Segundo Trabalho Prático de Algoritmos e Estruturas de Dados II

Guilherme Morbeck Rodrigues, João Vitor Alves Morais, Luan Liduário Silva

Professor: Rafael Sachetto Oliveira

Departamento de Ciências da Computação – Universidade Federal de São João del-Rei - UFSJ – Campus Tancredo Neves

São João del-Rei – MG – Brasil

1. Introdução

O presente trabalho tem por objetivo auxiliar no aprendizado de algoritmos de ordenação vistos em aula. São eles: Ordenação por Seleção, Inserção, *Shellsort*, *Quicksort*, *Heapsort*, *Mergesort*. Espera-se que com essa atividade os alunos possam analisar número de movimentações, o tempo de execução do algoritmo, além do número de comparações feitas pelo mesmo.

As comparações foram feitas com vetores de tamanho 20, 500, 5000, 10000 e 20000. As operações foram feitas com dois tipos de vetores: vetores com registros pequenos, os quais possuíam somente um atributo (uma chave inteira) e vetores com registros grandes, com 50 campos com *strings* de 50 caracteres além de uma chave inteira. Além disso, os algoritmos deveriam receber vetores em ordem aleatória, ordem crescente e ordem decrescente.

Cada operação com os algoritmos deveria ser repetida 10 vezes e o resultado final do tempo de execução é a média do tempo entre esses testes.

1.1. Algoritmos de ordenação:

1.1.2 Ordenação por seleção:

O método de ordenação por seleção é um dos algoritmos mais simples. Ele consiste em percorrer o vetor do início ao fim procurando o menor elemento e o colocando na primeira posição. Depois disso, o vetor era percorrido novamente partindo da segunda posição em busca do menor elemento e o colocando na segunda posição. Depois o mesmo processo é repetido até que reste apenas um elemento.

Esse tipo de ordenação, teoricamente, faz C(n) = n^2/2 – n/2 comparações entre chaves e M(n) = 3(n-1) movimentações de registros.

1.1.3 Ordenação por Inserção:

O algoritmo de ordenação por inserção é feito da seguinte forma: partindo do segundo elemento, o item é comparado com a posição anterior do vetor, caso seja menor os dois trocam de lugar e o item menor é comparado com seu novo antecessor (até que o anterior não seja menor ou que seu índice seja 0), caso não seja menor o índice é incrementado.

Nesse caso temos C(n) = n^2/4 + 3n/4 – 1 comparações no caso médio e M(n) = n^2/4 + 11n/4 – 3 movimentações em teoria.

1.1.3 Ordenação Shellsort:

O Shellsort funciona assim: os itens do vetor são comparados com um elemento h posições à frente do vetor (h = h*3+1, sendo que o valor inicial de h é 1 e ele aumenta com essa fórmula até que ele chegue o mais próximo possível do tamanho do vetor, sendo menor ou igual a ele). Se o item com índice menor for maior que o elemento com índice superior, então os dois trocam de lugar. Depois que todos os elementos forem analisados em relação aos h itens à frente, h é dividido por 3 (recebendo somente a parte inteira) e o processo se repete até que h seja 1.

1.1.4 Ordenação Quicksort:

O algoritmo Quicksort trabalha particionando o vetor. O vetor é rearranjado através de um pivô x escolhido de forma arbitrária. Com isso o vetor é dividido em duas partes: a parte da esquerda com elementos com chaves menores ou iguais ao pivô e a parte da esquerda com os elementos com chaves maiores que x. O vetor é percorrido a partir do primeiro elemento até que ele seja maior que o pivô (sempre que o elemento for menor o índice da esquerda é incrementado), ao mesmo tempo ele é percorrido do último elemento (no sentido contrário, ou seja, sempre que o elemento for maior seu índice é decrementado) até que o elemento seja menor que o pivô. Quando essas duas condições acontecerem os elementos das duas posições são trocados. O processo continua até que o índice da esquerda seja maior que o índice da direita, ou seja, quando eles se cruzarem. Depois disso, cada parte (menores e maiores que o pivô) serão duas novas partições e o processo se repete para cada uma delas (novos pivôs serão escolhidos para cada uma delas). Isso se repete até que o marcador incialmente à esquerda chega à última posição do vetor e o marcador que começou na direita chegue ao primeiro índice.

1.1.5 Ordenação Heapsort:

O método de ordenação Heapsort funciona como uma árvore binária. O método parte da primeira posição que possui pelo menos um filho (sendo n o tamanho do vetor, a fórmula para se chegar nessa posição é: (n-1), é feita uma análise se a posição atual possui filhos, caso tenha os filhos são comparados para saber quem é o maior e se o maior deles é maior que o pai, se for o caso eles trocam de posição. Depois dessa análise, o índice parar percorrer o vetor é decrementado em 1 e o processo se repete até que chegue ao início do vetor. Dessa forma, o maior item do vetor estará na primeira posição ou raiz, então ele é colocado na última posição do vetor e o processo se repete com os n-1 itens restantes.

1.1.6 Ordenação Mergesort:

O algoritmo de ordenação Mergesort funciona da seguinte forma: o vetor é dividido ao meio, depois seus dois subvetores são divididos ao meio e assim por diante, até que cada posição fique sozinha. Depois desse processo as posições são reagrupadas (primeiro de dois em dois, depois de quatro em quatro, até que o vetor seja totalmente reconstruído) e a cada reagrupamento elas são colocadas em ordem.

2 . Implementação

A cada função para *structs* pequenas existe uma igual para *structs* grandes, mudando somente o tipo da *struct* de entrada.

2.1 Estruturas de Dados

Duas TADs foram criadas: uma chamada “Pequeno” e a outra “Grande”. A estrutura “Pequeno” possui como atributo somente um inteiro que chamamos de “chave”. A estrutura “Grande” também possui o atributo “chave”, mas, além disso, ela possui uma matriz de caracteres 50x50, ou seja, 50 *strings* de 50 caracteres cada. Cada uma possui funções para cada de tipo de ordenação proposta no trabalho.

2.2 Funções

Como já foi dito, para cada operação específica para *structs* “Pequeno” existe uma também para “Grande”. Em todas as funções que tem o nome de um tipo ordenação criam variáveis para guardar os números de movimentações e comparações, as incrementa quando necessário e as imprime no final.

2.2.1 void realizaOperacoes(int tipoStruct,int t,int v,int ordena)

Recebe uma variável para definir o tipo de *struct* a ser criada, uma variável com o tamanho do vetor, uma para o tipo de preenchimento do vetor e uma para o tipo de ordenação. Com as três primeiras variáveis o vetor é criado e preenchido, depois disso a função de ordenação correspondente à escolha do usuário é chamada.

2.2.2 double tempo()

Função que retorna o tempo desde 1 de janeiro de 1970.

2.2.3 Pequeno* alocarVetorPequeno(Pequeno *vetor,int t) / Grande* alocarVetorGrande(Grande *vetor,int t)

Aloca um espaço na memória para um vetor do tipo “Pequeno”.

2.2.4 int vetorPequenoVazio(Pequeno *vetor) / int vetorGrandeVazio(Grande *vetor)

Testa se o vetor está vazio ou não.

2.2.5 void printVetorPequeno(Pequeno *vetor,int t) / void printVetorGrande(Grande *vetor,int t)

Imprime na tela o vetor recebido.

2.2.6 Pequeno *preencherVetorPequeno(Pequeno *vetor,int t,int v) / Grande *preencherVetorGrande(Grande *vetor,int t,int v)

Essa função recebe um vetor, seu tamanho e uma variável para indicar o tipo de preenchimento (ordem crescente, decrescente ou aleatória). Com esses dados o vetor é preenchido e retornado.

2.2.7 void selecaoVetorPequeno (int t, Pequeno *vetor) / void selecaoVetorGrande (int t, Grande *vetor)

Recebe um vetor e seu tamanho. Cria uma variável auxiliar para fazer as trocas de posições. O vetor é percorrido a partir do primeiro até o final, comparando os elementos para achar o que tem a menor chave. Depois o menor elemento é trocado com o elemento da primeira posição. O processo se repete para os elementos restantes, até que todas as posições estejam ordenadas.

2.2.8 void insercaoVetorPequeno (int t, Pequeno *vetor) / void insercaoVetorGrande (int t, Grande *vetor)

Recebe um vetor e seu tamanho. Começa a percorrer o vetor a partir da segunda posição. Cada posição é comparada com sua anterior, caso ela seja menor ela é analisada com mais uma anterior e assim por diante até que a anterior seja menor ou igual. Isso se repete até o final do vetor.

2.2.9 void shellsortVetorPequeno(Pequeno *vetor, int t) / void shellsortVetorGrande(Grande *vetor, int t)

Recebe um vetor e seu tamanho. Duas variáveis são criadas para poder comparar posições do vetor. A distancia entre essas duas variáveis é igual a h, sendo que h começa em 1 e sua fórmula é: h*3+1. O programa entra no laço e vai incrementado h enquanto ele é menor que o tamanho do vetor. Depois ele entra em outro laço (onde h é divido por 3 para que ele fique com o maior número da sequência de Knuth que seja menor que o tamanho do vetor). Nesse outro laço as posições são comparadas e com h posições a frente, se a com índice maior for menor que a outra elas trocam de lugar. Depois que o vetor é percorrido dessa forma o h é divido por 3 até que receba valor 1.

2.2.10 QUICKSORT

Próximas funções fazem parte do tipo de ordenação *Quicksort*.

2.2.10.1 void particionarVetorPequeno(int esq, int dir,int *i, int *j, Pequeno * vetor,long int *mov,long int*comparacao) / void particionarVetorGrande(int esq, int dir,int *i, int *j, Grande * vetor,long int *mov,long int*comparacao)

Recebe em qual posição a partição começa e onde termina. I e j recebem esq e dir, respectivamente. Um pivô é definido (no caso na metade do vetor). I começa a percorrer o vetor da esquerda para direita compara sua atual posição com o pivô até encontrar um elemento maior que o pivô, o mesmo acontece com o j mas ele percorre da direita para esquerda e compara para achar um elemento menor que o pivô. Quando as duas condições são atendidas, os elementos de i e j trocam de lugar e o processo volta a acontecer até que i seja maior que j.

2.2.10.2 void ordenarVetorPequeno(int esq, int dir, Pequeno * vetor,long int *mov,long int*comparacao) / void ordenarVetorGrande(int esq, int dir, Grande * vetor,long int *mov,long int*comparacao)

Essa função chama a função particionar com o vetor que foi recebido, com o inicio e o final deles. Depois disso, ela chama a si própria duas vezes: uma com a partição que estão os elementos menores que o pivô (caso não tenha somente um elemento nessa partição) e outra com a partição dos elementos maiores que o pivô (caso não tenha somente um elemento nessa partição).

2.2.10.3 void quicksortVetorPequeno(Pequeno *vetor, int t) / void quicksortVetorGrande(Grande *vetor, int t)

Chama a função ordenar com o vetor completo a ser ordenado.

2.2.11 HEAPSORT

Próximas funções fazem parte do tipo de ordenação *Heapsort*.

2.2.11.1 void refazerVetorPequeno(int esq, int dir, Pequeno *a,long int *comparacao,long int * mov) / void refazerVetorGrande(int esq, int dir, Grande *a,long int *comparacao,long int * mov)

Pega uma posição no vetor, compara seus dois filhos (seguindo a regra árvore binária) e acha o maior entre eles (caso a variável que vai receber o índico dos filhos seja maior que o tamanho do vetor, quer dizer que a posição atual não tem filhos e a sai da função). Depois o filho maior é comparado com o pai, caso seja maior eles trocam de posição.

2.2.11.2 void construirVetorPequeno(Pequeno * a, int n,long int * comparacao,long int *mov) / void construirVetorGrande(Grande * a, int n,long int * comparacao,long int *mov)

Cria uma variável para receber o tamanho do vetor divido por 2 mais 1, enquanto essa variável for maior que 0 a variável é decrementada em 1 (assim chega na primeira posição que não possui filhos) e a função refazer é chamada com essa variável indicando o início do vetor a ser comparado.

2.2.11.3 void heapsortVetorPequeno(Pequeno *a, int n) / void heapsortVetorGrande(Grande *a, int n)

Chama a função construir com o vetor e o tamanho dele menos 1. Depois pega a primeira posição do vetor (que estará com o valor mais alto) e coloca na ultima posição. Por ultimo chama a função refazer com o início do vetor e com o tamanho menos 1, enquanto a variável que indica o final do vetor não chegar a 0.

2.2.12 MERGESORT

Próximas funções fazem parte do tipo de ordenação *Mergesort*.

2.2.12.1 void mergeVetorPequeno(Pequeno *vetor, int t,long int *mov,long int *comparacao) / void mergeVetorGrande(Grande *vetor, int t,long int *mov,long int *comparacao)

Percorre as duas metades do vetor recebido simultaneamente comparando as posições, pega a menor q coloca no vetor temporário.

2.12.2 void mergeSortVetorPequeno(Pequeno *vetor, int t,long int * mov,long int * comparacao) / void mergeSortVetorGrande(Grande *vetor, int t,long int * mov,long int * comparacao)

Pega o tamanho do vetor e divide por dois. Depois chama a si própria duas vezes, cada vez com uma metade do vetor e depois chama a função para a reconstrução do vetor ordenando-o.

4. Tabelas para Análise

4.1 Struct Pequena, Preenchimento Crescente

4.1.1 Comparações

	20	500	5000	10000	200000
SelectionSort	190	124750	12497500	49995000	19999900000
InsertionSort	19	499	4999	9999	199999
ShellSort	0	0	0	0	0
QuickSort	54	3498	51822	113631	3137875
HeapSort	49	2528	32665	70763	1863770
MergeSort	40	2216	29804	64608	1730048

4.1.2 Movimentações

	20	500	5000	10000	200000
SelectionSort	57	1497	14997	29997	599997
InsertionSort	38	998	9998	19998	399998
ShellSort	84	4914	70168	150486	4134292
QuickSort	48	1020	11808	23616	524284
HeapSort	194	7340	90960	191420	4694748
MergeSort	176	8976	123616	267232	7075712

4.1.3 Tempo (tempo dado em microssegundos)

	20	500	5000	10000	200000
SelectionSort	0,000056	0,001442	0,057072	0,138893	42,36008
InsertionSort	0,000056	0,000074	0,000166	0,000417	0,005028
ShellSort	0,000071	0,000108	0,000868	0,00178	0,035173
QuickSort	0,000062	0,000168	0,001023	0,002146	0,036678
HeapSort	0,00006	0,000425	0,00343	0,006683	0,06754
MergeSort	0,000014	0,000207	0,004752	0,006274	0,058318

4.2 Struct Pequena, Preenchimento Decrescente

4.2.1 Comparações

	20	500	5000	10000	200000
SelectionSort	190	124750	12497500	49995000	19999900000
InsertionSort	209	125249	12502499	50004999	20000099999
ShellSort	30	1804	19212	53704	1436446
QuickSort	38	3006	46834	103644	2937892
HeapSort	47	2288	31310	67725	1796665
MergeSort	48	2272	32004	69008	18077808

4.2.2 Movimentações

	20	500	5000	10000	200000
SelectionSort	57	1497	14997	29997	599997
InsertionSort	228	125748	12507498	50014998	20000299998
ShellSort	114	6718	89380	204190	5570738
QuickSort	79	1767	19309	38617	824281
HeapSort	172	6670	83324	176972	4396864
MergeSort	176	8976	123616	267232	7075712

4.2.3 Tempo (tempo dado em microssegundos)

	20	500	5000	10000	200000
SelectionSort	0,000061	0,001711	0,062831	0,157196	50,114454
InsertionSort	0,000056	0,00214	0,07137	0,1812	61,6819
ShellSort	0,000062	0,000171	0,001304	0,003244	0,043013
QuickSort	0,000063	0,000304	0,001074	0,002275	0,040121
HeapSort	0,000063	0,000361	0,00322	0,006785	0,061379
MergeSort	0,000015	0,000247	0,002851	0,005013	0,060639

4.3 Struct Pequena, Preenchimento Aleatório

4.3.1 Comparações

	20	500	5000	10000	200000
SelectionSort	190	124750	12497500	49995000	19999900000
InsertionSort	130	61288	6254265	25032709	9999525590
ShellSort	37	3975	71875	166574	6861980
QuickSort	70	4837	65560	136074	3636371
HeapSort	48	2461	32657	70451	1843018
MergeSort	65	3852	55231	120459	3272618

4.3.2 Movimentações

	20	500	5000	10000	200000
SelectionSort	57	1497	14997	29997	599997
InsertionSort	149	61787	6259264	25042708	9999725589
ShellSort	121	8889	142043	317060	10996272
QuickSort	91	3944	51357	110510	2851730
HeapSort	185	7008	87116	184129	4549931
MergeSort	176	8976	123616	267232	7075712

4.3.3 Tempo (tempo dado em microssegundos)

	20	500	5000	10000	200000
SelectionSort	0,000064	0,002217	0,051708	0,138848	42,25156
InsertionSort	0,000061	0,00107	0,049166	0,107756	31,48718
ShellSort	0,000064	0,000334	0,005175	0,008605	0,089924
QuickSort	0,000071	0,000356	0,00364	0,007309	0,060131
HeapSort	0,000076	0,00038	0,004474	0,009247	0,079331
MergeSort	0,000018	0,000669	0,004697	0,009382	0,074088

4.4 Struct Grande, Preenchimento Crescente

4.4.1 Comparações

	20	500	5000	10000	200000
SelectionSort	190	124750	12497500	49995000	19999900000
InsertionSort	19	499	4999	9999	199999
ShellSort	0	0	0	0	0
QuickSort	54	3498	51822	113631	3137875
HeapSort	49	2528	32665	70763	1863770
MergeSort	40	2216	29804	64608	1730048

4.4.2 Movimentações

	20	500	5000	10000	200000
SelectionSort	57	1497	14997	29997	599997
InsertionSort	38	998	9998	19998	399998
ShellSort	84	4914	70168	150486	4134292
QuickSort	48	1020	11808	23616	524284
HeapSort	194	7340	90960	191420	4694748
MergeSort	176	8976	123616	267232	7075712

4.4.3 Tempo (tempo dado em microssegundos)

	20	500	5000	10000	200000
SelectionSort	0,000087	0,002288	0,115021	0,264972	257,269351
InsertionSort	0,000071	0,00061	0,004896	0,011282	0,071186
ShellSort	0,000098	0,001838	0,034194	0,070002	0,718247
QuickSort	0,000123	0,000686	0,006705	0,012385	0,112649
HeapSort	0,000172	0,003352	0,048944	0,070179	0,557953
MergeSort	0,000195	0,005618	0,104916	0,156732	3,192801

4.5 Struct Grande, Preenchimento Decrescente

4.5.1 Comparações

	20	500	5000	10000	200000
SelectionSort	190	124750	12497500	49995000	19999900000
InsertionSort	209	125249	12502499	50004999	20000099999
ShellSort	30	1804	19212	53704	1436446
QuickSort	38	3006	46834	103644	2937892
HeapSort	47	2288	31310	67725	1796665
MergeSort	48	2272	32004	69008	18077808

4.5.2 Movimentações

	20	500	5000	10000	200000
SelectionSort	57	1497	14997	29997	599997
InsertionSort	228	125748	12507498	50014998	20000299998
ShellSort	114	6718	89380	204190	5570738
QuickSort	79	1767	19309	38617	824281
HeapSort	172	6670	83324	176972	4396864
MergeSort	176	8976	123616	267232	7075712

4.5.3 Tempo

	20	500	5000	10000	200000
SelectionSort	0,000102	0,00252	0,118567	0,318224	296,138932
InsertionSort	0,000188	0,059642	3,894052	16,922225	786,105194
ShellSort	0,000099	0,002629	0,034009	0,069821	1,009488
QuickSort	0,000116	0,001097	0,013759	0,021913	0,203302
HeapSort	0,000138	0,003416	0,042631	0,067897	0,579671
MergeSort	0,000214	0,008951	0,096765	0,141092	3,008321

4.6 Struct Grande, Preenchimento Aleatório

4.6.1 Comparações

	20	500	5000	10000	200000
SelectionSort	190	124750	12497500	49995000	19999900000
InsertionSort	100	67813	6250782	25168383	9999504590
ShellSort	53	3532	65284	171731	7234125
QuickSort	54	3522	47378	107662	2876941
HeapSort	125	6710	91576	198151	5261755
MergeSort	63	3869	55241	120462	3272766

4.6.2 Movimentações

	20	500	5000	10000	200000
SelectionSort	57	1497	14997	29997	599997
InsertionSort	119	68312	6255781	25178382	9998723583
ShellSort	137	8446	135452	322217	11368417
QuickSort	91	4000	51424	109716	2834583
HeapSort	191	7028	87129	184190	4550026
MergeSort	176	8976	123616	267232	7075712

4.6.3 Tempo

	20	500	5000	10000	200000
SelectionSort	0,000093	0,002491	0,127012	0,280499	252,304093
InsertionSort	0,000108	0,034567	1,746322	8,061541	57,703524
ShellSort	0,00012	0,004085	0,059854	0,127328	2,097923
QuickSort	0,0001	0,001594	0,034279	0,055879	0,544026
HeapSort	0,000139	0,002643	0,052111	0,093588	1,187418
MergeSort	0,000197	0,009542	0,078301	0,143069	2,829869

5. Análise

5.1 *Selection Sort*

Como esperado as movimentações e comparações do *SelectionSort* se mantiveram as mesmas, independentemente do tipo de preenchimento do vetor e tipo de *struct*, porem dependente da quantidade de elementos, como pode ser visto nas tabelas do tópico anterior nas colunas de vinte elementos os números de comparações e movimentações não se alteraram.

Nas tabelas de tempo ficou evidente que o tipo de ordenação por seleção é improprio para arquivos grandes, pois gasta um tempo a mais considerável em sua execução do que os demais métodos, porém pela simplicidade do código gerado é bom para ser utilizado em arquivos menores.

5.2 *Insertion Sort*

As comparações realizadas pelo *InsertSort* se mostraram dependente do tipo de preenchimento inicial do vetor, enquanto realiza apenas dezenove comparações em um vetor já ordenado de 20 elementos, já no mesmo vetor, só que decrescente, realiza mais de duzentas comparações. O mesmo ocorre para as movimentações dentro do vetor.

Como o código gerado por esse tipo de ordenação é simples, ele é ideal para vetores ordenados ou quase ordenados, pois nos vetores inicialmente preenchidos de forma crescente foi mais lento poucas vezes, em um caso foi mais lento que ShellSort utilizando struct do tipo pequeno e um vetor de tamanho de dez mil. Ou seja é o método mais rápido caso os dados já estejam ordenados.

5.3 *Shell Sort*

Pelo método de implementação do *Shell Sort* utilizado, caso o vetor já esteja ordenado não se realiza nenhuma comparação de chaves, não realiza muitas comparações em vetores decrescentes.

É um dos métodos mais rápidos até o vetor de cinco mil elementos, se mostrando ideal para ser utilizado em vetores de tamanho moderado.

5.4 *Quick Sort*

Nos testes mostrou porque é conhecido como o mais rápido, independente da ordem inicial dos elementos não realiza muitas comparações e movimentações do vetor. Foi o mais rápido nos vetores grandes decrescentes e nos preenchidos aleatoriamente, mesmo quando não é o mais rápido ficou próximo de ser.

5.5 *Heap Sort*

Realiza poucas comparações comparado aos demais, e quanto maior foi o vetor preenchido de forma aleatório suas movimentações foi ficando entre as mais baixas. Se mostrou rápido o bastante para competir com o *QuickSort* em determinados testes, sendo ideal caso for necessário um método que seja rápido de complexidade constante e não necessita de memoria adicional.

5.6 *Merge Sort*

Suas comparações e movimentações não se alteram muito quanto ao tipo de preenchimento do vetor. Foi lento nos testes com structs do tipo grande comparado com os testes de structs pequena, foi o que teve maior diferença nos resultados quando se trocou de tipo de *struct*. Por seu alto uso de memória adicional não é bom para arquivos que tem elementos grandes

5.6 Tipo de *Struct*

Como esperado, quando foi utilizado *struct* que continha uma matriz cinquenta por cinquenta de char, todas as execuções foram mais lentas comparadas com os mesmos parâmetros de tamanho e preenchimento de vetor e do tipo de ordenação. Mas ficou visível a piora no desempenho com os métodos como *MergeSort*, e em vetores maiores o *QuickSort*, por outro lado o *SelectionSort*, e o *HeapSort* não sofreram tanto quanto os outros métodos.

6. Conclusão

Ao decorrer do trabalho o grupo teve algumas dificuldades na realização dos testes, pois a função de contar o tempo de execução estava ignorando as casas após a vírgula, com o problema resolvido, os vetores de duzentos mil elementos e dez mil demandaram muito tempo em seus testes.

Foi perceptível a diferença entre os tipos de ordenação, desde a complexidade dos códigos gerados até os resultados dos testes. Outro fator relevante foi o tamanho das *structs* utilizadas, a que continha apenas a chave do tipo inteiro exigiu menos tempo nas suas movimentações dentro do vetor, enquanto a *struct* que tinha uma matriz cinquenta por cinquenta do tipo char teve uma demanda maior nas movimentações.

O grupo concluiu que para escolher o tipo de ordenação a implementar é necessário avaliar várias pontos, como o tamanho e a quantidade dos elementos, também é importante analisar a ordem inicial, ou seja se o arquivo está em ordem crescente, ou decrescente, ou se está quase ordenado ou completamente aleatório.

Referências

ZIVIANI, N. **Projeto de Algoritmos**: com implementação em PASCAL e C. 2. ed. [S.l.]: Cengage Learning.