Instituto de Engenharia - Várzea Grande

ANÁLISE DE ORDENAÇÕES

GUSTAVO DA COSTA GONÇALVES
LUCAS STEFFENS DE OLIVEIRA

Cuiabá
Março de 2017

UNIVERSIDADE FEDERAL DE MATO GROSSO
Instituto de Engenharia - Várzea Grande

Análise de Ordenações

GUSTAVO DA COSTA GONÇALVES
LUCAS STEFFENS DE OLIVEIRA

Trabalho apresentado ao Instituto de Engenharias de Várzea Grande, em comprimento das exigências legais, sob orientação do Prof. Frederico S. Oliveira, para obtenção de nota da disciplina de Estrutura de Dados.

Março de 2017

Sumário

1 INTRODUÇÃO

A palavra ordernar significa: "dispor de forma organizada; arrumar, organizar", a grande vantagem da ordenação, ou seja a organização de determinados dados é a busca com uma maior facilidade, tendo em vista que a ordem inicial depende da classificação de como será ordenado, por exemplo no caso de números, pode-se escolher entre ordem crescente ou decrescente. em computação, existe diversos metódos para a organização de dados, discutiremos quais os metódos mais utilizados.

As ordem mais utilizadas são para organizar números de maneira crescente ou decrescente, e também para ordenar letras. Muitos pensadores criaram maneiras eficientes para ordenar dados ao longo do tempo que são utilizados até hoje, existe variações atuais de tais algoritmos.A proposta desse trabalho surge em analisar qual o algoritmo mais vantajoso para determinada situação, avaliar seu desempenho em tempo de processamento, em números de comparações e números de movimentações,

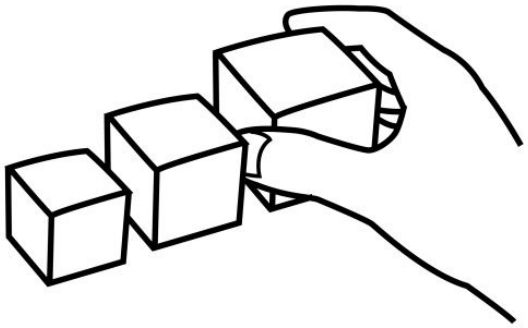

Figura 1: ORDENAÇÃO

1.1 ALGUNS ALGORITMOS UTILIZADOS

Esses algortimos apresentados são os mais utilizados, as notações do "Big - O"representam o custo de processamento do algoritmo.

1.1.1 Insertion Sort

- Método Semelhante a Ordenação de Baralho.
- Pior Caso: Ø(N)
- Melhor Caso: Ø(N^2)
- Estável
- Ø(1) Complexidade Espacial

1.1.2 Selection Sort

- Seleciona o menor elemento e coloca na posição correta
- Pior Caso: Ø(N^2)
- Melhor Caso: Ø(N^2)
- Não é estável
- Ø(1) Complexidade Espacial

1.1.3 Bubble Sort

- Vantagem: Simplicidade
- Pior Caso: Ø(N^2)
- Melhor Caso: Ø(N^2)
- Estável
- Ø(1) Complexidade Espacial

1.1.4 Merge Sort

- Método da Divisão e Conquista
- Divide o vetor recursivamente ao meio até restar um elemento no vetor, em seguida intercala os vetores
- Pior/Melhor Caso: Ø(N.log(N))
- Ø(N) Complexidade Espacial
- Estável

1.1.5 Heap Sort

- Assim Como o Selection Sort, ele seleciona o menor elemento e o insere na posição correta, utilizando uma estrutura de dados chamada Heap.
- Pior/Melhor Caso: Ø(N.log(N))
- Ø(N.log(N))Complexidade Espacial

1.1.6 Shell Sort

- Aplica o algoritmo Insertion Sort em subvetores que estão a uma amplitude H e segue reduzindo o valor da amplitude H ser igual 1.
- Melhor Caso: Ø(N.log(N))
- O valor da amplitude influência no resultado
- Não-Estável
- Caso médio: $Ø(N^{\frac{3}{2}})$ ou $Ø(N^{\frac{4}{3}})$
- Nunca foi provado matematicamente
- Ø(1) Complexidade Espacial

1.1.7 QuickSort

- Divisão e Conquista
- Melhor Caso: Ø(N.log(N))
- Pior Caso: $Ø(N^2)$

- Ordena em relação a um pivô, separando-os em duas partes: Elementos menores e maiores que o pivô. Em seguida, ordena recursivamente cada parte.
- Não-Estável
- Ø(1) Complexidade Espacial

2 DESENVOLVIMENTO

2.1 Cenário 1: Impacto de diferentes estruturas de dados

O primeiro cenário é dividido em três outros sub-cenários, onde cada um trata uma estrutura diferente e apresentam suas particularidades ao se aplicar os métodos de ordenação interna ensinados em sala de aula. Vale a pena resaltar que na análise quando se trara de comparações foram contabilizados apenas se eram testes realizados com elementos presentes no vetor e as cópias de registros foram contabilizadas três a cada troca de elemento.

2.1.1 A ordenação de vetores do tipo inteiro

Os vetores do tipo inteiro são os casos clássicos do problema de ordenação. Existem inumeros métodos para realizar esta operação, desde os simples e de fácil implementação aos mais sofisticados de implementação complexa.

Dentre os inumeros métodos foram analisados os mais populares nas cadeiras da disciplina Estrutura de Dados. Métodos estes já citados na introdução.

Os métodos foram testados para vetores de tamanho 1000 à 100000 (exceto o BubbleSort), e as quantidades de comparações, cópias de registros (Movimentações) e tempo por clock do processador foram salvos em saídas com extensão de texto (.txt), assim como o tamanho do vetor referente a execução e a semente utilizada pela função "srand"da biblioteca "time.h"para o sorteio dos números aleatórios que preenchem o vetor.

A analise de cada um dos sete métodos está listada nos subíndices abaixo.

- BubbleSort

O método do BubbleSort em português ordenação por bolha é uma das maneiras mais simples e intuitivas de se ordenar um vetor, pois seu método consiste em percorrer o vetor diversas vezes e colocar o maior elemento da sequência na posição final e no fim deste processo o vetor se encontra ordenado.Podemos notar que o método ordena com o auxilio da força bruta, uma vez que percorre inumeras vezes o vetor e realiza $\emptyset(N^2)$ comparações entre os elementos. Seu melhor caso é $\emptyset(N)$ que acontece quando o vetor se encontra ordenado. De praxe não é recomendado para vetores grandes pois é um algoritmo de ordem quadrática e tem custo computacional alto. Na análise os testes com o BubbleSort foram realizados até o tamanho de cem mil elementos, pois após essa quantia ele se torna inviável para a ordenação na máquina onde os testes foram realizados. Abaixo segue a tabela e os gráficos com todos os testes realizados na análise:

Tabela 1: Tabela da sáida gerada no testes do Bubble Sort

TAMANHO	TESTES	CÓPIAS	SEMENTE	TEMPO
1000	998001	688659	10	0
1000	998001	758391	100	0
1000	998001	768882	1000	0
1000	998001	749412	10000	0
1000	998001	768807	100000	0
1000	998001	763977	1000000	0
5000	24990001	16816575	10	0
5000	24990001	18401769	100	0
5000	24990001	18912321	1000	0
5000	24990001	18955143	10000	0
5000	24990001	18978006	100000	0
5000	24990001	18893400	1000000	0
10000	99980001	67946550	10	0
10000	99980001	73946862	100	0
10000	99980001	74402832	1000	0
10000	99980001	75643314	10000	0
10000	99980001	74686389	100000	0
10000	99980001	74274459	1000000	0
50000	2499900001	1692375558	10	14
50000	2499900001	1850540394	100	14
50000	2499900001	1871961795	1000	15
50000	2499900001	1883627457	10000	14
50000	2499900001	1872111915	100000	14
50000	2499900001	1865656203	1000000	15
100000	9999800001	6745184700	10	57
100000	9999800001	7412704137	100	57
100000	9999800001	7489244313	1000	58
100000	9999800001	7484974197	10000	58
100000	9999800001	7492374444	100000	57
100000	9999800001	7501660128	1000000	59

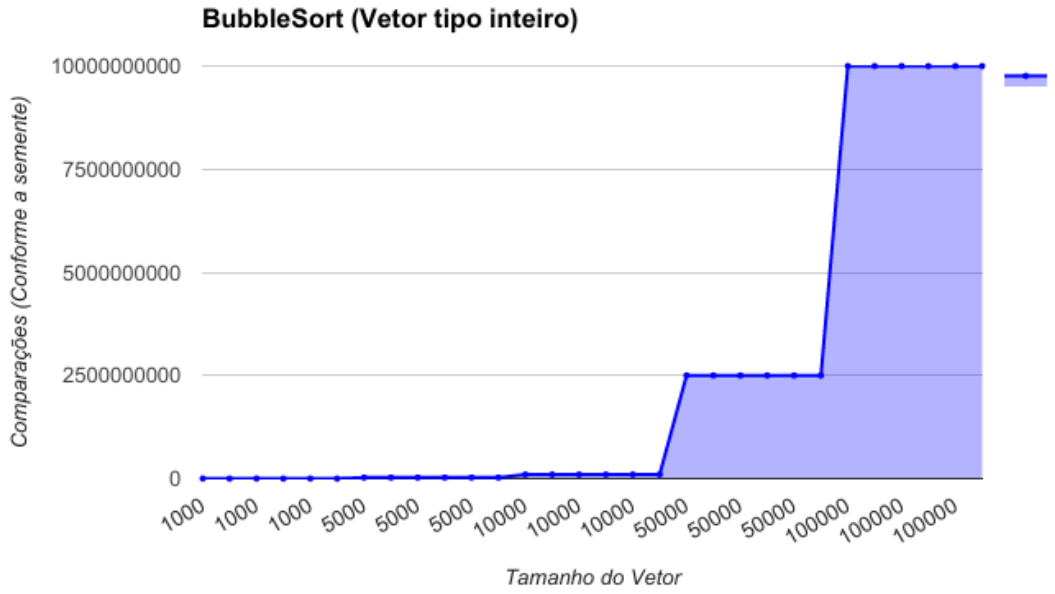

Figura 2: BubbleSort e a varição de comparações conforme o tamanho e a troca de semente.

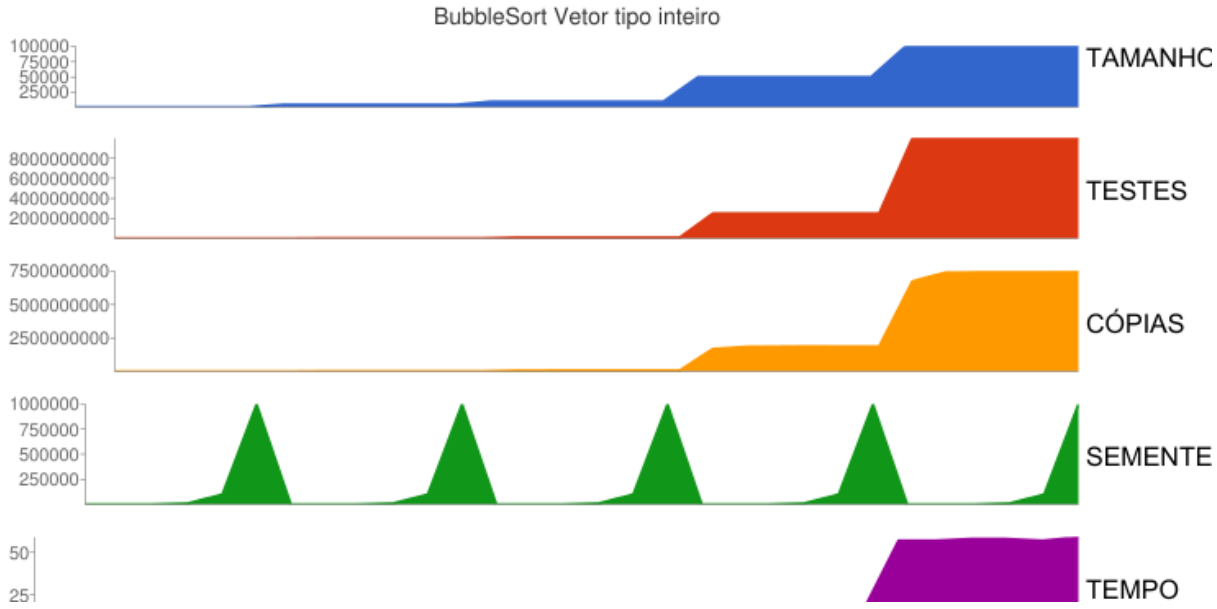

Figura 3: Gráfico referente a tabela BubbleSort e suas métricas.

Com os gráficos obtidos (Figura 2 e 3) fica claro o custo quadrático deste algoritmo e de quanto maior o vetor a ser ordenado maior a inviabilidade do método, nota-se também que ele não é sensível a mudança de semente (Figura 2) pois é um algoritmo estável e mesmo com sementes que preencheram o vetor com vários números repetidos ele não teve váriação no seu desenhepenho em número de comparações por este motivo.

O número de cópias de registro teve comportamento similar com os número de comparações apesar de ser menor, pois nem sempre que há uma comparação há copias de registro, o número de cópias é díficil de se controlar utilizando preenchimento aleatório dos vetores.É importante ressaltar que a cada movimentação foram contalibilzado três cópias de registro essas que advém das cópias realizadas pelas variavéis auxiliares pela troca em si dos elementos no vetor.

Observando os resultados obtidos oque mais chamou a atenção foi a variação de tempo do clock do processador conforme o aumento do tamanho do vetor (Figura 3). Nos casos de tamanho até mil elementos foram contabilizados zero clocks, mas partir dai começou a se tornar grande conforme o tamanho chegando a marcar 59 "segundos"de clock (processamento) para um vetor de cem mil elementos.

O Bubble Sort obteve a média aritmética de 1803666858 comparações e 178610955 cópias de registro. Esta média foi calculada somando todos os resultados obtidos na saída e dividida pelo número total de vetores,ou seja é a media no intervalo de mil a cem mil elementos com diferentes sequências.

- SelectionSort

O método do SelectionSort em português ordenação por seleção é uma das maneiras mais simples de ordenar um vetor, pois seu método consiste em percorrer o vetor diversas vezes e colocar o menor elemento da sequência na posição inicial e no fim deste processo o vetor se encontra ordenado. A cada vez que o algoritmo é executado ele compara N-1 vezes assim evitando de comparar os elementos que já estão na posição correta, mas mesmo assim tem o custo $\emptyset(N^2)$ comparações entre os elementos.

Não possui melhor caso, a condição do vetor já estar ordenado não evita que o algoritmo realize N^2 comparações.

Apesar de ser um método instável ele não apresentou sensibilidade a váriação da semente(Figura 4), mesmo quando os elementos do vetor provavelmente continham vários elementos repetidos. Foi confirmado que idependente da ordem que se encontram os elementos é realizado um número fixo de comparações e cópias de registro como era esperado deste método utilizando o preenchimento aleatório, pois quase sempre haverá trocas quando não se tem o controle de quais números compõe o vetor. Para tamanhos muito grandes as comparações não chegaram perto de seu custo usual, mas mesmo assim foram números altos que geram grandes custos computacionais.Quanto a váriação do tempo por clock do processador o algoritmo se mostrou uma boa opção para vetores de até dez mil elementos, isto levando em conta sua fácil implementação. O algortimo se sobressai quanto ao método BubbleSort que se tornou inviavel para vetores acima de cem mil elementos.

Obteve a média aritmética de 90187452429 comparações e 71428 cópias de registro. Esta média foi calculada somando todos os resultados obtidos na saída e dividida pelo número total de vetores,ou seja é a media no intervalo de mil a um milhão de elementos com diferentes sequências. Abaixo segue a tabela e os gráficos com todos os testes realizados na análise:

Tabela 2: Tabela da sáida gerada no testes do Selection Sort

TAMANHO	TESTES	CÓPIAS	SEMENTE	TEMPO
1000	499500	2997	10	0
1000	499500	2997	100	0
1000	499500	2997	1000	0
1000	499500	2997	10000	0
1000	499500	2997	100000	0
1000	499500	2997	1000000	0
5000	12497500	14997	10	0
5000	12497500	14997	100	0
5000	12497500	14997	1000	0
5000	12497500	14997	10000	0
5000	12497500	14997	100000	0
5000	12497500	14997	1000000	0
10000	49995000	29997	10	0
10000	49995000	29997	100	0
10000	49995000	29997	1000	0
10000	49995000	29997	10000	0
10000	49995000	29997	100000	0
10000	49995000	29997	1000000	0
50000	1249975000	149997	10	8
50000	1249975000	149997	100	8
50000	1249975000	149997	1000	8
50000	1249975000	149997	10000	8
50000	1249975000	149997	100000	8
50000	1249975000	149997	1000000	8
100000	4999950000	299997	10	32
100000	4999950000	299997	100	31
100000	4999950000	299997	1000	31
100000	4999950000	299997	10000	31
100000	4999950000	299997	100000	30
100000	4999950000	299997	1000000	30
500000	124999750000	1499997	10	825
500000	124999750000	1499997	100	819
500000	124999750000	1499997	1000	795
500000	124999750000	1499997	10000	838
500000	124999750000	1499997	100000	814
500000	124999750000	1499997	1000000	807
1000000	499999500000	2999997	10	3288
1000000	499999500000	2999997	100	3241
1000000	499999500000	2999997	1000	3302
1000000	499999500000	2999997	10000	3330
1000000	499999500000	2999997	100000	3247
1000000	499999500000	2999997	1000000	2821

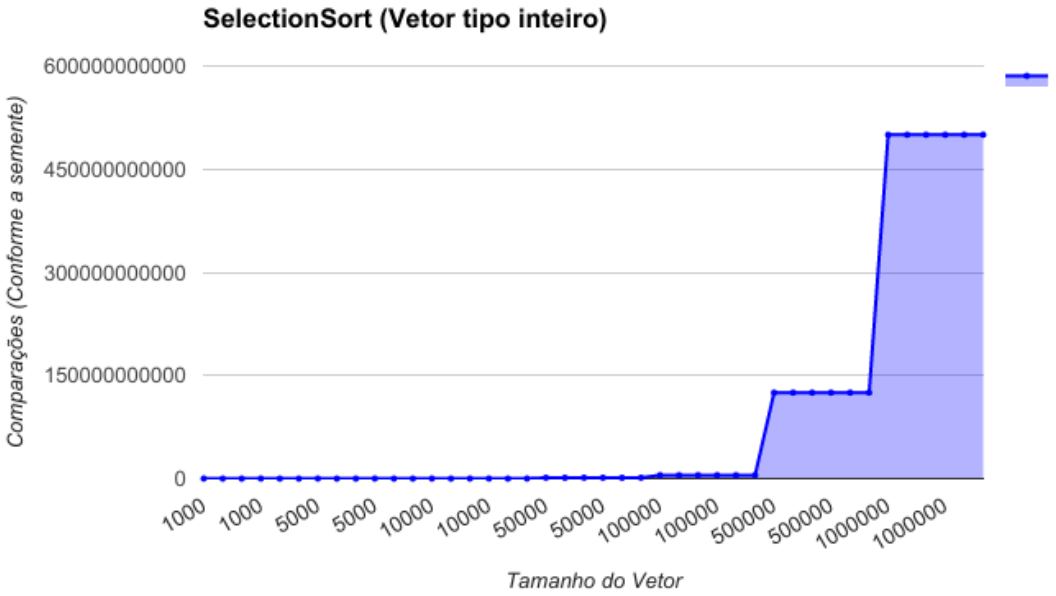

Figura 4: SelectionSort e a varição de conforme o tamanho e a troca de semente.

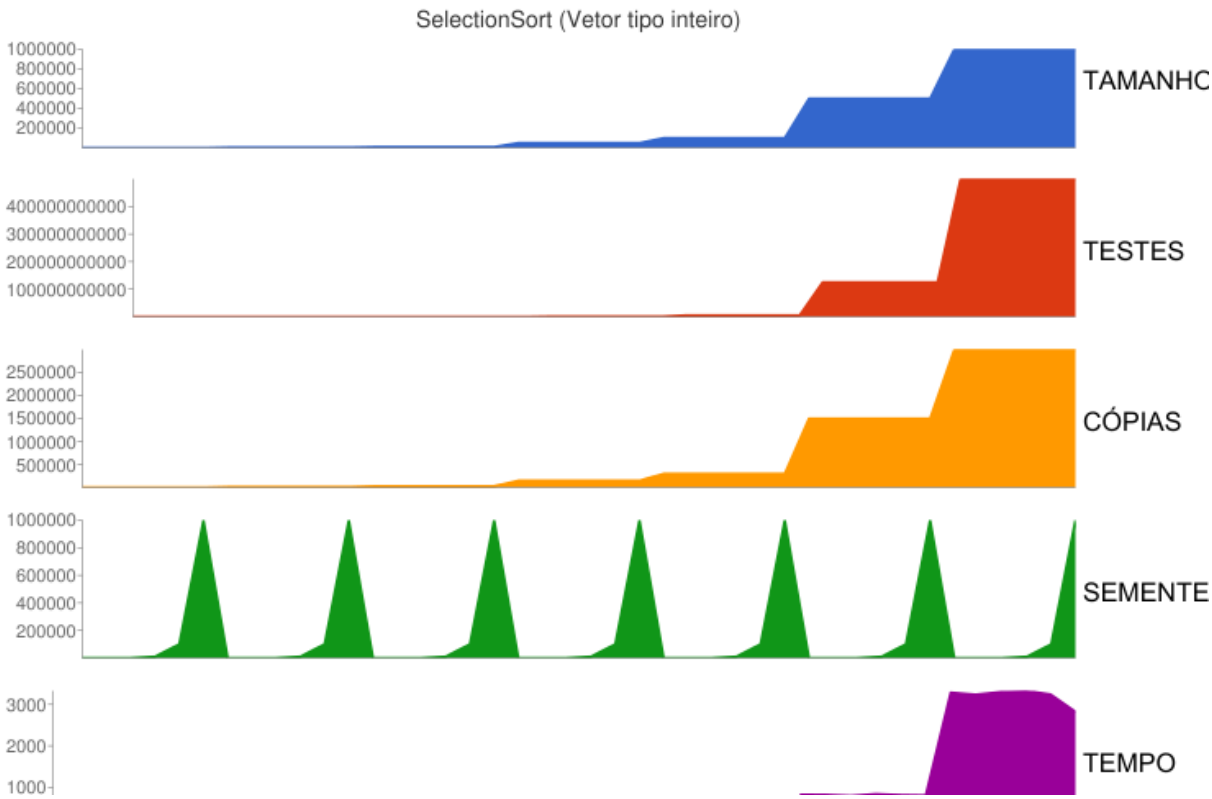

Figura 5: Gráfico referente a tabela SelectionSort e suas métricas.

- InsertionSort

O método do InsertionSort em português ordenação por inserção é também um dos métodos mais simples de ordenção, pois consiste em percorrer o vetor a partir da segunda posição e comparar com o elemento da esquerda, caso este elemento for maior é realizado a troca entre os elementos. É feito isto até que esta condição seja falsa com todos os elementos presentes no vetor, no final disto é garantido que o vetor está ordenado. Seu comportamento acaba sendo quádratico por realizar comparações com todos os elementos presentes no vetor logo seu pior caso é $\emptyset(N^2)$, e o melhor $\emptyset(N)$ que acontece quando o vetor está ordenado.

Não apresentou sensibilidade significativa a váriação da semente(Figura 6), mesmo quando os elementos do vetor provavelmente eram elementos repetidos. Se mostrou muito eficiênte para vetores com até dez mil elementos apesar do número alto de comparações e cópias de registros, mas foi considerado bom pois sua codificação idependente da linguaguem é simples.

Fez menos testes e cópias do que era esperado de seu "Ozão"de custo computacional mesmo com o do uso de números aleatórios,estes que a principio não alteram seu desempenho e justifica não ser sensível a troca de semente, pois se trata de um método éstavel.

Quanto ao tempo de execução, a partir de dez mil elementos o tempo de processamento aumentou, chegando até 1617 "segundos"de clock quando executou para um vetor de tamanho um milhão.Oque o faz ruim para vetores grandes, mas se sobresai comparando ao método BubbleSort que não terminou a execução para vetores com tamanho acima de cem mil, e se mostrou mais eficiente que o Selection no seu pior caso que contabilizou 3330 "segundos"de clock.

Obteve a média aritmética de 44265493371 comparações e 17868650653 cópias de registro. Esta média foi calculada somando todos os resultados obtidos na saída e dividida pelo número total de vetores,ou seja é a media no intervalo de mil a um milhão de elementos com diferentes sequências. Abaixo segue a tabela e os gráficos com todos os testes realizados na análise:

Tabela 3: Tabela da sáida gerada no testes do Insertion Sort

TAMANHO	TESTES	CÓPIAS	SEMENTE	TEMPO
1000	229277	687828	10	0
1000	252200	756597	100	0
1000	255882	767643	1000	0
1000	249771	749310	10000	0
1000	255334	765999	100000	0
1000	254212	762633	1000000	0
5000	5605041	16815120	10	0
5000	6131720	18395157	100	0
5000	6302219	18906654	1000	0
5000	6316797	18950388	10000	0
5000	6322766	18968295	100000	0
5000	6295443	18886326	1000000	0
10000	22643870	67931607	10	0
10000	24640892	73922673	100	0
10000	24798615	74395842	1000	0
10000	25213846	75641535	10000	0
10000	24891904	74675709	100000	0
10000	24754461	74263380	1000000	0
50000	564085042	1692255123	10	3
50000	616827780	1850483337	100	3
50000	623948246	1871844735	1000	4
50000	627809671	1883429010	10000	3
50000	624031589	1872094764	100000	4
50000	621835806	1865507415	1000000	3
100000	2248364928	6745094781	10	13
100000	2470900377	7412701128	100	15
100000	2496371782	7489115343	1000	15
100000	2494970129	7484910384	10000	15
100000	2497435611	7492306830	100000	15
100000	2500493222	7501479663	1000000	15
500000	56339127382	169017382143	10	339
500000	61897847452	185693542353	100	372
500000	62451850538	187355551611	1000	381
500000	62440871700	187322615097	10000	376
500000	62590273562	187770820683	100000	376
500000	62499823368	187499470101	1000000	375
1000000	225147673900	675443021697	10	1354
1000000	247589333082	742767999243	100	1489
1000000	249549183367	748647550098	1000	1501
1000000	249918339952	749755019853	10000	1504
1000000	249992799714	749978399139	100000	1503
1000000	250161109139	750483327414	1000000	1617

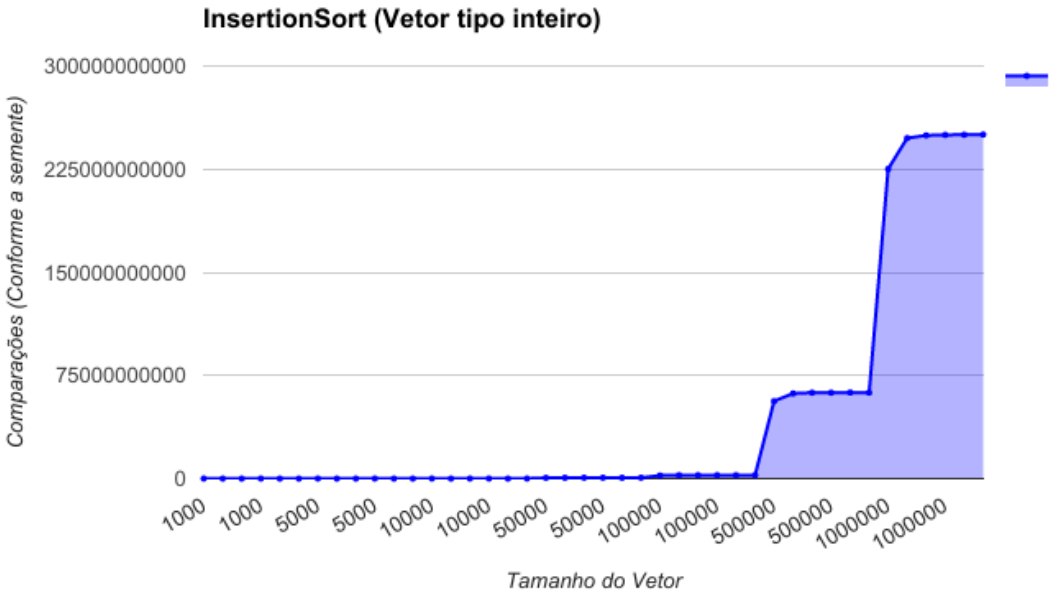

Figura 6: InsertionSort e a varição de comparações conforme o tamanho e a troca de semente.

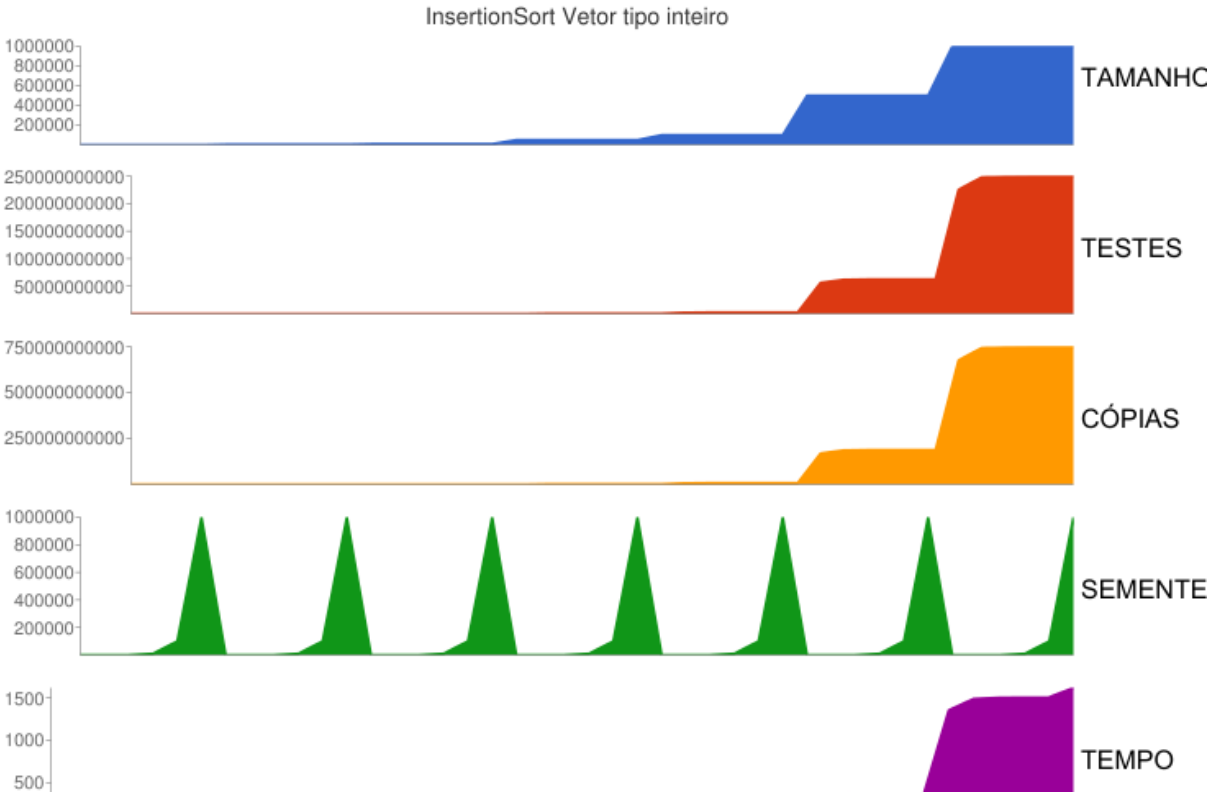

Figura 7: Gráfico referente a tabela InsertionSort e suas métricas.

- ShellSort

O método do ShellSort em que foi dado o nome em homenagem ao seu criador Donald Shell é um aperfeiçoamento do método InsertionSort, que ao invés de comparar apenas os elemetos adjacentes, trabalaha utilizando uma amplitude fazendo com que ocorram menos comparações e cópias de registro, no fim do ciclo há uma redução da amplitude e isto faz com que todos sejam testados e no fim estejam ordenados. Apenas o fato de comparar primeiramente itens que estejam mais longe faz que quando a amplitude diminua aconteçam poucas comparações e cópias pois alguns itens já estarão ordenados facilitando o método base InsertionSort.

Sua implementação já pode ser considerada de nível médio, pois não é tão intuitivo e necessita do conhecimento do método InsertionSort.

Diferente dos outros apresentados até o momento o método se mostrou bastante sensível a troca de semente (Figura 8), isto nos mostra que a utilização da amplitude teve maiores dificuldades quando tratou vetores com menos elemetos repetidos tendo que realizar mais testes conforme ordenava, logo também realizou mais cópias de registros, estas que foram proporcionais ao aumento das comparações.

Na implementação que resultou esta analise apenas cumpriu com o "Ozão"nos casos onde a semente era um número pequeno, estes que geram números repetidos que facilitaram a execução deste algoritmo. Era esperado um comportamento de Ø(N.log(N)) para todas execuções, oque não aconteceu vetores com menor quantidade de números repetidos.

Quanto ao tempo de execução ele se mostrou bastante eficaz em todos os tamanhos testados contabilizando zero "segundos"de clock, sobresaindo sobre os outros métodos apresentados.

Ao se comparar os algoritmos analisados até o momentos se mostrou o melhor apesar da sensibilidade a troca de semente.Teve a média aritmética de 7203651 comparações e 3875948 cópias de registro. Esta média foi calculada somando todos os resultados obtidos na saída e dividida pelo número total de vetores,ou seja é a media no intervalo de mil a um milhão de elementos com diferentes sequências.Abaixo segue a tabela e os gráficos com todos os testes realizados na análise:

Tabela 4: Tabela da sáida gerada no testes do Shell Sort

TAMANHO	TESTES	CÓPIAS	SEMENTE	TEMPO
1000	3101	9300	10	0
1000	6990	20967	100	0
1000	9773	29316	1000	0
1000	8441	25320	10000	0
1000	9212	27633	100000	0
1000	8828	26481	1000000	0
5000	16737	50208	10	0
5000	45705	137112	100	0
5000	58707	176118	1000	0
5000	66326	198975	10000	0
5000	68873	206616	100000	0
5000	72770	218307	1000000	0
10000	36661	109980	10	0
10000	97711	293130	100	0
10000	138559	415674	1000	0
10000	159993	479976	10000	0
10000	165105	495312	100000	0
10000	160899	482694	1000000	0
50000	187175	561522	10	0
50000	614033	1842096	100	0
50000	1033007	3099018	1000	0
50000	1174330	3522987	10000	0
50000	1165546	3496635	100000	0
50000	1392077	4176228	1000000	0
100000	390859	1172574	10	0
100000	1134380	3403137	100	0
100000	2205709	6617124	1000	0
100000	2943415	8830242	10000	0
100000	2759417	8278248	100000	0
100000	2773514	8320539	1000000	0
500000	1960012	5880033	10	0
500000	6544433	19633296	100	0
500000	15301060	45903177	1000	0
500000	19871272	59613813	10000	0
500000	21923756	65771265	100000	0
500000	20396860	61190577	1000000	0
1000000	3876927	11630778	10	0
1000000	12967277	38901828	100	0
1000000	29755804	89267409	1000	0
1000000	47124516	141373545	10000	0
1000000	49660288	148980861	100000	0
1000000	54263275	162789822	1000000	0

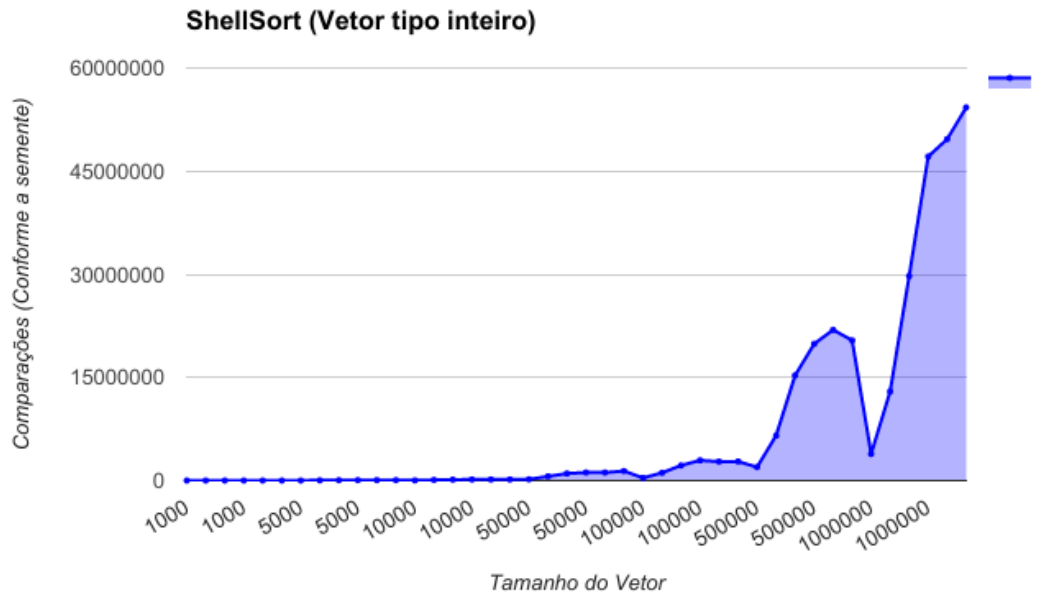

Figura 8: ShellSort e a varição de comparações conforme o tamanho e a troca de semente.

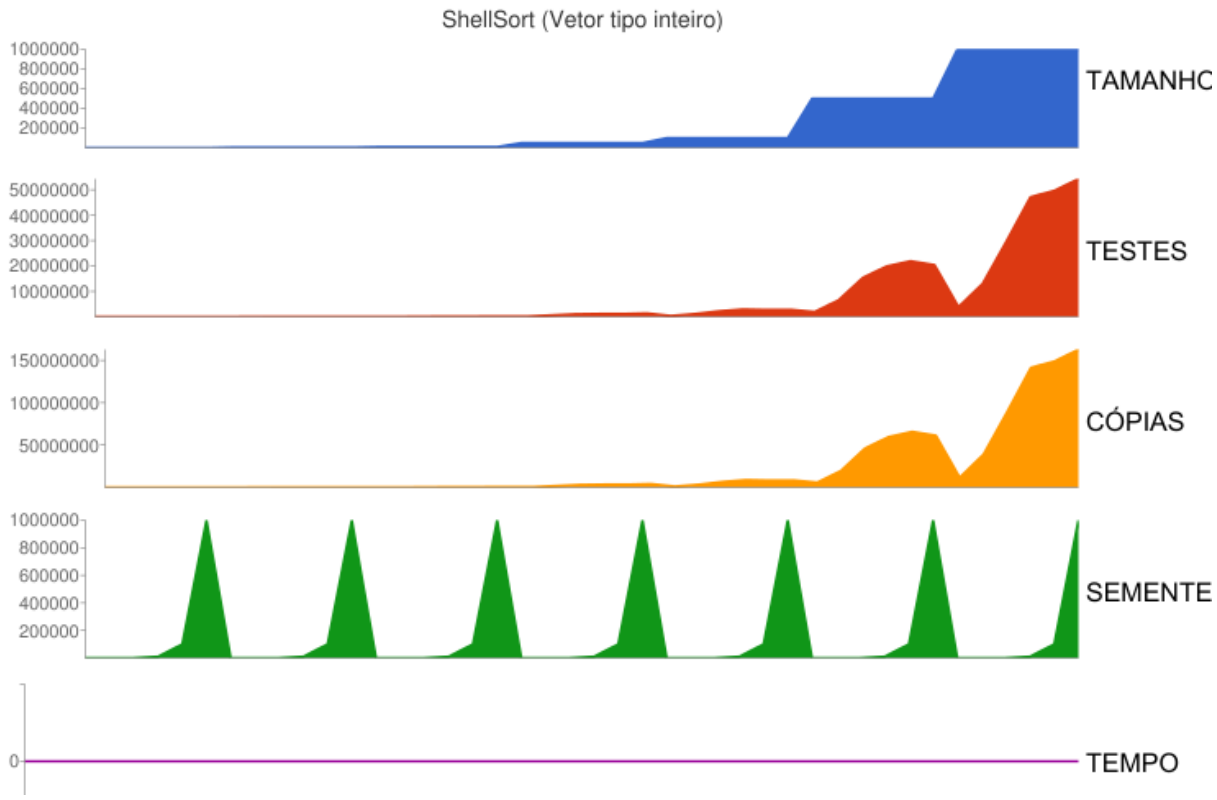

Figura 9: Gráfico referente a tabela ShellSort e suas métricas.

- HeapSort

O método HeapSort utiliza o príncipio de ordem de prioridade, e utiliza a estrutura Heap para realizar a ordenação. Pertence a família dos algortimos que utilizam recursão para realizar a ordenação, possui custo médio de Ø(N.log(N)) para todos os casos, é indicado para quando não pode-se tolerar variações de custo computacional. Sua implementação não tão simples pois necessita de conhecimento sobre recursão e também do funcionamento da estrutura heap, esta que tem comportamento parecido com uma árvore bínaria.

Nos testes reaizados o algoritmo não se mostrou muito sensível a mudança de semente (Figura 10) apesar de ser um método instável. Mas apesar disto demonstrou comportamento pior que o esparado no quesito comparações e cópias de registro, estrapolando seu "Ozão"deste os pequenos casos. Mas na prática realizou a execução rapidamente, tanto que registrou zero "segundos"de clock para seu processamento.

Ao compararar com os métodos até agora discutidos se mostrou o mais eficaz pois apesar de não registrar nenhum "segundo"de clock como o ShellSort ele realizou menos comparações no entanto realizou mais cópias de registros por utilizar a estrutura heap que a cada ordenção acaba realizando cópias "extras"para se reogarnizar.

Este método obteve a média aritmética de 5918459 comparações e 1360546 cópias de registro. Esta média foi calculada somando todos os resultados obtidos na saída e dividida pelo número total de vetores,ou seja é a media no intervalo de mil a um milhão de elementos com diferentes sequências.Abaixo segue a tabela e os gráficos com todos os testes realizados na análise:

Tabela 5: Tabela da sáida gerada no testes do Heap Sort

TAMANHO	TESTES	CÓPIAS	SEMENTE	TEMPO
1000	9833	24525	10	0
1000	11537	27063	100	0
1000	11641	27231	1000	0
1000	11770	27405	10000	0
1000	11662	27201	100000	0
1000	11681	27192	1000000	0
5000	64621	155253	10	0
5000	74581	170010	100	0
5000	75511	171006	1000	0
5000	75525	171027	10000	0
5000	75687	171288	100000	0
5000	75684	171270	1000000	0
10000	142107	337857	10	0
10000	164333	370116	100	0
10000	166216	372597	1000	0
10000	166035	372324	10000	0
10000	166220	372348	100000	0
10000	166291	372561	1000000	0
50000	859045	2000673	10	0
50000	992003	2195328	100	0
50000	1005205	2211225	1000	0
50000	1006155	2211129	10000	0
50000	1006753	2212761	100000	0
50000	1006532	2213046	1000000	0
100000	1841177	4254228	10	0
100000	2133926	4687935	100	0
100000	2160058	4722171	1000	0
100000	2162548	4724577	10000	0
100000	2163226	4724781	100000	0
100000	2163153	4724754	1000000	0
500000	10692315	24344907	10	0
500000	12376090	26832720	100	0
500000	12541778	27052884	1000	0
500000	12557922	27070167	10000	0
500000	12558624	27071310	100000	0
500000	12559942	27070929	1000000	0
1000000	22639593	51335037	10	0
1000000	26227734	56632962	100	0
1000000	26584800	57104235	1000	0
1000000	26616450	57143076	10000	0
1000000	26619805	57146016	100000	0
1000000	26619512	57142947	1000000	0

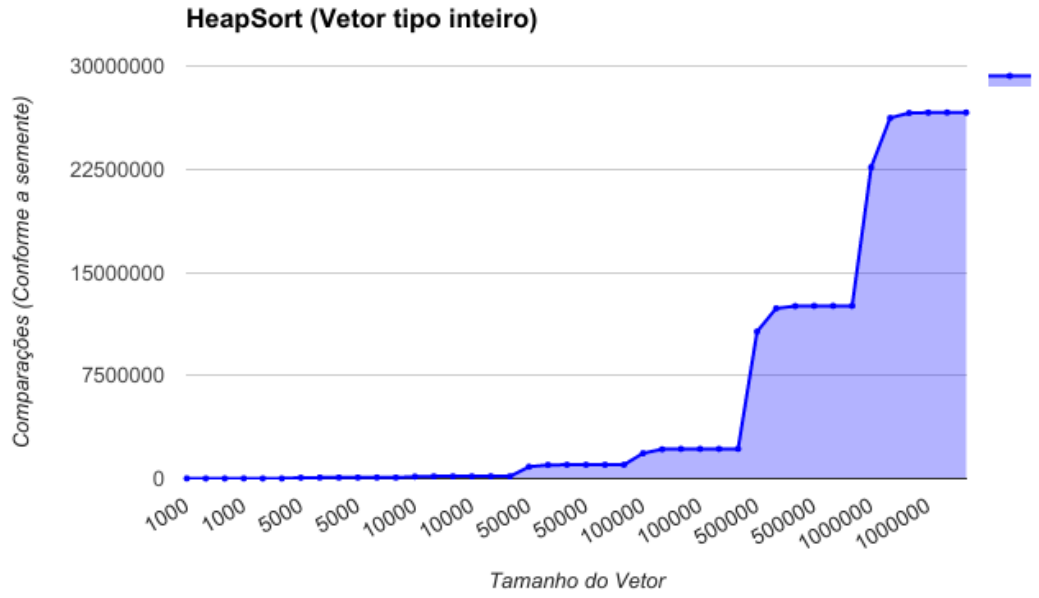

Figura 10: HeapSort e a varição de comparações conforme o tamanho e a troca de semente.

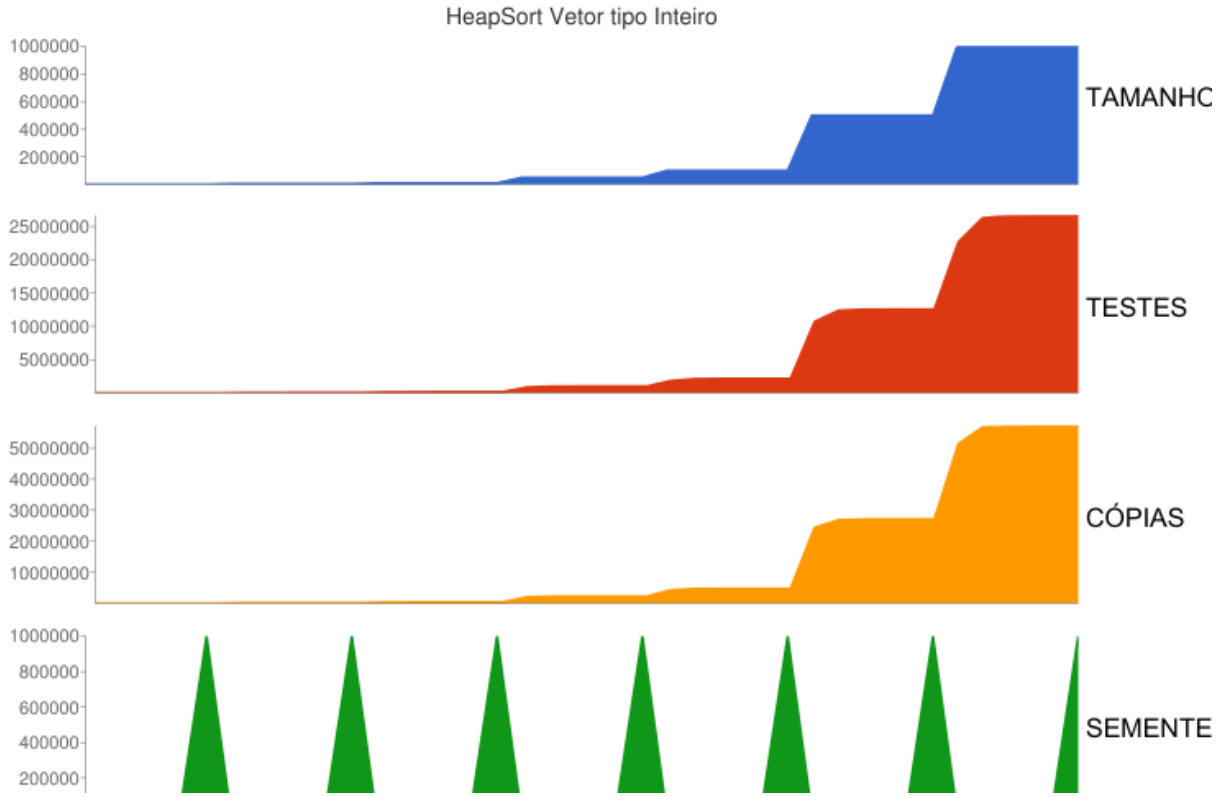

Figura 11: Gráfico referente a tabela HeapSort e suas métricas.

- MergeSort

O método do MergeSort em português ordenação por mistura, tem como princípio o método de divisão e consquista. O método divide os vetores ao meio até que não seja mais possivel realizar esta operação, logo após isto refaz o vetor mas ordenando da forma desejada, as divisões do vetor em subvetores são realizadas de forma recusiva.

Seu melhor caso é Ø(N.log(N)), que é também seu pior caso.É indicado para ordenações que não podem tolerar variações de custo computacional.

Sua implementação já pode ser considerada de nível médio alto, pois não é intuitivo e necessita do conhecimento prévio de sobre recursão.

Ao realizar a análise o algoritmo não mostrou muita sensibilidade a troca de semente (Figura 12), como era esperado sendo que seu custo esta sempre seja detro de Ø(N.log(N)) comparações, e também por se tratar de um método estavél.Logo é possivel concluir que idependente de quantos elementos são repetidos ou se o vetor já esta ordenado o custo será o mesmo, tanto para comparações quanto para cópia de registros.

Quanto ao tempo de execução ele se mostrou bastante eficaz em todos os tamanhos testados contabilizando zero "segundos"de clock, sobresaindo sobre os outros métodos apresentados pelo fato de não variar seu custo.

Obteve a média aritmética de 2117786 comparações e 475034 cópias de registro.Esta média foi calculada somando todos os resultados obtidos na saída e dividida pelo número total de vetores,ou seja é a media no intervalo de mil a um milhão de elementos com diferentes sequências.Abaixo segue a tabela e os gráficos com todos os testes realizados na análise:

Tabela 6: Tabela da sáida gerada no testes do MergeSort

TAMANHO	TESTES	CÓPIAS	SEMENTE	TEMPO
1000	3986	9987	10	0
1000	4394	9987	100	0
1000	4437	9987	1000	0
1000	4384	9987	10000	0
1000	4427	9987	100000	0
1000	4343	9987	1000000	0
5000	25858	61822	10	0
5000	27910	61822	100	0
5000	28222	61822	1000	0
5000	28283	61822	10000	0
5000	28132	61822	100000	0
5000	28213	61822	1000000	0
10000	55954	133631	10	0
10000	60762	133631	100	0
10000	61345	133631	1000	0
10000	61146	133631	10000	0
10000	61450	133631	100000	0
10000	61316	133631	1000000	0
50000	331044	784481	10	0
50000	360480	784481	100	0
50000	362798	784481	1000	0
50000	363434	784481	10000	0
50000	363235	784481	100000	0
50000	363106	784481	1000000	0
100000	706231	1668946	10	0
100000	770149	1668946	100	0
100000	775786	1668946	1000	0
100000	776499	1668946	10000	0
100000	776800	1668946	100000	0
100000	776576	1668946	1000000	0
500000	4040374	9475732	10	0
500000	4410931	9475732	100	0
500000	4444468	9475732	1000	0
500000	4447347	9475732	10000	0
500000	4447517	9475732	100000	0
500000	4446958	9475732	1000000	0
1000000	8536204	19951445	10	0
1000000	9317303	19951445	100	0
1000000	9389090	19951445	1000	0
1000000	9396025	19951445	10000	0
1000000	9395096	19951445	100000	0
1000000	9395003	19951445	1000000	0

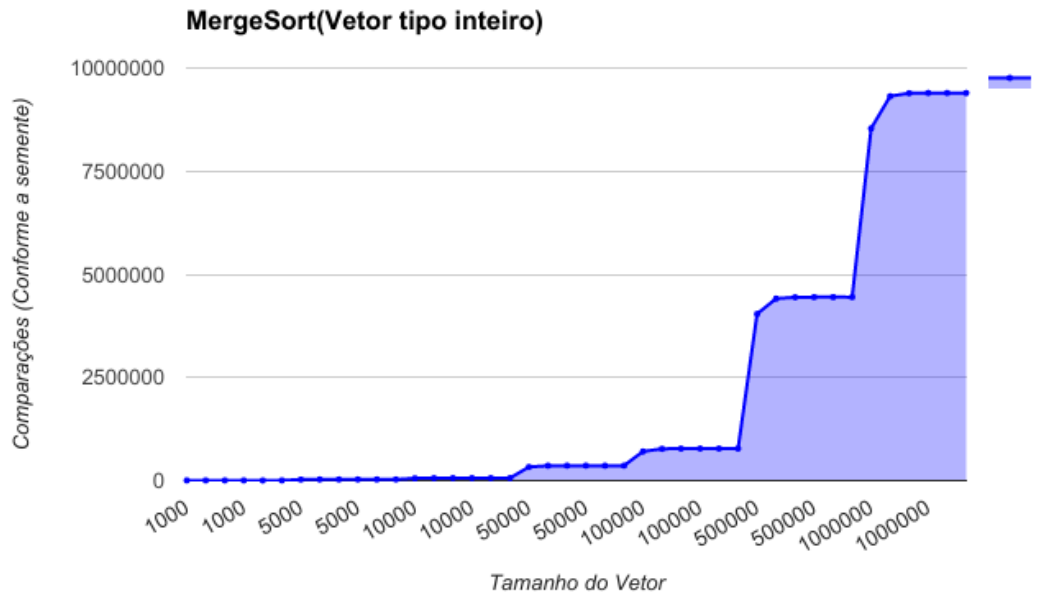

Figura 12: MergeSort e a varição de comparações conforme o tamanho e a troca de semente.

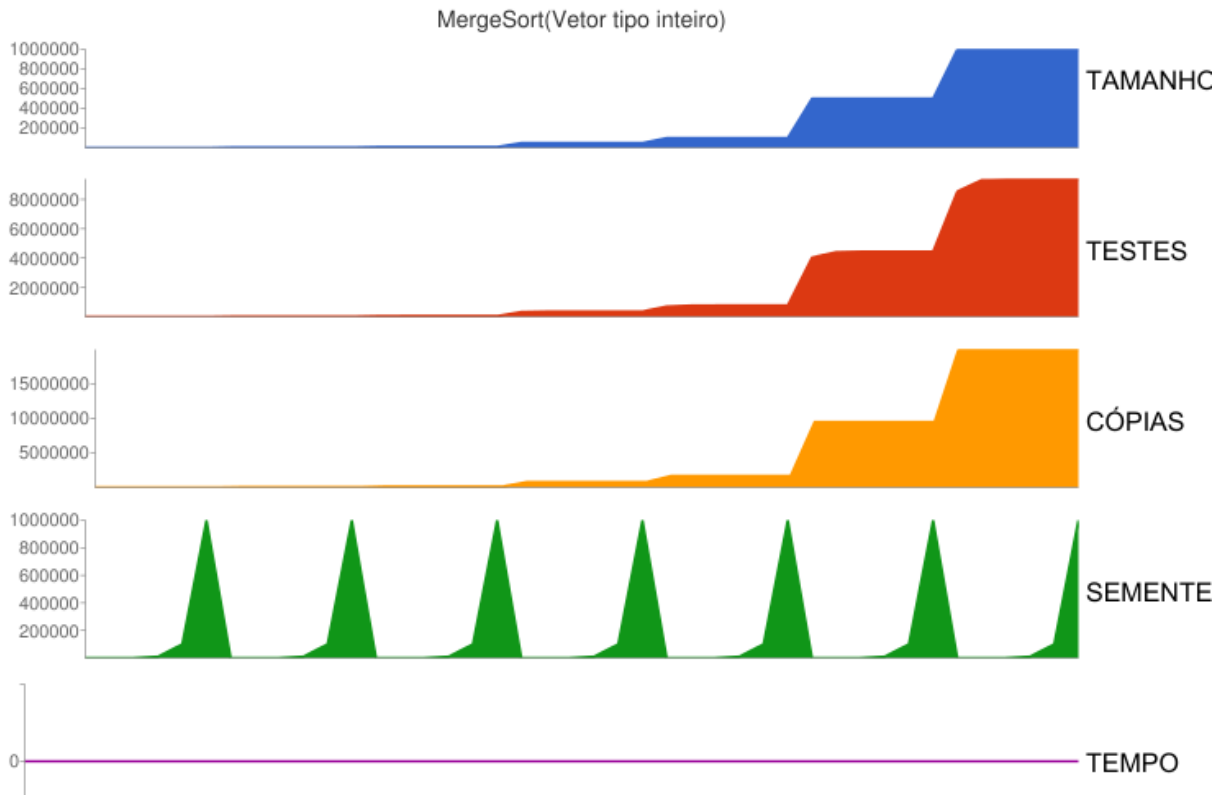

Figura 13: Gráfico referente a tabela MergeSort e suas métricas.

- QuickSort

O método QuickSort como o MergeSort utiliza o método de divisão e conquista, mas diferente do Merge que sempre divide o vetor no meio, este divide atráves da escolha de um pivô, e realiza este processo de forma recursiva até que não seja mais possivel a divisão. A escolha do pivô refelete em seu desenpenho logo é um grande desafio definir a maneira q será definida este pivô, e por isso este método possui inumeras maneiras de implementação (algumas serão discutidas no cenário 2), mas para a análise deste cenário foi escolhido a implementação passada em sala de aula que também é a mais popular que é a forma recursiva.

Seu melhor caso é Ø(N.log(N)), quando um bom pivô é escolhido, seu pior caso acontece quando o pivô escolhido é um elemento das extremidades do vetor, resultando em um custo total de $Ø(N^2)$.

Sua implementação já pode ser considerada de nível médio alto, pois não é intuitivo e necessita do conhecimento prévio de sobre recursão.

Logo para realizar a análise a priori já se é esperado um comportamento instável uma vez que preenchido com valores aleatórios a escolha do pivô pode ser comprometida e assim fazer com que o desenpenho caia ao considerar o esperado.

Mas na prática isto praticamente não acontece e o algoritmo manteu seu desenpenho de forma média variando pouco ao custo de Ø(N.log(N)).

Foi notado que o algoritmo sofre mudanças na quantidade de comparações e cópias de registros dependendo da semente, como pode ser observado na Figura 14 , isto pode ter acontecido pelo fato do algoritmo ser instável e também pela diminuição dos termos repetidos conforme o aumento do valor da semente.

Quanto ao tempo de execução ele se mostrou muito eficaz em todos os tamanhos testados contabilizando zero "segundos"de clock, sobresaindo sobre os outros métodos menos a se comparar ao MergeSort. Intuitivamente é esperado uma soberania do QuickSort sobre os outros algoritmos de ordenação apresentados, mas essa pequena discrepância no desempenho, pode ser justificada pelo problema do pivô, que é escolhido de forma dificultosa quando se trata de elementos aleatórios, as outras váriações do QuickSort presentes no cenário 2 podem fazer esta pequena diferença mudar com um melhor tratamento na hora de escolher o pivô.

Obteve a média aritmética de 2357979 comparações e 354784 cópias de registro.Esta média foi calculada somando todos os resultados obtidos na saída e dividida pelo número total de vetores,ou seja é a media no intervalo de mil a um milhão de elementos com diferentes sequências.Abaixo segue a tabela e os gráficos com todos os testes realizados na análise:

Tabela 7: Tabela da sáida gerada no testes do QuickSort

TAMANHO	TESTES	CÓPIAS	SEMENTE	TEMPO
1000	3056	11418	10	0
1000	6461	9165	100	0
1000	8743	7851	1000	0
1000	7899	7668	10000	0
1000	7731	7833	100000	0
1000	8354	7731	1000000	0
5000	20101	73722	10	0
5000	32096	61314	100	0
5000	39980	52290	1000	0
5000	44105	47790	10000	0
5000	45743	47445	100000	0
5000	47175	47010	1000000	0
10000	29589	163368	10	0
10000	54538	138381	100	0
10000	81087	116271	1000	0
10000	96996	104094	10000	0
10000	108044	100701	100000	0
10000	107805	100176	1000000	0
50000	137017	997119	10	0
50000	340087	857922	100	0
50000	412018	738132	1000	0
50000	558058	633573	10000	0
50000	640001	592782	100000	0
50000	648762	581877	1000000	0
100000	278086	2141514	10	0
100000	590834	1869186	100	0
100000	880839	1613139	1000	0
100000	1163961	1387725	10000	0
100000	1325489	1268991	100000	0
100000	1365739	1238562	1000000	0
500000	1068906	12461208	10	0
500000	2730724	11094417	100	0
500000	4187302	9790842	1000	0
500000	6137382	8466639	10000	0
500000	7607561	7455468	100000	0
500000	7760019	7079034	1000000	0
1000000	2845404	26353461	10	0
1000000	5758314	23671065	100	0
1000000	8522094	21014475	1000	0
1000000	12698726	18376758	10000	0
1000000	13787393	16265982	100000	0
1000000	16840892	14900925	1000000	0

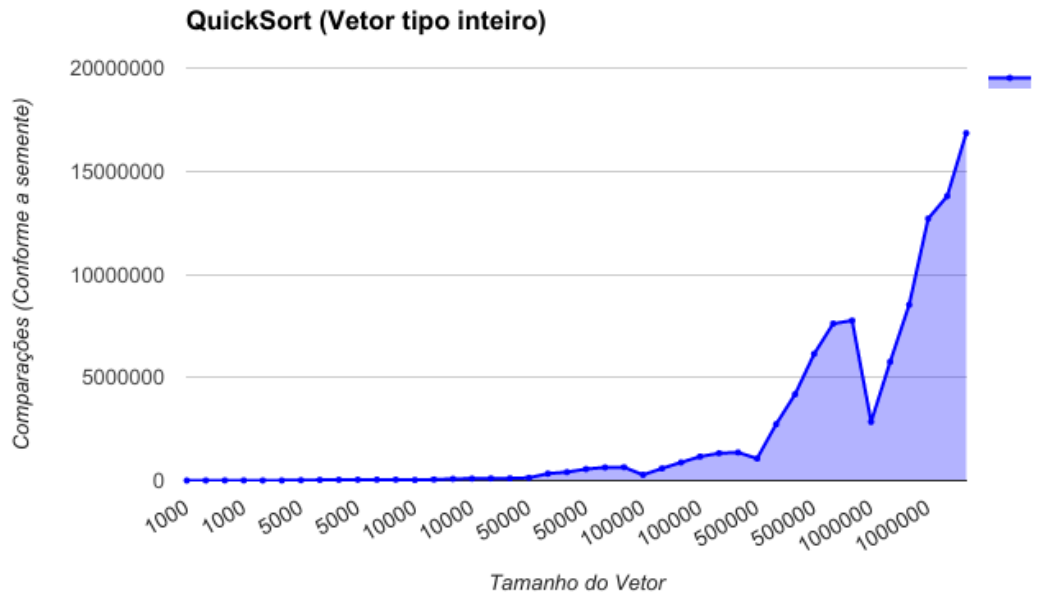

Figura 14: QuickSort e a varição de comparações conforme o tamanho e a troca de semente.

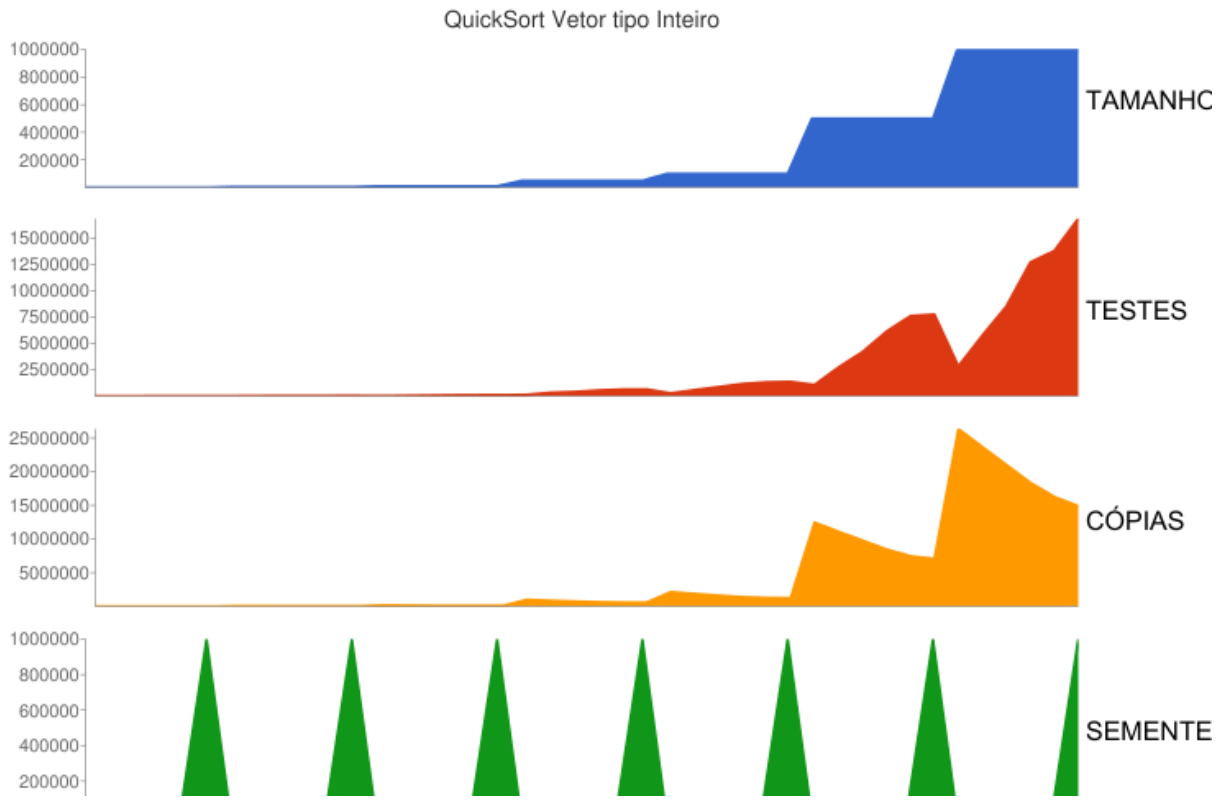

Figura 15: Gráfico referente a tabela QuickSort e suas métricas.

- Comparações entre os algoritmos apresentados neste subcenário

Sem dúvidas ao se comparar os métodos analisados acima, podemos concluir que o BubbleSort é o pior de todos, além de possuir um limite de cem mil elementos é oque possui a pior média de comparações e cópias de registros que se aproximaram da média final dos demais que realizaram para vetores de até um milhão de elementos.

O SelectionSort ficou logo em seguida do BubbleSort no quesito comparações sendo o segundo pior, mas se destacou entre todos quando se trata de cópias de registro pois seu método tende a realizar $3.(N-1)$ cópias de registros.

O InsertionSort ficou classificado como o terceiro pior método, apesar de ter realizado mais cópias de registro do que o método anterior, a diferença de comparações foi de 45921959058 que é bastante significativa, o tempo de processamento também foi um fator determinante para esta diferença.

O ShellSort como esperado realmente apresentou uma melhoria ao método InsertionSort realizando 4419345720 menos comparações e 178647747×10^1 cópias de registro considerando a média calculada.

O HeapSort já se destacou aos demais listados acima fazendo cerca de 1285192 comparações e 2515402 cópias de resgistro a menos que o ShellSort.

O QuickSort era o principal candidato a ser o melhor método, mas nos testes realizados não apresentou o desempenho esperado, isto pode ter ocorrido pelo método simplista da escolha do pivô, que pode causar um mal comportamento, logo podemos concluir que apesar de ótimo é dependente da escolha do pivô.

MergeSort acabou ficando como o melhor algoritmo de ordenação, pois cumpriu oque prometeu de ser um método estavél e que cumpre com seu "Ozão"idependente da situação em que se encontra os elementos do vetor, nos testes fez 240193 comparações a menos que o QuickSort, em número de cópias o Merge acabou realizando mais que o Quick, mas devido o fato de sempre mater um custo computacional apenas dependente do tamanho do vetor foi eleito o melhor.Abaixo segue a tabela com as médias de todos os testes.

Tabela 8: Médias de comparações e cópias de registro de cada algoritmo discutido

MÉTODO	Média TESTES	Média CÓPIAS
BubbleSort	1803666858	178610955
SelectionSort	90187452429	71428
InsertionSort	44265493371	17868650653
ShellSort	7203651	3875948
HeapSort	5918459	1360546
MergeSort	2117786	475034
QuickSort	2357979	354784

2.1.2 A ordenação de listas duplamente encadeadas

Listas duplamente encadeadas com elementos do tipo inteiro são um desafio para os métodos de ordenação apresentados em sala de aula, pois necessitam de mais cuidado e conhecimentos sobre ponteiros e TADs. Também acaba se tornando um desafio aplicar os métodos que utilizamos para vetores em listas pois de longe não é a melhor forma de se manter um lista ordenada, para isto a implementação de árvores AVL é mais eficaz. Logo inicialmente já podemos esperar menor desempenho dos algoritmos para ordenar as listas, alguns até se tornando inviáveis pela dificuldade de implemetação e o baixo desempenhos dos mesmos.

Dos métodos que foram utilizados no cenário anterior apenas foram implementados para listas duplamente encadeadas os métodos BubbleSort, SelectionSort,InsertionSort e ShellSort.

Os métodos foram testados para listas de tamanho 1000 ao limite para cada método estabelecido pela máquina que foi realizada a analise.As quantidades de comparações, cópias de registros (Movimentações) e tempo por clock do processador foram contabilizados e serão discutidos, apresentados em tabelas/gráficos. Como o desempenho de todas as implementações foi em geral péssima, e por isso foi realizado testes com a semente fixa igual a dez.

Todos as listas dos testes foram preenchidas aleatóriamente com a ajuda da função "srand"da biblioteca "time.h".

O limite de cada algoritmo foi definido após varias execuções onde por fim cada um apresentou um teto máximo de elementos.

A analise de cada um dos quatro métodos está listada nos subíndices abaixo.

- BubbleSort

O mesmo método já descrito no subcenário anterior foi aplicado, apenas foram realizadas alteraçoes na sintaxe dos códigos, utilizando uma variavel auxiliar para percorrer a lista e realizar as comparações. Não foi observado nenhuma surpresa ao realizar os testes,pois o algortimo continuou apresentando comportamento quadrático (Figuras 16 e 17) para todos os testes, estes que foram realizados com listas de tamanho 10000 à 301000 .O limite superior foi estabelecido pelo própio método, que não obteve sucesso em ordenar listas de ordem maior.

Com isto já foi perceptivél o quanto o tipo de estrutura utilizada influência no problema da ordenação.Todas as estatísticas estão apresentadas abaixo:

Tabela 9: Sáidas dos testes realizados com o método BubbleSort

TAMANHO	TESTES	CÓPIAS	SEMENTE	TEMPO
1000	230763	692286	10	0
51000	586288338	1758865011	10	26
101000	2885783115	8657349342	10	101
151000	8015722105	24047166312	10	239
201000	17105730541	51317191620	10	417
251000	31264063881	93792191640	10	618
301000	51663671538	154991014611	10	911

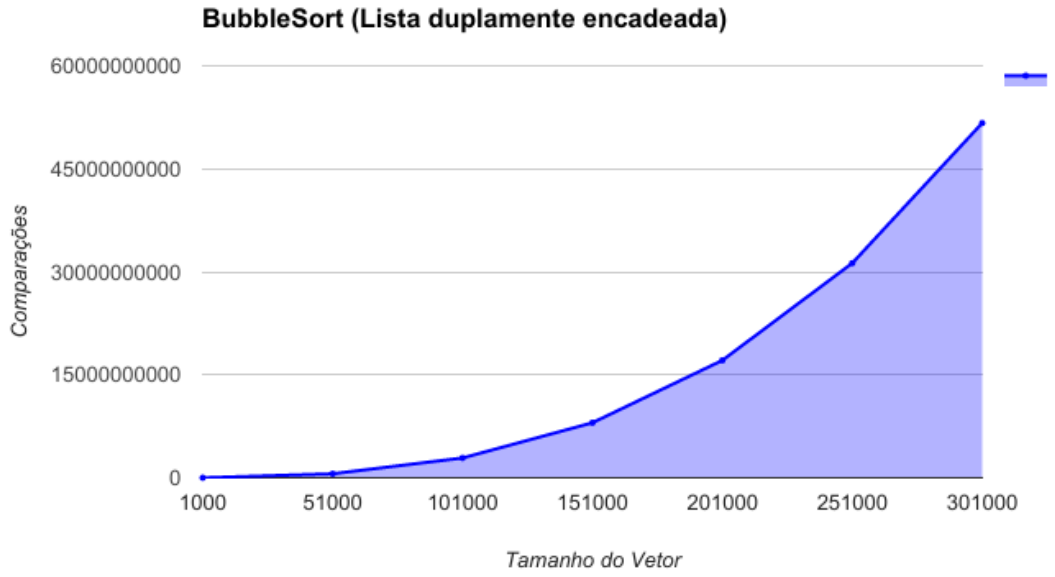

Figura 16: BubbleSort e a varição de comparações conforme o tamanho da lista.

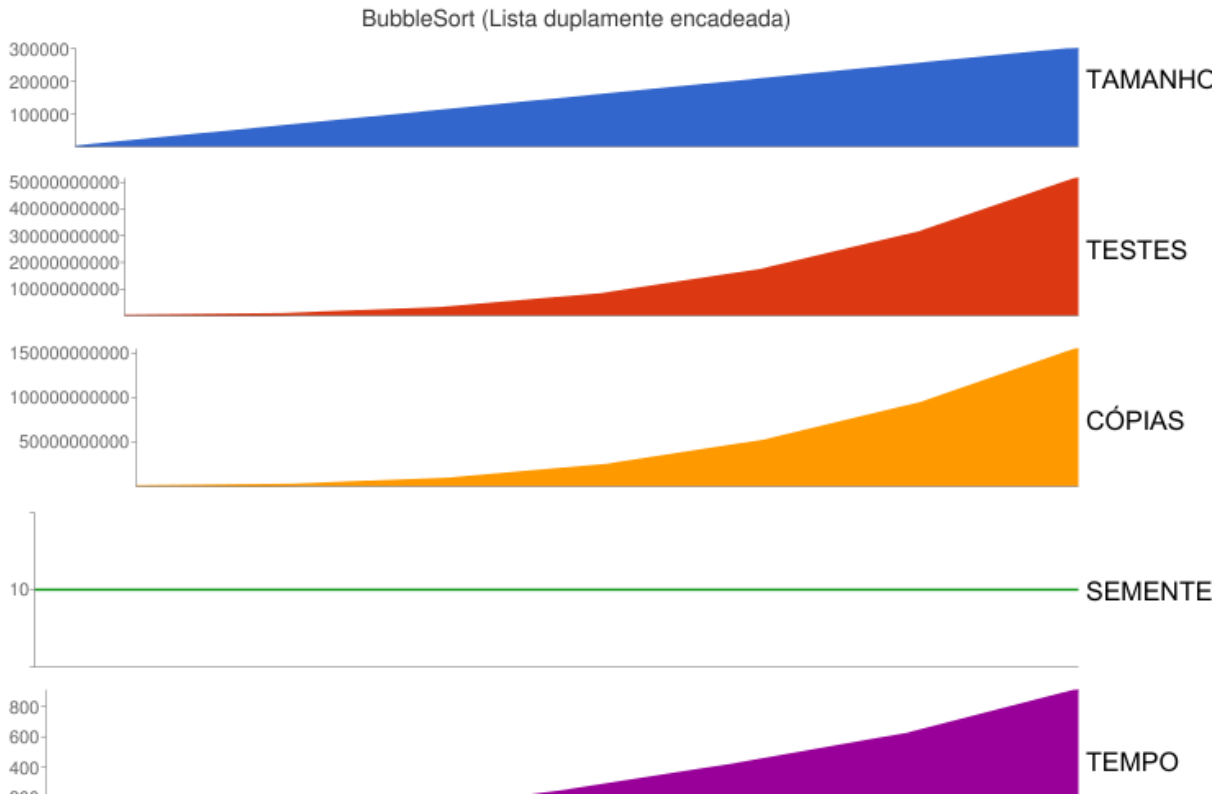

Figura 17: Gráfico referente a tabela BubbleSort e suas métricas.

Analisando os gráficos gerados podemos notar que todas as métricas exceto a semente, se tornaram exponenciais conforme o aumento do tamanho. Oque chama a antenção na Tabela 9 é o tempo de processamento, que chegou 911 "segundos"de clock do processador em seu pior caso.

Se compararmos com o cenário anterior onde o BubbleSort ordenou um vetor de cem mil elementos e levou 59 "segundos"de clock, para uma lista de cento e um mil acabou levando 101 "segundos. Isto reflete a sensibilidade do método dependendo da TAD utilizada. Outro aspecto relevante foi o fato de que na mesma máquina o método ordenou vetores do tipo inteiro com até cem mil elementos e para lista, apesar de mais processamento o algoritmo antingiu um teto de trezentos e um mil.

- SelectionSort

Como o BubbleSort o SelectionSort também foi descrito no primeiro subcenário para vetores do tipo inteiro, também foram realizados algumas mudanças na sintaxe e no método de compilação.

Não foi observado nenhuma surpresa ao realizar os testes,pois o algortimo continuou apresentando comportamento quadrático (Figuras 18 e 19) para todos os testes, estes que foram realizados com listas de tamanho 1000 ao limite superior de 451000 que foi estabelecido pelo própio método, que não obteve sucesso em ordenar listas de ordem maior.Todas as estatísticas estão apresentadas abaixo:

Tabela 10: Sáidas dos testes realizados com o método SelectionSort

TAMANHO	TESTES	CÓPIAS	SEMENTE	TEMPO
1000	498502	3000	10	0
51000	1300973002	156000	10	8
101000	6401422502	459000	10	35
151000	17801847002	912000	10	81
201000	38002246502	1515000	10	150
251000	69502621002	2268000	10	233
301000	114802970502	3171000	10	331
351000	176403295002	4224000	10	440
401000	256803594502	5427000	10	583
451000	358503869002	6780000	10	703

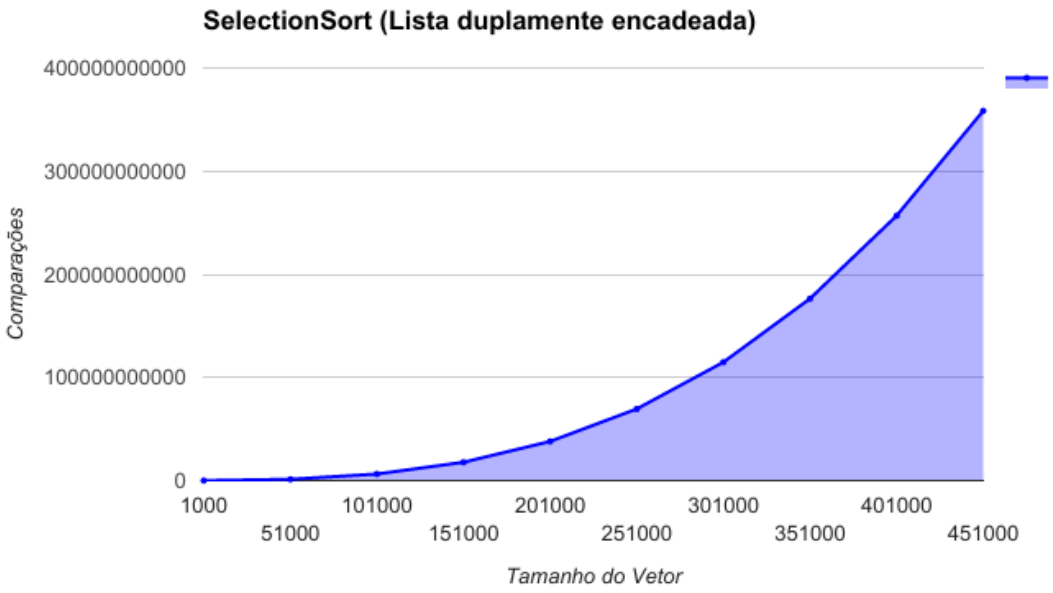

Figura 18: SelectionSort e a varição de comparações conforme o tamanho da lista.

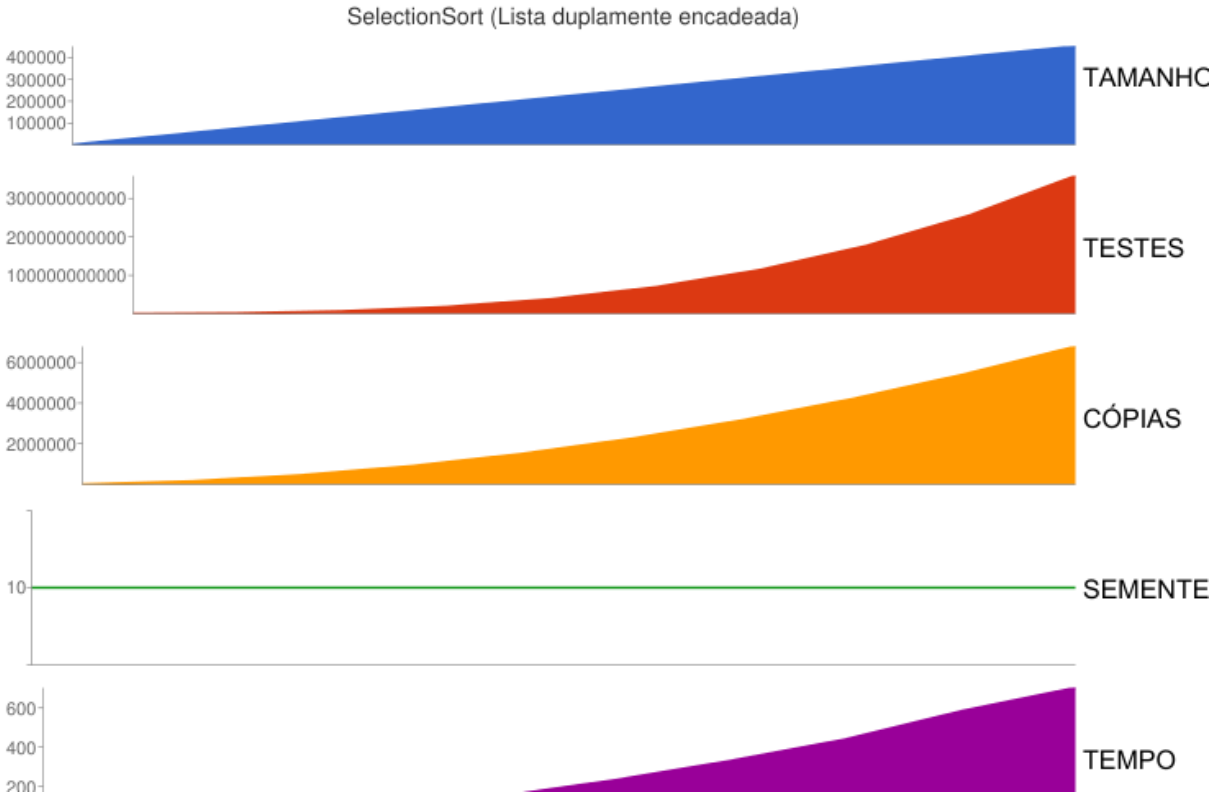

Figura 19: Gráfico referente a tabela Selection e suas métricas.

Comparado com o SelectionSort do subcenário anterior cabou se aproximando quanto custo de processamento pois para ordenar uma lista de 451000 levou 703 "segundos"de clock, e o para vetores do tipo inteiro levou 814 "segundos"para o melhor caso.

.

- InsertionSort

Método já descrito no primeiro subcenário para vetores do tipo inteiro, apenas também foram realizados algumas mudanças na sintaxe e no método de compilação.

Não foi observado nenhuma surpresa ao realizar os testes,pois o algortimo continuou apresentando comportamento quadrático para todos os testes (Figuras 20 e 21), estes que foram realizados com listas de tamanho 1000 ao limite superior de 451000 que foi estabelecido pelo própio método, que não obteve sucesso em ordenar listas de ordem maior.Todas as estatísticas estão apresentadas abaixo:

Tabela 11: Sáidas dos testes realizados com o método InsertionSort

TAMANHO	TESTES	CÓPIAS	SEMENTE	TEMPO
1000	214435	643302	10	0
51000	587159657	1761478968	10	11
101000	2897160953	8691482856	10	45
151000	8026376060	24079128177	10	105
201000	17125521328	51376563981	10	186
251000	31343564081	94030692240	10	270
301000	51715047547	155145142638	10	381
351000	79404966624	238214899869	10	506
401000	115564883017	346694649048	10	648
451000	161320733822	483962201463	10	689

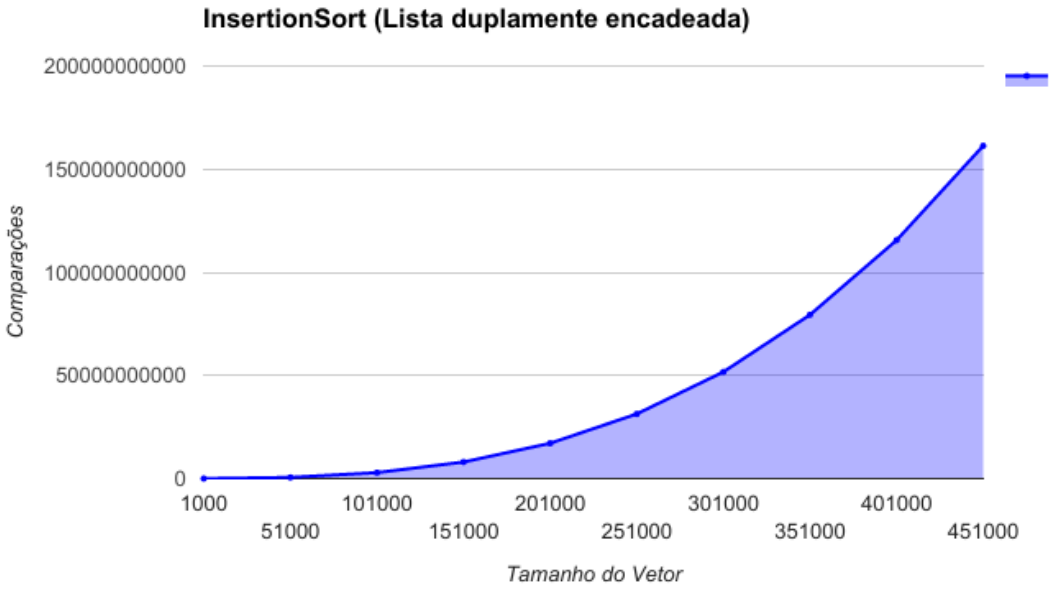

Figura 20: InsertionSort e a varição de comparações conforme o tamanho da lista.

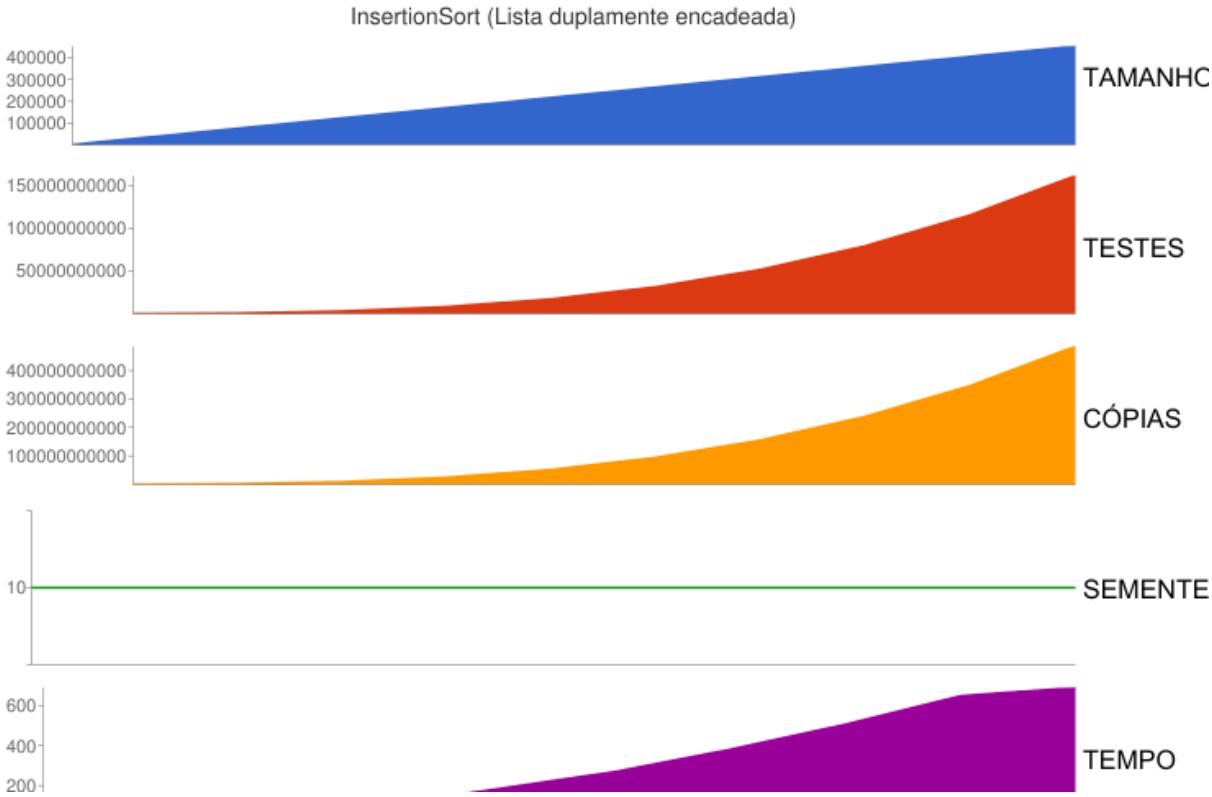

Figura 21: Gráfico referente a tabela Insertion e suas métricas.

Comparado com o InsertionSort do subcenário anterior ficou explicito que este método sofre ao ser aplicado em listas duplamentes encadeadas tanto na quantidade de comparações como em seu tempo de processamento que para uma lista de 451000 levou

689 "segundos"de clock, quanto para vetores do tipo inteiro de 500000 elementos o pior tempo foi de 376. Também foi perceptivel que se sobresaiu comparando com o método SelectionSort para listas.E é o melhor método para lista descrito até agora.

- ShellSort

Método que vem com a proposta de ser uma melhoria para o InsertionSort, mas que para a ordenação de listas, passa longe de apresentar alguma melhoria.

Foi uma surpresa ao realizar os testes,pois o algortimo aprensentou o pior resultado de todos os quatro algoritmos implementados para lista. Apresentou comportamento quadrático para todos os testes, estes que foram realizados para lista de até 50000 elementos, pois após isto o algoritmo se apresentou inviavel.Todas as estatísticas estão apresentadas abaixo:

Tabela 12: Sáidas dos testes realizados com o método ShellSort

TAMANHO	TESTES	CÓPIAS	SEMENTE	TEMPO
10000	38079	38078	10	6
20000	110200	110199	10	28
30000	225669	225668	10	70
40000	379752	379751	10	135
50000	572869	572868	10	229

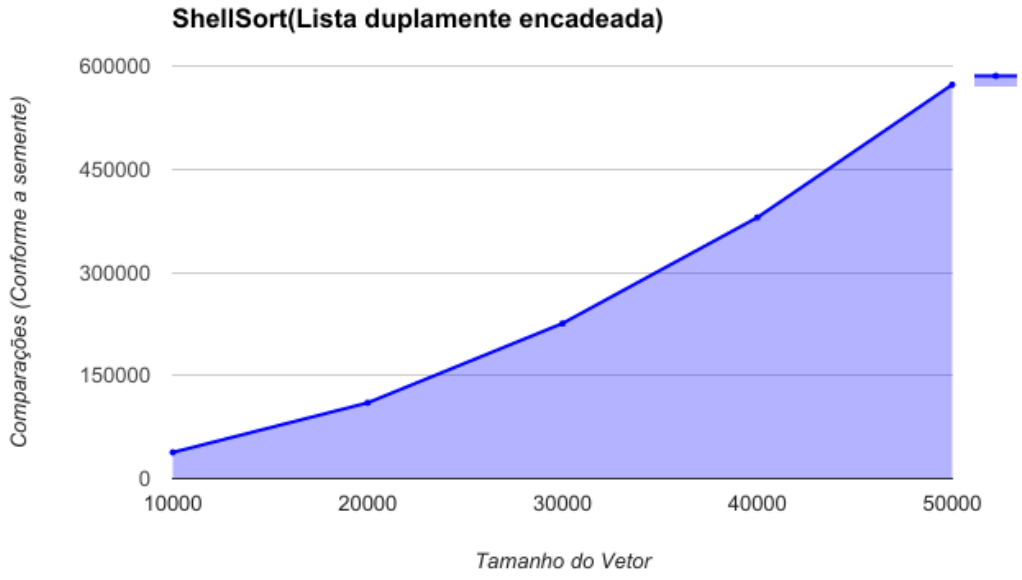

Figura 22: ShellSort e a varição de comparações conforme o tamanho da lista.

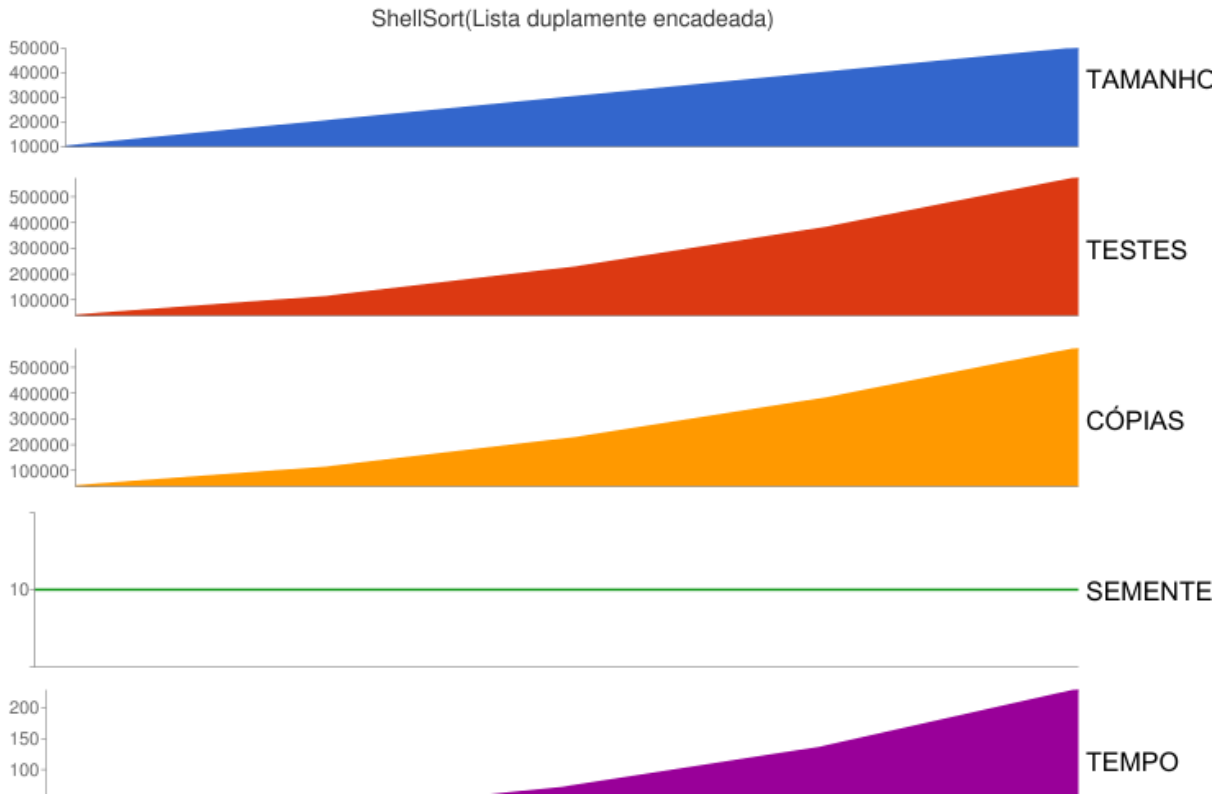

Figura 23: Gráfico referente a tabela Shell e suas métricas.

- Comparações entre os algoritmos apresentados neste subcenário

Ao analisar os dados obtidos nos testes ficou nítido que utilizar os métodos de ordenação apresentados em sala de aula, não são nada eficazes para ordenar listas duplamentes encadeadas. Vimos que métodos bons como ShellSort apresentaram comportamento totalmente adverso, e os demais desempenho ruim. Também podemos perceber que é mais eficiente criar a lista já ordenada, ou "ordernar"com outros métodos, que por fim também se tornam um grande desafio. Foram poucos algoritmos testados para listas pela dificuldade de implementação daqueles que utilizam recursão como ferramenta, mas já foram o bastante para perceber a dificuldade que é ordenar uma lista duplamente encadeada.

2.1.3 A ordenação de vetores struct

Os vetores de uma TAD criada, que possuem vários tipos de elementos, acabam tornando os métodos de ordenção lentos pois um vetor deste tipo, consome muita memória por conter inumeros elementos de vários tipos.

Os métodos por mais eficientes que sejam acabam não sendo tão bons pelo fato da máquina que executa a ordenação ter grandes dificuldades em percorrer e realizar cópias de resgistros destes tipo de elemento, uma vez que uma TAD não tem limites de tamanho. Logo ordenar se torna custoso, mas não pelo fato do método ser ruim mas sim trabalho "extra"que a máquina realiza para aplicar os algoritmos.

Esta dificuldade ficou explicita ao comparar o comportamento dos mesmos métodos para elementos do tipo inteiro.

Todos os métodos aplicados no cenário dos vetores do tipo inteiro foram aplicados neste cenário também. Alguns dos métodos infelizmente se tornaram inviaveis pela quantia de memória requerida, principalmente nos que utilizam recursão para realizar a ordenação.

Os algoritmos foram testados para vetores de 1000 ao limite imposto pela máquina em que foi realizado os testes. As quantidades de comparações, cópias de registro e tempo por clock do processador foram contabilizados e salvos em saídas de texto com a extensão (.txt), exceto para o algoritmo MergeSort que apresentou um erro inexplicavel e acabou sendo separado em outro arquivo que não gera saídas de texto.

Os vetores e os elementos da TAD foram preenchidos aleatóriamente pela função "srand"da biblioteca "time.h".

O limite de cada algoritmo foi definido após varias execuções onde por fim cada um apresentou um teto máximo de elementos.A analise de cada método está listada nos subíndices abaixo.

- BubbleSort

Este algoritmo como em suas implementações anteriores, não apresentou bons resultados e como nos vetores do tipo inteiro e para listas acabou sendo inviavel e se tornando limitado quanto aos testes. Para vetores do tipo struct foi possivel implementar apenas para vetores com até cem mil elementos com variações de sementes.

As estatiscas se encontram logo abaixo, o gráfico que compara o tamanho do vetor e as comparações conforme a mudança de semente foi ocultado pois resulta no mesmo do que o BubbleSort para vetores do tipo inteiro e pode ser verificado na Figura 2.

Tabela 13: Sáidas BubbleSort para vetores do tipo Struct

Tamanho N	Testes	Movimentacoes	Tempo	Semente
1000	998001	688659	0	10
1000	998001	758391	0	100
1000	998001	768882	0	1000
1000	998001	749412	0	10000
1000	998001	768807	0	100000
1000	998001	763977	0	1000000
5000	24990001	16816575	2	10
5000	24990001	18401769	3	100
5000	24990001	18912321	2	1000
5000	24990001	18955143	2	10000
5000	24990001	18978006	3	100000
5000	24990001	18893400	2	1000000
10000	99980001	67946550	12	10
10000	99980001	73946862	13	100
10000	99980001	74402832	12	1000
10000	99980001	75643314	11	10000
10000	99980001	74686389	11	100000
10000	99980001	74274459	12	1000000
50000	2499900001	1692375558	353	10
50000	2499900001	1850540394	603	100
50000	2499900001	1871961795	527	1000
50000	2499900001	1883627457	513	10000
50000	2499900001	1872111915	460	100000
50000	2499900001	1865656203	450	1000000
100000	9999800001	6745184700	1778	10
100000	9999800001	7412704137	1818	100
100000	9999800001	7489244313	1727	1000
100000	9999800001	7484974197	1835	10000
100000	9999800001	7492374444	1783	100000
100000	9999800001	7501660128	1831	1000000

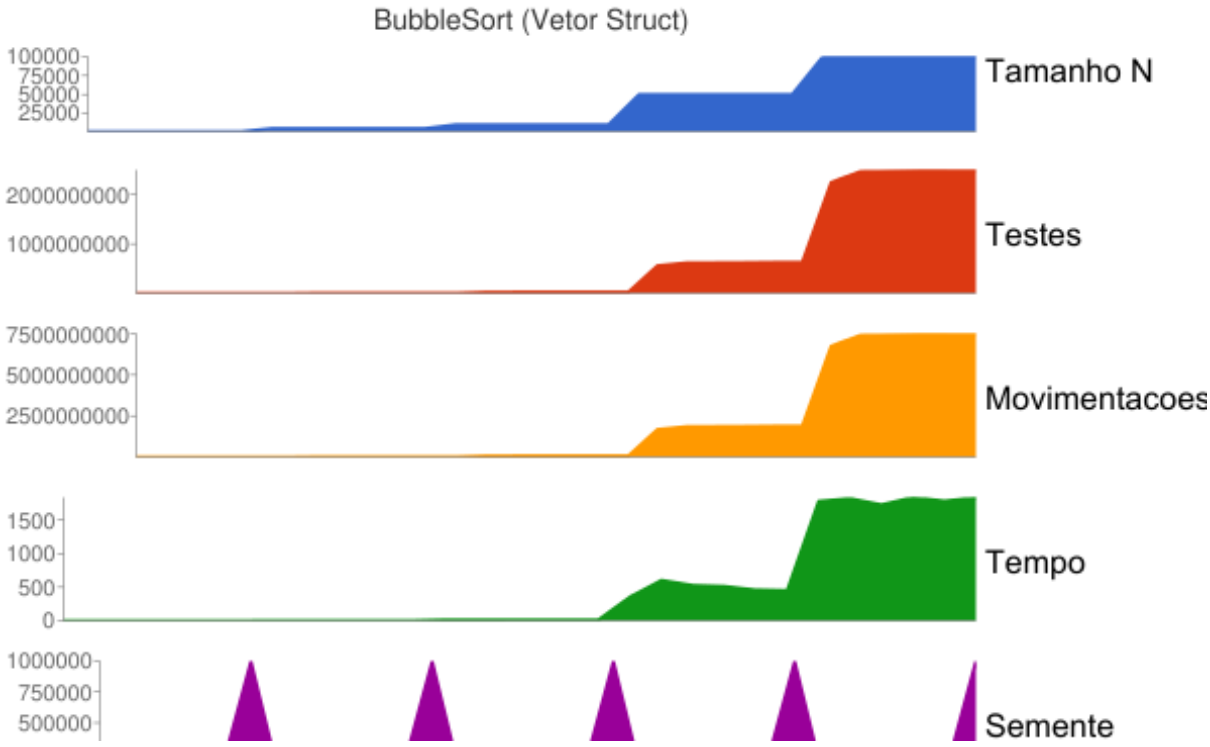

Figura 24: Gráfico referente a tabela do BubbleSort e suas métricas.

O primeiro ponto a ser notado é que a quantidade de comparações e cópias de registros entre os vetores do tipo inteiro para os de tipo struct foram praticamente idênticos, mostrando assim que o método da força bruta acaba ocorrendo idependendo da estrutura. Mas a real diferença foi o tempo de processamento que no pior caso para vetores do tipo inteiro foi de 59 "segundos"de clock , quanto para o tipo strcut chegou a 1835 em seu pior caso.

- SelectionSort

Como nas implementações anteriores o algoritmo acabou nao apresentando bons resultados, mantendo seu desempenho quanto a testes e cópias de registro se compararmos com os testes para vetores do tipo inteiro. Não apresentou sensibilidade a semente, porém ordenou vetores do tipo struct com no máximo cem mil elementos. Isto nos mostra o quanto realizar cópias de registro de vetores do tipo struct tem custo muito maior.

As estatiscas se encontram logo abaixo, o gráfico que compara o tamanho do vetor e as comparações conforme a mudança de semente foi ocultado pois resulta no mesmo do que para vetores do tipo inteiro e pode ser verificado na Figura 4.

Tabela 14: Sáidas SelectionSort para vetores do tipo Struct

Tamanho	Testes	Movimentacoes	Tempo	Semente
1000	499500	2997	0	10
1000	499500	2997	0	100
1000	499500	2997	0	1000
1000	499500	2997	0	10000
1000	499500	2997	0	100000
1000	499500	2997	0	1000000
5000	12497500	14997	0	10
5000	12497500	14997	0	100
5000	12497500	14997	0	1000
5000	12497500	14997	0	10000
5000	12497500	14997	0	100000
5000	12497500	14997	0	1000000
10000	49995000	29997	0	10
10000	49995000	29997	0	100
10000	49995000	29997	0	1000
10000	49995000	29997	0	10000
10000	49995000	29997	0	100000
10000	49995000	29997	0	1000000
50000	1249975000	149997	40	10
50000	1249975000	149997	46	100
50000	1249975000	149997	55	1000
50000	1249975000	149997	48	10000
50000	1249975000	149997	57	100000
50000	1249975000	149997	59	1000000
100000	4999950000	299997	231	10
100000	4999950000	299997	201	100
100000	4999950000	299997	197	1000
100000	4999950000	299997	176	10000
100000	4999950000	299997	147	100000
100000	4999950000	299997	187	1000000

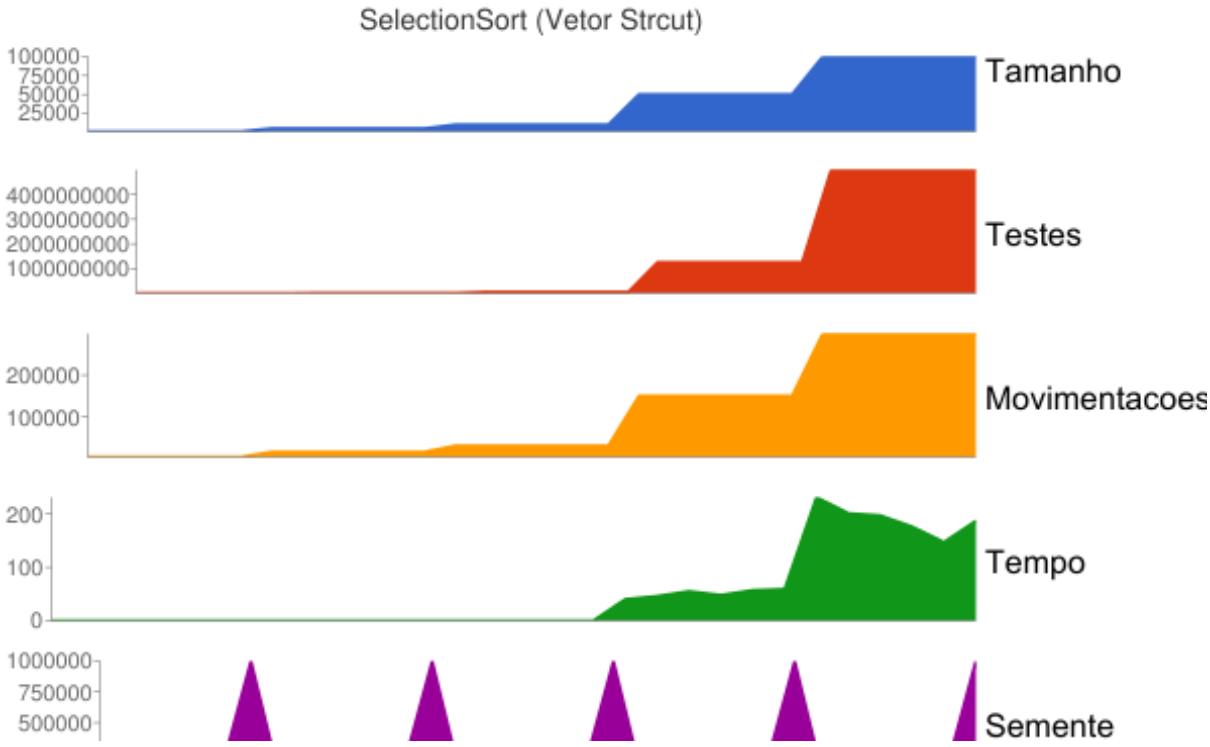

Figura 25: Gráfico referente a tabela do SelectionSort e suas métricas.

É percepctivel quanto o processamento é maior pois mantendo as mesmas estatisticas de cópias e testes, além de não ordenar elementos com mais de cem mil elementos levou 231 "segundos"de clock em seu pior caso enquanto para o mesmo número de elementos o SelectionSort para vetores do tipo inteiro levou 32 "segundos".

- InsertionSort

Como esperado as quantidades de comparações e cópias de registros também se aproximaram para os de vetores do tipo inteiro. Foi mantido a efeciência quanto as duas primeiras métricas, mas o tempo de processamento acabou sendo maior, algo que já era esperado se considerarmos que aconteceu o mesmo para os dois métodos anteriores deste subcenário. Também apenas ordenou vetores struct com até cem mil elementos.

As estatiscas se encontram logo abaixo, o gráfico que compara o tamanho do vetor e as comparações conforme a mudança de semente foi ocultado pois resulta no mesmo do que para vetores do tipo inteiro e pode ser verificado na Figura 6.

Tabela 15: Sáidas InsertionSort para vetores do tipo Struct

Tamanho N	Testes	Movimentacoes	Tempo	Semente
1000	231106	693315	0	10
1000	240691	722070	0	100
1000	248694	746079	0	1000
1000	246926	740775	0	10000
1000	245481	736440	0	100000
1000	240508	721521	0	1000000
5000	5662606	16987815	3	10
5000	6197873	18593616	4	100
5000	6226749	18680244	4	1000
5000	6174709	18524124	4	10000
5000	6200614	18601839	4	100000
5000	6217845	18653532	3	1000000
10000	22787540	68362617	15	10
10000	24776205	74328612	16	100
10000	25089490	75268467	16	1000
10000	25105207	75315618	17	1000
10000	25137572	75412713	15	100000
10000	24797148	74391441	19	1000000
50000	561923647	1685770938	545	10
50000	623869171	1871607510	496	100
50000	625292837	1875878508	512	1000
50000	624781815	1874345442	454	10000
50000	624999511	1874998530	448	100000
50000	624066439	1872199314	452	1000000
100000	2249410731	6748232190	1677	10
100000	2481988817	7445966448	1793	100
100000	2493911475	7481734422	1754	1000
100000	2504561326	7513683975	1827	10000
100000	2498442020	7495326057	1686	100000
100000	2497379159	7492137474	1724	1000000

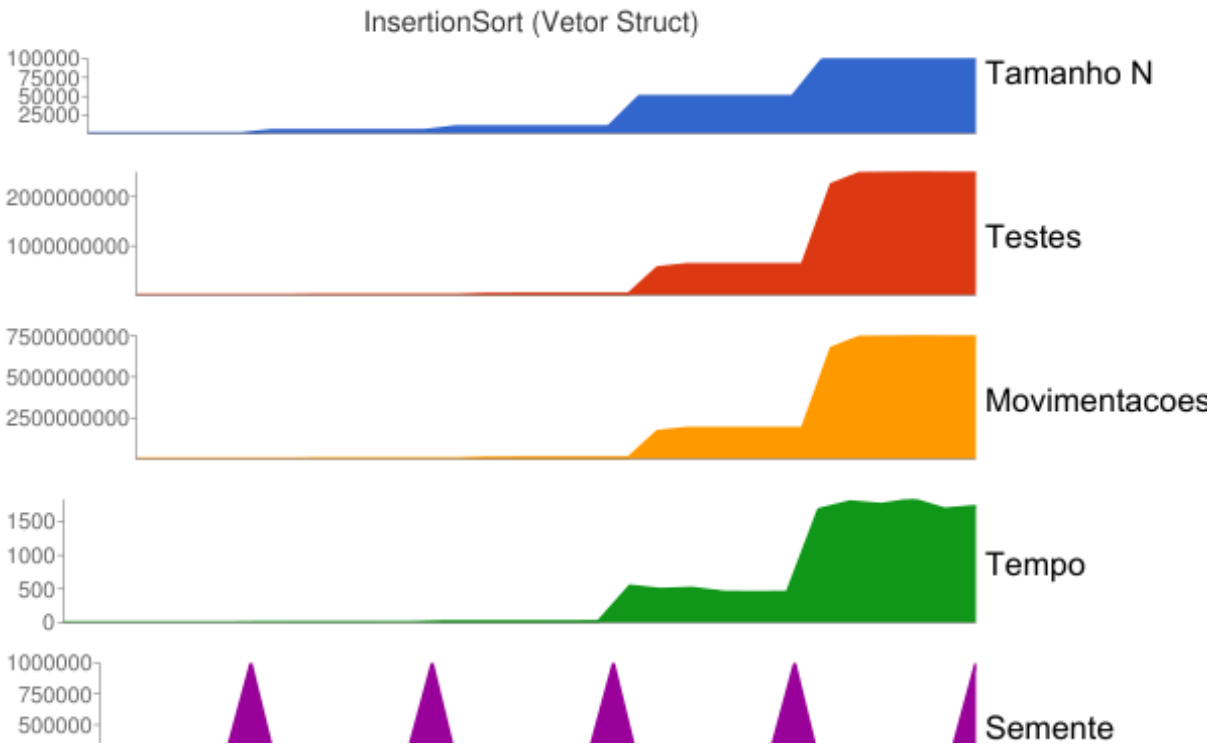

Figura 26: Gráfico referente a tabela do InsertionSort e suas métricas.

Foi percepctivel quanto o processamento é maior pois mantendo praticamente as mesmas estatisticas de cópias e testes, além de não ordenar elementos com mais de cem mil elementos levou 1827 "segundos"de clock em seu pior caso enquanto para o mesmo número de elementos o InsertionSort para vetores do tipo inteiro levou 15 "segundos".

- ShellSort

Com a análise deste algortimo podemos concluir que ao aplicar os métodos de ordenação para vetores struct a quantidade de comparações e cópias sera a mesma que para vetores struct, a diferença esta no processamento.

Este método diferente dos anteriores conseguiu ordenar vetores com até um milhão de elementos, se sobresaindo quanto aos métodos discutidos neste subcenário.

As estatiscas se encontram logo abaixo, o gráfico que compara o tamanho do vetor e as comparações conforme a mudança de semente foi ocultado pois resultara na mesma sensibilidade conforme a mudança de semente do que para vetores do tipo inteiro e pode ser verificado na Figura 8.

Tabela 16: Sáidas ShellSort para vetores do tipo Struct

Tamanho N	Tempo	Testes	Movimentacoes	Semente
1000	0	3262	9783	10
1000	0	6510	19527	100
1000	0	8487	25458	1000
1000	0	8650	25947	10000
1000	0	8776	26325	100000
1000	0	8830	26487	1000000
5000	0	17230	51687	10
5000	0	41874	125619	100
5000	0	63318	189951	1000
5000	0	70818	212451	10000
5000	0	70166	210495	100000
5000	0	67089	201264	1000000
10000	0	36308	108921	10
10000	0	93880	281637	100
10000	0	142095	426282	1000
10000	0	173972	521913	10000
10000	0	153082	459243	100000
10000	0	161551	484650	1000000
50000	0	186517	559548	10
50000	0	577169	1731504	100
50000	0	955748	2867241	1000
50000	0	1257846	3773535	10000
50000	0	1142899	3428694	100000
50000	0	1134480	3403437	1000000
100000	0	377455	1132362	10
100000	0	1082718	3248151	100
100000	0	2415356	7246065	1000
100000	1	2811815	8435442	10000
100000	1	2835449	8506344	100000
100000	1	3107359	9322074	1000000
500000	1	1994437	5983308	10
500000	3	6926506	20779515	100
500000	6	13557824	40673469	1000
500000	7	19689317	59067948	10000
500000	8	21580793	64742376	100000
500000	8	21104964	63314889	1000000
1000000	2	3988450	11965347	10
1000000	7	12744594	38233779	100
1000000	13	29609677	88829028	1000
1000000	17	43555449	130666344	10000
1000000	19	46811200	140433597	100000
1000000	20	51081548	153244641	1000000

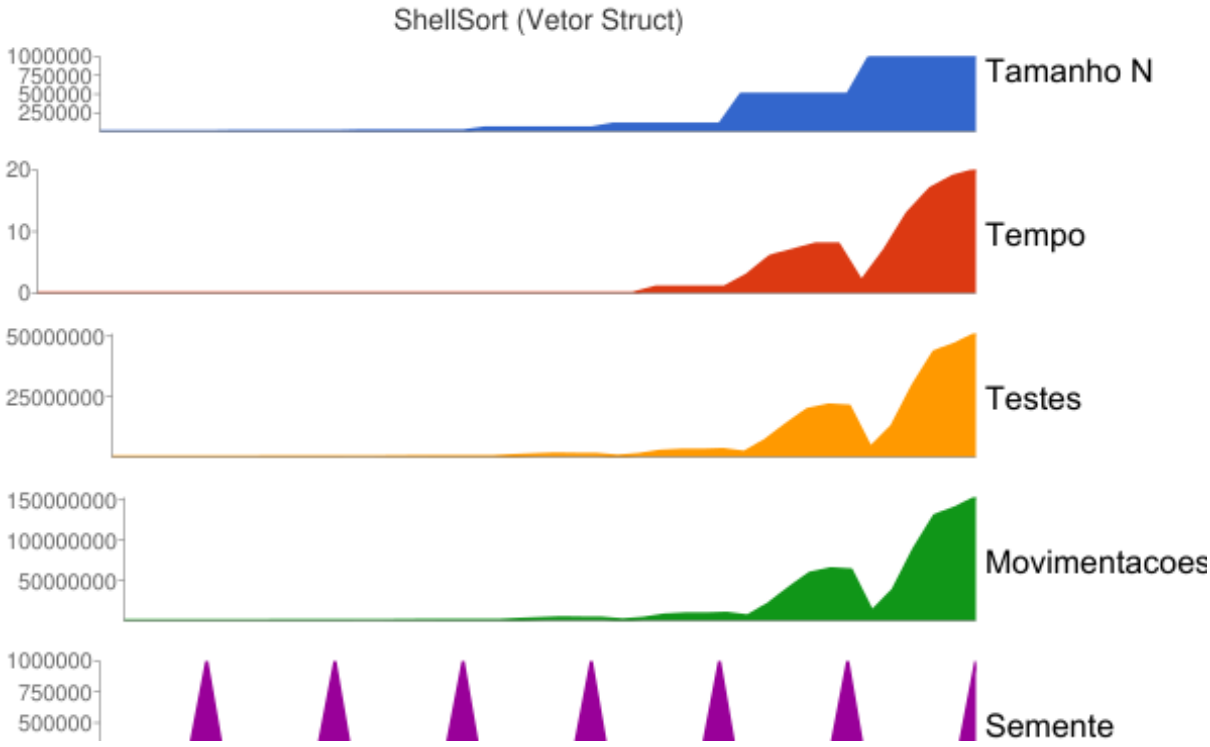

Figura 27: Gráfico referente a tabela do ShellSort e suas métricas.

O fator de diferenciação quanto ao método para vetores do tipo inteiro foi o tempo de processamento, pois as structs naturalmente levam mais por ocuparem mais memória. Em seu pior caso o vetor de strcut levou 20 "segundos"de clock, quanto para vetores do tipo inteiro o tempo foi constantemente 0.

- HeapSort

Cópias de registro e testes foram similar para o método para vetores inteiros, também como o ShellSort conseguiu ordenar vetores de até um milhão de elementos, o diferencial continuou sendo o tempo de processamento que se tornou maior.

As estatiscas se encontram logo abaixo, o gráfico que compara o tamanho do vetor e as comparações conforme a mudança de semente foi ocultado pois resulta no mesmo que os de vetores do tipo inteiro e pode ser verificado na Figura 10.

Tabela 17: Sáidas HeapSort para vetores do tipo Struct

Tamanho N	Tempo	Testes	Movimentacoes	Semente
1000	0	9870	24432	10
1000	0	11610	27168	100
1000	0	11677	27189	1000
1000	0	11694	27282	10000
1000	0	11723	27357	100000
1000	0	11633	27132	1000000
5000	0	64648	155232	10
5000	0	74554	169671	100
5000	0	75637	171396	1000
5000	0	75796	171309	10000
5000	0	75632	171096	100000
5000	0	75843	171492	1000000
10000	0	142905	339351	10
10000	0	164103	369795	100
10000	0	166403	372909	1000
10000	0	166529	372762	10000
10000	0	166343	372765	100000
10000	0	166450	373026	1000000
50000	0	859915	2000874	10
50000	0	992463	2194293	100
50000	0	1005574	2211324	1000
50000	0	1006260	2212194	10000
50000	0	1006646	2212893	100000
50000	0	1006961	2213052	1000000
100000	0	1844208	4261476	10
100000	0	2132462	4687104	100
100000	0	2159614	4720335	1000
100000	0	2162630	4724979	10000
100000	0	2162864	4724100	100000
100000	0	2163643	4724769	1000000
500000	2	10689220	24343746	10
500000	3	12378248	26839224	100
500000	4	12542335	27052668	1000
500000	4	12557639	27068667	10000
500000	4	12558538	27071136	100000
500000	4	12558950	27071094	1000000
1000000	5	22647578	51343782	10
1000000	7	26230680	56636907	100
1000000	8	26581971	57105588	1000
1000000	8	26614602	57139638	10000
1000000	9	26620016	57144909	100000
1000000	9	26618460	57140997	1000000

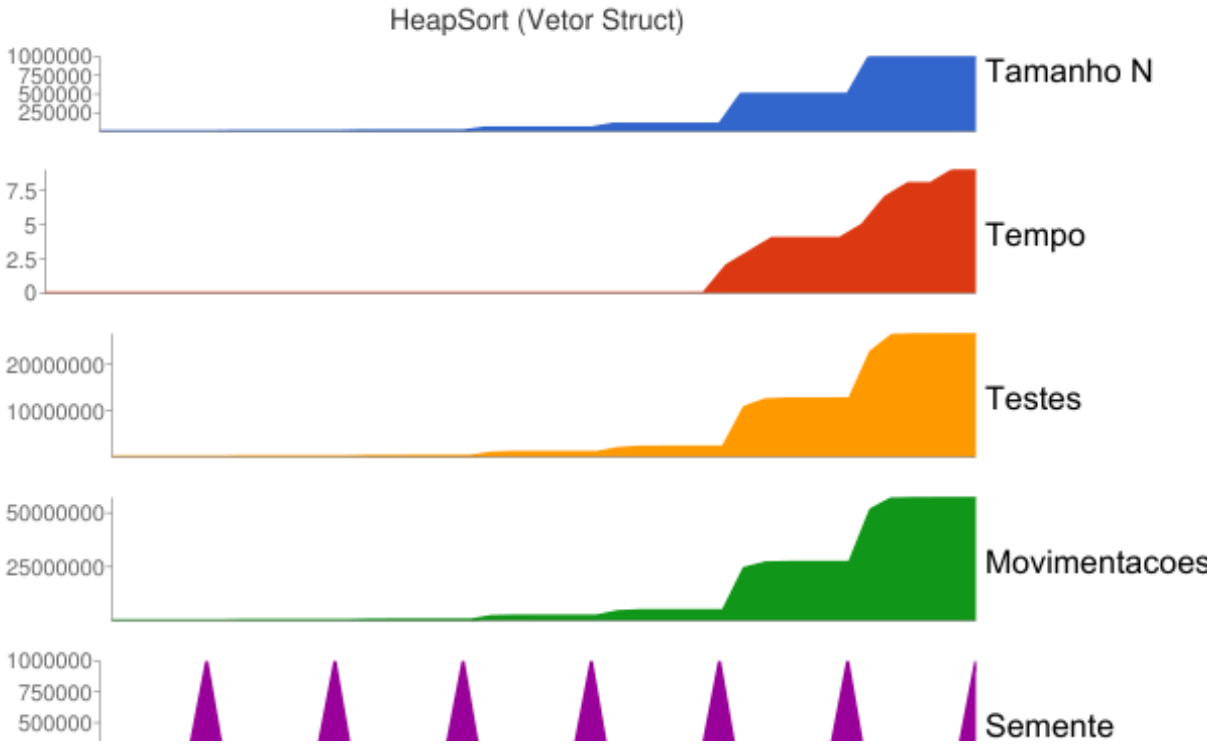

Figura 28: Gráfico referente a tabela do HeapSort e suas métricas.

O fator de diferenciação novamente quanto ao método para vetores do tipo inteiro foi o tempo de processamento. Em seu pior caso o vetor de strcut levou 9 "segundos"de clock, quanto para vetores do tipo inteiro o tempo foi constantemente 0.

- MergeSort

O método MergeSort foi a grande surpresa deste subcenário, pois diferente de para vetores do tipo inteiro onde ele se apresentou o melhor por sua estabilidade quanto ao custo, para vetores struct se mostrou o pior.

Pode ser devido ao fato da implementação, onde foi notado que corriqueiramente ao utilizar a recursão o ponteiro que é utilizado como vetor temporário perde o endereço de memória e não consegue liberara-la, fazendo assim que o método se torne inviavel, tanto que foi retirado da função principal na qual gerava as saídas (.txt) e foi tratado como um caso particular.

Mesmo assim não apresentou bons resutados ordenando apenas para vetores de até 700000 elementos. Tratando-o como caso particular devido a dificulade de implementação encotrada foi realizado apenas quatro testes ordenando vetores a partir de 100000 elemento.

As estatiscas se encontram logo abaixo, o gráfico que compara o tamanho do vetor e as comparações conforme a mudança de semente foi ocultado pois resulta no mesmo que os de vetores do tipo inteiro e pode ser verificado na Figura 12.

Tabela 18: Sáidas MergeSort para vetores do tipo Struct

Tamanho	Testes	Movimentações	Semente	Tempo
100000	706849	1668946	10	1
300000	3260441	7144678	100	4
500000	7704953	16620410	1000	6
700000	14151851	30271855	10000	10

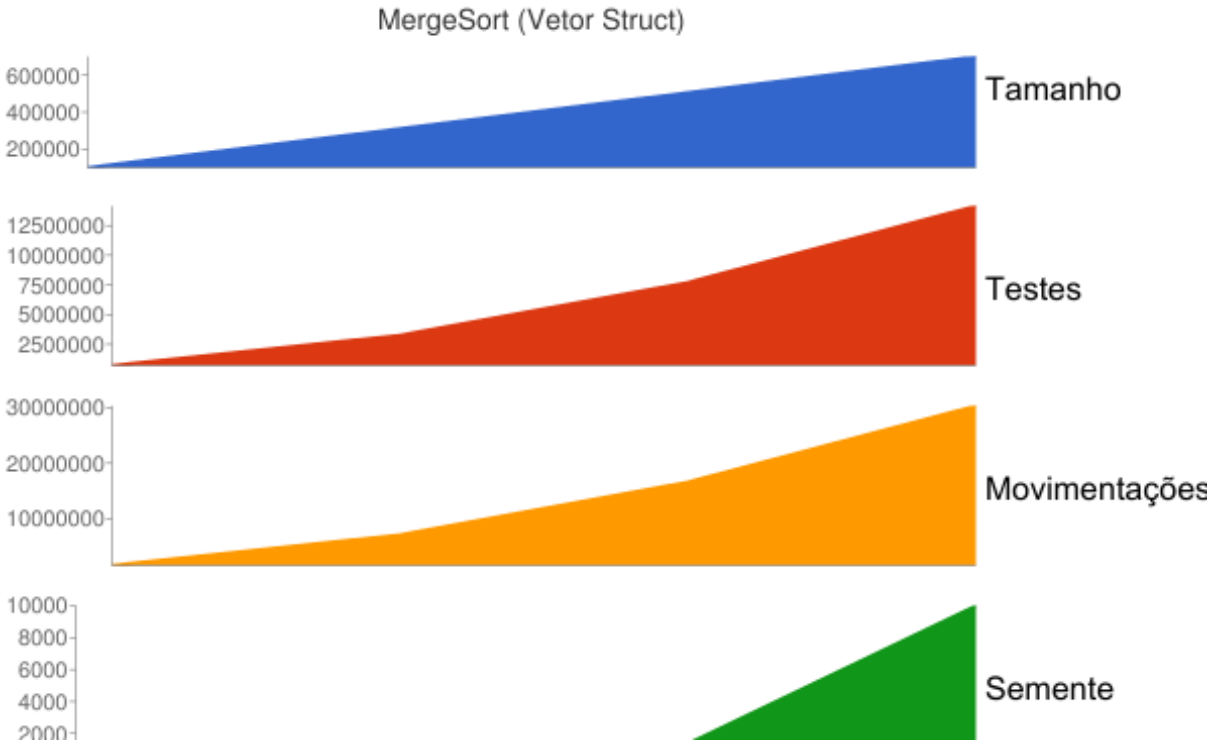

Figura 29: Gráfico referente a tabela do HeapSort e suas métricas.

Apesar da dificuldade de implementação, apresentou bons resultados quanto seu processamento, mas ficando atrás do HeapSort que acima foi apresentado que para um milhão de elementos levou 9 "segundos"de clock , quanto o MergeSort para 700000 levou 10 "segundos".

- QuickSort

O método QuickSort também manteve o números de comparações e cópias de registro parecidos com para os vetores do tipo inteiro, havendo apenas uma pequena discrepância. Comos os anteriores também teve problemas com o tamanho da struct e usou mais processamento.

As estatiscas se encontram logo abaixo, o gráfico que compara o tamanho do vetor e as comparações conforme a mudança de semente foi ocultado pois resulta no mesmo que os de vetores do tipo inteiro e pode ser verificado na Figura 14.

Tabela 19: Sáidas QuickSort para vetores do tipo Struct

Tamanho N	Testes	Movimentacoes	Tempo	Semente
1000	2828	11517	0	10
1000	5688	9102	0	100
1000	7380	8058	0	1000
1000	7462	7866	0	10000
1000	9010	7572	0	100000
1000	7847	7743	0	1000000
5000	12348	75150	0	10
5000	25497	62160	0	100
5000	45030	52029	0	1000
5000	46955	47496	0	10000
5000	46077	47517	0	100000
5000	48316	46962	0	1000000
10000	25111	165033	0	10
10000	54165	138450	0	100
10000	81702	116355	0	1000
10000	110545	102093	0	10000
10000	108420	100632	0	100000
10000	115809	99060	0	1000000
50000	128255	997806	0	10
50000	269635	865071	0	100
50000	428460	734691	0	1000
50000	520024	638619	0	10000
50000	604111	595599	0	100000
50000	609898	587958	0	1000000
100000	269784	2139225	0	10
100000	565720	1871730	0	100
100000	893074	1615839	0	1000
100000	1217127	1379787	0	10000
100000	1286755	1269513	0	100000
100000	1259088	1251501	0	1000000
500000	1374552	12422706	4	10
500000	3225156	11051250	3	100
500000	4496377	9734343	4	1000
500000	6322271	8482341	3	10000
500000	7003374	7501512	3	100000
500000	8038644	7045536	3	1000000
1000000	2619120	26330994	11	10
1000000	6429334	23535756	9	100
1000000	9111661	20981928	8	1000
1000000	11584045	18434073	6	10000
1000000	15626338	16139070	7	100000
1000000	16348061	14957991	6	1000000

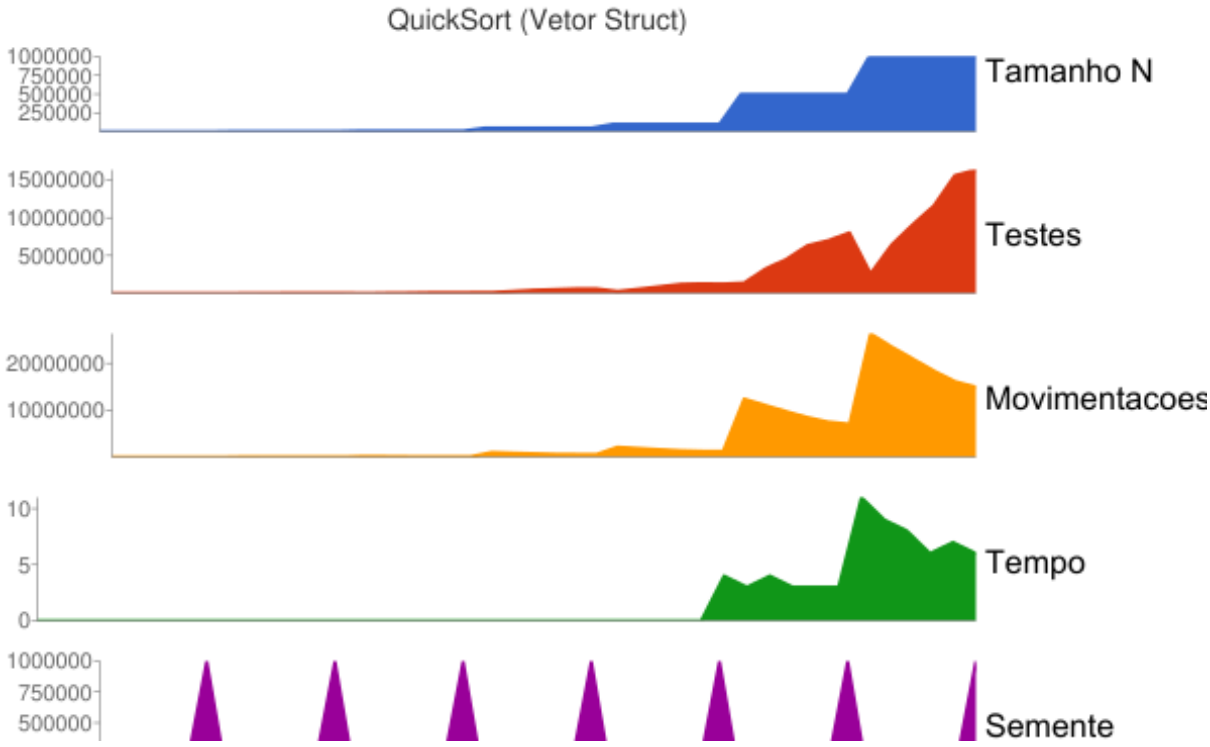

Figura 30: Gráfico referente a tabela do QuickSort e suas métricas.

Apresentou bons resultados, realizando para vetores de até um milhão de elementos, apenas sofrendo com a quantia de processamento que em seu pior caso levou 11 "segundos"de clock, e o mesmo algortimo para vetores inteiros mantese o tempo em 0.

- Comparações entre os algoritmos apresentados neste subcenário

A principal caracteristica que difere dos métodos para vetores inteiros foi nítidamente a dificuldade de processamento das structs por ocuparem mais memória.

Sem dúvidas ao se comparar os métodos analisados acima, podemos concluir que o BubbleSort, InsertionSort e SelectionSort são inviaveis para vetores structs, pois limitaram-se a ordenar apenas vetores de até cem mil elementos e com tempo de processamento muito elevado.

O ShellSort apresentou-se viavel uma vez que o tempo de processamento só foi significativo para vetores de um milhão de elementos chegando a marcar apenas 20 "segundos"de clock.

O HeapSort já se destacou aos demais, sendo considerado o melhor nos testes realizados, pois obteve as melhores estatisticas quanto ao tempo de processamento marcando 9 "segundos"de clock em seu pior caso.

O QuickSort era mais uma vez o principal candidato a ser o melhor método, mas nos testes realizados apesar de ter ótimo desempenho teve mais dificuldades quanto ao processamento apresentado em seu pior caso 11 "segundos"de clock, e foi esta minima diferença quando ao HeapSort que o deixou em segundo lugar entre os métodos de ordenação para vetores struct.

MergeSort acabou apesar de ordenar para um número grande de elementos sua implementação não se comportou bem, conseguindo assim se tornar o método menos indicado por não ser seguro. Suas estatisticas até foram boas mas se compararmos o tempo de processamento acaba ficando atrás do Quick e Heap.

2.2 Cenário 2: Impacto de variações do Quicksort

O Quicksort é um algoritmo bom por ser relativamente fácil de ser implementado e realiza a ordenação sem a necessidade de um vetor auxiliar. O tempo médio de execução é Ø(nlogn) onde n é o número do tamanho do vetor, porém é preciso cuidado na implementação a fim de evitar o pior caso Ø(n^2), é provavelmente o algoritmo mais utilizado.

Neste cenário foram comparados o desempenho de diferentes variações do Quicksort para ordenar um conjunto de n inteiros armazenados em um vetor.

Foram feitos testes com 5 sementes diferentes e variando o valor de n (tamanho do vetor), feito uma média de tempo de processamento, número de comparações e número de trocas.

Neste caso o que faz toda a diferença é a escolha do pivô na parte do particionamento o pior caso acontece quando o pivô eleito é o maior ou o menor elemento do vetor, para tentar fugir do pior caso é feita tais variações.

- Quicksort Recursivo - Esse é o clássico Quicksort que funciona na premissa da divisão e conquista, escolhe um pivô e faz a partição.
- Quicksort Mediana - Essa variação do Quicksort Recursivo escolhe o pivô para particionar como a mediana de 3 elementos. Foi implementada da seguinte maneira: Pivô = (V[meio] + V[*i] + V[*j])/3; o mediana de 5 foi testado, porém seus resultados não foram satisfatórios, a diferença de processamento foi minima e de comparações também, por isso a escolha ficou com k = 3.
- Quicksort Inserção - Serve como uma otimização para caso de sequências pequenas, a melhor implementação foi a de m = 100, pois otimizou o tempo de processamento, porém seus números de comparações e movimentações são bastantes expressivos.
- Quicksort Empilha Inteligente - versão otimizada do Quicksort Recursivo, processa primeiro o lado menor da partição
- Quicksort Iterativo - Nessa variação foram implementadas pilhas auxiliares
- Quicksort Empilha Inteligente - versão otimizada do Quicksort Interativo processa o lado menor da partição

	Comparações	Movimentações	Tempo(s)
Tamanho do N	**1000**		
QUICKSORT RECURSIVO	10027	29807	0
QUICKSORT MEDIANA	13003	29851	0
QUICKSORT EMPILHA RECURSIVO	10027	35888	0
QUICKSORT INSERCAO	20359	22684	0
QUICKSORT INTERATIVO	267085	42751	0
QUICKSORT EMPILHA INTERATIVO	320948	34290	0

	Comparações	Movimentações	Tempo(s)
Tamanho do N	**5000**		
QUICKSORT RECURSIVO	53454	202499	0
QUICKSORT MEDIANA	78876	203007	0
QUICKSORT EMPILHA RECURSIVO	53454	236443	0
QUICKSORT INSERCAO	120965	96263	0
QUICKSORT INTERATIVO	9448727	215324	0
QUICKSORT EMPILHA INTERATIVO	10722237	215324	0

	Comparações	Movimentações	Tempo(s)
Tamanho do N	**10000**		
QUICKSORT RECURSIVO	101970	441533	0
QUICKSORT MEDIANA	163586	443821	0
QUICKSORT EMPILHA RECURSIVO	101970	498836	0
QUICKSORT INSERCAO	271459	218310	0
QUICKSORT INTERATIVO	27730297	432402	0
QUICKSORT EMPILHA INTERATIVO	32769414	347200	0

	Comparações	Movimentações	Tempo(s)
Tamanho do N	**50000**		
QUICKSORT RECURSIVO	101042	438126	0
QUICKSORT MEDIANA	162830	437746	0
QUICKSORT EMPILHA RECURSIVO	101042	491449	0
QUICKSORT INSERCAO	287535	217296	0
QUICKSORT INTERATIVO	25179134	432955	0
QUICKSORT EMPILHA INTERATIVO	30217048	347825	0

	Comparações	Movimentações	Tempo(s)
Tamanho do N	**100000**		
QUICKSORT RECURSIVO	1001852	5948349	0
QUICKSORT MEDIANA	2009091	5946940	0
QUICKSORT EMPILHA RECURSIVO	1001852	6656158	0
QUICKSORT INSERCAO	3645884	3417312	0
QUICKSORT INTERATIVO	525247391	4314341	2
QUICKSORT EMPILHA INTERATIVO	425344890	3460205	2

Figura 31: Métricas dos Quicksort

	Comparações	Movimentações	Tempo(s)
Tamanho do N	**500000**		
QUICKSORT RECURSIVO	5053604	35315614	0
QUICKSORT MEDIANA	11138393	35248335	0
QUICKSORT EMPILHA RECURSIVO	5053605	38326492	0
QUICKSORT INSERCAO	21626374	21673439	0
QUICKSORT INTERATIVO	316152130	21653898	78
QUICKSORT EMPILHA INTERATIVO	216256139	17395860	68

	Comparações	Movimentações	Tempo(s)
Tamanho do N	**1000000**		
QUICKSORT RECURSIVO	9904335	75149331	0
QUICKSORT MEDIANA	23302084	75115151	0
QUICKSORT EMPILHA RECURSIVO	9904335	81185840	0
QUICKSORT INSERCAO	45098993	48287346	0
QUICKSORT INTERATIVO	42806327	43413431	276
QUICKSORT EMPILHA INTERATIVO	223610276	34899177	275

Figura 32: Métricas dos Quicksort

Podemos observar que devido a implementação do quicksort interativo com pilhas auxiliares adiciona-se um custo a mais tornando o algoritmo quase inviavel para n maiores que 500000

podemos ver o que o melhor em comparações e movimentações é o QuickSort Recursivo em termos de processamento o mais efetivo foi o quicksort Inserção, faz a ordenação em um tempo menor. Desempenho em relação ao tempo, para n maiores nessa parte foram retirado o quicksort interativo e o empilha inteligente da versão interativa, pois para n muito grande seu tempo de processamento é muito alto para a capacidade da máquina. A análise foi feita da seguinte maneira, foram implementados variáveis globais para fazer a contagem a cada comparação de elementos e cada troca de posição.

Tamanho do N	10000000
	Tempo(s)
QUICKSORT RECURSIVO	2
QUICKSORT MEDIANA	2
QUICKSORT EMPILHA RECURSIVO	3
QUICKSORT INSERCAO	1

Tamanho do N	20000000
	Tempo(s)
QUICKSORT RECURSIVO	5
QUICKSORT MEDIANA	5
QUICKSORT EMPILHA RECURSIVO	6
QUICKSORT INSERCAO	3

Tamanho do N	35000000
	Tempo(s)
QUICKSORT RECURSIVO	10
QUICKSORT MEDIANA	10
QUICKSORT EMPILHA RECURSIVO	11
QUICKSORT INSERCAO	6

Tamanho do N	450000000
	Tempo(s)
QUICKSORT RECURSIVO	13
QUICKSORT MEDIANA	13
QUICKSORT EMPILHA RECURSIVO	14
QUICKSORT INSERCAO	8

Tamanho do N	50000000
	Tempo(s)
QUICKSORT RECURSIVO	15
QUICKSORT MEDIANA	16
QUICKSORT EMPILHA RECURSIVO	17
QUICKSORT INSERCAO	10

Tamanho do N	70000000
	Tempo(s)
QUICKSORT RECURSIVO	20
QUICKSORT MEDIANA	19
QUICKSORT EMPILHA RECURSIVO	23
QUICKSORT INSERCAO	13

Figura 33: Métricas dos Quicksort

Depois desses testes foram eliminados o quicksort interativo e quicksort inteligente interativo, para prosseguir com os valores de n maiores e ter uma noção de qual é o mais rápido para n com tamanho grande. Podemos ver claremente que o quicksor insertion leva vantagem em questão de rapidez, pois seu tempo de processamento é cada vez menor quando aumentamos cada vez mais o tamanho do n. Todas as métricas foram alternando as sementes e realizada sua média.

2.3 Cenário 3: Quicksort X Mergesort X Heapsort X Countingsort X Gnomesort

Neste cenário foi escolhido a melhor variação do Quicksort, que com base nos teste o resultado foi o Quicksort Inserção, com os algoritmos Heapsort, Mergesort, Countingsort e o Gnome sort

- Countingsort - É um algoritmo de ordenação que funciona por contagem, outro método, a vantagem ocorre quando você conhece os valores, a desvantagem é que consome muita memória, pois ele vai alocando de tal forma até ordernar o vetor. veja sua análise na tabela 20.
- GnomeSort - Esse método é parecido com o Insertion sort o elemento é levado até sua posição correta com uma seqüencia grande de trocas assim como o Bubble sort, percorrendo o vetor e comparando seus elementos dois a dois, quando ele encontra um elemento que está na posição incorreta, ou seja, um número maior antes de um menor, ele troca a posição dos elementos, e volta com este elemento até que encontre o seu respectivo lugar. veja sua análise na tabela 21;

Tabela 20: Análise do CountingSort

Tamanho do N	Memória Alocada	Tempo em seg
1000	12,000 bytes	0
5000	60,000 bytes	0
10000	120,000 bytes	0
50000	600,000 bytes	0
100000	1,200,000 bytes	0
500000	6,000,000 bytes	0
1000000	12,000,000 bytes	0
10000000	120,000,000 bytes	2
100000000	1,200,000,000 bytes	32

A tabela ilustra muito bem o CountingSort, um algoritmo muito eficiente e rápido, porém demanda de muita memória do computador, o que o torna as vezes inviável, mas para vetores de tamanhos pequenos e valores conhecidos é uma boa alternativa. ele foi analisado de maneira separada pois demanda de muita mémoria, foi utilizado o recurso valgrind do linux.

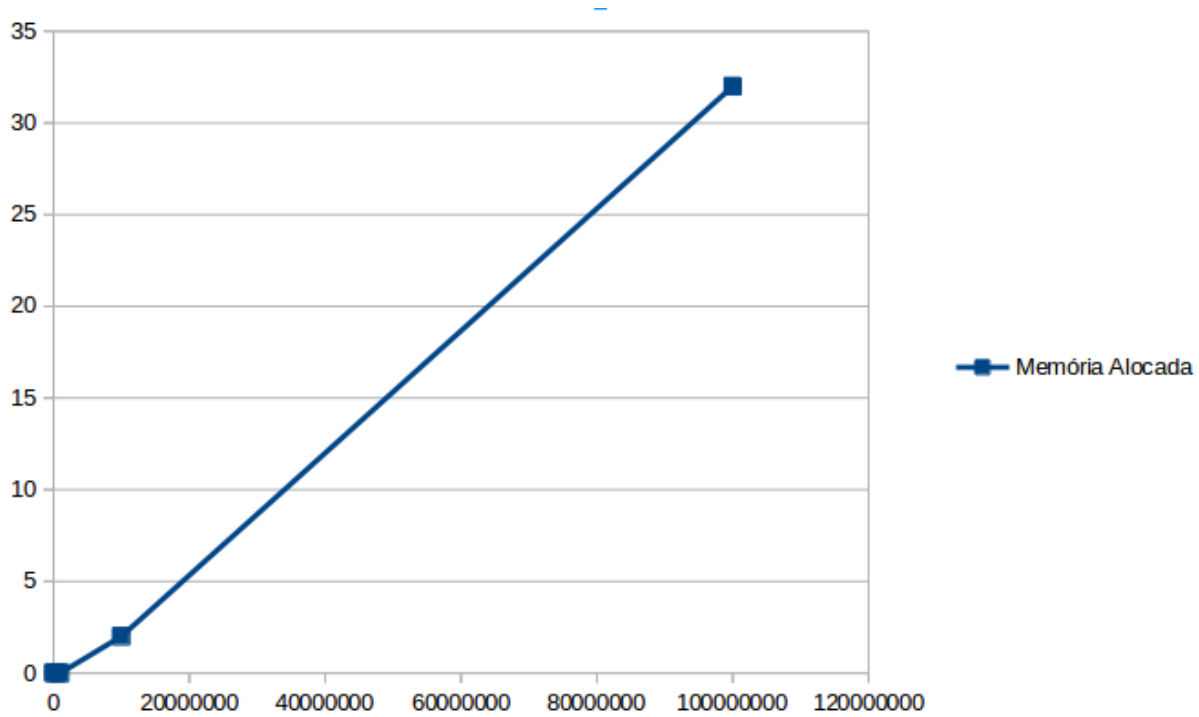

Figura 34: Gráfico CountingSort

Tabela 21: Saidas GnomeSort

Tamanho N:	Tempo:	Testes:	Movimentacoes:	Semente:
1000	0	232338	691338	10
1000	0	255095	759330	100
1000	0	258783	770361	1000
1000	0	252696	752097	10000
1000	0	259175	771540	100000
1000	0	257539	766635	1000000
5000	0	5619059	16828572	10
5000	0	6146164	18408657	100
5000	0	6316722	18920199	1000
5000	0	6332893	18968703	10000
5000	0	6337277	18981864	100000
5000	0	6312278	18906861	1000000
10000	0	22676845	67973598	10
10000	0	24677837	73973808	100
10000	0	24827605	74422860	1000
10000	0	25243434	75670347	10000
10000	0	24920891	74702703	100000
10000	0	24787138	74301453	1000000
50000	3	564225184	1692390459	10
50000	3	616972267	1850618214	100
50000	3	624132187	1872096756	1000
50000	4	628020866	1883762628	10000
50000	4	624176377	1872229176	100000
50000	4	622030456	1865791398	1000000
100000	15	2248674769	6745454394	10
100000	16	2471189547	7412971641	100
100000	16	2496661527	7489384995	1000
100000	16	2495281348	7485244134	10000
100000	16	2497725650	7492576992	100000
100000	17	2500783125	7501749414	1000000
500000	392	56340777722	169019483613	10
500000	419	61899292293	185694892149	100
500000	421	62453685508	187358058045	1000
500000	413	62442321268	187323964011	10000
500000	425	62591922535	187772767659	100000
500000	427	62501555061	187501665213	1000000

	Comparações	Movimentações	Tempo(s)
Tamanho do N	**1000**		
QUICKSORT INSERÇÃO	22005	18809	0
HEAPSORT	11655	27237	0
MERGESORT	9987	19974	0
GNOMESORT	252519	751584	0

	Comparações	Movimentações	Tempo(s)
Tamanho do N	**5000**		
QUICKSORT INSERÇÃO	128657	105777	0
HEAPSORT	75730	171357	0
MERGESORT	61822	123644	0
GNOMESORT	6268571	18775731	0

	Comparações	Movimentações	Tempo(s)
Tamanho do N	**10000**		
QUICKSORT INSERÇÃO	282592	210955	0
HEAPSORT	166554	372780	0
MERGESORT	133631	267262	0
GNOMESORT	24991957	74915886	0

	Comparações	Movimentações	Tempo(s)
Tamanho do N	**50000**		
QUICKSORT INSERÇÃO	1516856	1143416	0
HEAPSORT	1006357	2212710	0
MERGESORT	784481	1568962	0
GNOMESORT	623988008	1871664072	5

	Comparações	Movimentações	Tempo(s)
Tamanho do N	**100000**		
QUICKSORT INSERÇÃO	3217214	2356686	0
HEAPSORT	2162699	4723929	0
MERGESORT	1668946	3337892	0
GNOMESORT	1787612005	1068468686	23

	Comparações	Movimentações	Tempo(s)
Tamanho do N	**500000**		
QUICKSORT INSERÇÃO	18054268	12493496	0
HEAPSORT	12557503	27069987	0
MERGESORT	9475732	189514464	0
GNOMESORT	2102832673	2010530759	298

Figura 35: Métricas do Cenário 3

A partir de agora podemos ver a diferenças entres os algoritmos do cenário 3, o QuickSort inserção ainda leva vantagem na questão do tempo, já em questão de movimentação e trocas o MergeSort leva vantagem.

3 COMPILAÇÃO E EXECUÇÃO

Para a compilação e execução do cenário 1 no sistema LINUX.

Para o subcenário vetores do tipo inteiro:

- Acessar a pasta do trabalho, dentro da subpasta CENARIO 1, em VETORES DO TIPO INTEIRO.
- Digitar no terminal gcc analise.c -o nomedometodoescolhido
- Os métodos que podem ser escolhidos são : bubblesort, selectionsort, insertionsort, shellsort, heapsort, mergesort e quicksort .
- Após isto Digite ./nomedometodoescolhido entrada.txt saida.txt
- O algoritmo será executado para o método escolhido, lembrando que para alguns é preciso editar a entrada .txt presente no mesmo diretório para que o algortimo conclua a ordenação.

  Para o subcenário Listas duplamente encadeadas:

- Acessar a pasta do trabalho, dentro da subpasta CENARIO 1, em LISTA DUPLAMENTE ENCADEADA. Para cada método será necessário uma compilação.
- Para BubbleSort digite no terminal: gcc BubbleLista.c -o bubble.out
- Digite ./bubble.out e o algoritmo será executado.
- Para SelectionSort digite no terminal: gcc SelectionLista.c -o selection.out
- Digite ./selection.out e o algoritmo será executado.
- Para InsertionSort digite no terminal: gcc InsertionLista.c -o insertion.out
- Digite ./insertion.out e o algoritmo será executado.
- Para ShellSort digite no terminal: gcc ShellLista.c -o shell.out
- Digite .shell.out e o algoritmo será executado.

Para o subcenário Vetores Struct:

- Acessar a pasta do trabalho, dentro da subpasta CENARIO 1, em VETORES TIPO STRUCT.
- Para os métodos bubblesort, selectionsort, insertionsort, shellsort, heapsort e quicksort.
- Digitar no terminal gcc analisevetorstruct.c -o nomedometodoescolhido .
- Após isto Digite ./nomedometodoescolhido entrada.txt saida.txt
- Para o caso especial do MergeSort:
- Digitar no terminal gcc mergesortstruct.c -o merge.out
- Após isto Digite ./merge.out o algoritmo será executado.

Para a compilação e execução do CENÁRIO 2 no sistema LINUX :

- Acessar a pasta do trabalho, na subpasta CENÁRIO 2
- Abrir o terminal
- Digitar gcc quicksort.c
- Digitar ./a.out
- O algoritmo será executado.

Para a compilação e execução do CENÁRIO 3 no sistema LINUX:

- Acessar a pasta do trabalho,na subpasta CENÁRIO 3
- Abrir o terminal
- Digitar gcc cenario3.c
- Digitar ./a.out
- O algoritmo será executado.

No cenário 3, o método CountingSort teve um arquivo separado, para a compilar basta:

- Acessar a pasta do trabalho,na subpasta CENÁRIO 3
- Abrir o terminal
- Digitar no terminal gcc counting.c
- Digitar ./a.out ou utilizar o valgrind ./a.out para a verificação de memória
- O algoritmo será executado

OBS : No cenário 2 e 3 é necessário a alteração do tamanho máximo e e da semente no código para cada execução, para a análise completa.

4 CONCLUSÃO

Este trabalho teve o objetivo de fixar os conhecimentos obtidos em sala de aula, como também realizar o teste prático da eficiência dos algoritmos para diferentes tipos de estrutura de dados (Cenário 1). Também mostrou que a maneira que se implementa um algoritmo de ordenação é importante, pois ele pode se tornar mais eficiênte apenas com algumas alterações (Cenário 2).Foi perceptivél também como as limitações de hardware, atrapalham a execução dos algoritmos, assim limitando-os quando se tratava de ordenações com vários elementos. A principal dificuldade encontrada foi a limitação da máquina pois esta atrapalhou a realização da análise, existindo testes que demoraram horas para gerar as estatiscas, isto quando o sistema não travava e vinha a se reiniciar fazendo que todos os dados fossem perdidos. Mas apesar das limitações pode-se tirar boas conclusões sobre cada algoritmo e suas possiveis situações.

Referências

[1] CELES, W.; CERQUEIRA, R.; RANGEL, J. L. *Introdução a Estrutura de Dados: Com técnicas de programação em C*. 4º edição. ed. [S.l.]: Campus, 2004.

[2] ZIVIANI, N. *Projeto de Algoritmos*. [S.l.]: Thomson, 2004.

[3] BUBBLESORT. Disponível em: <https://pt.wikipedia.org/wiki/Bubble$_sort$>.

[4] MERGESORT. Disponível em: <https://pt.wikipedia.org/wiki/Merge$_sort$>.

[5] SELECTIONSORT. Disponível em: <https://pt.wikipedia.org/wiki/Selection$_sort$>.

[6] QUICKSORT. Disponível em: <https://pt.wikipedia.org/wiki/Quicksort>.

[7] AVA Lista Dupla. Disponível em: <http://200.129.241.132/mdl$_ie$ − $cuv/mod/resource/view.php?id = 3921$>.

[8] QUICKSORT Interativo. Disponível em: <https://www.ime.usp.br/ pf/algoritmos/aulas/solucoe
it.html>.