UNIVERSIDADE FEDERAL DA PARAÍBA (UFPB) – JOÃO PESSOA
Centro de Informática (CI)

Ordenação 03

Alunos: Higor Anjos e Marcos Alves

Orientador: Leonardo Bezerra

Disciplina: Análise e Projeto de Algoritmos

UNIVERSIDADE FEDERAL DA PARAÍBA (UFPB) – JOÃO PESSOA
Centro de Informática (CI)

Conteúdo

1. Introdução

O presente trabalho, de caráter estritamente experimental, visa promover a discussão dos efeitos das complexidades de cada algoritmo de ordenação por comparação na prática. Os algoritmos mostrados no trabalho são de autoria dos alunos supracitados e podem ser encontrados em seus repositórios no GitHub.

Como forma de avaliar o tempo de execução de cada algoritmo em determinadas situações, estes foram submetidos a rotinas de testes configurados da seguinte forma:

- Conjunto 1: vetores 10% ordenados.
- Conjunto 2: vetores 50% ordenados.
- Conjunto 3: vetores 90% ordenados.

Ainda, os algoritmos foram implementados nas linguagens de programação C++ e Ruby e seus desempenhos em cada uma delas também foram avaliados e comparados.

2. Análise do tempo de execução na linguagem C++

Todos os algoritmos receberam 5 vetores de cada conjunto exposto anteriormente e suas médias foram calculadas para se obter o tempo médio de execução sobre cada conjunto de entrada. Vale ressaltar que os programas foram compilados com a flag de otimização O2 do compilador g++ como exemplo mostrado a seguir:

g++ -std=c++11 -O2 main.cpp -o main

2.1 Insertion Sort

O primeiro algoritmo a ser avaliado foi o Insertion Sort. Sua implementação será abstraída pelo fato desta já ter sido tratada em trabalhos anteriores, mas seu funcionamento e a influência no desempenho são fundamentais para a análise do tempo de execução.

Como mostrado na figura 1, o Insertion Sort tem tempo de execução menor em entradas com menor grau de entropia (mais ordenados). A linha amarela está muito abaixo das linhas azul e vermelha, significando que seu desempenho é bem melhor para este tipo de entrada.

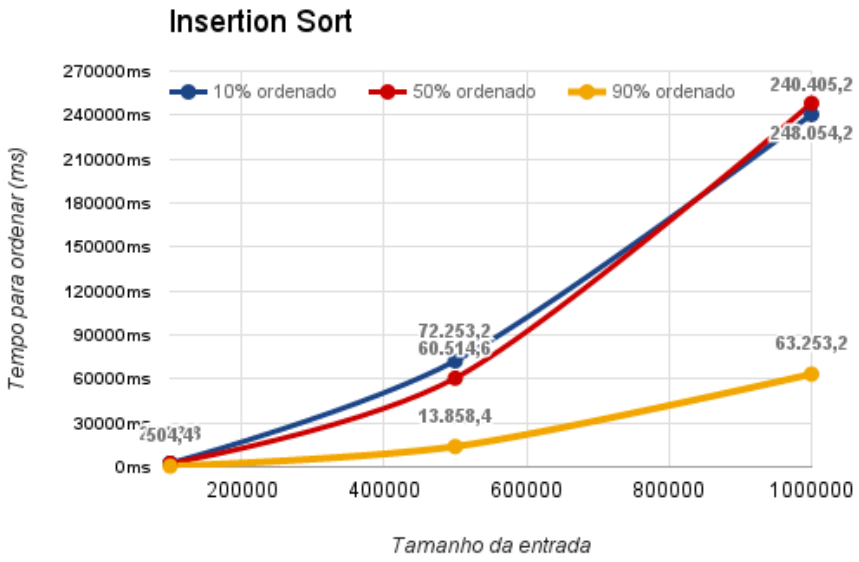

Figura 1: Gráfico comparativo do tempo de ordenação do algoritmo Insertion Sort com vetores 10, 50 e 90 por cento ordenados e com tamanho de 100.000, 500.000 e 1.000.000.

Podemos observar ainda que as curvas de crescimento assemelham-se a quadráticas à medida que o tamanho do vetor aumenta, justificando a complexidade assintótica

$O(n^2)$.

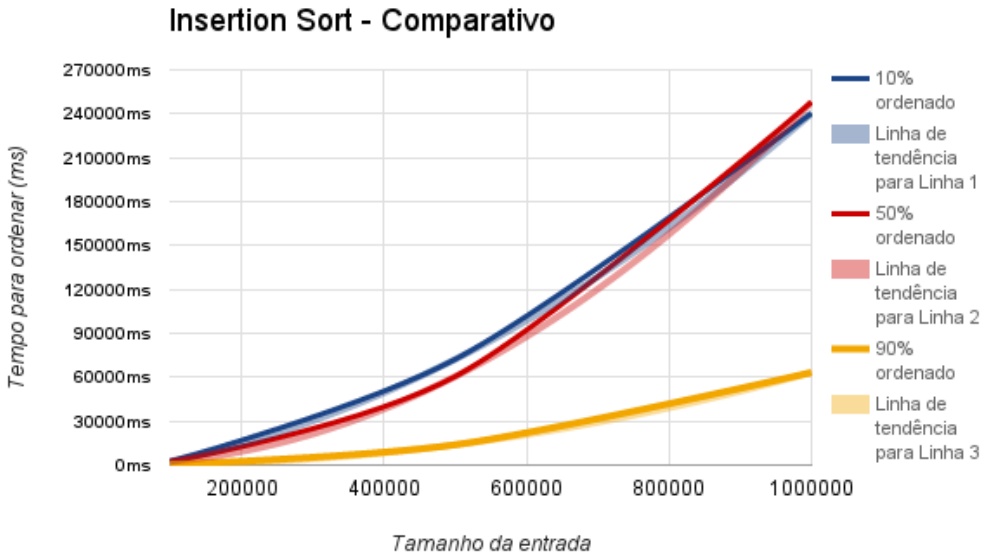

Figura 2: Gráfico comparativo entre as curvas resultantes dos testes e as de tendência polinomial de grau 2.

2.2 Selection Sort

O segundo algoritmo avaliado foi o Selection Sort. Devido às características deste algoritmo, seu tempo de execução varia muito pouco em relação ao tamanho da entrada, não possuindo um melhor caso visível. A figura 3 ilustra o tempo de execução do Selection Sort com as entradas já citadas.

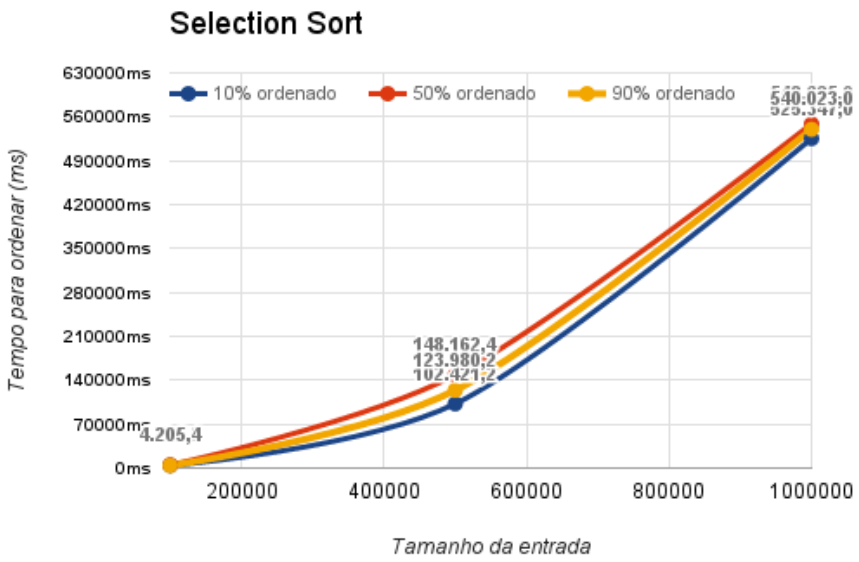

Figura 3: Gráfico comparativo do tempo de ordenação do algoritmo Selection Sort com vetores 10, 50 e 90 por cento ordenados e com tamanho de 100.000, 500.000 e 1.000.000.

2.3 Heap Sort

Começando a análise de algoritmos com a filosofia “dividir para conquistar” temos o Heap Sort. Este algoritmo possui funcionamento similar ao Selection Sort, diferenciando-se por utilizar a estrutura de heap máximo para melhorar seu desempenho. Assim, seus resultados estão mostrados na figura 4.

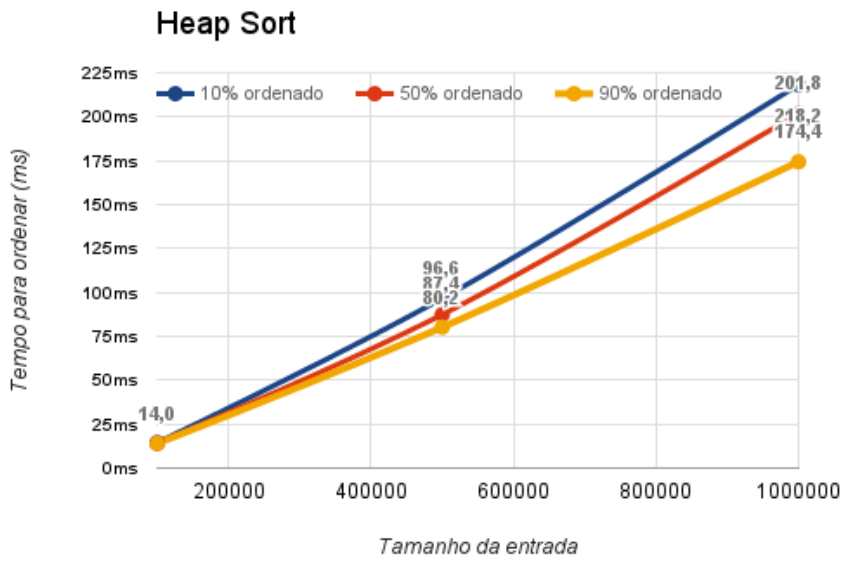

Figura 4: Gráfico comparativo do tempo de ordenação do algoritmo Heap Sort com vetores 10, 50 e 90 por cento ordenados e com tamanho de 100.000, 500.000 e 1.000.000.

2.4 Merge Sort

Este algoritmo também se enquadra na filosofia de divisão e conquista, como já é de nosso conhecimento. Seu desempenho foi bastante similar ao anterior, como visto na figura 5. Um fato interessante de se observar é que foram utilizados vetores auxiliares na divisão que eram alocados na pilha, como forma de melhorar o desempenho do algoritmo, mas quando o mesmo foi submetido aos testes com entradas de tamanho 1.000.000, o espaço da pilha do processo não foi o suficiente e o programa foi encerrado pelo sistema operacional. Para sanar tal problema, os vetores auxiliares foram alocados dinamicamente, no espaço de heap.

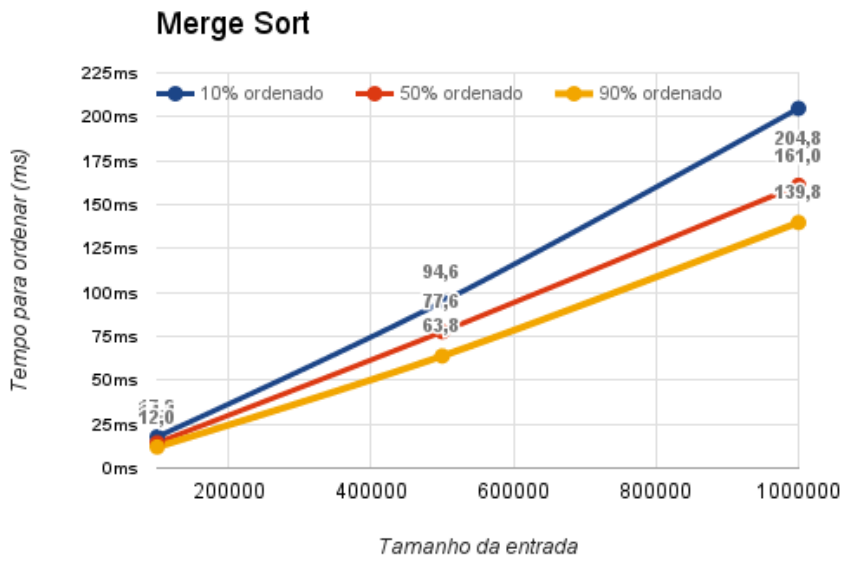

Figura 5: Gráfico comparativo do tempo de ordenação do algoritmo Merge Sort com vetores 10, 50 e 90 por cento ordenados e com tamanho de 100.000, 500.000 e 1.000.000.

2.5 Quick Sort

O algoritmo que mostrou obter o melhor desempenho foi o Quick Sort, como esperado. Assim como os demais que compartilham a característica de divisão e conquista, seus resultados variaram pouco com a mudança de entropia e mantiveram-se na mesma proporção. Um fato interessante de ser observado é a estreita relação entre a eficiência e a escolha do pivô. Os testes mostrados na figura 6 foram realizados com a escolha do pivô na última posição, se a escolha fosse realizada pelo primeiro elemento, o tempo seria totalmente diferente do exposto e alcançaria valores compatíveis com o Insertion Sort.

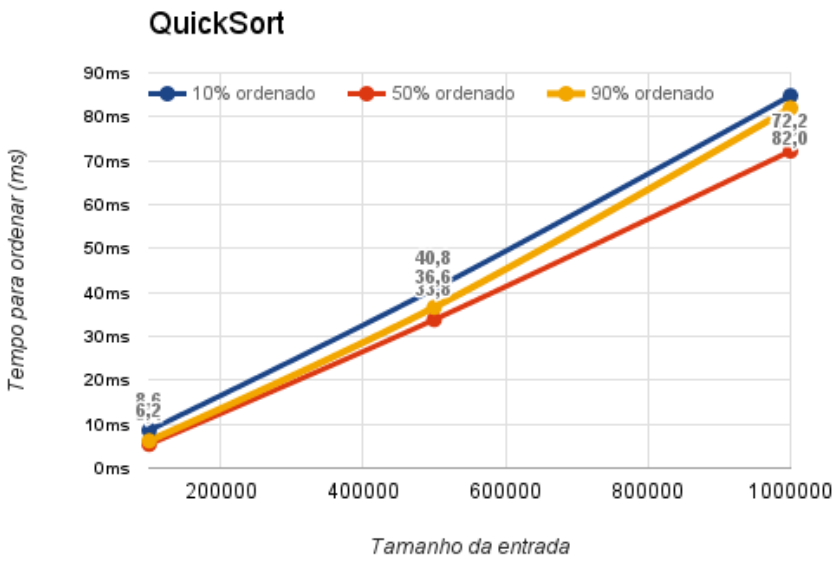

Figura 6: Gráfico comparativo do tempo de ordenação do algoritmo Quick Sort com vetores 10, 50 e 90 por cento ordenados e com tamanho de 100.000, 500.000 e 1.000.000.

2.6 Comparação entre os algoritmos

Para melhor analisar o desempenho de todos de forma geral, a figura 7 compara o desempenho de todos os algoritmos citados nesse relatório. É possível observar que os algoritmos de divisão e conquista são bem mais eficientes que os demais sendo até difícil diferenciá-los quando mostrados juntos.

Figura 7: Gráfico comparativo do tempo de ordenação dos algoritmos Selection Sort, Insertion Sort, Heap Sort, Merge Sort e Quick Sort com vetores 90% ordenados e com tamanho de 1.000.000.

2.7 Ordenação Padrão da Linguagem

A linguagem C++ possui o método de ordenação padrão da Standard Library (STL) chamado sort. Esta função opera sobre qualquer objeto que possa ser iterado e, segundo a documentação da linguagem, possui complexidade O(nlogn). Após ser submetido à mesma rotina de testes, o método obteve desempenho mais eficiente que os demais e seus resultados estão mostrados na figura 8. Isso se deve ao fato da utilização de vários algoritmos conhecidos como o Quick Sort, Heap Sort e Insertion Sort. Este método tem o Quick Sort como algoritmo principal, mas por ter o pior caso sendo $O(n^2)$, o método mantém salvo a profundidade da recursão para que, quando passe de $2logn$, o Heap

Sort seja utilizado. Esse procedimento que combina os dois algoritmos é chamado de Intro Sort.

Além disso, a escolha do pivô também é otimizada selecionando a mediana de 3 pivôs aleatórios. Por fim, quando o tamanho do vetor diminui na árvore de recursão, o algoritmo Insertion Sort é utilizado. Isso pode ser justificado pelo o que vimos anteriormente, quando o Insertion Sort mostra ser bem eficiente com vetores menores e quase ordenados.

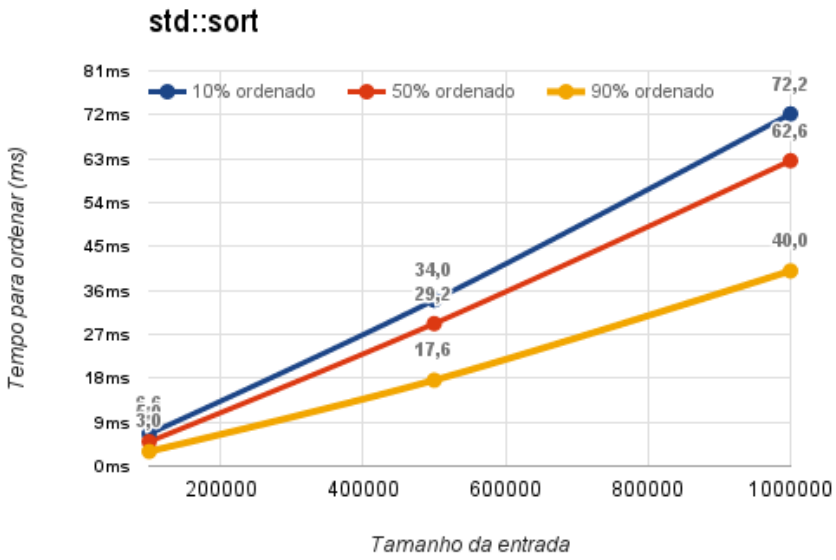

Figura 8: Gráfico comparativo do tempo de ordenação do método sort da biblioteca padrão de C++ com vetores 10, 50 e 90 por cento ordenados e com tamanho de 100.000, 500.000 e 1.000.000.

3. Análise do tempo de execução na linguagem Ruby

Levando em consideração os conjuntos descritos inicialmente, e as entradas estão definidas da seguinte forma:

- Entrada A: 1 000 000
- Entrada B: 500 000
- Entrada C: 100 000

Os algoritmos foram analisados de acordo com a tabela abaixo:

Algoritmo	Entrada
Heap	A, B e C
Insertion	Apenas C
Merge	A, B e C
Quick	A, B e C
Selection	Apenas C

Tabela 1: Como cada algoritmo foi analisado

Configurações do ambiente de testes:

Processador	Memória
Intel® Core™ i3-4005U CPU @ 1.70GHz x 4	4 GB DDR3L 1600 MHz

Tabela 2: Como cada algoritmo foi analisado

3.1 Insertion Sort

Observado apenas na entrada do tipo C (como mostra a tabela 2), o Insertion mostrou sua características principal: Trabalhar com vetores parcialmente ordenados. Na figura 9, apesar de ser apenas um tipo de entrada, podemos notar a diferença de acordo com o grau de ordenação do arquivo de entrada.

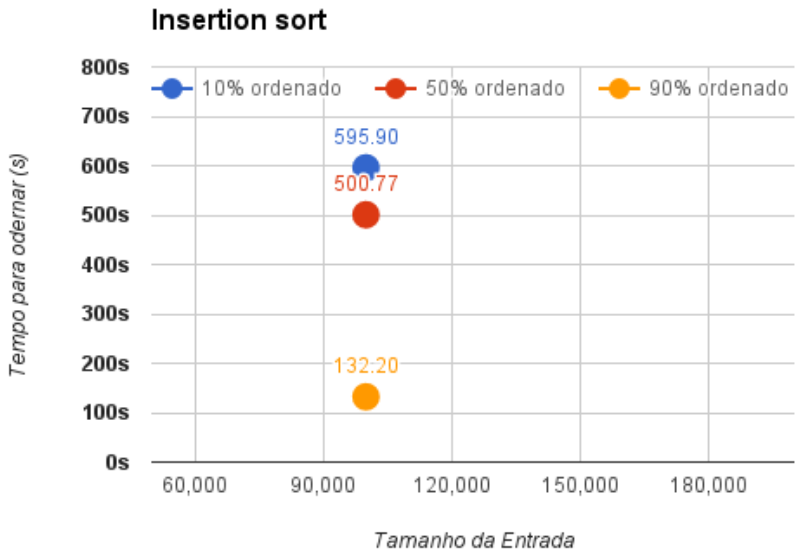

Figura 9: Gráfico comparativo do tempo de ordenação do algoritmo Insertion Sort com vetores 10, 50 e 90 por cento ordenados e com tamanho de 100.000, 500.000 e 1.000.000.

3.2 Selection Sort

O que causou mais problemas em questão de tempo, o Selection independente da entrada ele verifica todos com todos por isso sua complexidade $O(n^2)$. Observando a figura 10 fica claro a proximidade dos tempos, visto que não importa o grau de ordenação

do vetor de entrada, este algoritmo foi avaliado apenas na entrada do tipo C (como mostra a tabela 2).

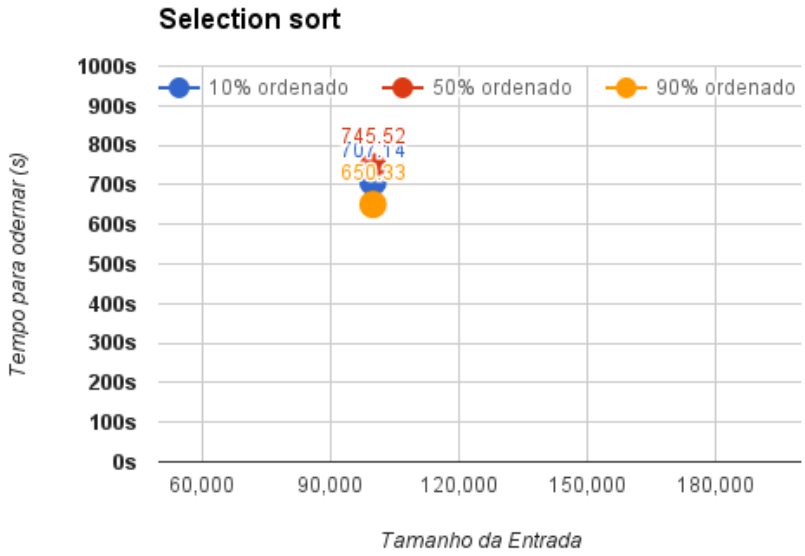

Figura 10: Gráfico comparativo do tempo de ordenação do algoritmo Selection Sort com vetores 10, 50 e 90 por cento ordenados e com tamanho de 100.000, 500.000 e 1.000.000.

3.3 Heap Sort

Este algoritmo não verifica se o vetor está ordenado, utiliza uma estrutura heap para ordenar os elementos. Podemos notar na figura 11 que não há uma distinção de tempo entre os vetores parcialmente ordenados.

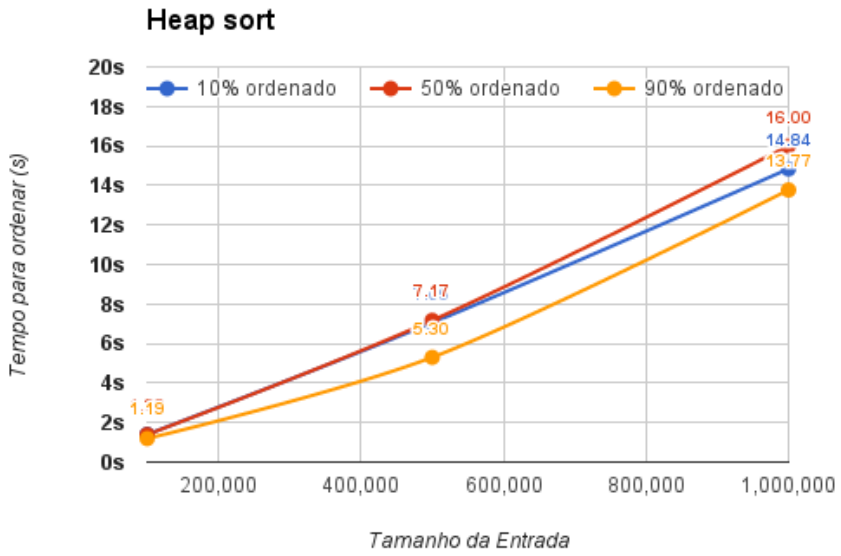

Figura 11: Gráfico comparativo do tempo de ordenação do algoritmo Heap Sort com vetores 10, 50 e 90 por cento ordenados e com tamanho de 100.000, 500.000 e 1.000.000.

3.4 Merge Sort

Apesar de não ser um algoritmo *in-place*, o tempo obtido pelo Merge sort foi extremamente satisfatório. Podemos na figura 12, que assim como Heap sort, houve a proximidade nos tempos. Vale ressaltar que a linguagem Ruby tem um excelente mecanismo de cópia, o que possibilitou este algoritmo ficar em primeiro dentre os demais.

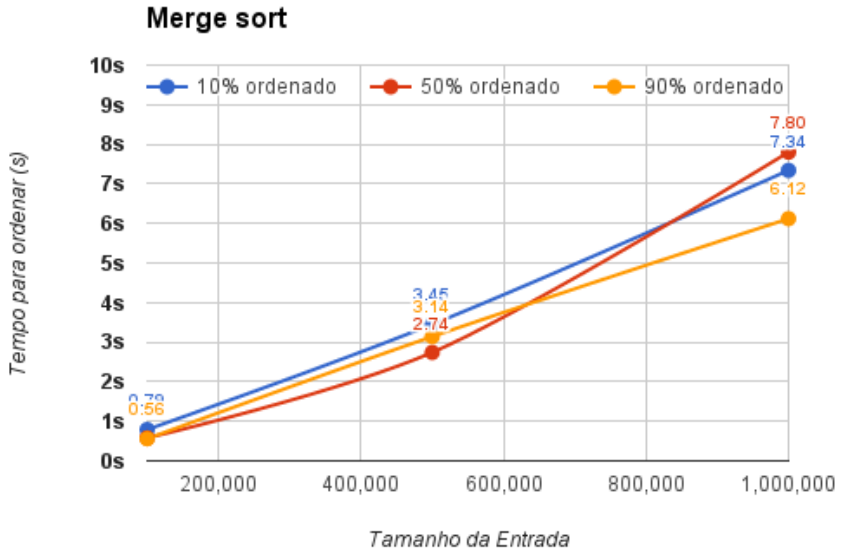

Figura 12: Gráfico comparativo do tempo de ordenação do algoritmo Merge Sort com vetores 10, 50 e 90 por cento ordenados e com tamanho de 100.000, 500.000 e 1.000.000.

3.5 Quick Sort

O Quick sort apresentou, como era esperado, um bom desempenho. Vale ressaltar que a escolha do pivô para todos casos foi o último elemento. Podemos notar na figura 13, que o tempo se elevou de acordo com o grau de ordenação do vetor, como estamos escolhendo um extremo para ser o pivô, a configuração do gráfico mostra como uma boa escolha do pivô melhora o tempo deste algoritmo.

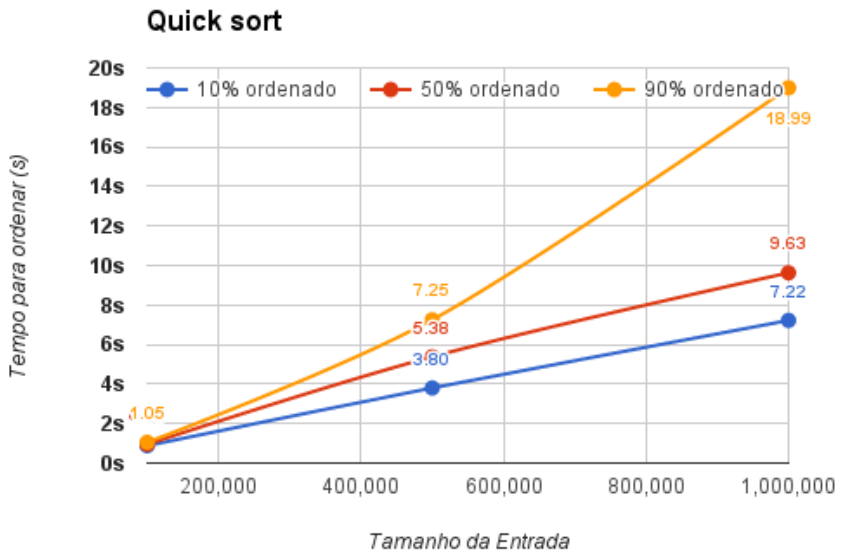

Figura 13: Gráfico comparativo do tempo de ordenação do algoritmo Quick Sort com vetores 10, 50 e 90 por cento ordenados e com tamanho de 100.000, 500.000 e 1.000.000.

3.6 Comparação entre os algoritmos

Comparando apenas os algoritmos de divisão e conquista, devido ao tempo levado pelo insertion e selection, podemos notar que o Merge e o Quick ficaram praticamente empatados, exceto quando o grau de ordenação é maior.

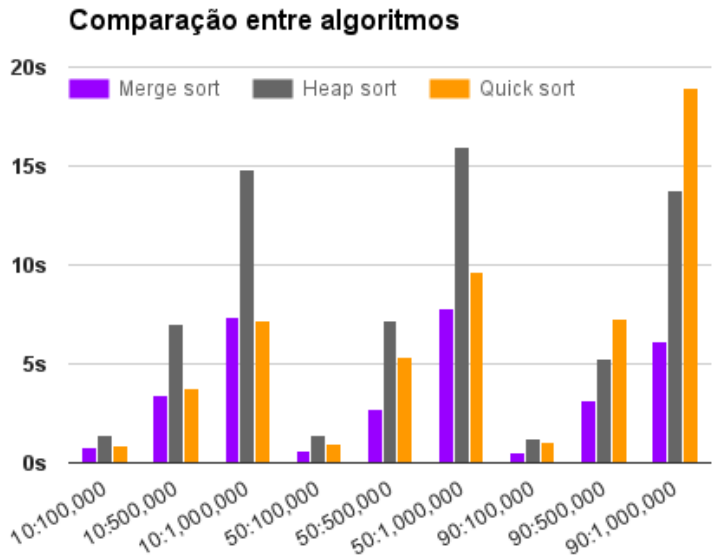

Figura 14: Gráfico comparativo do tempo de ordenação dos algoritmos Selection Sort, Insertion Sort, Heap Sort, Merge Sort e Quick Sort com vetores 90% ordenados e com tamanho de 1.000.000.

3.7 Ordenação Padrão da Linguagem

Podemos perceber como Ruby utiliza as principais características dos algoritmos de ordenação, apesar de não ficar claro na leitura do código fonte da linguagem. Podemos notar na figura 15 que todas a ordenações foram feitas abaixo 1 segundo, o que não ocorreu em nenhum dos algoritmos analisados anteriormente.

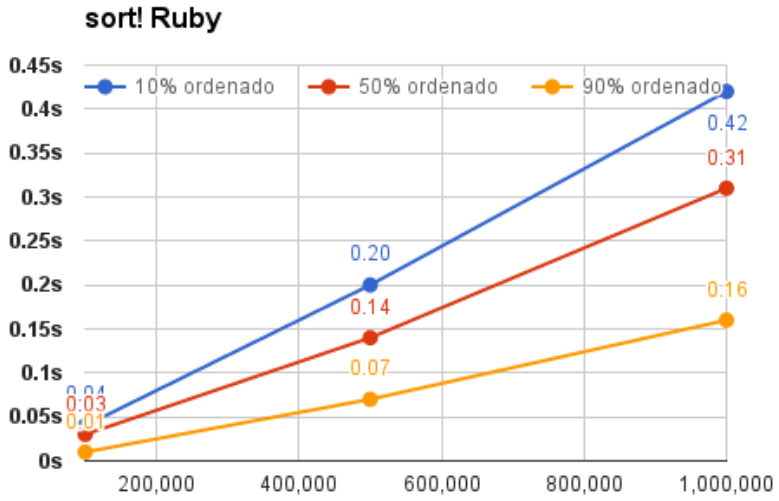

Figura 15: Gráfico comparativo do tempo de ordenação do método sort da biblioteca padrão de Ruby com vetores 10, 50 e 90 por cento ordenados e com tamanho de 100.000, 500.000 e 1.000.000.

4. C++ vs Ruby

As linguagens objetos deste estudos se diferem em vários aspectos que acabam por influenciar no modo em que os dados são tratados e, por consequência, na eficiência de cada uma.

A linguagem C++ possui características de alto desempenho e dá possibilidade ao programador de construir com o nível de abstração desejado, podendo trabalhar com orientação a objetos, programação imperativa, programação genérica e *metaprogramming*. Além disso, C++ possui tipagem estática e seus compiladores modificam os códigos, quando possível, para que estes rodem de forma otimizada. Ou seja, esta linguagem possui características determinantes para utilização em cenários onde o desempenho é prioridade, mesmo sendo uma linguagem de propósito geral.

Ruby por sua vez, é uma linguagem de programação interpretada multiparadigma, de tipagem dinâmica, com gerenciamento de memória automático, planejada e desenvolvida para ser usada como linguagem de script. Uma de suas principais características é a expressividade que possui e simplicidade na escrita dos códigos.

Após apresentação de características de ambas, já é possível prever o comportamento dos algoritmos quando submetidos às rotinas de testes já citadas anteriormente. Como forma de obter um dado visual sobre a diferença entre as linguagens, a figura 16 agrupa os resultados anteriores de ambas as linguagens em um mesmo contexto. Para prover uma disposição mais clara e limpa, foram considerados apenas os resultados dos algortimos baseados em divisão e conquista sobre vetores de tamanho 1.000.000 do Conjunto 3 (90% ordenado).

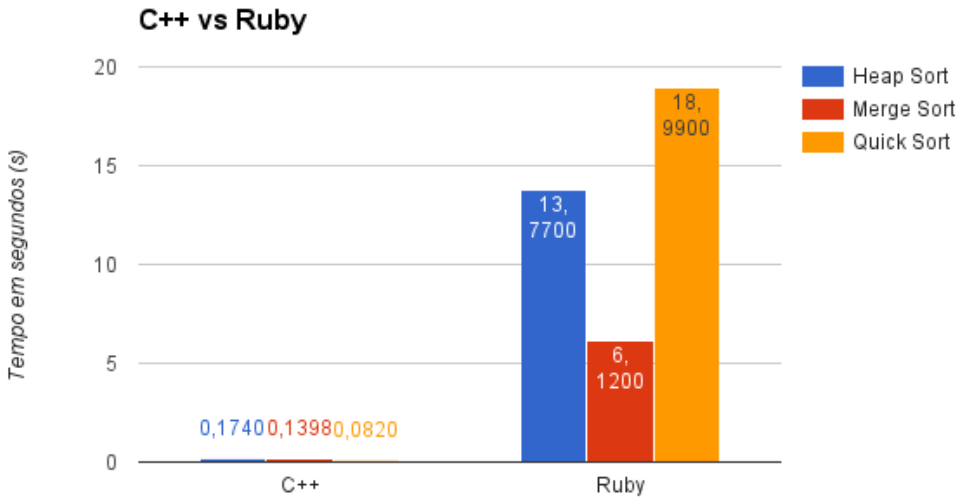

Figura 16: Gráfico comparativo do tempo de ordenação dos algoritmos Heap, Merge e Quick Sort sobre um vetor de tamanho 1.000.000 e 90% ordenado nas linguagens C++ e Ruby.

É possível notar a grande diferença de desempenho entre as duas linguagens. Pelas características já citadas anteriormente,a linguagem Ruby deve checar constantemente os tipos antes de realizar operações. Procedimento este que acaba por gerar flags e tests a mais, que geram custos. Ainda, na linguagem Ruby, tudo é tratado como Objeto e ocupa mais espaço em memória, pois junto com o dado propriamente dito, são carregadas diversas outras informações que compõem um objeto.

Em C++, por outro lado, tipos primitivos podem ser manipulados sem estarem encapsulados em objetos e todos os seus tipos são verificados em tempo de compilação, não havendo necessidade de performar checagens durante tempo de execução.

Vale ressaltar que dentre vários aspectos, estes foram elicitados e considerados mais

influenciadores nos resultados pelos integrantes deste trabalho, podendo haver muitos outros que possam influenciar de uma forma bem mais efetiva.

Dessa forma, acreditamos que apesar de analisarmos assintóticamente o desempenho dos algoritmos de ordenação apresentados neste estudo, existem muitas variáveis que influenciam de fato no desempenho que apenas na prática podem ser identificadas e contornadas e, por isso, há grande discrepância no tempo de processamento de algoritmos de mesma complexidade.