Comparando vetores aleatórios em algoritmos de ordenação:. BubbleSort, InsertionSort e Quicksort

Cleber Soares[1], **Sthefany Oliveira** [1], **Victor Souzal**[1]

[1]Faculdade de Computação e engenharia elétrica
Universidade Federal do Sul e Sudeste do Pará (UNIFESSPA)
68505-080 – Marabá – PA – Brasil

{cleber.soares,sthefany.oliveira,victoor.soouza}@unifesspa.edu.br

Abstract. *Ordering algorithms are computational methods for organizing and / or ordering a list of numbers or words according to their individualities, they can take care of each case that best suits their needs and then improve certain problems that are related to the retrieval of data in lists, that is, facilitate the search of information. This article presents its implementation, execution and comparative among three sorting algorithms that are: Insertion Sort, Bubble Sort, and Quick Sort. For the purpose of analyzing your results and checking the positives and negatives of each.*

Resumo. *Os algoritmos de ordenação são métodos computacionais para organizar e ou ordenar lista de números ou palavras, de acordo com suas individualidades, podem atender cada caso que se adeque melhor a sua necessidade para então melhorar certos problemas que estejam relacionados a recuperação de dados em listas, ou seja, facilitam a busca das informações. Este artigo apresenta a sua implementação, execução e comparativo entre três algoritmos de ordenação que são: Insertion Sort, Bubble Sort, e Quick Sort. Com a finalidade de analisar seus resultados e verificar os pontos positivos e negativos de cada um.*

Introdução

Na computação existem vários tipos de algoritmos que tem diferentes técnicas para realizar operações de ordenação, tendo como fim facilitar a busca de conjuntos de dados, seja para organiza-los ou classifica-los em ordem crescente ou decrescente. Dentre os vários algoritmos, o que serão apresentados neste artigo são: Insertion Sort, Bubble Sort, e Quick Sort.

O Insertion Sort tem funcionamento simples, constitui-se de que a cada passo a partir do segundo elemento do vetor, selecionar o próximo item da sequência e colocá-lo na posição de acordo com o critério da ordenação.

O Bubble Sort, funciona da seguinte forma, são comparados dois elementos do vetor para trocá-los de posição, de forma que os maiores elementos sejam colocados na última posição. Ele geralmente não é recomendado para programas que precisam de velocidade e que operam com grandes volumes de dados.

O Quick Sort, funciona tem um método de ordenação rápido baseando- se no conceito de dividir e conquistar, então é selecionado um elemento que é nomeado de pivô,

divide a lista de entrada em dois subconjuntos para então realizar o mesmo procedimento nos dois menores até torna-la unitária. E é bastante utilizada por se adequar a uma ampla e variedade de situações.

Neste trabalho iremos fazer uma comparação entre esses três algoritmos e tentar concluir qual o melhor algoritmo para ordenação em diferentes casos de teste.

Algoritmos

Bubble Sort

O Bubble Sort é o tipo mais antigo e mais simples usado para ordenações. Ele funciona comparando cada item da lista com o item do lado dele, e efetua a troca se o valor na posição que está sendo analisada for maior que o da posição após a dele. O algoritmo repete este processo até passar por todas as posições da lista. Isto faz com que os valores maiores "flutuem" para o final da lista, enquanto os valores menores "afundem" para o início da lista [da Silva Nascimento et al. 2016].

Tal algoritmo usa a estrategia de troca e seu pior caso O(n^2) e melhor caso O(n).

Algorithm 1 Bubble Sort

```
1:  function BUBBLESORT(A, n)
2:      for i = 0 to i < (n − 1) do
3:          for j = 0 to j < (n − i − 1) do
4:              if A[j] > A[j + 1] then
5:                  temp = A[j]
6:                  A[j] = A[j + 1]
7:                  A[j + 1] = temp
8:              end if
9:          end for
10:     end for
11: end function
```

Insertion Sort

O algoritmo consiste em uma ordenação percorrendo o vetor da esquerda para direita, e conforme avanca, vai ordenando os valores da sua esquerda. Possui um melhor caso em forma ordenada com complexidade de O(n) e os casos medio e pior com complexidade de O(n2) [Santos and Tarôco].

Algorithm 2 Insertion Sort

```
1: function INSERTIONSORT(A, p, r)
2:     for j = 2 to n do
3:         temp = A[j]
4:         i = j − 1
5:         while (i > 0) and(A[i] > temp) do
6:             A[i + 1] = A[i]
7:             i = i − 1
8:         end while
9:         A[i + 1] = temp
10:    end for
11: end function
```

Quick Sort

Em primeiro momento iremos falar sobre a função partition que é de extrema importância para o Quick Sort.

A função partition elege um dos elementos da sequência como elemento pivô. Após essa eleição, cada elemento da sequência é comparado com o pivô e, se for menor que o pivô, deverá ficar à esquerda desse, se for maior que o pivô, deverá ficar à direita. Ao final dessa etapa, o elemento pivô fica na sua posição final da seqüência ordenada [Prado 2005].

Essa função irá determinar o nivel de balanciamento do quick sort que terá grande importância em seu desempenho.

Algorithm 3 Partition

```
1: function PARTITION(A, p, r)
2:     x = A[r]
3:     i = p − 1
4:     for j = p to r − 1 do
5:         if A[j] <= x then
6:             temp = A[i]
7:             A[i] = A[j]
8:             A[j] = temp
9:         end if
10:    end for
11:    temp = A[i + 1]
12:    A[i + 1] = A[r]
13:    A[r] = temp
14:    return i + 1
15: end function
```

O Quicksort é um algoritmo do tipo dividir para conquistar para ordenação cujo tempo de execução no pior caso é $O(n^2)$ sobre uma seqüência de entrada de n números pior caso do Quicksort ocorre quando todos os particionamentos de todos os níveis de

recursão ocorrem da pior maneira possível, ou seja, o elemento escolhido como pivô pela função particiona for sempre maior ou o menor elemento da seqüência [Prado 2005].

O melhor caso do Quicksort ocorre quando todos os particionamentos em todos os níveis de recursão forem ótimos, ou seja, quando toda vez que a função particiona for chamada, ela divida a seqüência exatamente ao meio ($\log_2$n) [Prado 2005].

Algorithm 4 Quick Sort

```
1: function QUICKSORT(A, p, r)
2:     if p < r then
3:         q = PARTITION(A, p, r)
4:         QUICKSORT(A, p, q − 1)
5:         QUICKSORT(A, q + 1, r)
6:     end if
7: end function
```

Algumas caracteristicas em relação ao método utilizado

Algoritmo	Médoto				
	Troca	Seleção	Inserção	Mistura	Partição
Bubble Sort	Sim	Sim	Sim	Sim	Não
Quick Sort	Não	Não	Não	Sim	Sim
Insertion Sort	Sim	Sim	Sim	Não	Não

Algumas caracteristicas em relação ao tipo do algoritmo

Algoritmo	Tipo					
	Lento	Rápido	In-Place	OutOff-Place	Adaptável	Não Adaptáve
Bubble Sort	Sim	Não	Sim	Não	Não	Sim
Quick Sort	Não	Sim	Sim	Não	Sim	Não
Insertion Sort	Sim	Não	Sim	Não	Não	Sim

Algoritmo	Tipo			
	Recursivo	Não Recursivo	Grande Volume	Pequeno Volume
Bubble Sort	Não	Sim	Não	Sim
Quick Sort	Sim	Não	Sim	Sim
Insertion Sort	Não	Sim	Não	Sim

Configurações

Para fim de comparação tais algoritmos foram executados em uma mesma maquina em todos os testes de caso, segue a baixo as configurações utilizadas:

Processador: 4x Intel(R) Core(TM) i5-4690K CPU @ 3.50GHz
Memoria: 8075MB
Sistema Operacional: KDE neon User Edition 5.13
Kernel: Linux 4.13.1-041301-generic x86-64
Disco: ATA WDC WD10EZEX-21W

O vetores utilizados para a ordenação são vetores de tamanhos 5000, 10000 e 15000 em todos os tamanhos há vetores completamente aleatório(entre 0 a 100000), quase ordenado e quase invertido. Nos vetores quase ordenado foram ordenados 90% do vetor aleatório atribuindo esses valores ao mesmo e os 10% restantes foi atribuído de forma aleatória, de forma semelhante foi feito com os vetores quase invertidos com os valores ordenados sendo atribuido inversamente.

Os gráficos foram gerados a parti do número de pulsos de clock decorridos desde inicio da ordenação até seu final e a quantidade de atribuições e comparações em cada ordenação, foi obtido 5 resultados para cada tipo de vetor (vetores de 5000: aleatório, quase ordenado, quase invertido; vetores de 10000: aleatório, quase ordenado, quase invertido; vetores de 15000: aleatório, quase ordenado, quase invertido) e realizado a media a fim de gerar os gráficos, como por exemplo a tabela a seguir.

Random				
Arquivo	Tamanho	Bubble Sort	Insertion Sort	Quick Sort
1	5000	45915	15458	456
2	5000	45150	15431	450
3	5000	45412	15555	455
4	5000	45649	15473	448
5	5000	45901	15232	449
	Media	45605.4	15429.8	451.6

Resultados

Com as configurações utilizadas tivemos os seguintes resutados:

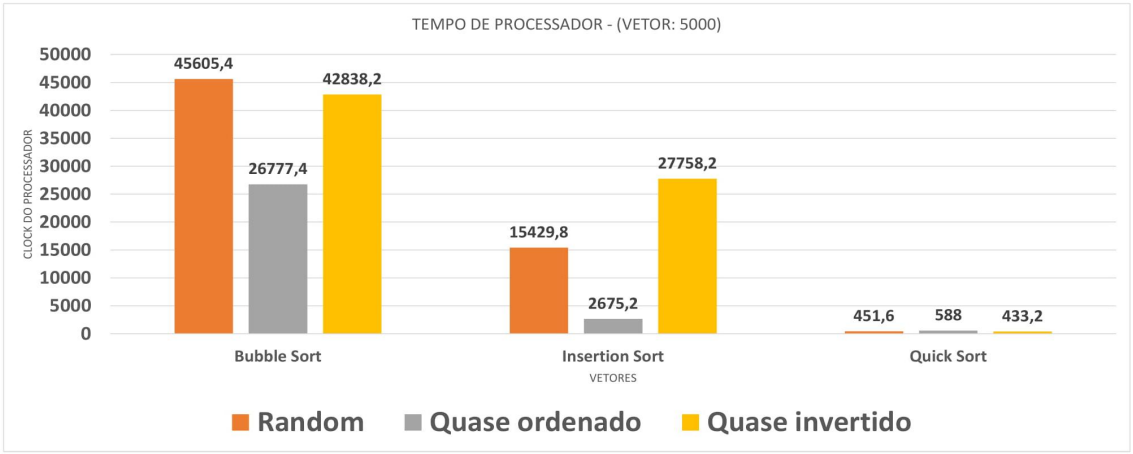

Figura 1. Gráfico tempo de processador (5000)

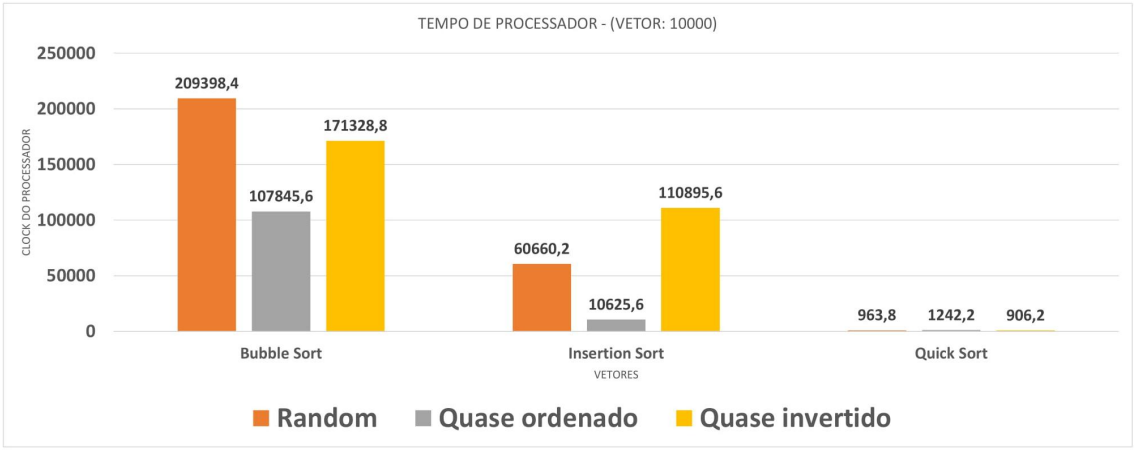

Figura 2. Gráfico tempo de processador (10000)

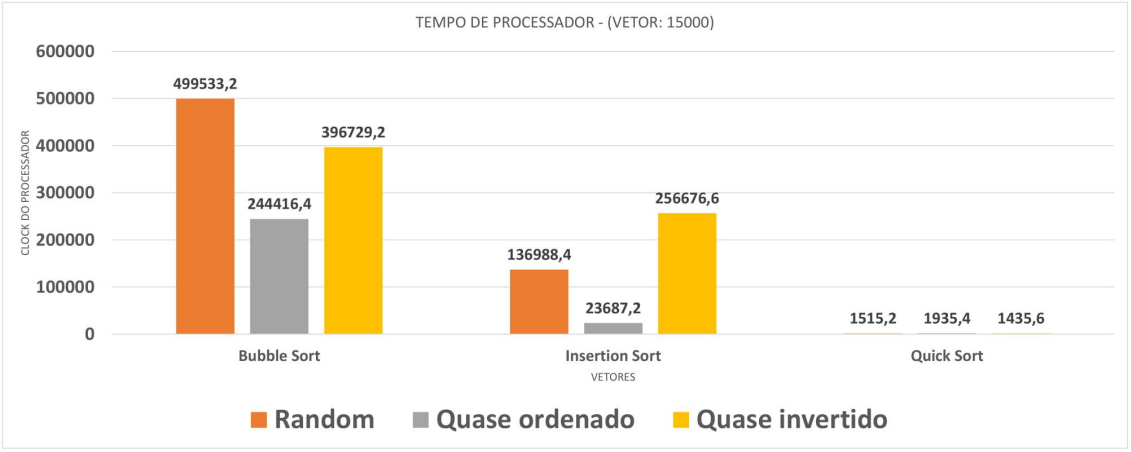

Figura 3. Gráfico tempo de processador (15000)

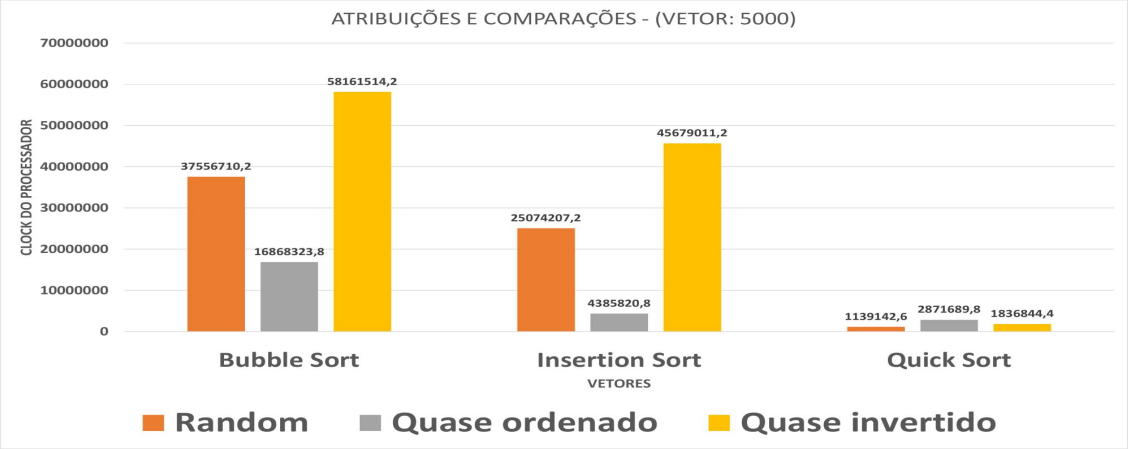

Figura 4. Gráfico atribuições e comparações (5000)

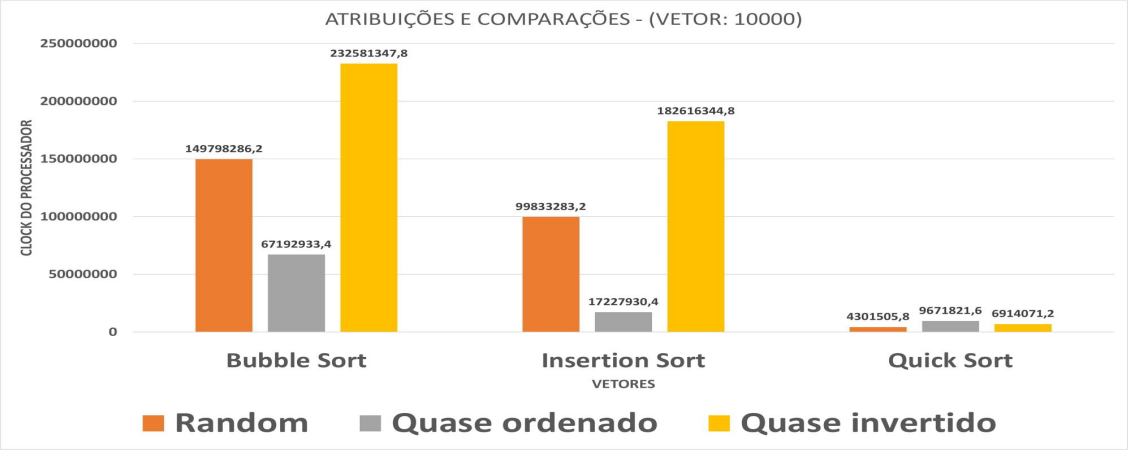

Figura 5. Gráfico atribuições e comparações (10000)

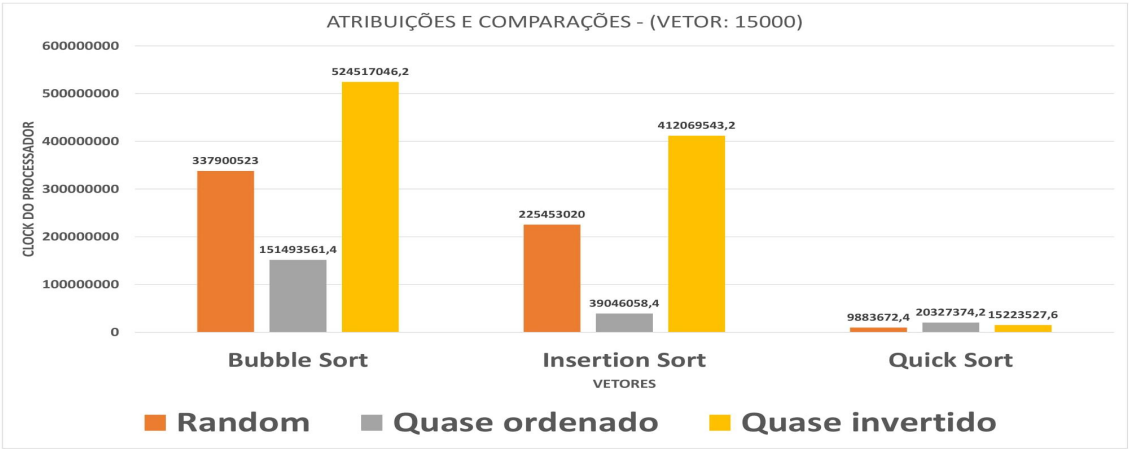

Figura 6. Gráfico atribuições e comparações(15000)

Considerações Finais

A análise sobre os resultados processados permitiu avaliar: que para todos os vetores independemente do tamanho ou sua configuração inicial Quick Sort prevaleceu como melhor tanto em desempenho quanto em velocidade, tendo seu pior caso sempre o

vetor quase ordenado diferente dos demais algoritmos que tiveram ambos como melhor caso os vetores quase ordenados e seu melhor caso foram com os vetores quase invertido.

O segundo melhor algoritmo foi o Insertion Sort que como foi dito antes seu melhor caso para os vetores de todos os tamanhos foi os quase ordenados e seu pior caso foi quase invertido.

Por ultimo temos o Bubble Sort que saiu como algoritmos mais lento e com mais atribuições e comparações em todos casos seu melhor caso foi com os vetores quase ordenados e o pior caso com os vetores aleatórios.

Referências

da Silva Nascimento, J., Mozzaquatro, P. M., and Antoniazzi, R. L. (2016). Análise e comparação de algoritmos implementados em java. *Simpósio de Pesquisa e Desenvolvimento em Computação*, 1(1).

Folador, J. P., Neto, L. N. P., and Jorge, D. C. (2014). Aplicativo para análise comparativa do comportamento de algoritmos de ordenação. *Revista Brasileira de Computação Aplicada*, 6(2):76–86.

Prado, J. A. S. (2005). Análise experimental do quicksort probabilístico com gerador de números pseudo-aleatórios penta-independente.

Santos, E. and Tarôco, J. Uma comparaçao de algoritmos de ordenaçao baseados em comparaçao.