Análise Comparativa dos Algoritmos de Ordenação

Bruna T. Silva[1], **Gabriel G. Conejo**[2]

[1]Departamento de Ciência da Computação
CCT – Universidade do Estado de Santa Catarina (UDESC)
Joinville - SC - Brasil

{silvatavares.bruna,gabrielgconejo}@gmail.br

Abstract. *The objective of this paper is to explain the and compare the complexities shown by a number of sorting algorithms. The algorithms studied here are all based on already implemented algorithms and our own implementations as well, they are as follows: Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Counting Sort and Bucket Sort.*

Resumo. *O objetivo deste artigo é explicar e comparar as complexidades obtidas por algoritmos de ordenação. Os algoritmos estudados aqui são todos baseados em implementações já existentes bem como nosssas próprias implementações, são eles: Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Counting Sort e Bucket Sort.*

1. Introdução

Algoritmos de ordenação são uma classe de algoritmos amplamente utilizada, tanto acadêmicamente como competitivamente e até mesmo no mercado de trabalho. Por este motivo existem diversos algoritmos de ordenação diferentes, para que desta forma o programador possa utilizar o algoritmo que mais se aplica ao problema a ser resolvido.

Dentre os inúmeros algoritmos existentes alguns se destacam, seja por sua fácil implementação, ou por seu desempenho ou até mesmo por fins didáticos. O objetivo deste trabalho é, a partir destes algoritmos de ordenação mais populares estudar e comparar como a complexidade de cada algoritmo se comporta em certas situações.

Para coletar os dados e ter uma amostragem precisa para a comparação os algoritmos foram rodados com diferentes tamanhos de vetores, 25000, 50000, 75000, 100000 e 1000000 respectivamente e com diferentes formas de distribuíção de valores nestes vetores, de forma crescente, decrescente, aleatória e aleatória com um número 100000000 em um espaço qualquer, nestas condições os algoritmos foram executados cinco vezes para cada configuração e a média dos resultados foi levada em conta e para a realização dos testes foram utilizados três computadores diferentes com processadores diferentes.

2. Bubble Sort

O algoritmo bubble sort é o mais simples de todos os algoritmos testados, sua complexidade de tempo é:

$$O(n^2) \tag{1}$$

Para todos os casos testados a complexidade é a mesma, logo os resultados obtidos pelo mesmo foram muito parecidos em todos os testes. Sua complexidade de espaço por outro lado é considerada muito boa, a mesma é:

$$O(1) \tag{2}$$

Utilizando uma abordagem em que o vetor já estava ordenado de forma crescente o algoritmo demonstrou levar um tempo menor para computar o resultado, mesmo sua complexidade sendo a mesma para todos os casos. O resultado para o vetor de tamanho 25000 foi 1,151459s, já para o tamanho 50000 foi 4,644879s, para 75000 foi 10,303064s, para 100000 foi 18,12249s e 1000000 foi 139m28.981s.

Já para um vetor ordenado decrescente os resultados foram, para 25000 foi 1,426641s, para 50000 foi 5,687056s, para 75000 foi 12,492529s, para 100000 foi 22,047783s e para 1000000 foi 142m15.549s.

Com a abordagem aleatória foram constatados os resultados, para 25000 foi 2,151778s, para 50000 foi 8,87043s, para 75000 foi 20,238321s, para 100000 foi 35,851265s, para 1000000 foi 151m23.406s.

Por último utilizando a abordagem aleatória com um elemento 100000000, para 25000 foi 2,219056s, para foi 50000 foi 8,863322s, para 75000 foi 19,992016s, para 100000 foi 35,658722s, para 1000000 foi 148m46.207s.

É notável que este algoritmo não apresenta um desempenho satisfatório, pois mesmo utilizando vetores relativamente pequenos o mesmo apresenta mais de um segundo de tempo para a computação do resultado.

3. Insertion Sort

O algoritmo insertion sort também é um algoritmo simples de ser implementado e dos algoritmos básicos é o mais rápido[3], e assim como o bubble a sua complexidade para os casos de teste é:

$$O(n^2) \tag{3}$$

Como dito anteriormente, esta complexidade deixa a desejar pois é considerada péssima para o desempenho do algoritmo. Porém, assim como o bubble sua complexidade de espaço é boa tendo a notação assintótica de:

$$O(1) \tag{4}$$

Os resultados obtidos com o vetor ordenado de forma crescente foi, para 25000 foi 0,001272s, para 50000 foi 0,000821s, para 75000 foi 0,001062s, para 100000 foi 0,001418s, para 1000000 foi 0,014228s.

Já para o vetor ordenado de forma decrescente, para 25000 foi 0,001298s, para 50000 foi 0,001037s, para 75000 foi 0,00107s, para 100000 foi 0,001422s, para 1000000 foi 0,024175s.

Com o vetor organizado de forma aleatória obtivemos para 25000 foi 1,07123s, para 50000 foi 4,335445s, para 75000 foi 9,807844s, para 100000 foi 17,436335s, para 1000000 foi 1774,068726s.

Por último, com o vetor organizado de forma aleatória e um valor 100000000 em um local qualquer obtivemos. Para 25000 foi 1,204947s, para 50000 foi 4,825184s, para 75000 foi 10,841021s, para 100000 foi 19,301056s, para 1000000 foi 1935,159424s.

Apesar da complexidade deste algoritmo ser a mesma do bubble, este teve um desempenho muito melhor, mas ainda assim percebe-se que o tempo apresentado ao final de cada teste ainda é alto, mostrando que este algoritmo não é o ideal a ser utilizado em certas circunstâncias, vetor de tamanho grande e aleatório por exemplo.

4. Quick Sort

O algoritmo quick sort é considerado um algoritmo de ordenação de fácil implementação e complexidade relativamente boa por seu desempenho[5], sua complexidade de tempo é para o caso médio:

$$O(nlogn) \tag{5}$$

Para a realização dos testes foram implementadas diversas variações deste algoritmo, o motivo para isto é que quando o pivô era o primeiro elemento do vetor uma falha de segmentação ocorria para o vetor de tamanho 1000000, o motivo para isto é o espaço que o quick utiliza somado a complexidade de tempo do caso que é específico, sua complexidade de espaço e tempo para o caso são respectivamente:

$$O(logn) \tag{6}$$
$$O(n^2) \tag{7}$$

Este algoritmo foi testado de duas formas, utilizando o primeiro elemento como pivô e um elemento qualquer como pivô.

4.1. Quick Primeiro Elemento Como Pivô

Utilizando o primeiro elemento como pivô é possível obter resultados não satisfatórios para o vetor ordenado de forma crescente, devido ao aumento de sua complexidade.

Os resultados obtidos para um vetor ordenado de forma crescente foram. Para 25000 foi 1,537377s, Para 50000 foi 6,051425s, Para 75000 foi 13,669278s, Para 100000 foi 25,1018728s, Para 1000000 obteve-se o erro de segmentação aos 13 minutos.

Já para o vetor ordenado de forma decrescente obteve-se, para 25000 foi 0,004295s, para 50000 foi 0,006946s, para 75000 foi 0,010689s, para 100000 foi 0,014662s, para 1000000 foi 0,317846s.

Para o vetor deserdenado de forma aleatória obteve-se, para 25000 foi 0,004465s, para 50000 foi 0,006375s, para 75000 foi 0,009831s, para 100000 foi 0,013464s, para 1000000 foi 0,302032s.

Por último para o vetor aleatório com um elemento 100000000 em um posição qualquer , para 25000 foi 0,004624s, para 50000 foi 0,006678s, para 75000 foi 0,010352s, para 100000 foi 0,01404s, para 1000000 foi 0,314575s.

Com o pivô sendo o primeiro elemento do vetor é notável que o desempenho não é o preferível, porém a seguir o mesmo algoritmo é utilizado com a diferença na escolha do pivô, que desta vez é aleatório.

4.2. Quick Elemento Qualquer Como Pivô

Após o teste com o primeiro elemento como pivô foi realizado o teste para qualquer elemento. Os resultados obtidos por cada um dos testes estão listados abaixo.

Os resultados obtidos para um vetor ordenado de forma crescente foram. Para 25000 foi 0,003492s, para 50000 foi 0,004821s, para 75000 foi 0,00734s, para 100000 foi 0,010165s, para 1000000 foi 0,117889s.

Já para o vetor ordenado de forma decrescente obteve-se, para 25000 foi 0,004449s, para 50000 foi 0,006749s, para 75000 foi 0,010191s, para 100000 foi 0,013888s, para 1000000 foi 0,167242s.

Para o vetor deserdenado de forma aleatória obteve-se, para 25000 foi 0,004298s, para 50000 foi 0,006619s, para 75000 foi 0,010185s, para 100000 foi 0,013945s, para 1000000 foi 0,166698s.

Por último para o vetor aleatório com um elemento 100000000 em um posição qualquer , para 25000 foi 0,004483s, para 50000 foi 0,006714s, para 75000 foi 0,010563s, para 100000 foi 0,014116s, para 1000000 foi 0,166303s.

O tempo em relação ao teste utilizando o primeiro elemento como pivô diminuiu em cinquenta por cento nos testes realizados, salvo o teste com o vetor ordenado de forma crescente que melhorou muito além dos cinquenta por cento. Por este motivo é aconselhado utilizar um pivô aleatório ou escolher o elemento central como pivô.

5. Merge Sort

O algoritmo merge sort não é um algoritmo fácil de ser implementado, o mesmo pode ser implementado utilizando vetores ou árvores. Para este trabalho o mesmo foi implementado utilizando um vetor.

A complexidade deste algoritmo, assim como a do quick sort, é:

$$O(nlogn) \tag{8}$$

A diferença entre eles no entanto é que o merge sort não possui um caso em que sua complexidade aumente, ou seja, sua complexidade de tempo é sempre a mesma. Já sua complexidade de espaço é:

$$O(n) \tag{9}$$

Uma complexidade de espaço um pouco mais alta que as visatas anteriormente, porém é uma complexidade considerada boa.

Os resultados dos testes realizados com este algoritmo estão listados abaixo.

Os resultados obtidos para um vetor ordenado de forma crescente foram. Para 25000 foi 0,00707s, para 50000 foi 0,012205s, para 75000 foi 0,018803s, para 100000 foi 0,02551s, para 1000000 foi 0,286757s.

Já para o vetor ordenado de forma decrescente obteve-se, para 25000 foi 0,007101s, para 50000 foi 0,011942s, para 75000 foi 0,018483s, para 100000 foi 0,024798s, para 1000000 foi 0,285766s.

Para o vetor deserdenado de forma aleatória obteve-se, para 25000 foi 0,008032s, para 50000 foi 0,015304s, para 75000 foi 0,023892s, para 100000 foi 0,032178s, para 1000000 foi 0,375656s.

Por último para o vetor aleatório com um elemento 100000000 em um posição qualquer , para 25000 foi 0,008018s, para 50000 foi 0,015171s, para 75000 foi 0,023586s, para 100000 foi 0,031846s, para 1000000 foi 0,371876s.

6. Heap Sort

O segredo do heap sort é uma estrutura de dados chamada heap, esta faz com que o vetor seja visto como uma árvore binária[4]. O algoritmo heap sort foi implementado em uma única função e assim como o merge sort têm a complexidade de tempo:

$$O(nlogn) \tag{10}$$

Sua complexidade será a mesma para qualquer um dos casos de teste realizados, é um algoritmo estável por este motivo e também por utilizar uma árvora para realizar suas comparações. Sua complexidade de espaço é:

$$O(1) \tag{11}$$

O que o torna um bom contraponto para o merge sort, justamento por sua complexidade de espaço menor.

Os testes realizados com o heap sort foram feios apenas com um algoritmo implementado, seus resultados são apresentados abaixo.

Os resultados obtidos para um vetor ordenado de forma crescente foram. Para 25000 foi 0,007454s, para 50000 foi 0,014134s, para 75000 foi 0,022055s, para 100000 foi 0,030187s, para 1000000 foi 0,36281s.

Já para o vetor ordenado de forma decrescente obteve-se, para 25000 foi 0,002115s, para 50000 foi 0,001196s, para 75000 foi 0,001784s, para 100000 foi 0,002373s, para 1000000 foi 0,024546s.

Para o vetor deserdenado de forma aleatória obteve-se, para 25000 foi 0,007237s, para 50000 foi 0,014159s, para 75000 foi 0,022176s, para 100000 foi 0,03049s, para 1000000 foi 0,405284s.

Por último para o vetor aleatório com um elemento 100000000 em um posição qualquer , para 25000 foi 0,007568s, para 50000 foi 0,01443s, para 75000 foi 0,02259s, para 100000 foi 0,031023s, para 1000000 foi 0,410692s.

É possível observar que a diferença de desempenho entre o heap, merge e quick (qualquer elemento como pivô) não é tão grande, isto se deve por sua complexidade de tempo ser a mesma para os casos testados, logo, a escolha entre eles pode fácilmente ser arbitrária caso necessário.

7. Counting Sort

O algoritmo counting sort foi implementado de duas maneiras para a realização dos testes, o primeiro algoritmo utilizado ocorria erro de segmentação para alguns casos de teste, por este motivo foi criado um novo.

Como parâmetros para a função foram usados um ponteiro (nos demais algoritmos um vetor era utilizado) e também o valor mínimo e máximo que estavam contidos no vetor.

O algoritmo counting sort têm complexidade de espaço proporcional ao tamanho do vetor e deixa de ser eficiente quando um valor é muito maior do que o tamanho do vetor[2, 5].

O algoritmo counting sort têm complexidade de tempo:

$$O(n + k) \tag{12}$$

$$k = nroEspaçosVetor \tag{13}$$

Esta complexidade não muda para nenhum caso de teste, o que torna este um algoritimo de baixa complexidade e, consequentemente, de bom desempenho. Sua complexidade de espaço é:

$$O(k) \tag{14}$$

$$k = nroEspaçosVetor \tag{15}$$

Sua complexidade de espaço é considerada relativamente boa, os testes realizados constam a seguir.

Os resultados obtidos para um vetor ordenado de forma crescente foram. Para 25000 foi 0,000841s, para 50000 foi 0,000474s, para 75000 foi 0,000706s, para 100000 foi 0,000936s, para 1000000 foi 0,009178s.

Já para o vetor ordenado de forma decrescente obteve-se, para 25000 foi 0,000762s, para 50000 foi 0,001495s, para 75000 foi 0,000637s, para 100000 foi 0,000832s, para 1000000 foi 0,008364s.

Para o vetor deserdenado de forma aleatória obteve-se, para 25000 foi 0,000426s, para 50000 foi 0,000443s, para 75000 foi 0,000658s, para 100000 foi 0,000878s, para 1000000 foi 0,008809s.

Por último para o vetor aleatório com um elemento 100000000 em um posição qualquer , para 25000 foi 0,000457s, para 50000 foi 0,000442s, para 75000 foi 0,00066s, para 100000 foi 0,000874s, para 1000000 foi 0,008817s.

Ao final dos testes fica claro o alto desempenho deste algoritmo, o mesmo leva pouco mais de um centésimo de segundo em todos os casos de teste, mostrando que consegue ter um desempenho superior aos algoritmos explicitados anteriormente.

8. Bucket Sort

O algoritmo bucket sort implementado cria buckets do tamanho de cada espaço do vetor de números, e o vetor de buckets têm o mesmo tamanho do vetor de números, por este motivo ele pode ser extremamente rápido[1].

O algoritmo têm uma complexidade de tempo igual para seu caso médio e melhor caso, porém maior para seu pior caso. A complexidade do caso médio e do pior caso são respectivamente:

$$O(n + k) \tag{16}$$

$$k = nroEspaçosVetor \tag{17}$$

$$O(n^2) \tag{18}$$

E sua complexidade de espaço é:

$$O(n) \tag{19}$$

Os testes realizados foram realizados com apenas um algoritmo e os resultados são apresentados abaixo.

Os resultados obtidos para um vetor ordenado de forma crescente foram. Para 25000 foi 0,000858s, para 50000 foi 0,001533s, para 75000 foi 0,001892s, para 100000 foi 0,001506s, para 1000000 foi 0,011173s.

Já para o vetor ordenado de forma decrescente obteve-se, para 25000 foi 0,000294s, para 50000 foi 0,000597s, para 75000 foi 0,000886s, para 100000 foi 0,001237s, para 1000000 foi 0,011447s.

Para o vetor deserdenado de forma aleatória obteve-se, para 25000 foi 0,000758s, para 50000 foi 0,001117s, para 75000 foi 0,001853s, para 100000 foi 0,002346s, para 1000000 foi 0,0033501s.

Por último para o vetor aleatório com um elemento 100000000 em um posição qualquer , para 25000 foi 0,0006903s, para 50000 foi 0,0012097s, para 75000 foi 0,0020283s, para 100000 foi 0,0024927s, para 1000000 foi 0,0036271s.

Percebe-se que, assim como o counting sort, o bucket sort têm um desempenho muito bom comparado com os algoritmos anteriores, porém o counting sort ainda é melhor no quesito tempo.

9. Gráficos

Esta sessão apresenta os gráficos dos resultados obtidos, cada gráfico representa um tipo de ordenação do vetor.

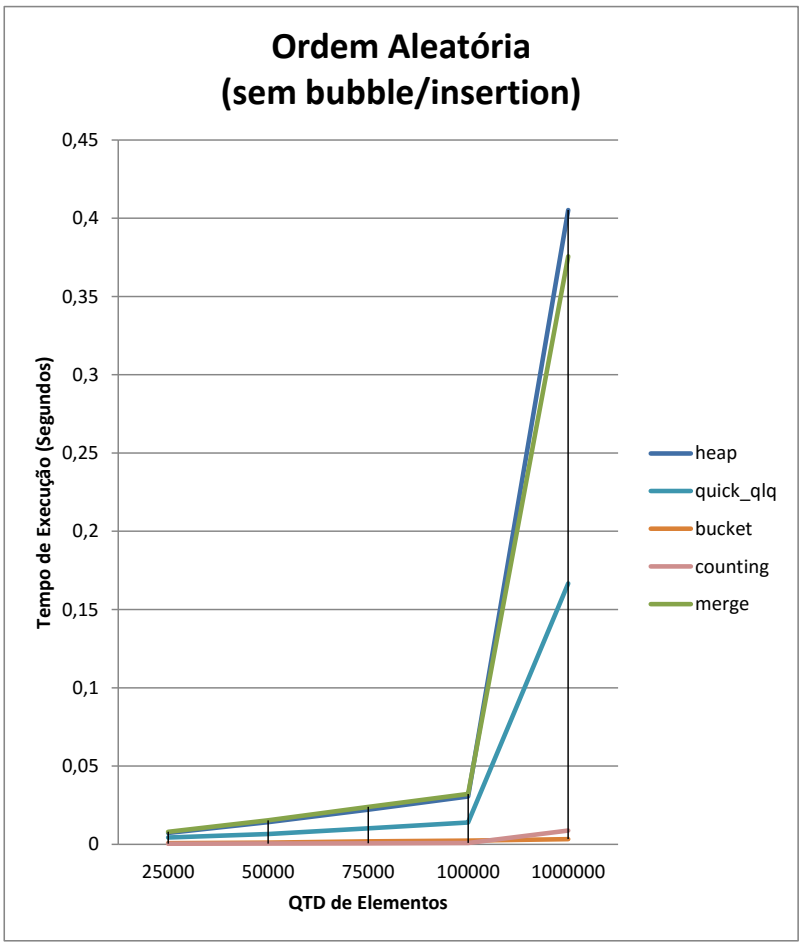

Figura 1. Ordem aleatória sem Bubble/Insertion

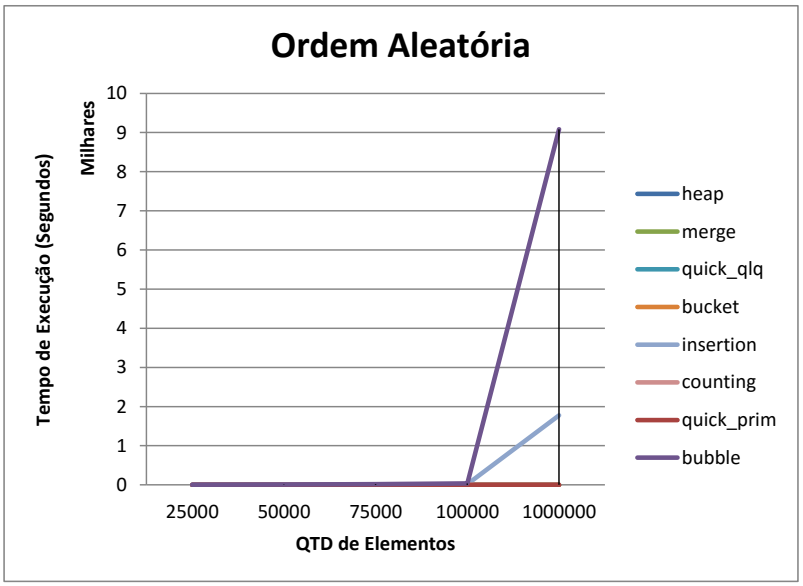

Figura 2. Ordem aleatória

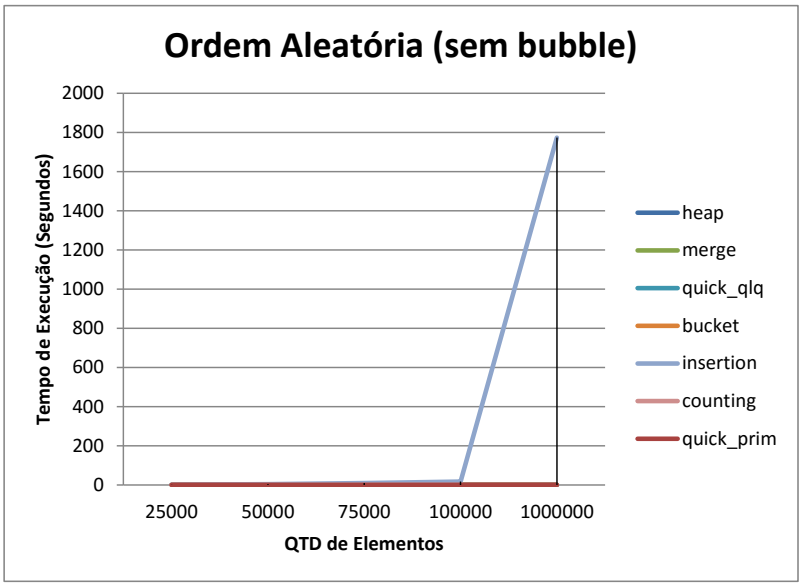

Figura 3. Ordem aleatória sem bubble

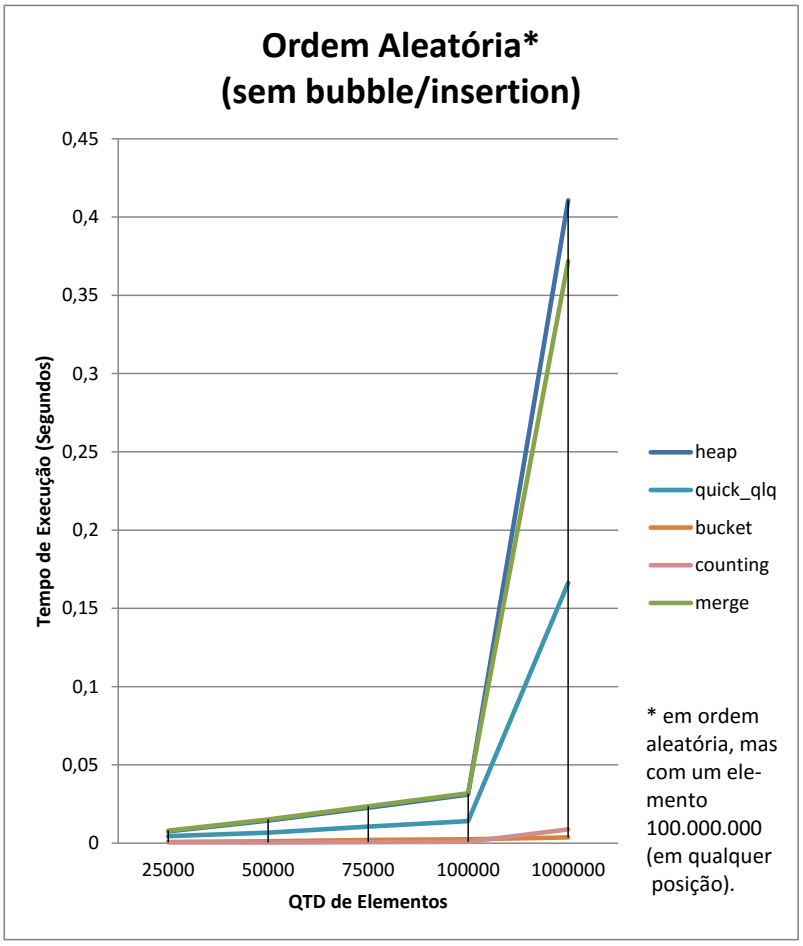

Figura 4. Ordem aleatória com elemento 100000000 sem Bubble/Insertion

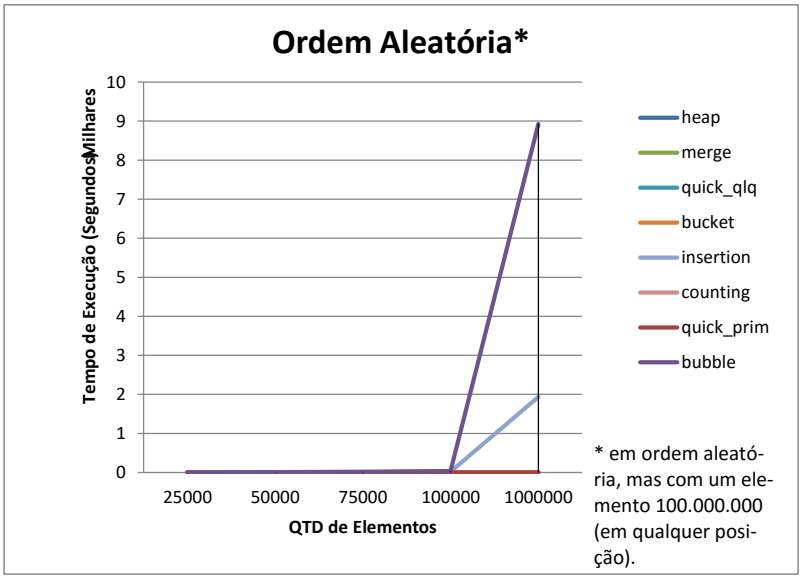

Figura 5. Ordem aleatória com elemento 100000000

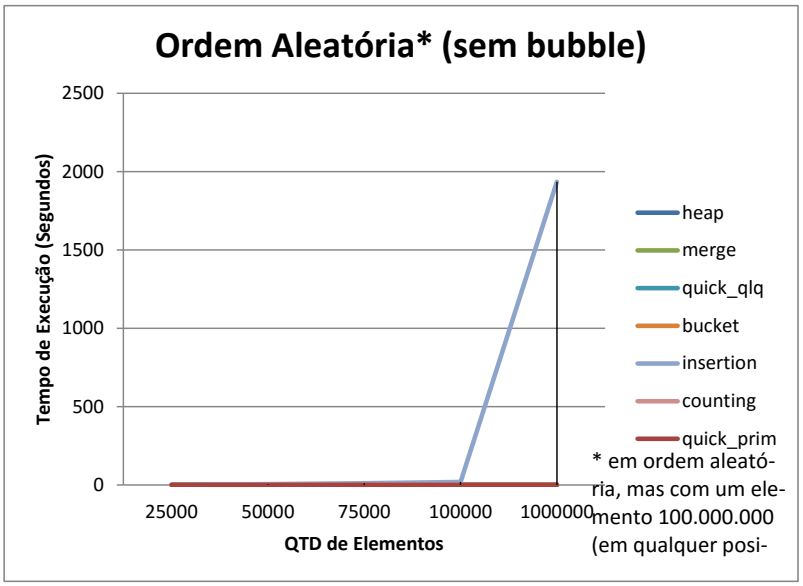

Figura 6. Ordem aleatória com elemento 100000000 sem bubble

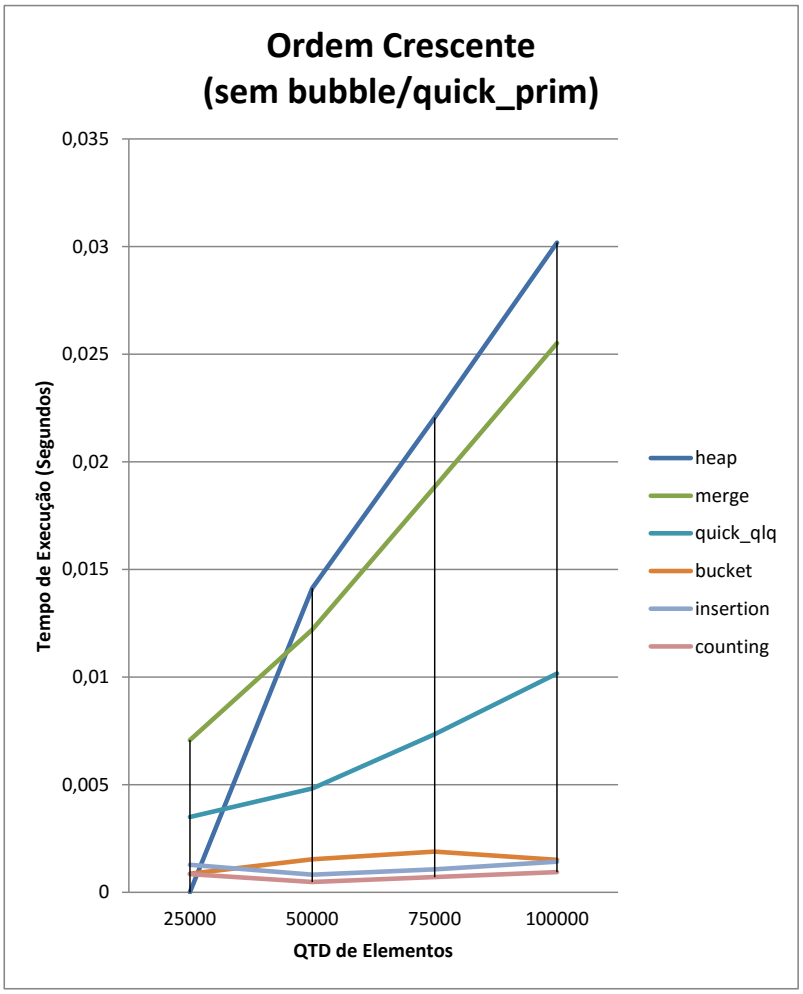

Figura 7. Ordem crescente sem Quick Primeiro Elemento como Pivô e sem Bubble

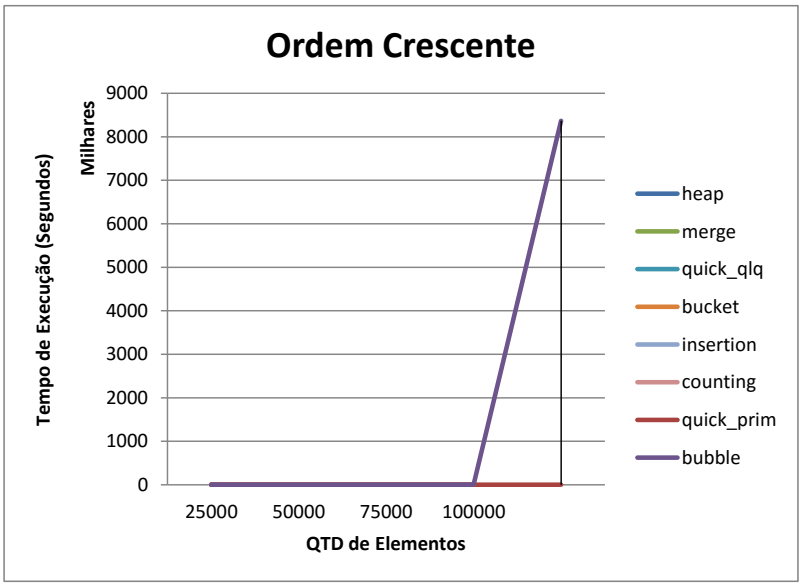

Figura 8. Ordem crescente

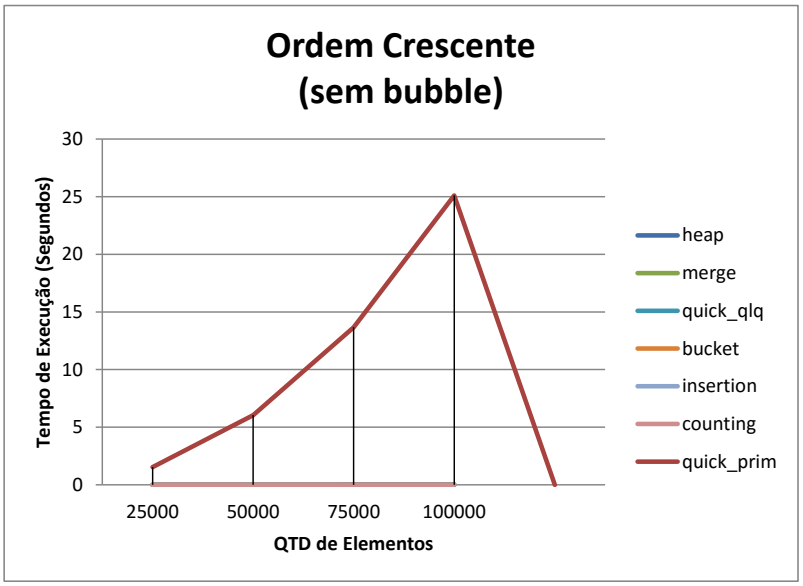

Figura 9. Ordem crescente sem Bubble

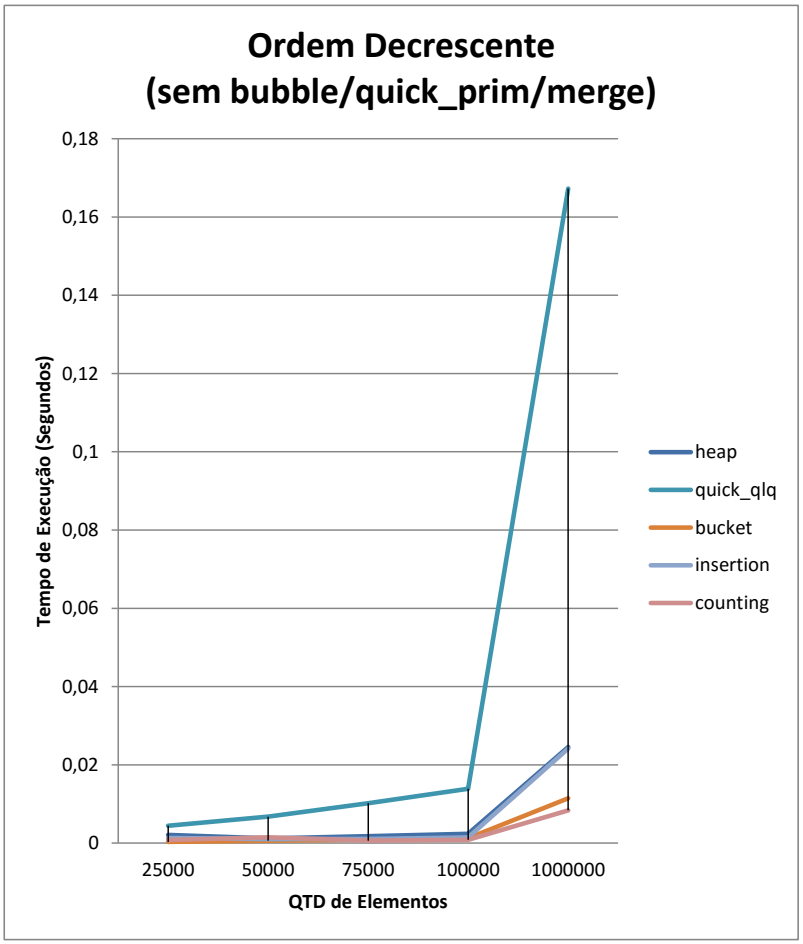

Figura 10. Ordem decrescente sem Quick Primeiro Elemento como Pivô, sem Bubble e sem Merge

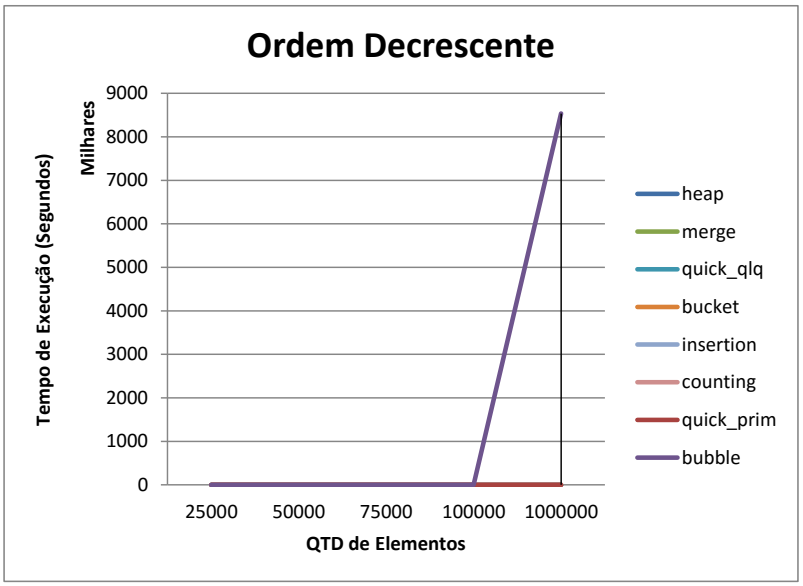

Figura 11. Ordem decrescente

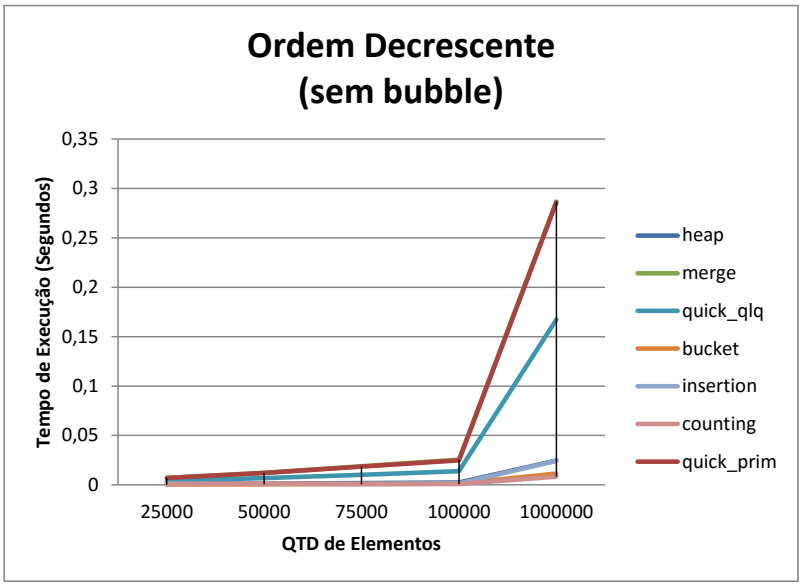

Figura 12. Ordem decrescente sem Bubble

10. Conclusão

Concluímos com este trabalho que para selecionar um algoritmo de ordenação no momento de resolver algum problema deve-se levar alguns fatores em conta.

É necessário averiguar o tamanho do problema a ser resolvido, se for um problema pequeno, um algoritmo de implementação mais simples pode ser o suficiente.

Já sobre os algoritmos em si, é notável o comportamento de suas complexidades, algoritimos de complexidade quadrática obtiveram o pior desempenho, em contrapartida algoritmos logarítimicos obtiveram desempenho médio e algorítmos lineares obtiveram o melhor desempenho, destacando-se o counting sort com os melhores tempos.

Referências

[1] Bucket sort. http://www.growingwiththeweb.com/2015/06/bucket-sort.html. Acesso em, 13 de Setembro, 2016.

[2] Counting sort. http://www.geeksforgeeks.org/counting-sort/. Acesso em, 13 de Setembro, 2016.

[3] Insertion sort. http://www.ft.unicamp.br/liag/siteEd/definicao/insertion-sort.php. Acesso em, 13 de Setembro, 2016.

[4] Paulo Feofillof. Heap sort. http://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html. Acesso em, 13 de Setembro, 2016.

[5] Austin G Walters. Counting sort in c. http://austingwalters.com/counting-sort-in-c/. Acesso em, 13 de Setembro, 2016.