Universidade Federal da Paraíba
Centro de Informática

Análise e Projeto de Algoritmos

Relatório Ordenação 02

Alunos	Diego Filipe Souza de Lima (C++) - 11408104
	Matheus Lima Moura de Araújo (Java) - 11409518
Professor	Leonardo Cesar Teonacio Bezerra

João Pessoa, 05 de fevereiro de 2017

Conteúdo

1 Introdução

O presente trabalho, proposto durante as aulas da disciplina de Análise e Projeto de Algoritmos na Universidade Federal da Paraíba, tem como principal objetivo a realização de um estudo prático e experimental com o intuito de observar e analisar, em um contexto real, o comportamento de algoritmos de ordenação como o Selection sort, Insertion Sort, Merge sort, Quick sort e Heap sort em duas linguagens de programação diferentes, Java e C++. Além disso, também será comparado os algoritmos implementados com os algoritmos de ordenação padrão das linguagens citadas.

1.1 Ambiente de Testes

Na implementação desse trabalho foram utilizadas as linguagens de programação C++ e o compilador G++, Java 1.8.0_121 e o compilador javac. Os testes executados nos programas desenvolvidos em C++ e Java foram realizados em uma máquina com 8GB(gigabyte) de memória DDR3, processador Intel® CoreTM i7-6500U CPU @ 2.50GHz x 4 e sistema operacional Ubuntu 16.04. As instâncias utilizadas para realização dos testes de ordenação foram disponibilizadas pelo professor Leonardo Bezerra no site da disciplina [1].

1.2 Procedimento Experimental

Para obtenção dos dados utilizados na análise do nosso experimento, executamos os algoritmos de uma forma específica. Foram utilizados três subconjuntos diferentes possuindo as seguintes características: O primeiro subconjunto contém vetores 10% ordenados (90% de entropia). O segundo subconjunto contém vetores 50% ordenados (50% de entropia). O terceiro subconjunto contém vetores 90% ordenados (10% de entropia). Cada subconjunto é dividido em três tamanhos de vetores (100.000, 500.000 e 1.000.000 de elementos) contendo dez exemplos de cada tipo.

Cada grupo de instâncias com mesma característica, ou seja, grupos de dez instâncias, foram executadas e calculadas o tempo médio transcorrido entre cada um dos cinco algoritmos. Ao final são comparados os resultados entre os diferentes algoritmos, linguagens e seus respectivos algoritmos padrões de ordenação. Caso o tempo do algoritmo ultrapasse 5 minutos, desconsideraremos a instância em questão, marcando o tempo como *overtime*.

A fim de analisar os dados obtidos, além de tabelas, foram construídos gráficos utilizando a biblioteca de plotagem para python Matplotlib onde o custo médio de tempo são utilizados para gerar os pontos.

A implementação dos algoritmos será abstraída, pois foi o foco em trabalhos anteriores, mas seu funcionamento e a influência no desempenho são fundamentais para a análise do tempo de execução. Outro ponto a ser ressaltado é que para evitar informações repetitivas nesse relatório, a análise de complexidade e informações cruciais sobre os algoritmos, serão informadas na sessão de análise do Java, ficando para a sessão em C++ as informações mais estritamente sobre a análise nessa linguagem.

2 Algoritmos de ordenação

2.1 JAVA

2.1.1 Selection sort

A ordenação por seleção (do inglês, *selection sort*) é um algoritmo de ordenação considerado simples. Ele é baseado em analisar todo o vetor, buscando o menor elemento, e só assim efetuar a troca. E assim é feito sucessivamente com os $n-1$ elementos restantes, até os últimos dois elementos. Por conta disso, para todos os casos sua complexidade será $O(n^2)$, pois sempre serão executados os dois laços do algoritmo, o externo e o interno. Devido a estas características, este algoritmo apresentou um tempo de execução elevado em relação aos demais analisados, sendo este o único que possui casos que ultrapassam o tempo limite proposto de 5 minutos. Isso ocorreu em dois casos específicos, o primeiro foi quando o vetor possuía tamanho de 1.000.000 de elementos e nível de entropia de 90% e no segundo caso possuía mesmo tamanho da primeira situação, porém com 50% de entropia. Para o último exemplo de 1.000.000 de elementos, devido ao seu baixo nível de entropia houve um menor número de trocas no vetor, mesmo tendo que percorrer todo vetor, obtendo assim um resultado dentro do limite, porém bastante próximo ao limiar estabelecido. A tabela 1 apresenta os resultados obtidos por este algoritmo.

A partir dos resultados obtidos podemos perceber as principais característicos do algoritmo, visto que seu tempo de execução possui baixa variância com relação ao tamanho de entrada.

Entropia	Tamanho do vetor	Tempo médio (ms)
90%	100.000	3293
90%	500.000	103645
90%	1.000.000	*overtime*
50%	100.000	3085
50%	500.000	84432
50%	1.000.000	*overtime*
10%	100.000	3001
10%	500.000	72752
10%	1.000.000	293482

Tabela 1: Resultados vetor ordenado por Selection Sort - JAVA

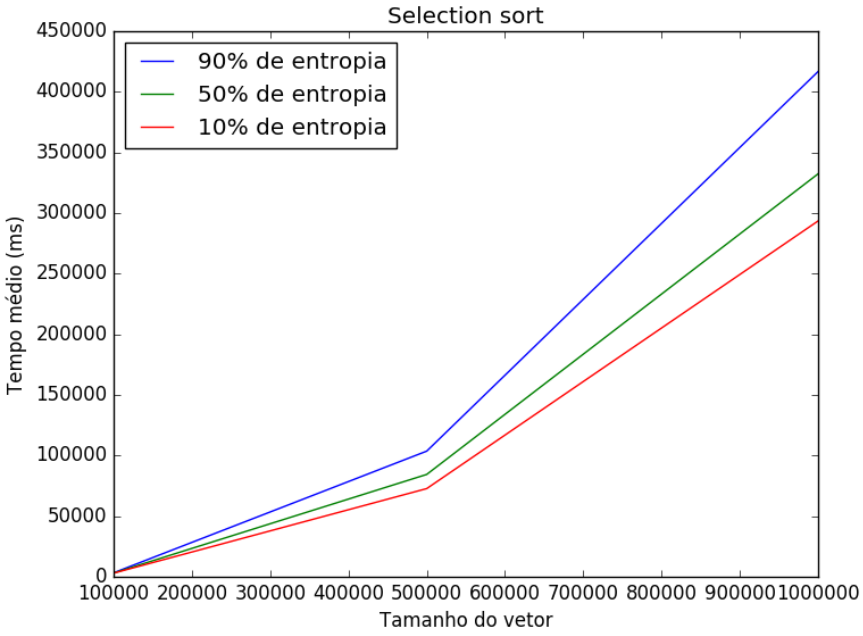

Figura 1: Gráfico comparativo do desempenho do Selection Sort com diferentes tipos de entrada

2.1.2 Insertion sort

O *insertion sort* também é considerado um algoritmo simples de ordenação, eficiente quando aplicado a um pequeno número de elementos. Isso justifica o grande "salto" nos resultados à medida que aumentou o número de instâncias. Este algoritmo apresentou um melhor desempenho comparado ao *selection sort*, visto que a principal característica do *insertion sort* con-

siste em ordenar o vetor utilizando um sub-vetor ordenado localizado em seu início, e a cada novo passo, acrescenta-se a este sub-vetor mais um elemento, até que atinja o último elemento do vetor tornando-o assim ordenado, ou seja, não são todos os casos em que o algoritmo necessita passar por todo o arranjo. Um exemplo disso está na figura 2, visto que a linha vermelha está bem abaixo das linhas azul e verde, uma vez que o vetor possui baixa entropia e o *insertion sort* opera com uma menor complexidade, diferentemente do *selection sort* que sempre opera com $O(n^2)$. A tabela 2 apresenta os resultados obtidos por este algoritmo.

Entropia	Tamanho do vetor	Tempo médio (ms)
90%	100.000	1221
90%	500.000	32363
90%	1.000.000	121592
50%	100.000	974
50%	500.000	24533
50%	1.000.000	94589
10%	100.000	253
10%	500.000	6159
10%	1.000.000	24061

Tabela 2: Resultados vetor ordenado por Insertion Sort - JAVA

Os resultados confirmam o fato de que quanto menor entropia (ou maior nível de ordenação), maior será o desempenho do *insertion sort*. É notável ainda que as curvas de crescimento assemelham-se a quadráticas à medida que o tamanho do vetor aumenta, justificando assim sua complexidade assintótica $O(n^2)$. O gráfico da figura 2 demonstra bem esta situação.

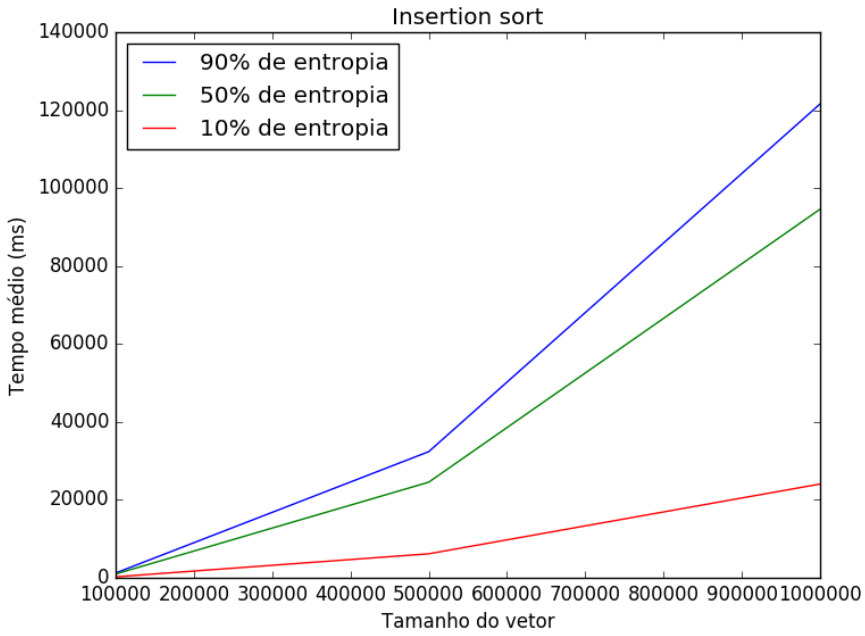

Figura 2: Gráfico comparativo do desempenho do Insertion Sort com diferentes tipos de entrada

2.1.3 Quick sort

Quick sort utiliza o conceito de elemento pivô para dividir o problema em subproblemas. A partir de um pivô, o algoritmo rearranja as chaves de modo que as chaves "menores"precedam as chaves "maiores". Em seguida o *quick sort* ordena as duas sub listas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada. O pivô utilizado neste exemplo é sempre o primeiro elemento do vetor. É importante ressaltar que o processo de escolha de pivô está fortemente associado ao desempenho do algoritmo.

O desempenho deste algoritmo comparado aos dois anteriores é consideravelmente melhor, tornando-os inviáveis para este tipo de situação comparado ao *quick sort*.

O gráfico presente na figura 3 apresenta crescimentos semelhantes à medida que o número de elementos aumenta. Tal comportamento foi semelhante ao do *merge sort*, presente no gráfico da figura 4.

Entropia	Tamanho do vetor	Tempo médio (ms)
90%	100.000	10
90%	500.000	50
90%	1.000.000	106
50%	100.000	7
50%	500.000	41
50%	1.000.000	86
10%	100.000	5
10%	500.000	33
10%	1.000.000	66

Tabela 3: Resultados vetor ordenado por Quick Sort - JAVA

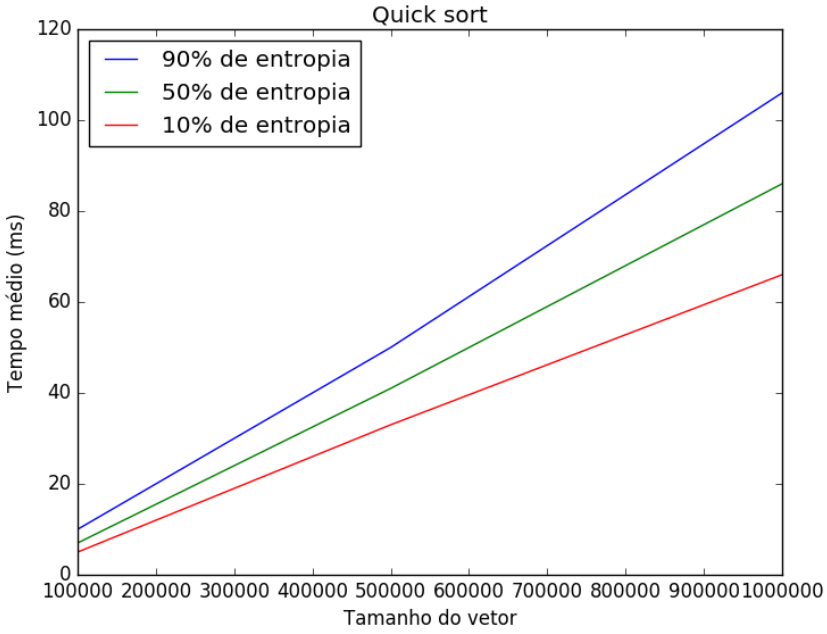

Figura 3: Gráfico comparativo do desempenho do Quick Sort com diferentes tipos de entrada

2.1.4 Merge sort

Método de ordenação por divisão e conquista, ou seja, o *merge sort* sempre divide o problema de forma balanceada (gerando subproblemas de mesmo tamanho) de modo recursivo. Podemos associar a sua execução com uma árvore binária. Por conta disso sua complexidade é $O(nlog(n))$, visto que a altura h da árvore de execução é $O(logn)$ e a intercalação de dois vetores

ordenados pode ser feita em tempo linear, com isso temos assintoticamente $O(n)$, ficando assim $O(n*log(n))$. A grande desvantagem deste método é que ele requer o dobro de memória, ou seja, precisa de um vetor com as mesmas dimensões do vetor que está sendo classificado.

Entropia	Tamanho do vetor	Tempo médio (ms)
90%	100.000	11
90%	500.000	55
90%	1.000.000	116
50%	100.000	6
50%	500.000	37
50%	1.000.000	79
10%	100.000	3
10%	500.000	19
10%	1.000.000	42

Tabela 4: Resultados vetor ordenado por Merge Sort - JAVA

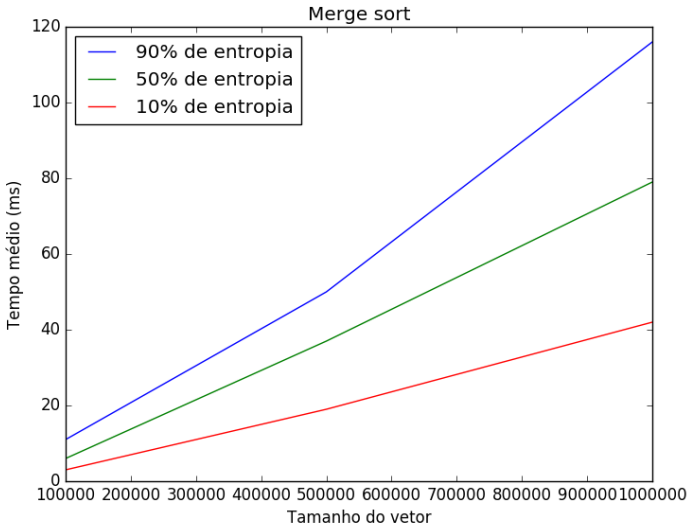

Figura 4: Gráfico comparativo do desempenho do Merge Sort com diferentes tipos de entrada

2.1.5 Heap sort

Heap sort é um método de ordenação cujo princípio de funcionamento é o mesmo utilizado para a ordenação por seleção. Pra maior eficiência, o algoritmo *heap sort* utiliza uma estrutura de dados chamada *heap*. *Heap* é um vetor que pode ser visto como uma árvore binária incompleta.

O *heap sort* serve para aplicações que não podem tolerar eventualmente um caso desfavorável. Seu comportamento é sempre $O(nlog(n))$. Uma desvantagem é que ele é um algoritmo não estável, além disso, seu anel interno é bastante complexo se comparado com o do *quick sort*. O *heap sort* obteve a maior taxa de crescimento de tempo dentre todos os algoritmos analisados neste trabalho, à medida que a quantidade de elementos do vetor aumentava.

A principal característica do *heap sort* se mostrou presente no gráfico da figura 5, já que seu tempo de execução possui comportamento semelhante em diferentes níveis de entropia.

Entropia	Tamanho do vetor	Tempo médio (ms)
90%	100.000	13
90%	500.000	76
90%	1.000.000	167
50%	100.000	11
50%	500.000	64
50%	1.000.000	139
10%	100.000	9
10%	500.000	50
10%	1.000.000	106

Tabela 5: Resultados vetor ordenado por Heap Sort - JAVA

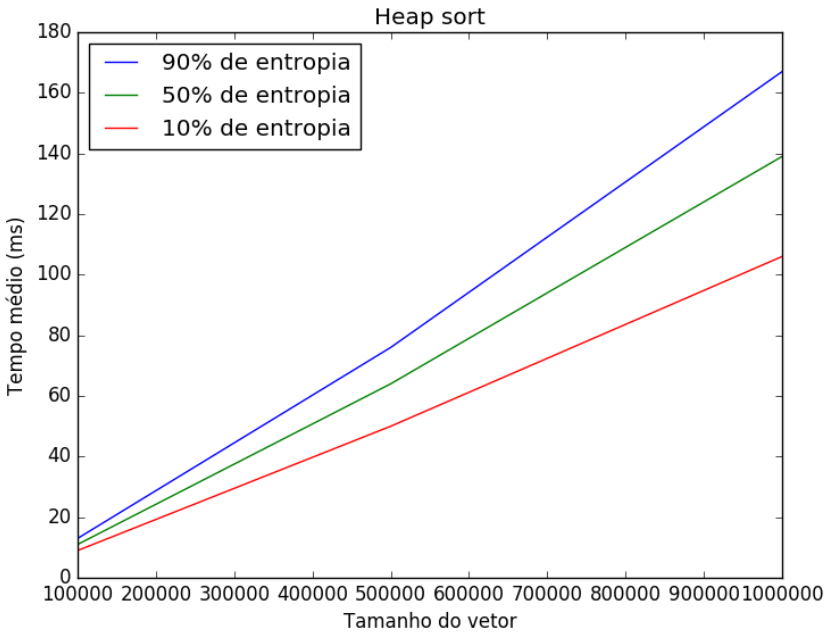

Figura 5: Gráfico comparativo do desempenho do Heap Sort com diferentes tipos de entrada

2.1.6 Java sort

A linguagem Java possui um método de ordenação padrão presente no pacote *java.util* que pode ser facilmente utilizado chamando o método *Arrays.sort()* e ordenando um *array* de forma crescente. Esta função, segundo a documentação oficial da linguagem[2], possui complexidade $O(n * log(n))$. Após ser submetido a mesma rotina de testes, o método em questão superou, em questão de eficiência, todos os outros algoritmos analisados neste relatório e seus resultados estão mostrados na figura 7. Isso se deve ao fato da utilização e aprimoramento de métodos já conhecidos de acordo com algumas características dos dados de entrada.

Este método têm como base o *quick sort*, porém com ele utiliza um duplo pivô[3] para a ordenação dos elementos e oferece complexidade $O(n * log(n))$ em muitos conjuntos de dados e é tipicamente mais rápido que o *quick sort* convencional, com um único pivô. O gráfico da figura 7 ilustra bem esta situação. Para casos onde o tamanho do vetor é pequeno (< 17), o algoritmo *insertion sort* é utilizado.

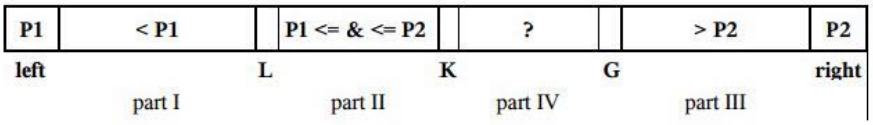

Figura 6: Exemplo de particionamento com duplo pivô

A figura 6 exemplifica como funciona o particionamento a partir de um duplo pivô. P1 deve ser menor que P2, caso contrário eles são trocados. O próximo elemento $[K]$ da parte IV é comparado com dois pivôs P1 e P2, e colocado na correspondente parte I, II ou III. Os ponteiros L, K e G são alterados nas direções correspondentes. As etapas 4 - 5 são repetidas enquanto $K <= G$. O elemento pivô P1 é trocado com o último elemento da parte I, o elemento pivô P2 é trocado com o primeiro elemento da parte III. As etapas 1 - 7 são repetidas recursivamente para cada parte I, parte II e parte III.

Entropia	Tamanho do vetor	Tempo médio (ms)
90%	100.000	10
90%	500.000	37
90%	1.000.000	82
50%	100.000	5
50%	500.000	33
50%	1.000.000	65
10%	100.000	2
10%	500.000	17
10%	1.000.000	34

Tabela 6: Resultados vetor ordenado por Java Sort

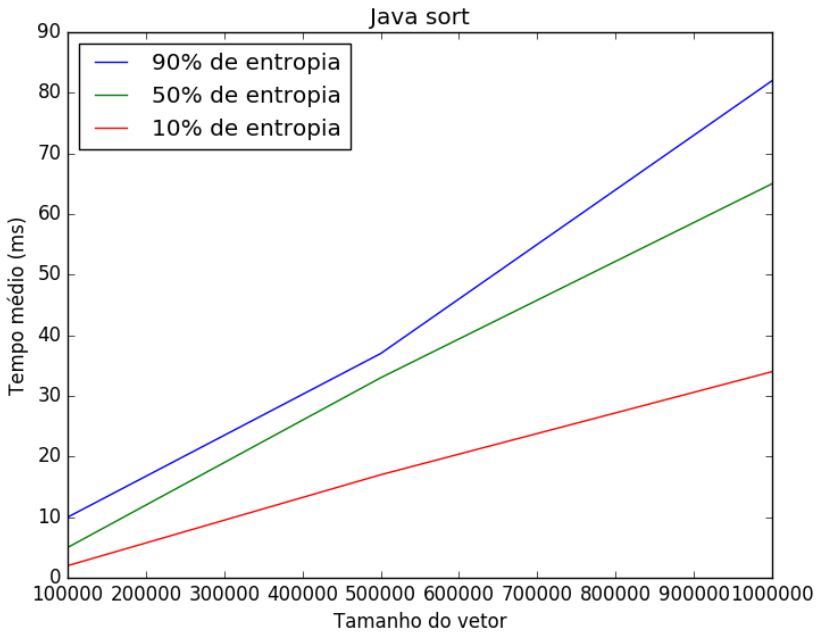

Figura 7: Gráfico comparativo do desempenho do Java Sort com diferentes tipos de entrada

2.1.7 Comparação entre os algoritmos

Comparando apenas os algoritmos de divisão e conquista, devido ao alto tempo levado pelo *insertion* e *selection*, podemos notar que o algoritmo padrão da linguagem JAVA ganha em todos os casos dos algoritmos tradicionais *quick sort*, *merge sort* e *heap sort*. Em alguns casos, o *quick sort* possui desempenho próximo ao do padrão Java, o que já era esperado, visto que possuem implementações parecidas.

Além disso, é importante comentar que em alguns casos o *merge sort* supera o *quick sort*. Isso se deve ao fato que nesses casos a escolha do pivô não favoreceu o desempenho do *quick sort*, já que em todos os casos o pivô é o primeiro elemento da partição.

A seguir serão apresentados alguns gráficos com características distintas para melhor ilustrar todos esses casos.

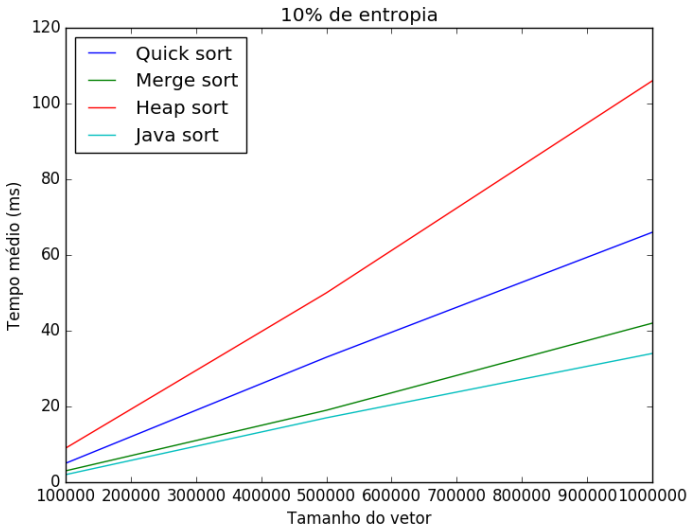

Figura 8: Gráfico comparativo do tempo de ordenação dos algoritmos Heap Sort, Merge Sort, Quick Sort e Java sort com vetores com 10% de entropia e com tamanhos de 100.000, 500.000 e 1.000.000.

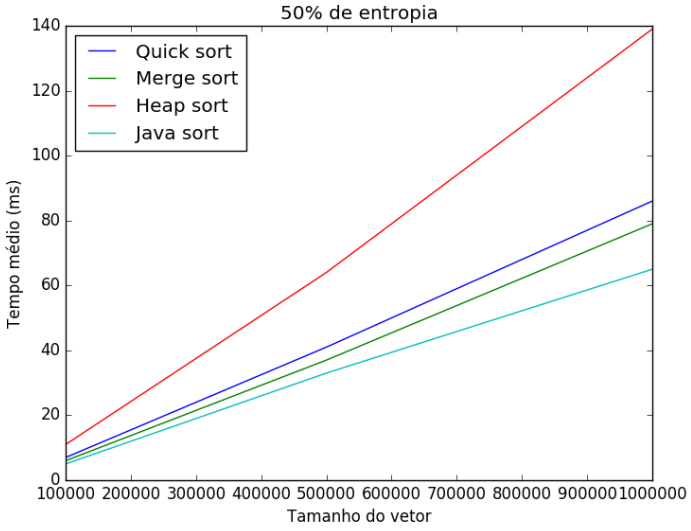

Figura 9: Gráfico comparativo do tempo de ordenação dos algoritmos Heap Sort, Merge Sort, Quick Sort e Java sort com vetores com 50% de entropia e com tamanhos de 100.000, 500.000 e 1.000.000.

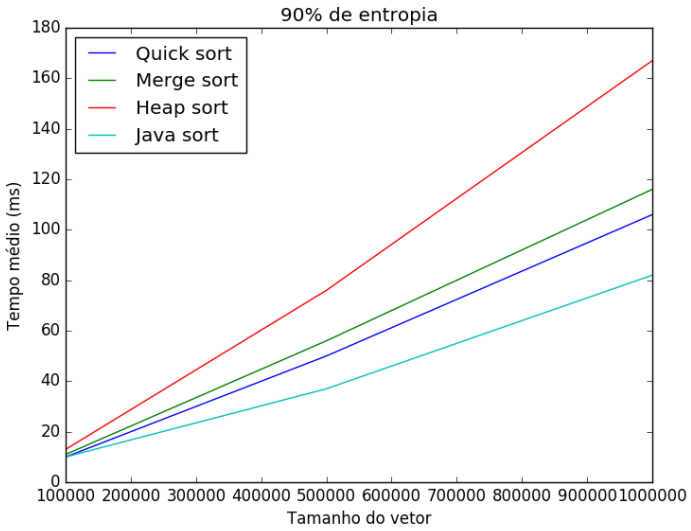

Figura 10: Gráfico comparativo do tempo de ordenação dos algoritmos Heap Sort, Merge Sort, Quick Sort e Java sort com vetores com 90% de entropia e com tamanhos de 100.000, 500.000 e 1.000.000.

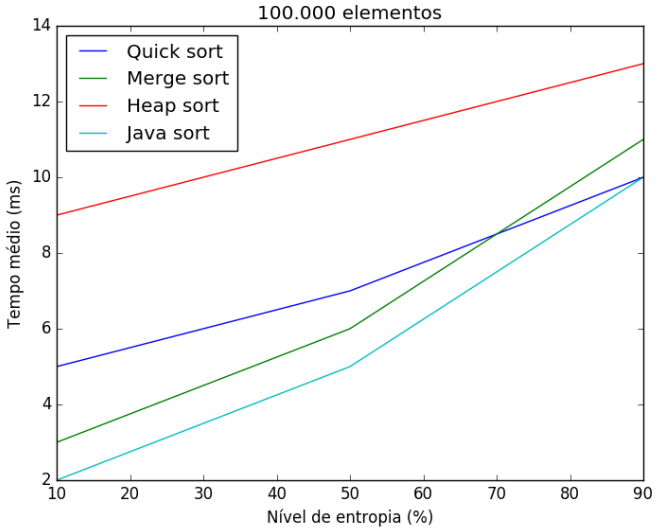

Figura 11: Gráfico comparativo do tempo de ordenação dos algoritmos Heap Sort, Merge Sort, Quick Sort e Java sort com vetores de tamanho 100.000 e com níveis de entropia de 10%, 50% e 90%

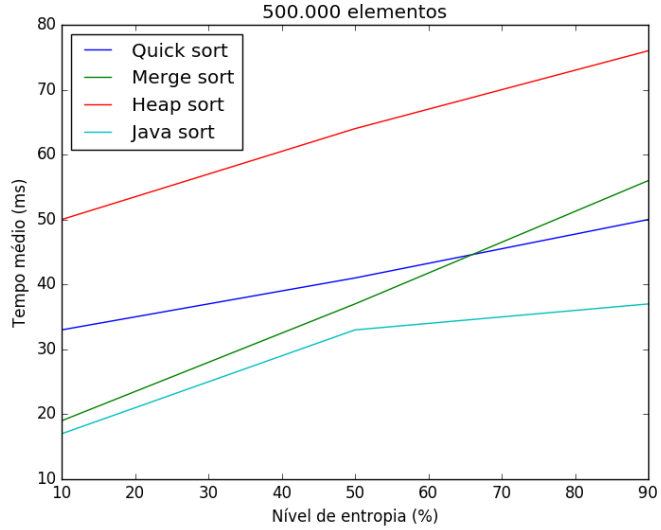

Figura 12: Gráfico comparativo do tempo de ordenação dos algoritmos Heap Sort, Merge Sort, Quick Sort e Java sort com vetores de tamanho 500.000 e com níveis de entropia de 10%, 50% e 90%

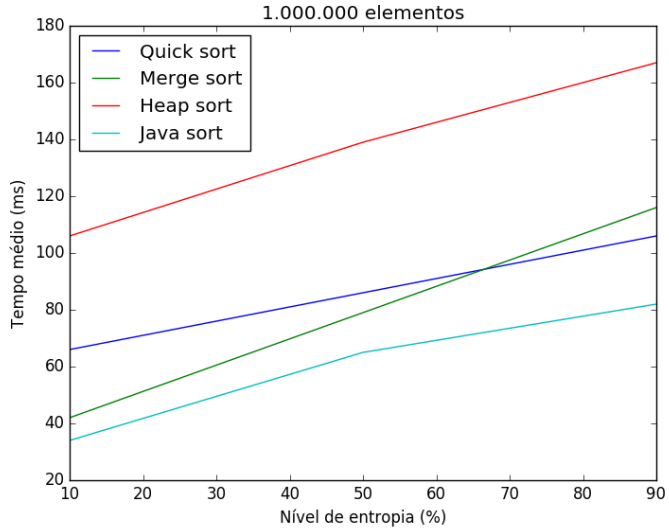

Figura 13: Gráfico comparativo do tempo de ordenação dos algoritmos Heap Sort, Merge Sort, Quick Sort e Java sort com vetores de tamanho 1.000.000 e com níveis de entropia de 10%, 50% e 90%

2.2 C++

2.2.1 Selection sort

O primeiro algoritmo a ser analisado é o selection sort, como já foi citado, ele tem complexidade quadrática, o que resulta em uma baixa eficiência no tempo de ordenação. Esse algoritmo foi o que obteve os piores resultados, e como é possível observar na tabela 7, caso seja fixado o tamanho da instância e varie o grau de entropia, os resultados continuam bem próximos, assim como se fixarmos o grau de entropia e variarmos o tamanho. Outra breve observação é que o algoritmo estourou o tempo limite dessa análise, que era de 5 minutos, para todas as instâncias com 1 milhão de elementos, independentemente do grau de entropia.

Entropia	Tamanho do vetor	Tempo médio (ms)
90%	100.000	5026
90%	500.000	124853
90%	1.000.000	*overtime*
50%	100.000	5015
50%	500.000	124729
50%	1.000.000	*overtime*
10%	100.000	5008
10%	500.000	124828
10%	1.000.000	*overtime*

Tabela 7: Resultados vetor ordenado por Selection Sort - C++

Observação: Na figura 14 as linhas que representam as entropias estão sobrepostas, dando a impressão de não existir as demais linhas.

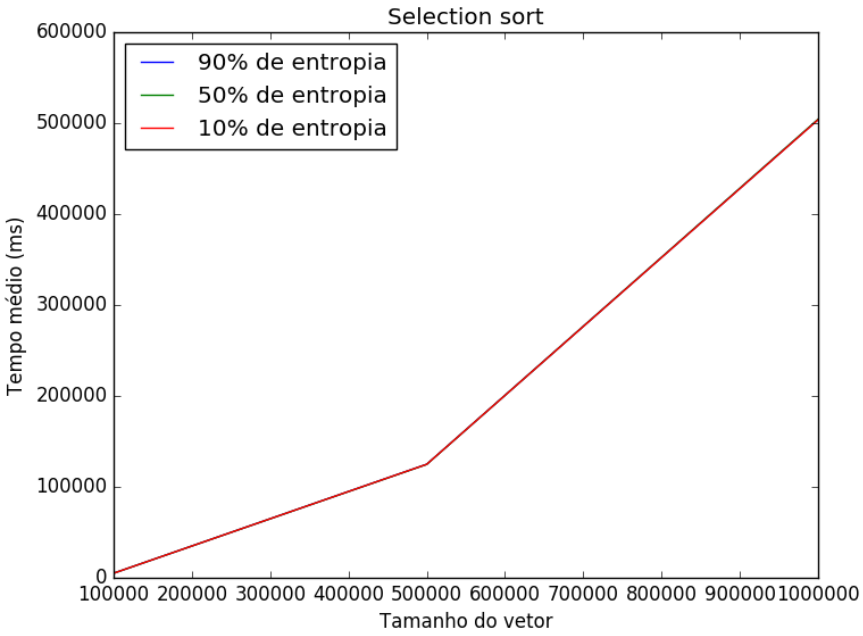

Figura 14: Gráfico comparativo do desempenho do Selection Sort com diferentes tipos de entrada.

2.2.2 Insertion sort

O ultimo dos algoritmos de complexidade média quadrática, o insertion sort teve resultados não satisfatórios, mas esperados, pois devido a sua complexidade, ele é mais eficiente apenas do que o selection sort. Um fato a se observar nessa análise, é uma importante característica desse algoritmo, a qual faz seu tempo cair bruscamente a medida que os vetores estão mais ordenados (com uma menor entropia).

Entropia	Tamanho do vetor	Tempo médio (ms)
90%	100.000	1697
90%	500.000	43134
90%	1.000.000	173566
50%	100.000	1273
50%	500.000	33112
50%	1.000.000	134149
10%	100.000	321
10%	500.000	8308
10%	1.000.000	32868

Tabela 8: Resultados vetor ordenado por Insertion Sort - C++

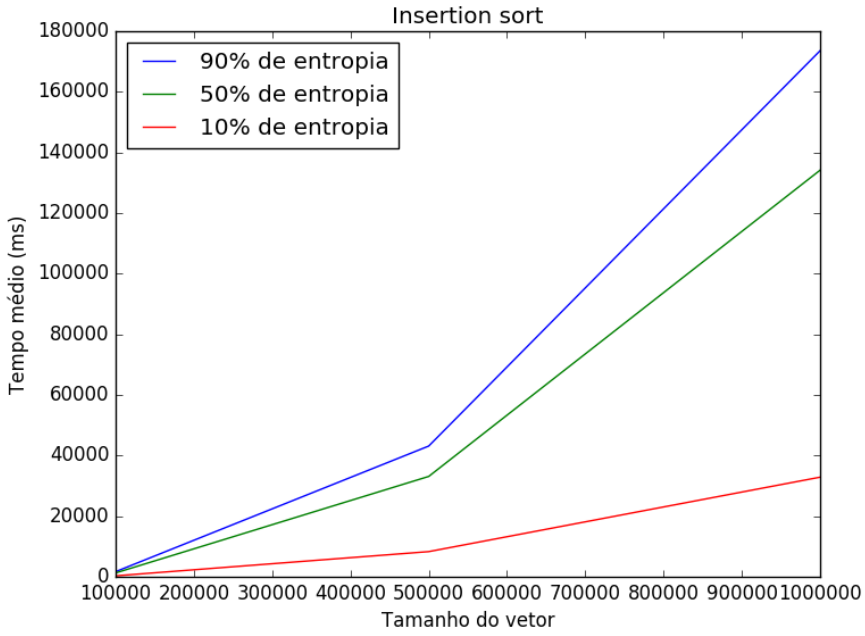

Figura 15: Gráfico comparativo do desempenho do Insertion Sort com diferentes tipos de entrada

2.2.3 Quick sort

Dando início aos algoritmos que utilizam a abordagem de "divisão e conquista", temos o quick sort, que confirmou a expectativa, sendo o mais eficiente, e obtendo um desempenho notavelmente superior, assim como os outros

métodos que utilizam a mesma abordagem. A eficiência dos métodos que utilizam essa abordagem varia bastante de acordo com o grau de entropia, e no caso especial desse algoritmo um fator determinante é a escolha do pivô. Diferente da implementação em Java, onde o pivô foi o primeiro elemento, em C++ o pivô foi o ultimo elemento. Caso tivéssemos implementado a escolha do pivô semelhante a de Java, os resultados seriam totalmente diferentes.

Entropia	Tamanho do vetor	Tempo médio (ms)
90%	100.000	9
90%	500.000	42
90%	1.000.000	88
50%	100.000	6
50%	500.000	37
50%	1.000.000	78
10%	100.000	7
10%	500.000	38
10%	1.000.000	78

Tabela 9: Resultados vetor ordenado por Quick Sort - C++

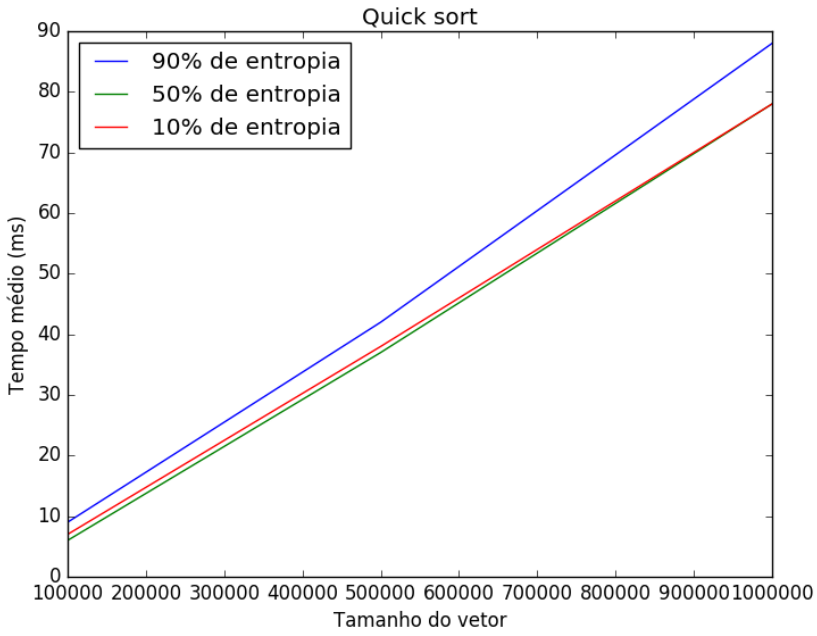

Figura 16: Gráfico comparativo do desempenho do Quick Sort com diferentes tipos de entrada

2.2.4 Merge sort

*M*ais um algoritmo que utiliza a abordagem de "divisão e conquista", o merge sort obteve um desempenho bem próximo ao heap sort. Inicialmente na sua implementação, foram utilizados vetores auxiliares nas divisões em subproblemas, mas quando o algoritmo foi submetido a entradas de tamanhos maiores, como por exemplo as instancias de 1 milhão de elementos, o espaço em memoria na pilha de execução não foi suficiente para esse grande numero de alocações, e o processo era encerrado pelo sistema operacional. Como solução para tal problema, a implementação foi alterada e os vetores passaram a ser alocados dinamicamente, no espaço de heap.

Entropia	Tamanho do vetor	Tempo médio (ms)
90%	100.000	16
90%	500.000	79
90%	1.000.000	165
50%	100.000	11
50%	500.000	62
50%	1.000.000	128
10%	100.000	8
10%	500.000	45
10%	1.000.000	94

Tabela 10: Resultados vetor ordenado por Merge Sort - C++

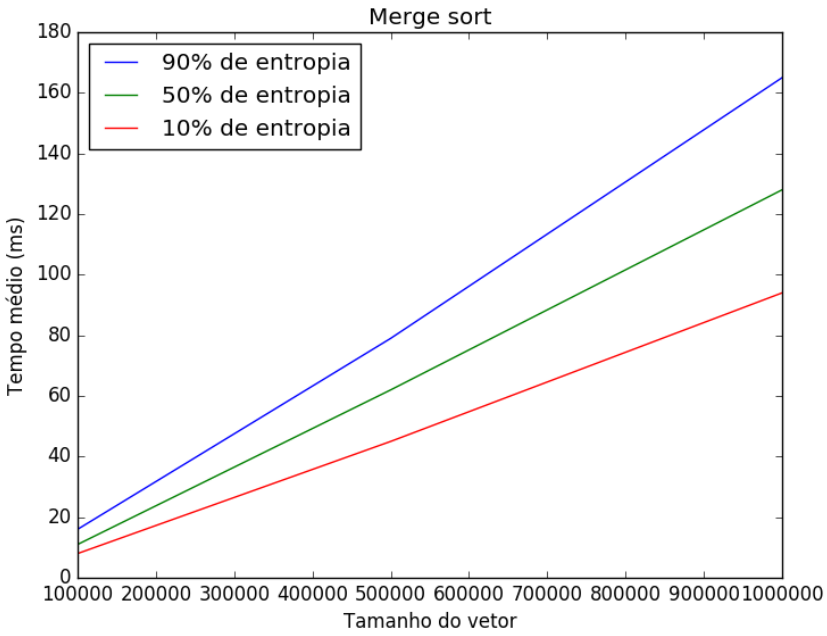

Figura 17: Gráfico comparativo do desempenho do Merge Sort com diferentes tipos de entrada

2.2.5 Heap sort

O heap sort obteve um resultado bastante satisfatório, ficando atras apenas do quick sort. A grande responsável por essa alta eficiência é uma estrutura denominada de heap, que auxilia a ordenação do vetor. Para observarmos a eficiência dessa estrutura, basta apenas compararmos os tempos do heap sort com do selection sort, pois caso a estrutura de heap seja retirada, os dois algoritmos se tornam equivalentes.

Entropia	Tamanho do vetor	Tempo médio (ms)
90%	100.000	11
90%	500.000	64
90%	1.000.000	139
50%	100.000	9
50%	500.000	55
50%	1.000.000	118
10%	100.000	7
10%	500.000	42
10%	1.000.000	88

Tabela 11: Resultados vetor ordenado por Heap Sort - C++

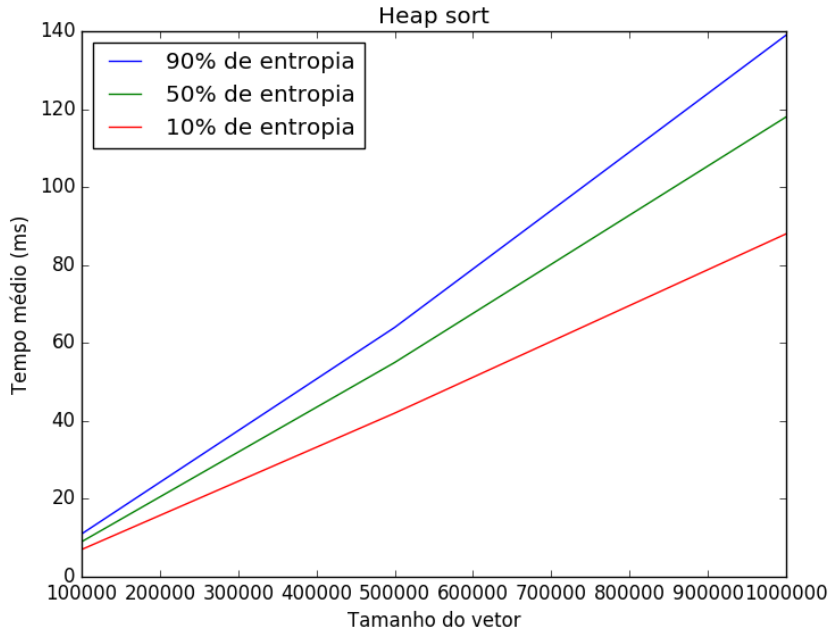

Figura 18: Gráfico comparativo do desempenho do Heap Sort com diferentes tipos de entrada

2.2.6 C++ sort

Agora faremos uma breve análise sobre o método de ordenação padrão da Standard Library (STL), chamado sort. Segundo a documentação da linguagem C++, esse método funciona sobre qualquer objeto que possa ser iterado, e tem complexidade $O(n * log(n))$. Esse método foi submetido aos mesmos

testes que os cinco métodos anteriores, e obteve o melhor desempenho entre todos. Se comparamos ele com o quick sort que foi o melhor dentre os cinco que foram implementados, vemos que ainda assim o método sort é bem mais eficiente, principalmente quando as instâncias aumentam de tamanho.

O grande motivo da alta eficiência desse algoritmo, é que ele utiliza uma combinação de 3 algoritmos, quick sort, heap sort, insertion sort. O algoritmo tomado como principal é o quick sort, mas como ele tem complexidade de pior caso de ordem quadrática, o método mantem salvo a profundidade de recursão de $O(2log(n))$, para que o heap sort seja utilizado. Esse mecanismo de combinação se chama Intro Sort.

Outras informações valem ser ressaltadas, a primeira é que a escolha do pivô é feita por intermédio da mediana de 3 pivôs aleatórios, o que contribui muito para o aumento da eficiência. A segunda informação é que em um certo estágio da arvore de recursão, quando os vetores estão menores e mais ordenados, é utilizado o algoritmo insertion sort, que é muito eficientemente nesse caso, como vimos anteriormente.

Entropia	Tamanho do vetor	Tempo médio (ms)
90%	100.000	6
90%	500.000	35
90%	1.000.000	75
50%	100.000	5
50%	500.000	29
50%	1.000.000	63
10%	100.000	3
10%	500.000	17
10%	1.000.000	37

Tabela 12: Resultados vetor ordenado por C++ - sort

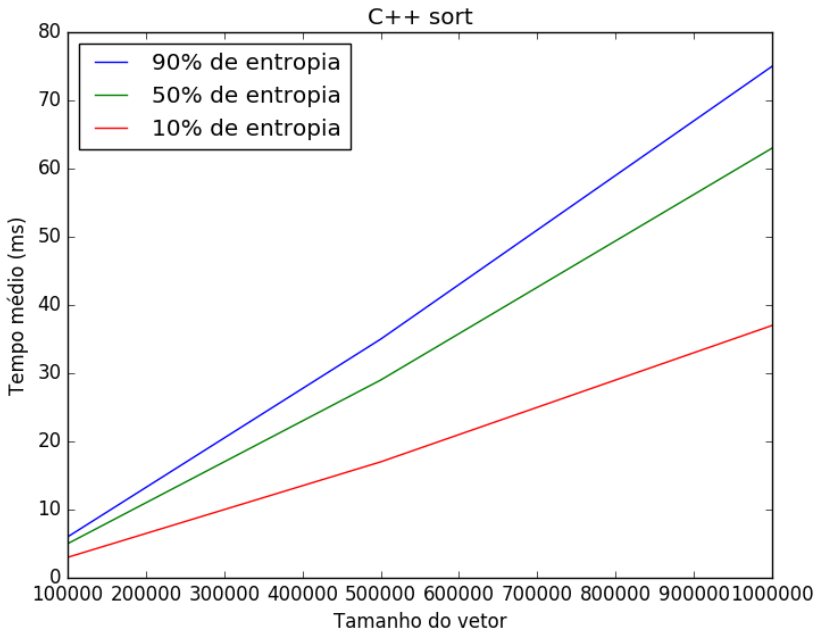

Figura 19: Gráfico comparativo do desempenho do C++ Sort com diferentes tipos de entrada

2.2.7 Comparação entre os algoritmos

Nessa sessão faremos uma breve análise sobre os 5 algoritmos que foram implementados. É fácil identificar, observando a figura abaixo, que os algoritmos que utilizam a abordagem de "divisão e conquista"são bem mais eficientes, sendo até difícil de diferencia-los em uma representação gráfica caso sejam expostos os algoritmos de ordem quadrática. Isso tudo acontece pelo fato da complexidade dos algoritmos que utilizam essa abordagem, serem da ordem $O(nlog(n))$.

Outra observação é que em uma representação gráfica fica mais evidente o quanto o selection sort é ineficiente em relação aos demais.

Diante de tudo que foi apresentado nessa sessão sobre os algoritmos implementados em C++, vimos que caso precise-se optar por um algoritmo de ordenação o melhor é optar pelo método padrão da linguagem, pois esse método passou por um número enorme de testes e situações e foi otimizado de tal modo que obtenha os melhores resultados, mas também foi possível observar que podemos implementar algoritmos não tão complexos como os padrões das linguagens e mesmo assim termos resultados satisfatórios.

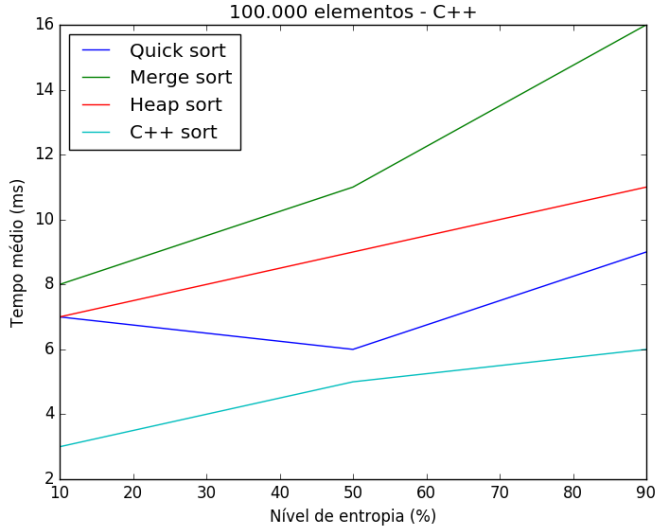

Figura 20: Gráfico comparativo do desempenho dos diferentes algoritmos, com entrada de tamanho 100000 elementos.

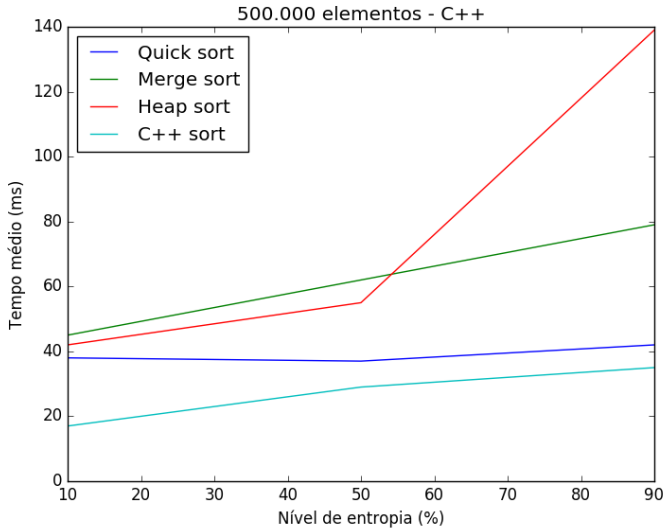

Figura 21: Gráfico comparativo do desempenho dos diferentes algoritmos, com entrada de tamanho 500000 elementos.

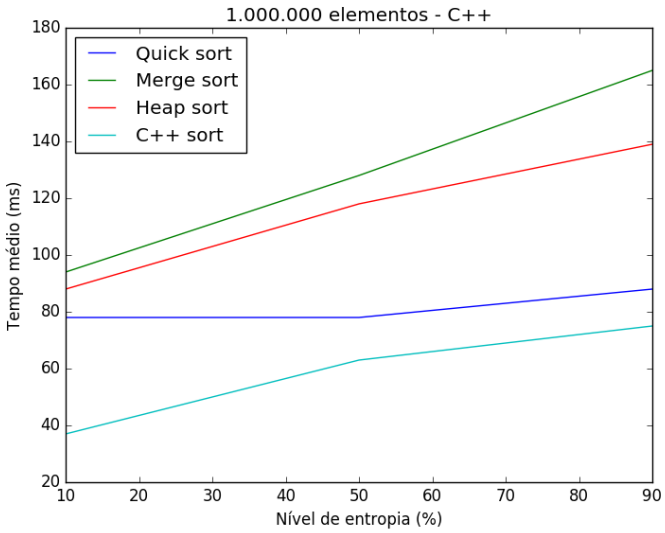

Figura 22: Gráfico comparativo do desempenho dos diferentes algoritmos, com entrada de tamanho 1000000 elementos.

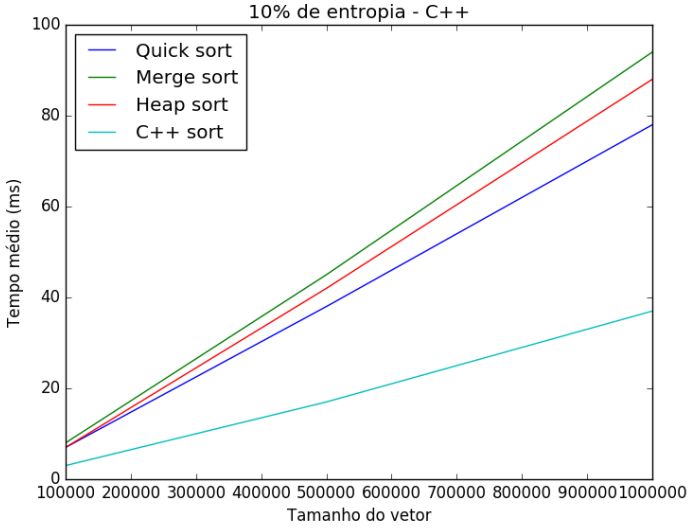

Figura 23: Gráfico comparativo do desempenho dos diferentes algoritmos, com 10% de entropia.

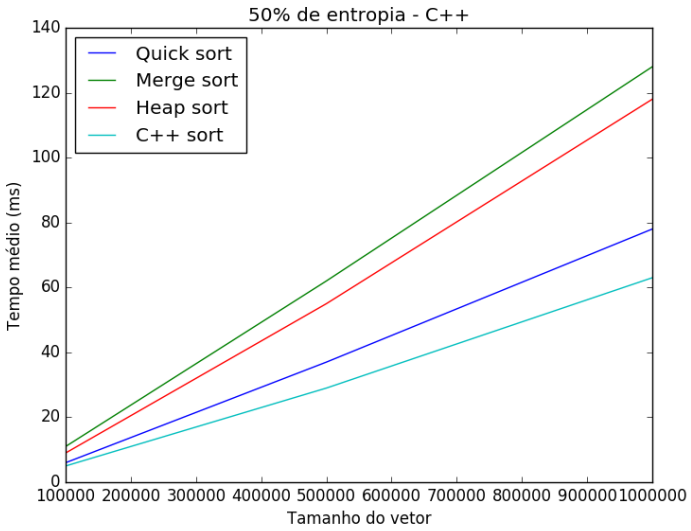

Figura 24: Gráfico comparativo do desempenho dos diferentes algoritmos, com 50% de entropia.

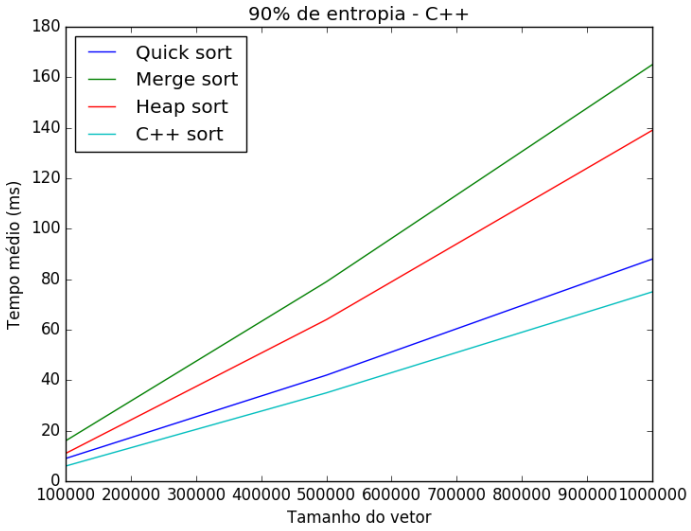

Figura 25: Gráfico comparativo do desempenho dos diferentes algoritmos, com 90% de entropia.

3 Java vs C++

As linguagens presentes neste trabalho possuem características distintas que acabam por influenciar na eficiência final de cada algoritmo.

O Java é uma linguagem de programação orientada a objetos de uso geral, simultânea, baseada em classe e estaticamente tipada que usa uma sintaxe semelhante a C ++, mas incompatível. Ele se baseia em uma máquina virtual Java (JVM) para ser seguro e altamente portátil, no qual o código é compilado para *byte-code*, e em seguida, interpretado em tempo de execução. O desempenho de um *byte-code* Java depende de como suas tarefas dadas são geridos pelo *host* da JVM, e como o ela explora os recursos do hardware do computador e sistema operacional ao fazê-lo. Além de executar um programa Java compilado, computadores que executam aplicativos Java, em geral, também deve executar a máquina virtual Java (JVM), enquanto programas em C ++ compilados podem ser executados sem uso de aplicativos externos, influenciando no desempenho dos programas C++ em detrimento aos do Java. Porém com os avanços e otimizações dos compiladores java e da JVM este problema foi se tornando cada vez menor.

A linguagem de programação C++ possui características de alto desempenho e permite que o programador trabalhe com um nível de abstração desejado. Ela também possui igualmente ao Java o conceito de orientação a objeto e ainda programação imperativa, programação genérica. Outro ponto forte dessa linguagem é que seus compiladores, sempre que possível, modificam o código para otimizar o desempenho, além do fato de possuir tipo tipagem estática. Após apresentações das características de cada linguagem vamos analisar graficamente o desempenho na prática de cada linguagem. Como forma de obter um dado visual mais claro e objetivo sobre a diferença entre as linguagens, foram selecionados alguns algoritmos para estudo dos resultados em um mesmo contexto. Foram considerados apenas os resultados dos algoritmos baseados em divisão e conquista sobre vetores de tamanho 1.000.000 com diferentes níveis de entropia.

Não houve superioridade total de uma linguagem em detrimento de outra, visto que os resultados foram equilibrados. No caso em particular do *quick sort*, na implementação em Java o pivô foi sempre o primeiro elemento da partição e na implementação em C++ o pivô foi o último elemento da partição. Por isso em algumas entradas o Java foi melhor do que o C++ e vice-versa. Para o algoritmo padrão de cada linguagem os resultados foram bastante semelhantes, com o algoritmo em Java mais rápido com menor entropia, porém ao decorrer dos níveis ele é ultrapassado pelo algoritmo em C++, porém com baixíssimas diferenças entre eles, mostrando a alta eficiência de ambos.

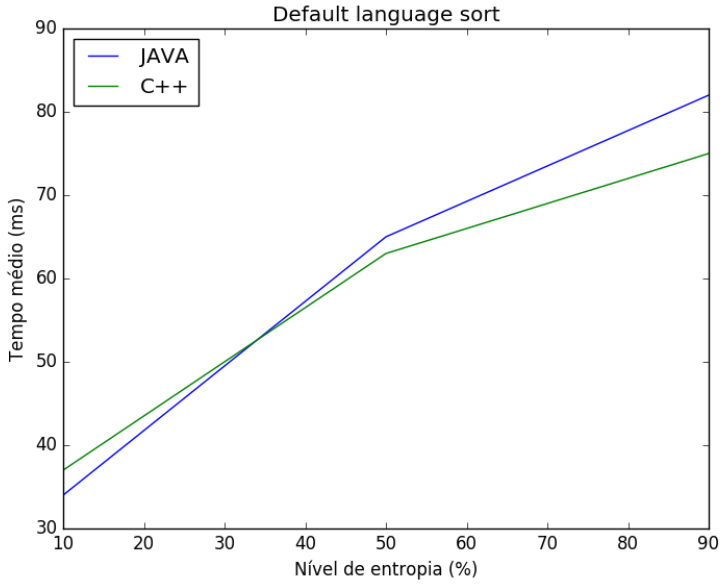

Figura 26: Gráfico comparativo do desempenho dos algoritmos padrões das linguagens com 1.000.000 elementos e diferentes tipos de entropia

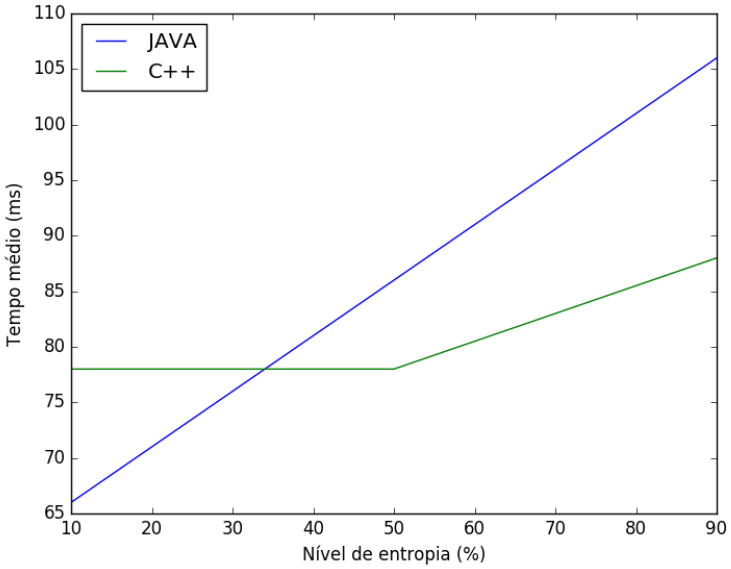

Figura 27: Gráfico comparativo do desempenho do algoritmo quick sort nas duas linguagens com 1.000.000 elementos e diferentes tipos de entropia

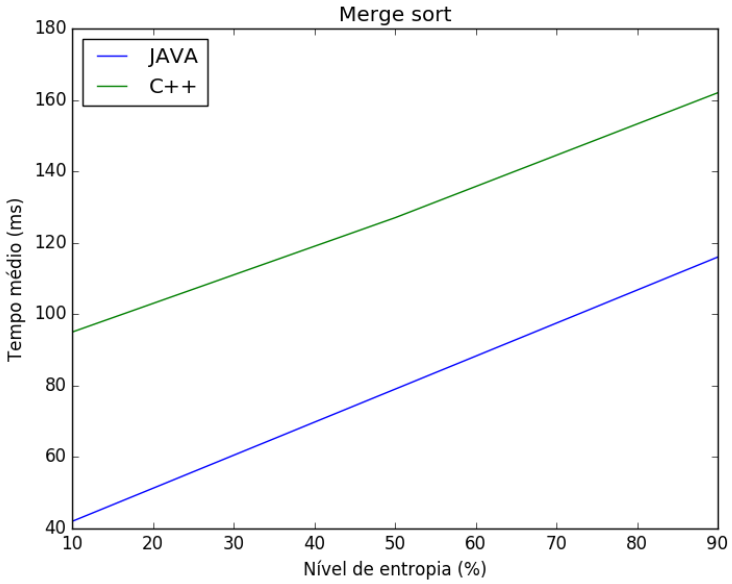

Figura 28: Gráfico comparativo do desempenho do algoritmo merge sort nas duas linguagens com 1.000.000 elementos e diferentes tipos de entropia

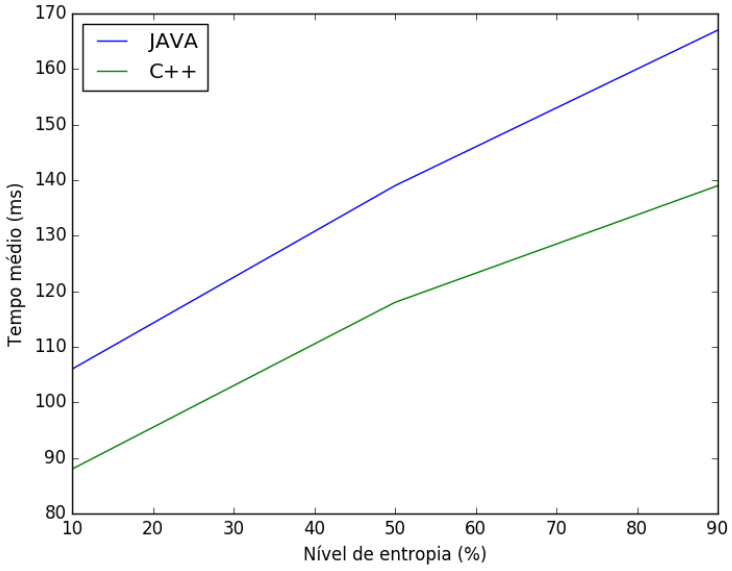

Figura 29: Gráfico comparativo do desempenho do algoritmo heap sort nas duas linguagens com 1.000.000 elementos e diferentes tipos de entropia

É importante ressaltar que dentre todos os aspectos mencionados, estes foram considerados por nós os mais influenciadores para os resultados, porém podem existir vários outros fatores pertinentes a linguagem que não foram citados e que podem ter sido potencialmente decisivos nos resultados.

Em suma, acreditamos que apesar de analisarmos assintoticamente o desempenho dos algoritmos de ordenação apresentados neste trabalho, existem muitas variáveis que influenciam de fato no desempenho dos algoritmos que só podem ser notados quando aplicados na prática e, por isso, há grandes diferenças no tempo de processamento de algoritmos de mesma complexidade. Com isso, o conhecimento teórico aliado ao prático são peças primordiais na hora de escolher qual método utilizarmos.

Referências

[1] Leonardo Cesar Teonacio Bezerra. Instâncias disponíveis em http://leobezerra.ci.ufpb.br/disciplinas/. Acessado em: 28/01/2017.

[2] Documentação oficial da linguagem JAVA. *Oracle Java doc.* Disponível em https://docs.oracle.com/javase/8/docs/api/. Acessado em: 28/01/2017.

[3] Vladimir Yaroslavskiy. Dual-pivot quicksort algorithm. disponível em http://codeblab.com/wp-content/uploads/2009/09/dualpivotquicksort.pdf. 2009. Acessado em: 28/01/2017.