outras heap, assim terminado de ordenar o vetor.

Mesmas funções porem para estruturas de dados diferentes:

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

imagem 15

```
250 }
251 void construir_complexa(teste *vet, int raiz, int fundo){
252     int pronto, filhoMax;
253     teste tmp;
254
255     pronto = 0;
256     while ((raiz*2 <= fundo) && pronto!=1) {
257         if (raiz*2 == fundo) {
258             filhoMax = raiz * 2;
259         }
260         else if (vet[raiz * 2].chave > vet[raiz * 2 + 1].chave) {
261             filhoMax = raiz * 2;
262         }
263         else {
264             filhoMax = raiz * 2 + 1;
265         }
266
267         if (vet[raiz].chave < vet[filhoMax].chave) {
268             tmp = vet[raiz];
269             vet[raiz] = vet[filhoMax];
270             vet[filhoMax] = tmp;
271             raiz = filhoMax;
272         }
273         else {
274             pronto = 1;
275         }
276     }
277 }
```

```
277 }
278 void heapsort_complexa(teste *vet, int n ){
279     int i;
280     teste aux;
281
282     for (i = (n / 2); i >= 0; i--) {
283         construir_complexa(vet, i, n - 1);
284     }
285
286     for (i = n-1; i >= 1; i--) {
287         aux = vet[0];
288         vet[0] = vet[i];
289         vet[i] = aux;
290         construir_complexa(vet, 0, i-1);
291     }
292 }
```

imagem 16

Pag 12

Mergesort:

A ordenação Mergesort é composta por duas funções a merge que responsável pelas repartições e comparações do vetor e a própria Mergesorte que organiza esse processo.

Imagem 17

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

```
159 void merge_sort_int(apenas_chave *vet, int inicio, int fim) {  
160     if (inicio < fim) {  
161         int meio = (inicio + fim) / 2;  
162         merge_sort_int(vet, inicio, meio);  
163         merge_sort_int(vet, meio + 1, fim);  
164         merge_int(vet, inicio, meio, fim);  
165     }  
166 }
```

<159>Esta função deve receber um vetor , um indicador do inicio e fim do vetor.

<160-165> Este if denotado ate quando a recursividade deve ocorrer , para assim ordenar corretamente.

<161> É calculado o meio para logo após chamar a função.

<162-164> É chamada recursivamente a função e a merge é posta em ação para fazer o processo de partição e comparação.

Pag 13.

Função merge

Imagem 18

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

```
167
168 void merge_int(apenas_chave *vet, int inicio, int meio, int fim) {
169     apenas_chave *temp;
170     temp = (apenas_chave*)malloc((fim - inicio + 1) * sizeof(apenas_chave));
171     int p1, p2, tamanho, i, j, k;
172     int fim1 = 0, fim2 = 0;
173     tamanho = fim - inicio + 1;
174
175     p1 = inicio;
176     p2 = meio + 1;
177
178     if (temp != NULL) {
179         for (i = 0; i < tamanho; i++) {
180             if (!fim1 && !fim2) {
181                 if (vet[p1].chave < vet[p2].chave)
182                     temp[i] = vet[p1++];
183                 else
184                     temp[i] = vet[p2++];
185
186                 if (p1 > meio)
187                     fim1 = 1;
188                 if (p2 > fim)
189                     fim2 = 1;
190             } else {
191                 if (!fim1)
192                     temp[i] = vet[p1++];
193                 else
194                     temp[i] = vet[p2++];
195             }
196         }
197         for (j = 0, k = inicio; j < tamanho; j++, k++)
198             vet[k] = temp[j];
199     }
200     free(temp);
201 }
```

<168> A função deve receber um vetor, um número representando o inicio o meio e o fim.

<178> Este primeiro “if” checa se a alocação foi feita corretamente na memoria, se não nada ocorre.

<179-196> Neste “for” ocorre a partição do vetor e as trocas.

<181-184> Executa as combinações ordenando.

<186-189> Estes “if” checam se o vetor já terminou.

<197-198> Transfere os dados do vetor auxiliar para o vetor original.

<200> libera a memoria alocada no inicio da função.

O mesmo funcionamento das funções porém com as outra struct

Imagem 19

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

pag 14.

```
353 }
354 void merge_sort_complexa(teste *vet, int inicio, int fim){
355     if (inicio < fim) {
356         int meio = (inicio + fim) / 2;
357         merge_sort_complexa(vet, inicio, meio);
358         merge_sort_complexa(vet, meio + 1, fim);
359         merge_complexa(vet, inicio, meio, fim);
360     }
361 }
362
```

Imagem 10

```
363 void merge_complexa(teste *vet, int inicio, int meio, int fim){
364     teste *temp;
365     temp = (teste*)malloc((fim - inicio + 1) * sizeof(teste));
366     int p1, p2, tamanho, i, j, k;
367     int fim1 = 0, fim2 = 0;
368     tamanho = fim - inicio + 1;
369
370     p1 = inicio;
371     p2 = meio + 1;
372
373     if (temp != NULL) {
374         for (i = 0; i < tamanho; i++) {
375             if (!fim1 && !fim2) {
376                 if (vet[p1].chave < vet[p2].chave)
377                     temp[i] = vet[p1++];
378                 else
379                     temp[i] = vet[p2++];
380
381                 if (p1 > meio)
382                     fim1 = 1;
383                 if (p2 > fim)
384                     fim2 = 1;
385             } else {
386                 if (!fim1)
387                     temp[i] = vet[p1++];
388                 else
389                     temp[i] = vet[p2++];
390             }
391         }
392         for (j = 0, k = inicio; j < tamanho; j++, k++)
393             vet[k] = temp[j];
394     }
395 }
```

testes:

-O tempo de execução médio condiz ao tempo real do computador onde foi feito os testes.

Algoritmo exemplo de como foi conseguido o tempo:

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

pag 15

```
tp-2 > C main.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "ordenacoes.h"
4  #include <sys/time.h>
5  #include <sys/resource.h>
6
7
8  int main()
9  {
10     int n = 20;
11     apenas_chave v[20];
12
13     // tempo real-----
14     struct timeval start, end;
15
16
17     // tempo real-----
18     encher_vetor_int(v,n);
19     quicksort_int(v, n-1);
20     printf("\n");
21     // tempo real-----
22     gettimeofday(&start, NULL);
23     gettimeofday(&end, NULL);
24     long double startTime = start.tv_sec + start.tv_usec / 1000000.0L;
25     long double endTime = end.tv_sec + end.tv_usec / 1000000.0L;
26     // tempo real -----
27     printf("tempo real\n");
28     printf("Tempo inicial = %Lf\n", startTime);
29     printf("Tempo final = %Lf\n", endTime);
30     printf("Tempo decorrido = %Lf\n", endTime - startTime);
31     printf("\n");
32     //tempo real -----
33
34
35
36 }
```

<4> É usado a biblioteca “<sys/time.h>” .

<14> É declarado o inicio e o fim onde sera calculado o tempo da diferença das duas

<24-25>É calculado o tempo e dividido por 1000000.

<26-31>é mostrado as informações no terminal.

- As chaves dos vetores foram escolhidas por uma função aleatória.

Função de preencher o vetor

```
207 void encher_vetor_int(apenas_chave *vet, int n){
208
209
210     srand(time(NULL));
211     for(int i = 0 ; i < n ; i++){
212         vet[i].chave = rand();
213     }
214
215 }
```

<207> A função recebe um vetor e o tamanho dele.

<210> É iniciado o gerador de números aleatórios usando a biblioteca “<time.h>”.

<211-213> Percorre o vetor e o preenche.

Feito por : Vitor Ferreira França
< a – b> isto denota de uma certa a linha a outra b.

Pag 16

-Estrutura de quando o vetor precisa ser alocado na declaração

```
teste *v = (teste*)malloc(n*sizeof(teste)) ;
```

Por seleção:

A complexidade é $O(n^2)$

1-Com 20 elemento:

Com apenas a chave

Tempo de execução medio = 0,000003.

Número médio de comparações = 190.

Número médio de movimentações = 57.

Com a chave e as strings

Tempo de execução médio: 0,000001.

Número médio de comparações: 190.

Número médio de movimentações: 57.

2-Com 500 elementos:

Com apenas a chave

Tempo de execução medio = 0,000005.

Número médio de comparações = 124750.

Número médio de movimentações = 1497.

Com a chave e as strings

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 17

Tempo de execução medio = 0.00000.

Número médio de comparações = 124750.

Número médio de movimentações = 1497.

3-Com 5000 elementos:

Com apenas a chave

Tempo de execução médio: 0,000003.

Número médio de comparações:12497500.

Número médio de movimentações:14,997

Com a chave e as strings

(o vetor teve de ser alocado dinamicamente)

Tempo de execução médio: 0.000004.

Número médio de comparações:12497500.

Número médio de movimentações:14,997

4-Com 10000 elementos:

Com apenas a chave

Tempo de execução médio: 0,00000.

Número médio de comparações:49995000.

Número médio de movimentações:29997.

Com a chave e as strings

(o vetor teve de ser alocado dinamicamente)

Tempo de execução médio: 0,000005.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 18.

Número médio de comparações:49995000.

Número médio de movimentações:29997

5- Com 200000 elementos:

Com apenas a chave

Tempo de execução médio: 0,000001.

Número médio de comparações: 199999^{10} .

Número médio de movimentações: 599997.

Com a chave e as strings

(o vetor teve de ser alocado dinamicamente)

Tempo de execução médio: 0.000002.

Número médio de comparações: 199999^{10} .

Número médio de movimentações: 599997.

Conclusão: O número de movimentações foi menor em todos mostrando que houve muitas comparações e poucas movimentações, o tempo médio não mudou de um pro outro independentemente da quantidade de itens. mostrando assim a eficiência do algoritmo de forma homogênea, algoritmo de implementação e logica simples ,porem com numero maior de itens demandava mais do computador.

Por inserção:

Complexidade é $O(N^2)$

1-Com 20 elemento:

Com apenas a chave

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 19.

Tempo de execução médio: 0.000003.

Número médio de comparações: 114.

Número médio de movimentações:152.

Com a chave e as strings

Tempo de execução medio =0.000004.

Número médio de comparações =114.

Número médio de movimentações =152.

2-Com 500 elementos:

Com apenas a chave

Tempo de execução médio: 0.000002.

Número médio de comparações: 125249.

Número médio de movimentações: 63872.

Com a chave e as strings

Tempo de execução medio =0.000002.

Número médio de comparações =125249.

Número médio de movimentações =63872.

3-Com 5000 elementos:

Com apenas a chave

Tempo de execução médio: 0.00000.

Número médio de comparações: 6253749.

Número médio de movimentações: 6263747.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 20

Com a chave e as strings

(o vetor teve de ser alocado dinamicamente)

Tempo de execução médio: 0.000004.

Número médio de comparações: 6253749.

Número médio de movimentações: 6263747.

4-Com 10000 elementos:

Com apenas a chave

Tempo de execução médio: 0.000003.

Número médio de comparações:25007499.

Número médio de movimentações:25027497.

Com a chave e as strings

(o vetor teve de ser alocado dinamicamente)

Tempo de execução médio: 0.000005.

Número médio de comparações:25007499.

Número médio de movimentações:25027497.

5- Com 200000 elementos:

Com apenas a chave

Tempo de execução médio: 0.000004.

Número médio de comparações:1000015^10.

Número médio de movimentações:1000055^10.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 21.

Com a chave e as strings

(o vetor teve de ser alocado dinamicamente)

Tempo de execução médio: 0.000002.

Número médio de comparações:1000015^10.

Número médio de movimentações:1000055^10.

Conclusão: Na ordenação por inserção o número de movimentação foi maior que o de comparações, porém o tempo que o computador precisou foi basicamente o mesmo, algoritmo de logica simples e implementação simples, porem exigiu mais do computador com numero maior de itens .

Shellsort :

Complexidade deste algoritmo é de $O(n \log n)$.

1- Com 20 eleemntos:

Com apenas a chave

Tempo de execução médio: 0.000002.

Número médio de comparações:80.

Número médio de movimentações:63.

Com a chave e as strings

Tempo de execução médio: 0.000003.

Número médio de comparações:80.

Número médio de movimentações:53.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 22.

2-Com 500 elemetos:

Com apenas a chave

Tempo de execução médio: 0.0000004.

Número médio de comparações:2323.

Número médio de movimentações:2001.

Com a chave e as strings

Tempo de execução médio: 0.00000003.

Número médio de comparações:2323.

Número médio de movimentações:1993.

3-Com 5000 elementos:

Com apenas a chave

Tempo de execução médio: 0.0000003.

Número médio de comparações:24822.

Número médio de movimentações: 22004.

Com a chave e as strings

Tempo de execução médio: 0.0000006.

Número médio de comparações:24822.

Número médio de movimentações: 20002.

4-Com 10000 elementos:

Com apenas a chave

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 23.

Tempo de execução médio: 0.000002.

Número médio de comparações:65432.

Número médio de movimentações:50134.

Com a chave e as strings

Tempo de execução médio: 0.000005.

Número médio de comparações:65432.

Número médio de movimentações:49023.

5-Com 200000 elementos:

Com apenas a chave

Tempo de execução médio: 0.00002.

Número médio de comparações:89442719.

Número médio de movimentações:80231453.

Com a chave e as strings

Tempo de execução médio: 0.000004.

Número médio de comparações:89442719.

Número médio de movimentações:81150431.

Conclusão: O tempo para a ordenação ser feita em geral é mais rápida em comparação a ordenação pro “seleção” e por “inserção”, e os números de comparações são grandes também, algoritmo simples de entender a logica porem um pouco mais complexo para se por em pratica .

Quicksort:

A complexidade desse algoritmo é de $O(n \log n)$.

1-Com 20 elementos:

Com apenas a chave

Tempo de execução médio: 0.0000004.

Número médio de comparações:92.

Número médio de movimentações:73.

Com a chave e as strings

Tempo de execução médio: 0.0000004.

Número médio de comparações:92.

Número médio de movimentações:62.

2-Com 500 elementos:

Com apenas a chave

Tempo de execução médio: 0.0000008.

Número médio de comparações:3107302.

Número médio de movimentações:234505.

Com a chave e as strings

Tempo de execução médio: 0.0000005.

Número médio de comparações:3107302.

Número médio de movimentações:224321.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 25.

3-Com 5000 elementos:

Com apenas a chave

Tempo de execução médio: 0.000003.

Número médio de comparações:42585854.

Número médio de movimentações:4141230.

Com a chave e as strings

Tempo de execução médio: 0.0000005.

Número médio de comparações:42585854.

Número médio de movimentações:3251250.

4-Com 10000 elementos:

Com apenas chave

Tempo de execução médio: 0.0000025.

Número médio de comparações:92101201.

Número médio de movimentações: 8120352.

Com a chave e as strings

Tempo de execução médio: 0.0000004.

Número médio de comparações:92101201.

Número médio de movimentações: 90163512.

5-Com 200000 elementos:

Com apenas chave

Tempo de execução médio: 0.0000035.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 26.

Número médio de comparações: 2441213429.

Número médio de movimentações: 2323416234.

Com a chave e as strings

Tempo de execução médio: 0.000026 .

Número médio de comparações: 2441213429.

Número médio de movimentações: 2124325719.

Conclusão: O número de comparações foi grande e os de movimentações também, porém nesse algoritmo com números maiores de itens não houve a demora dos outros algoritmos ou ocorreu de o computador onde os testes foram feitos de “travar”, algoritmo simples de entender e fácil de aplicar .

Heapsort:

A complexidade da heap sort é $O(n \log n)$.

1-Com 20 elementos:

Com apenas chave

Tempo de execução médio: 0.000003.

Número médio de comparações:228 .

Número médio de movimentações: 132.

Com a chave e as strings

Tempo de execução médio: 0.0000003.

Número médio de comparações:228.

Número médio de movimentações: 121.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 27.

2-Com 500 elementos:

Com apenas chave

Tempo de execução médio: 0.0000003.

Número médio de comparações:125748 .

Número médio de movimentações: 100235.

Com a chave e as strings

Tempo de execução médio: 0.000003 .

Número médio de comparações:125748.

Número médio de movimentações: 112027.

3-Com 5000 elementos:

Com apenas chave

Tempo de execução médio: 0.000003.

Número médio de comparações: 12502498.

Número médio de movimentações: 10205200.

Com a chave e as strings

Tempo de execução médio: 0.0000006.

Número médio de comparações: 12502498.

Número médio de movimentações: 1140139.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 28.

4- Com 10000 elementos:

Com apenas chave

Tempo de execução médio: 0.000005.

Número médio de comparações: 50004998.

Número médio de movimentações:42344248.

Com a chave e as strings

Tempo de execução médio: 0.000002.

Número médio de comparações: 50004998.

Número médio de movimentações:40201700.

5-Com 200000 elementos:

Com apenas chave

Tempo de execução médio: 0.000005.

Número médio de comparações: 20000200090.

Número médio de movimentações:172324240.

Com a chave e as strings

Tempo de execução médio: 0.000006.

Número médio de comparações: 20000200090.

Número médio de movimentações:130120251.

Conclusão: O algoritmo por ser em parte recursivo exige nas comparações logo as movimentações são grandes, e por utilizar de heaps, porem o tempo dela comparada a outras é mais rápido menos que a quicksort, da

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 29.

mesma forma onde foi testado o tempo para o algoritmo ordenar com mais itens foi mais rápido, assim o "travamento" foi mais curto, porém na criação pratica do algoritmo foi mais complexa comparadas as outras antes dela.

Mergesort:

A complexidade desse algoritmo é de $O(n \log n)$

1- Com 20 elementos:

Com apenas chave

Tempo de execução médio: 0,0000025.

Número médio de comparações:26.

Número médio de movimentações:17.

Com a chave e as strings

Tempo de execução médio: 0.0000004.

Número médio de comparações:26.

Número médio de movimentações:15.

2-Com 500 elementos:

Com apenas chave

Tempo de execução médio: 0.0000004.

Número médio de comparações:502.

Número médio de movimentações:307.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 30.

Com a chave e as strings

Tempo de execução médio: 0.0000005 .

Número médio de comparações:502.

Número médio de movimentações:297.

3-Com 5000 elementos:

Com apenas chave

Tempo de execução médio: 0.0000001.

Número médio de comparações:8191.

Número médio de movimentações:6892.

Com a chave e as strings

Tempo de execução médio: 0.0000008 .

Número médio de comparações:8191.

Número médio de movimentações:6798.

4-Com 10000 elementos:

Com apenas chave

Tempo de execução médio: 0.0000005.

Número médio de comparações: 20479.

Número médio de movimentações: 16830.

Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 31.

Com a chave e as strings

Tempo de execução médio: 0.0000005.

Número médio de comparações: 20479.

Número médio de movimentações: 16524.

5-Com 200000 elementos:

Com apenas chave

Tempo de execução médio: 0.000003.

Número médio de comparações: 399935.

Número médio de movimentações:223432.

Com a chave e as strings

Tempo de execução médio: 0.0000007.

Número médio de comparações: 399935.

Número médio de movimentações:221420.

Conclusão: O algoritmo possui velocidade, porém na criação do algoritmo, tive bastante dificuldade na interpretação logica e pratica o algoritmo.

Conclusões gerais

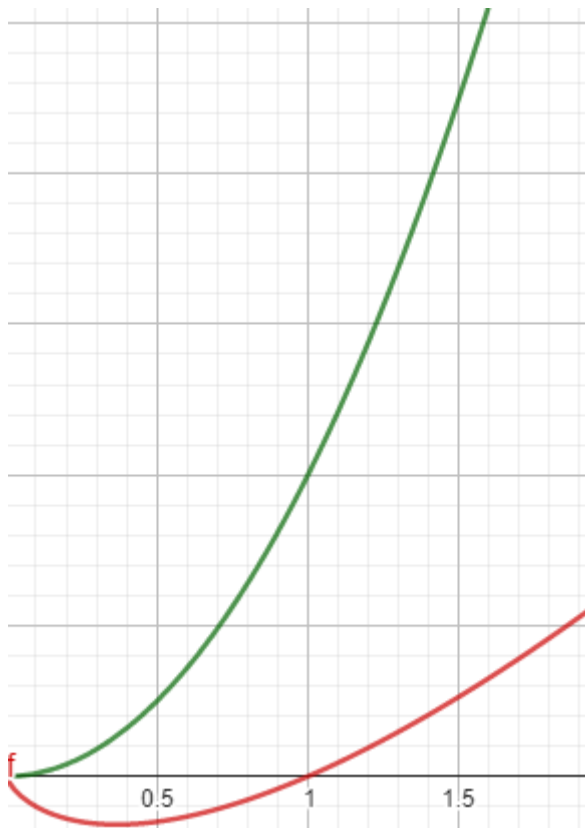
-O algoritmo melhor para grandes itens, em questão de velocidade foi o quicksort.

-Para menor número de itens o mais recomendado ao uso é a ordenação por inserção.

-A ordenação por inserção e seleção são os mais simples a respeito de logica e implementação para o programador.

-A ordenação por heapsort, exige uma quantidade de comparações e trocas maiores que as outras formas, assim não sendo recomendado para computadores com menor capacidade.

Graficos de comparações em complexidade :



Feito por : Vitor Ferreira França

< a – b> isto denota de uma certa a linha a outra b.

Pag 33.

-A linha verde corresponde aos algoritmos de inserção e seleção

-A vermelha corresponde a heapsort, quicksort, shellsort e mergesort.

Tempos médios em todos os testes de cada tipo de ordenação:

Seleção: 0.00000195.

Inserção: 0.0000023.

Shellsort: 0.00000142.

Quicksort:0.000000118

Heapsort:0.000000145.

Mergesort:0.00000607.

Dificuldade na produção do trabalho:

-As dificuldades encontradas na produção do trabalho foi a interpretação logica dos algoritmos e a implementação das mesmas em formar de código, com alguns deles como a “heapsort” e a “mergesort” sendo as mais complicadas pois trabalham com ideias novas para mim.

Bibliografias:

- <https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html>

- https://www.youtube.com/watch?v=yXluViYl-DM&ab_channel=KrishnaTeaches

- <https://www.youtube.com/watch?v=RZbg5oT5Fgw>

- https://www.youtube.com/watch?v=ECdLOLaIVx8&ab_channel=PontoAcad%C3%AAmico

Feito por : Vitor Ferreira França

$\langle a - b \rangle$ isto denota de uma certa a linha a outra b.