Estudo sobre algoritmos de ordenação

Leonardo Valério Anastácio

[1]Departamento de Ciência da Computação – Universidade do Estado de Santa Catarina

Resumo. *Este trabalho visa analisar os principais algoritmos de ordenação encontrados na literatura - Bubble, Insert, Merge, Heap, Quick (utilizando duas técnicas diferentes de pivoteamento). Além desses considerados indispensáveis, será também explorado dois algoritmos de ordenação considerados com complexidade linear - Bucket e Counting.*

1. Detalhes de Implementação

1.1. Estruturamento

O projeto foi desenvolvido na linguagem C++, para organização do código utilizou-se um namespace pai chamado de "sort", e dentro desse bloco seria definido namespaces filhos para implementação de cada algoritmo de ordenação. A princípio, toda essa parte do projeto seria abstraída em uma Classe, entretanto, após algumas codificações foi visto que essa classe seria estática. Ou seja, em nenhum momento do código iríamos, por exemplo, instanciar uma Classe QuickSort para ordenar um vetor não seria conveniente. Para isso a utilização do namespace, que simplifica o código no momento de executar os algoritmos. Assim, para executar um método de ordenação, por exemplo HeapSort, basta lançar em qualquer parte do código "sort::heap::run(vetor)", sem a necessidade de instanciar classes. Por esse motivo, e pela simplicidade de implementação, foi escolhido trabalhar com namespaces.

1.2. Arquivos de entrada

Serão gerados em execução 4 tipos de entrada, que serão vetores com 25K, 50k, 75k, 100k, 1M de tamanho, esses serão alimentados com valores numéricos entre 0 e tamanho do vetor. Cada vetor será atribuído ao namespace "sort" e são representado por um "vector< long long >". Além disso, cada um possui quatro versões diferentes de acordo com a especificação do trabalho.

1.3. Execução dos algoritmos

Os testes foram desempenhados em um computador com 4 núcleos físicos e que contava com 8gb de memória RAM DDR3. Visto que a máquina de execução dos testes é multicore, cada teste do algoritmo foi executado em um processo filho, ou seja, a cada teste é executado simultaneamente os 4 tipo de vetores no algoritmo informado. A utilização de processos ao invés de threads se deve ao fato de não existir maneira de interromper/parar/matar uma thread, ou seja, é preciso usar flags o que tornaria a implementação custosa e diferente para cada algoritmo. Enquanto que ao utilizar multi processamento, o processo pai a todo momento pode interromper um processo filho, assim, ao passar 5 minutos (time out) o processo pai cancela/mata a execução do filho.

2. Definições

Os elementos dos vetores não se repentem e vão de 0 até o tamanho - 1 do vetor. A seguir segue uma lista de consulta para auxílio nos gráficos:

- Ordenado -> Vetor Ordenado de forma Crescente
- Ordenado (*) -> Vetor Ordenado de forma Decrescente
- Randon -> Vetor desordenado
- Randon (*) -> Vetor desordenado e com elemento 100.000.000
- 25K -> Vetor com 25.000 elementos
- 50K -> Vetor com 50.000 elementos
- 75K -> Vetor com 75.000 elementos
- 100K -> Vetor com 100.000 elementos
- 1M -> Vetor com 1 milhão de elementos

3. Bubble Sort

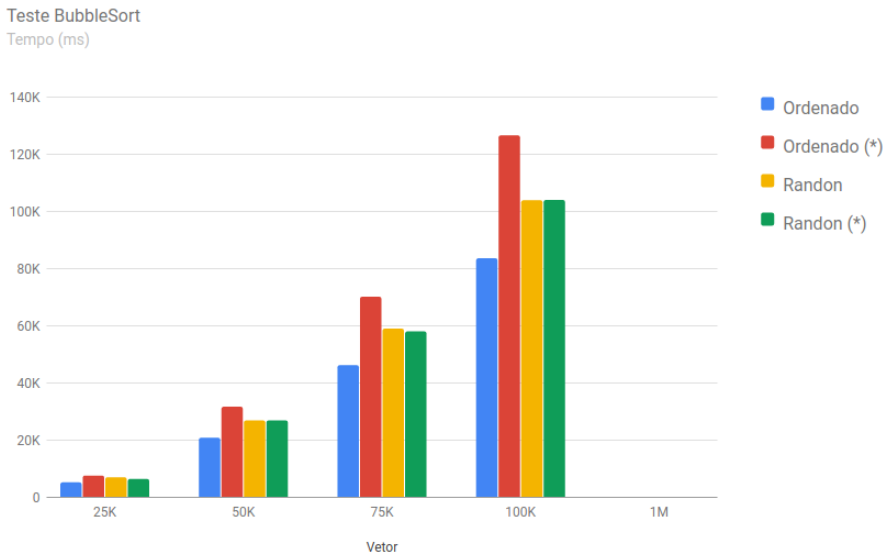

Figure 1. Desempenho do BubbleSort

Considerado o método de ordenação mais simples, possui complexidade $\theta(N^2)$, ou seja, em toda execução percorrerá o vetor de início a fim N vezes, seu único benefício é o espaço constante $O(1)$. Pois todas suas trocas são feitas no vetor de origem. Seu pior caso é caracterizado quando ordena um vetor organizado de forma decrescente, pois a cada iteração estará fazendo trocas. Por consequência, seu melhor caso é quando o vetor está de forma crescente, pois em nenhum momento entrará na condição de troca de elementos. Seu tempo ao executar o vetor de 1M elementos excedeu o limite aceitável de 5 minutos, logo seu resultado foi desconsiderado.

4. Insertion Sort

Como o Bubble, o Insert é um algoritmo Naive, possui complexidade para pior caso na ordem $O(N^2)$, é caracterizado quando o vetor está de forma decrescente, pois a cada iteração serão feitas novas trocas. Melhor caso é de ordem $\Omega(N)$, é quando o vetor está organizado de forma crescente. No gráfico é difícil de reparar a barra do melhor caso,

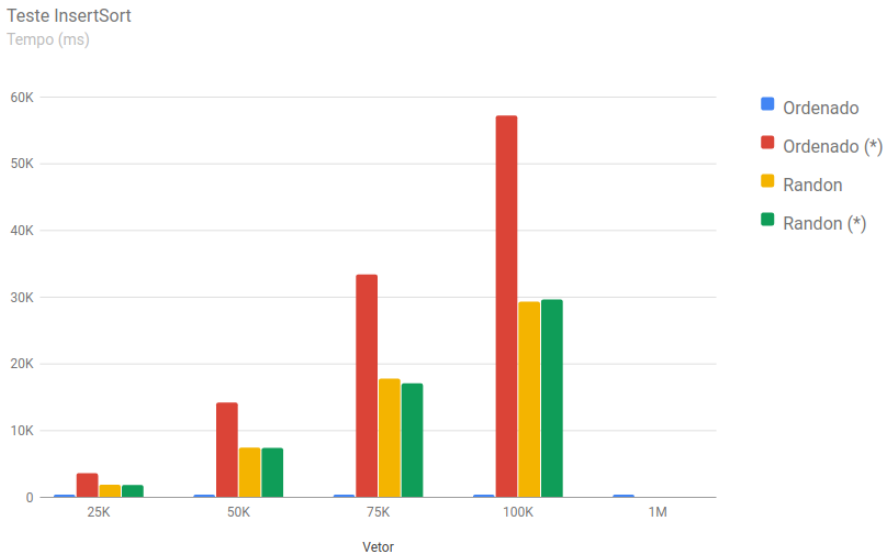

Figure 2. Desempenho do InsertSort

pois nessa ocasião específica esse algoritmo obteve desempenho ótimo com tempo médio de 125 ms. Já para seu pior caso estourou o tempo limite aceitável de 5 minutos, ou seja superior a 300K ms. E para os vetores com elementos randomicos obteve o resultado esperado, que seria próximo a média do melhor e pior caso.

5. Merge Sort

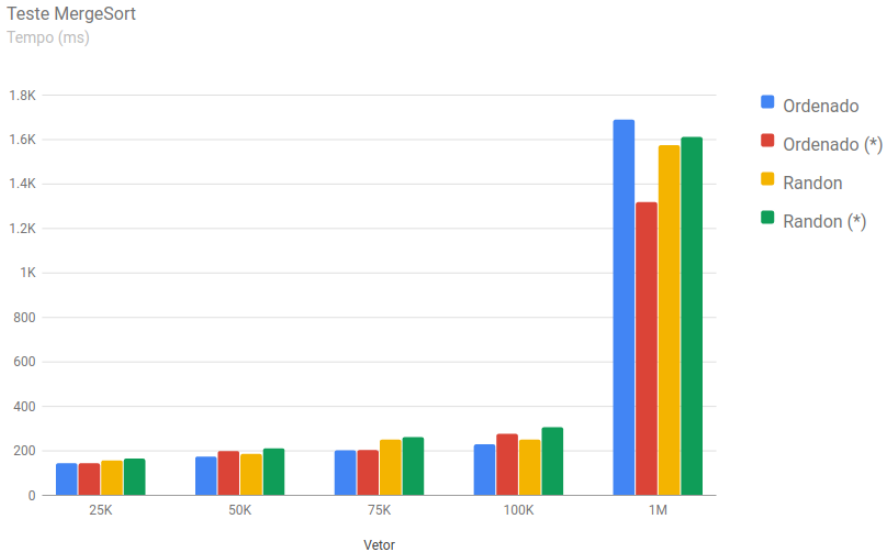

Figure 3. Desempenho do MergeSort

No gráfico da Figura 3 percebe-se que não há melhor caso para o mergeSort, pois a cada teste efetuado tanto o melhor quanto o melhor caso alterava. Isso é devido ao fato, creio eu, de ser um algoritmo estável, além disso, possui complexidade na ordem de $\theta(nlog(n))$. Quanto a sua complexidade de espaço $O(n)$, pois o algoritmo cria uma cópia do vetor original para cada chamada recursiva, caracterizando a sua principal desvantagem.

6. QuickSort

Assim como o Mergesort utiliza divisão e conquista, sua complexidade de espaço para o melhor caso é $\Omega(log(n))$ que é a altura máxima da árvore de recursão e tempo $\Omega(nlog(n))$, que é identificado quando o algoritmo de particionamento divide a lista em duas partes iguais. Além disso, seu espaço e tempo para o pior caso é $O(n)$ e $O(n^2)$, ou seja a partir de um pivoteamento ruim o algoritmo gera uma árvore recursiva de forma linear, que deteriora sua complexidade de tempo tornando igual um algoritmo Naive.

6.1. QuickSort (1) - Pivô fixo

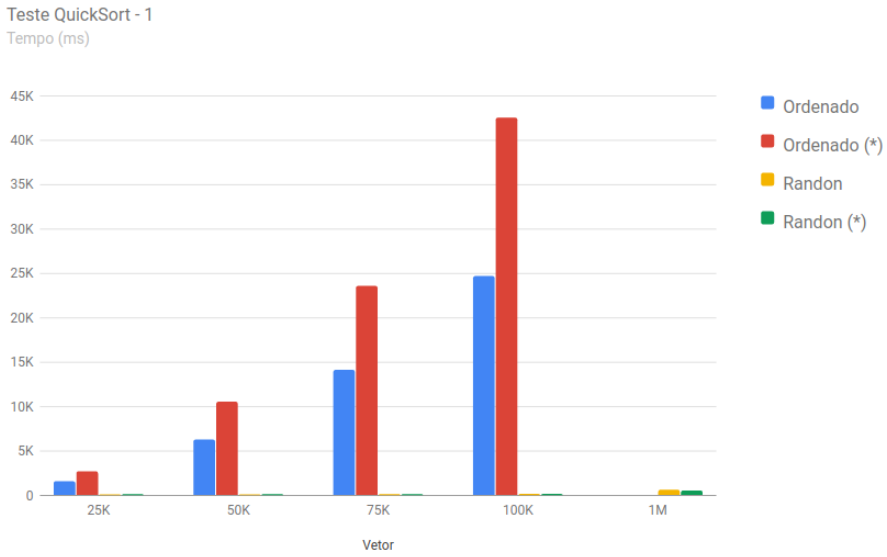

Figure 4. Desempenho do Quicksort (1)

No gráfico da figura 4 percebe o péssimo desempenho do quicksort em sua versão de pivô fixo na tentativa de organizar vetores ordenados. Em contrapartida, ao ordenar vetores randômicos esse algoritmo obteve desempenho ótimo, visto que para organizar o vetor 1M precisou de 500ms, ou seja, um tempo três vezes mais rápido que o MergeSort no mesmo teste. O tempo para ordenar os vetores organizados 1M (crescente/decrescente) foram desconsiderados pois estouraram o tempo limite de 5 minutos.

6.2. QuickSort (2) - Pivô aleatório

No gráfico da figura 5 está o tempo do QuickSort usando técnica de pivoteamento aleatório, obteve um resultado muito mais sólido e constante que sua versão com pivô fixo. Percebeu-se que a cada novo teste executado o melhor e o pior caso alteravam, visto que algumas vezes o pivô escolhido particionava com mais eficiência a árvore que outras vezes.

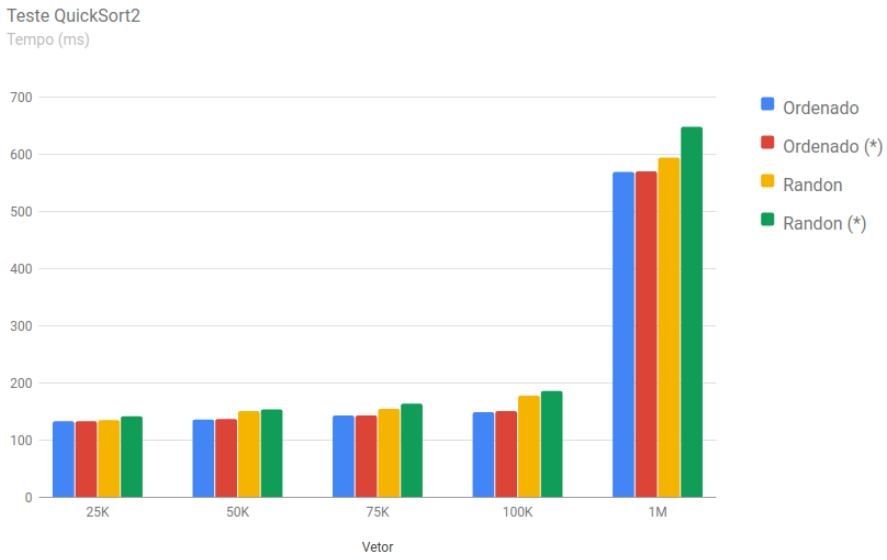

Figure 5. Desempenho do Quicksort (2)

7. HeapSort

O Heap sort assim como o Merge é um algoritmo de ordem de tempo $\Omega(nlog(n))$ e espaço $\Omega(n)$ (para a Heap). Nos testes feitos, como ilustra a figura 6, este algoritmo mostrou mais eficiência ao ordenar vetores organizados (crescente/decrescente). Creio que mais trocas na heap são feitas quando um vetor desorganizado é ordenado usando esse algoritmo.

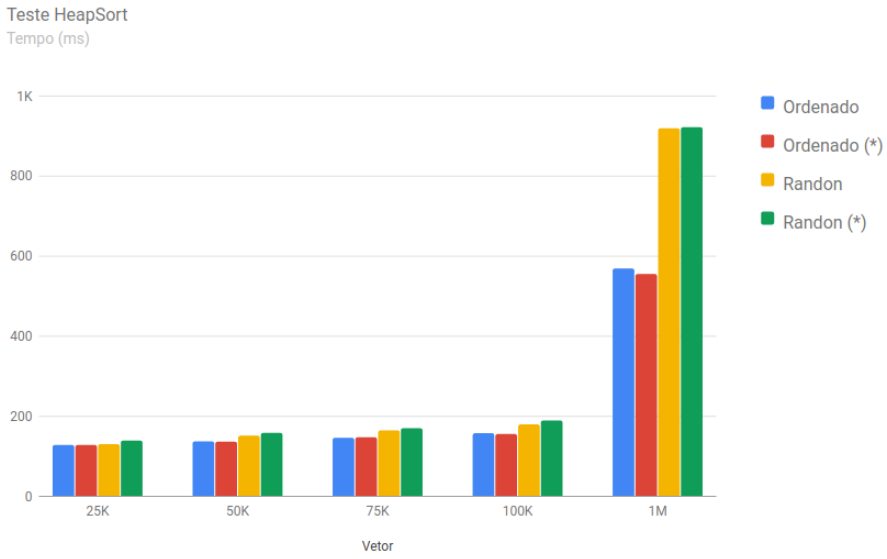

Figure 6. Desempenho do Heapsort

8. Counting e Bucket Sort

O Bucket e o Counting apresentaram resultados muito eficientes para organização dos vetores em ordem, e aleatório. Isso porque foi testado um caso extremamente específico que os valores vão de 0 até o tamanho-1 do vetor, nessas ocasiões esses tipo de algoritmos lineares obtém um ótimo desempenho. Apesar do counting possuir desempenho melhor

que o bucket na ordenação de vetores em ordem e aleatório, em todos os teste do vetor Randon(*) ele se deteriorou. Já o Bucket mostrou um pouco mais de dificuldade para organizar esse tipo de vetor, mas mesmo assim apresentou um resultado uniforme e coerente com o esperado. Além disso, cada bucket foi representado como um ”std::set”, os elementos são inseridos em ordem com complexidade de $\theta(nlog(n))$.

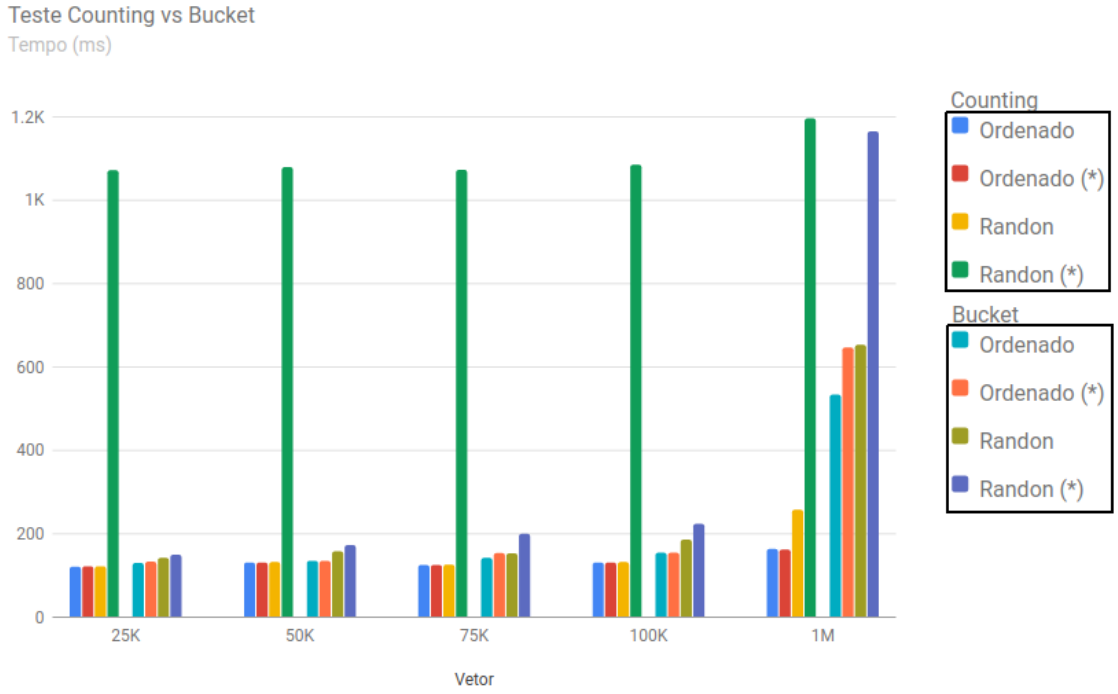

Figure 7. Desempenho do Counting e Bucket Sort

9. Conclusões

10. Teste na categoria Crescente

No teste da figura 8 nota-se que o melhor algoritmo encontrado para ordenar um vetor, já ordenado de forma crescente, é o Insert Sort com o tempo de 129ms, enquanto que o heap e o quick(2) obtiveram pontuações semelhantes, já o merge demonstrou pouco desempenho comparado aos outros algoritmos para esse tipo de entrada.

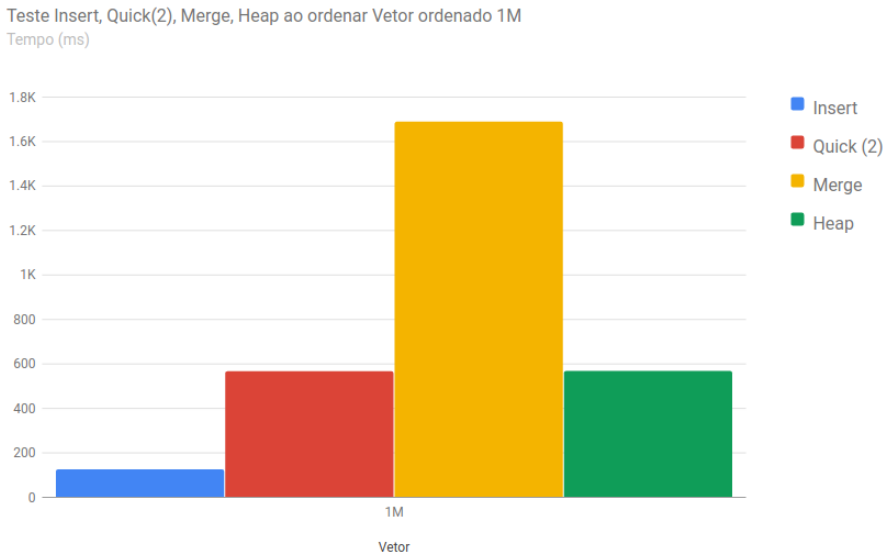

Figure 8. Teste Insert, Quick(2), Merge, Heap ao ordenar Vetor ordenado 1M

11. Teste na categoria Randon

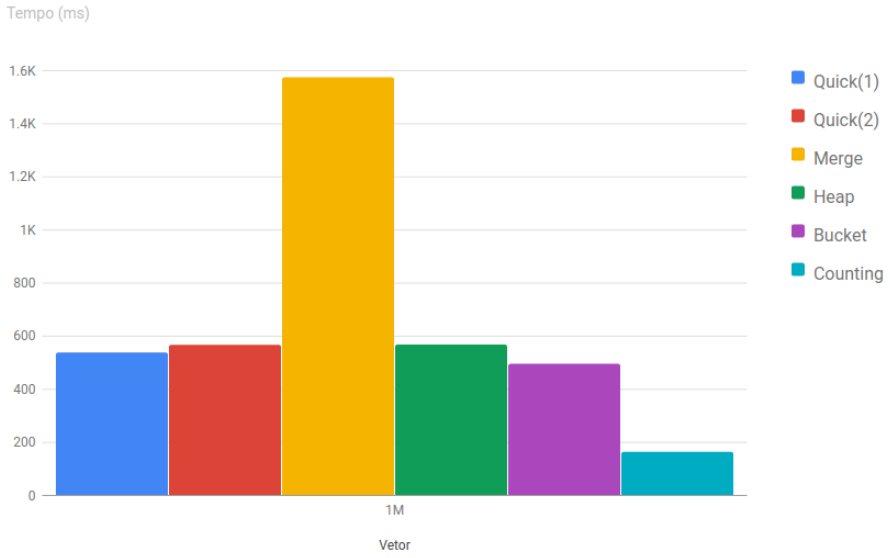

Figure 9. Teste Quick (1) VS Quick (2) VS Merge VS Heap VS Bucket VS Counting Vetor Randon 1M

O teste da figura 9 evidenciou a eficiência dos algoritmos de ordenação Bucket e Counting Sort, o bucket com 498ms e Counting com 166ms. Mas devemos levar em consideração que a distribuição dos elementos do vetor testado é uniforme o que praticamente torna esses algoritmos com tempo linear. O terceiro mais ágil foi o QuickSort com tempo de 541ms, praticamente empatado com o heap 571ms. Os algoritmos Naive não foram registrados pois o tempo estourou o limite aceitável.

12. Teste na categoria Decrescente

O teste da figura 10 mostra um resultado muito semelhante ao da figura 8, pois agora o vetor não está crescente e sim organizado de forma decrescente. Notamos que o melhor

algoritmo continua sendo counting com 164ms. É importante frisar que nesse teste tanto o Insert, Bubble, e Quick(1) apresentaram time out devido suas características.

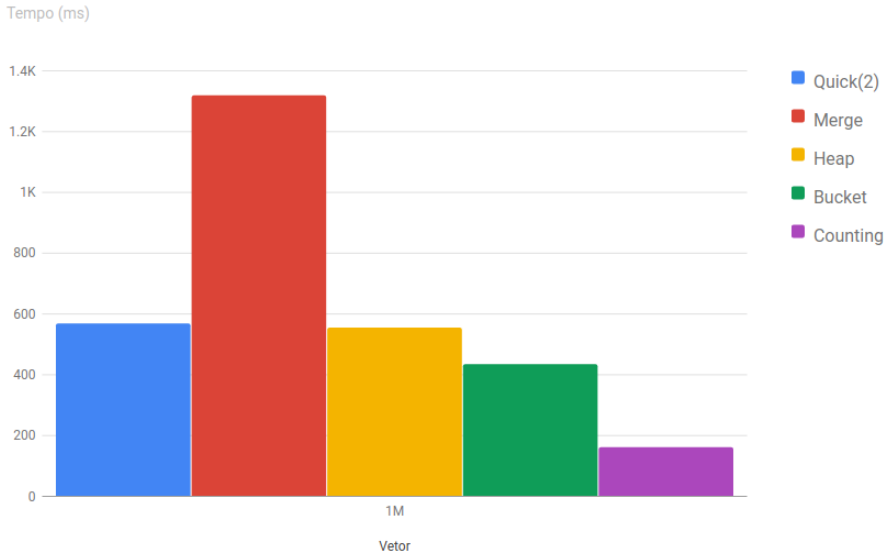

Figure 10. Teste Quick (2) VS Merge VS Heap VS Bucket VS Counting Vetor Decrescente 1M

13. Teste na categoria Random com elemento 100.000.000

O interessante do teste da figura 11 é que mesmo sendo um dos piores casos tanto para o bucket quanto para o counting, eles obtiveram resultados mais eficientes que o merge sort. Pode-se afirmar que os QuickSort's empataram em desempenho e o heap alcançou o terceiro melhor resultado.

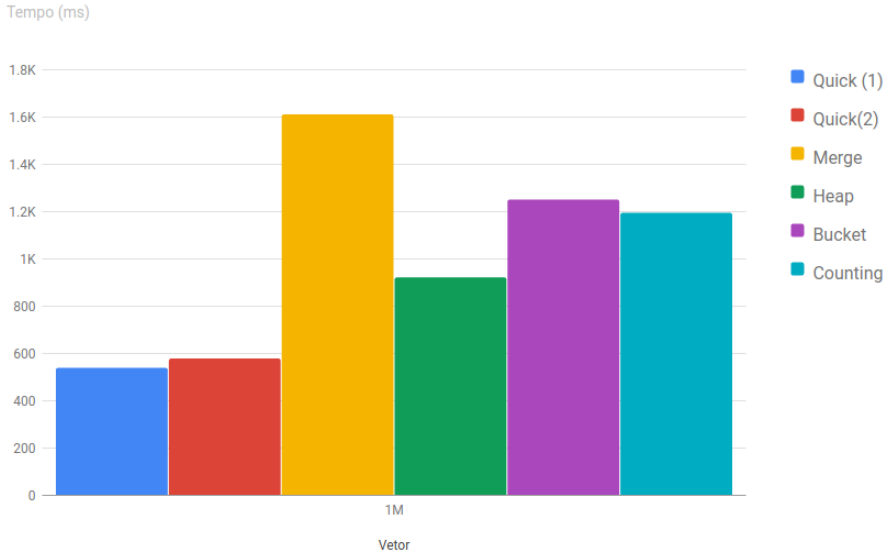

Figure 11. Teste Quick(1) Quick (2) VS Merge VS Heap VS Bucket VS Counting Vetor Decrescente 1M