FCT – Faculdade de Ciência e Tecnologia

Bacharelado em Ciência da Computação

Trabalho Prático

Projeto de comparação entre análise experimental e assintótica utilizando algoritmos de ordenação.

Caroline Martins Alves e Lucas Bernardo de Souza

Presidente Prudente

2022

Bubble Sort versão original.

Análise assintótica

A seguir é mostrado o principal trecho de código desse algoritmo:

```
1.      Para n ← 1 até dimensão faça
2.         Início
3.         Para i ← 0 até dimensão – 1 faça
4.            Início
5.            Se(X[i] > X[i+1])
6.            Então início
7.               Aux ← X[i]
8.               X[i] ← X[i+1]
9.               X[i+1] ← aux
10.          Fim
11.        Fim
12.     Fim
```

Verifica-se que o número de iterações do primeiro laço é ‘dimensão’ que é o tamanho do vetor e o segundo laço é uma iteração a menos que o primeiro, mas como está interno ao primeiro temos o produto entre a quantidade de iterações dos dois laços, ou seja, o produto entre dimensão e dimensão – 1 determina a quantidade de iterações desse algoritmo. Para um vetor de tamanho n o algoritmo realizará $n(n – 1) = n^2 – n$ comparações. Logo podemos concluir que a execução do algoritmo BUBBLE SORT é $O(n^2)$.

Para qualquer que seja o tamanho e as características do vetor de entrada esse algoritmo se comportará da mesma forma e não apresentará situações de melhores ou piores casos. (FIGURA 1)

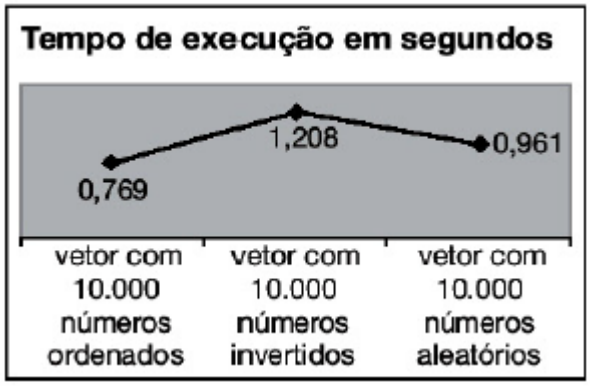
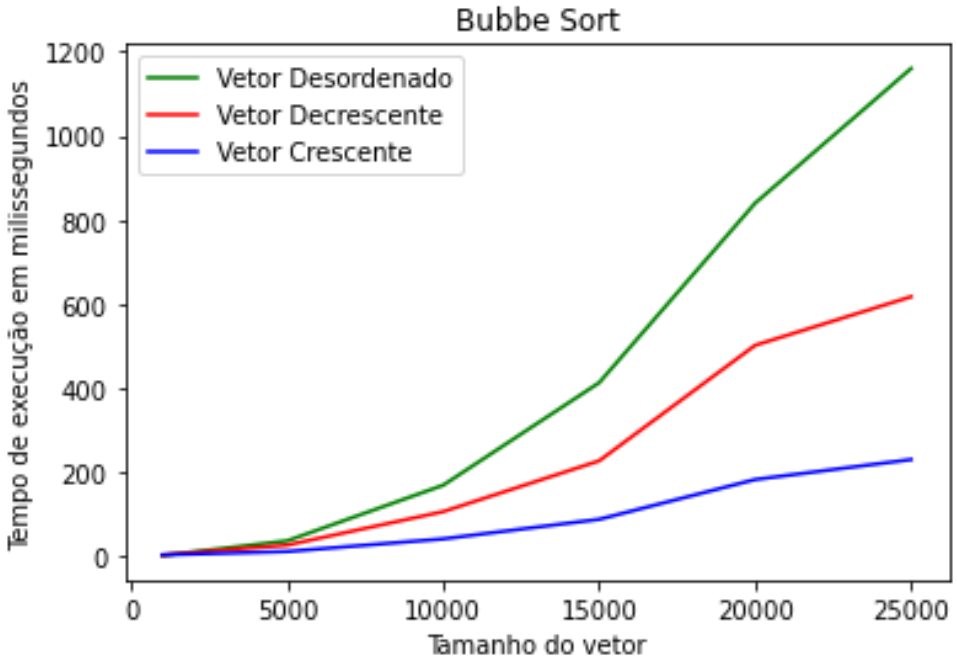


Figura 1 ASCENCIO, Ana; ARAÚJO, Graziela. Estrutura de dados algoritmos, análise de complexidade e implementações em Java e C/C++. (2010, p. 27)

Análise experimental



Na análise experimental é observável um comportamento semelhante ao da análise assintótica, contudo aqui ocorreu uma peculiaridade em que o caso médio obteve um pior desempenho em relação ao pior caso em que o vetor estava ordenado de forma decrescente. O comportamento da função, entretanto é o mesmo ou semelhante, quanto maior a entrada mais operações são realizadas e consequentemente o tempo de execução é maior.

Bubble Sort melhorado.

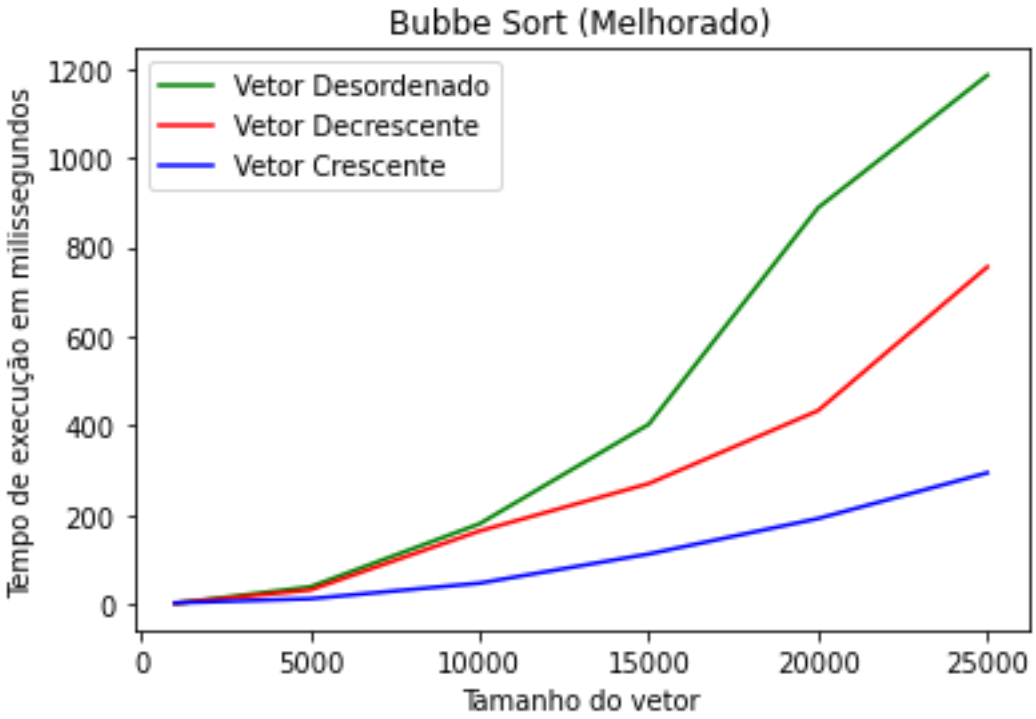
Análise assintótica

A seguir é mostrado o trecho principal desse algoritmo

```
1.      Para n ← 0 até dimensão faça
2.         Início
3.         estaOrdenado ← true
4.         Para i ← 0 até dimensão -1 faça
5.            Início
6.            Se(X[i] > X[i+1])
7.            Então início
8.               Aux ← X[i]
9.               X[i] ← X[i+1]
10.              X[i+1] ← aux
11.              estaOrdenado ← false
12.           Fim
13.         Se(estaOrdenado)
14.         Então retorna X
15.         Fim
16.      Retorna X
```

Esse algoritmo é semelhante a primeira versão do bubble sort, com a diferença que ele verifica se o vetor já está ordenado. O número de iterações no primeiro laço é dimensão e no segundo dimensão -1, logo temos no pior caso *n(n-1)* iterações, que corresponde ao valor assintótico $O(n^2)$. Já no melhor caso o número de operações diminui, pois o condicional do segundo laço não é satisfeito, contudo a complexidade do algoritmo continua sendo $O(n^2)$.

Análise experimental



Na análise experimental, o resultado obtido foi muito semelhante ao da primeira versão do algoritmo e assim como na primeira versão o caso médio que é composto por vetores desordenados obteve pior desempenho.

Quick Sort

Análise assintótica

A ideia desse algoritmo é dividir o vetor em duas partes, o que é feito por uma função. Nesse procedimento o vetor é particionado na posição j, de modo que todos os elementos do lado esquerdo de j são menores ou iguais ao elemento denominado pivô. O tempo de execução é limitado pelo tamanho do vetor. Isso ocorre uma vez que o algoritmo

compara todos os elementos do vetor com o pivô enquanto os índices atenderem a condição i < j. Logo, o procedimento partição realizará *O(n)* comparações.

O tempo de execução depende se o particionamento é ou não balanceado. O pior caso ocorre quando o procedimento de particionamento produz uma região com n-1 elementos e outra com somente um elemento. O seu tempo no pior caso é $\Theta(n^2)$. O melhor caso ocorre quando o procedimento de particionamento produz duas regiões de tamanho n/2. O seu tempo de execução no melhor caso é $T(n) = \Theta(n \times log\ n)$.

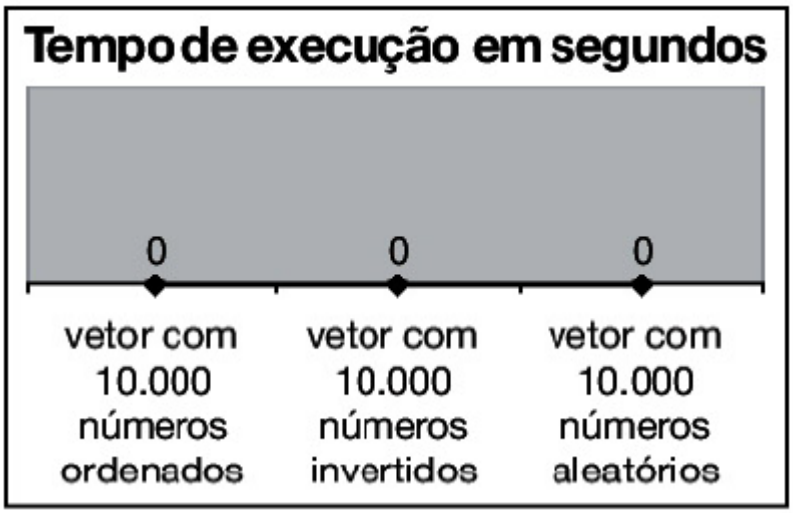
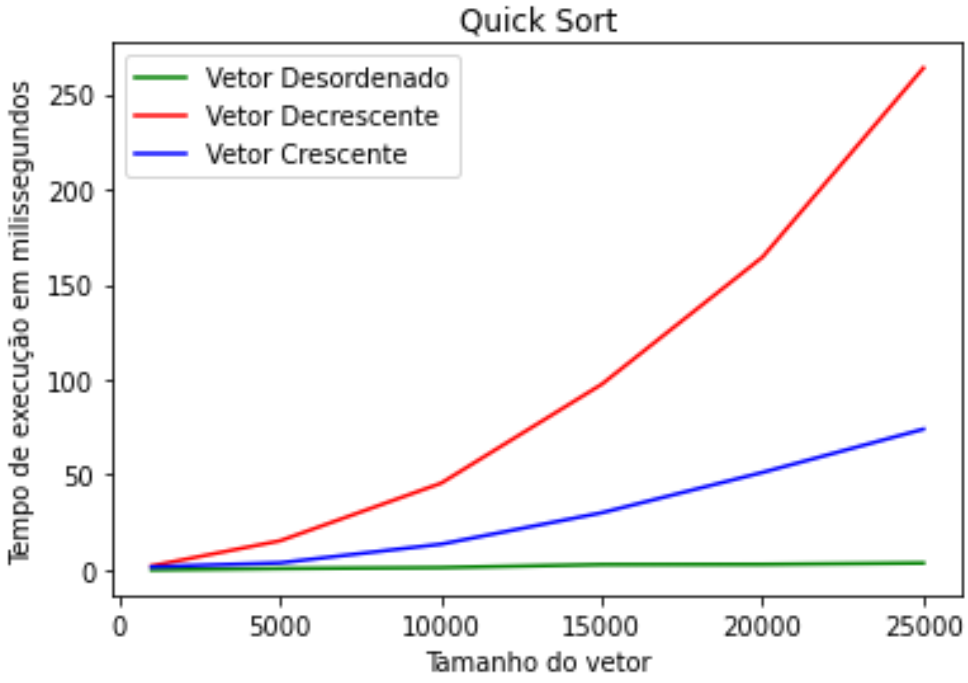
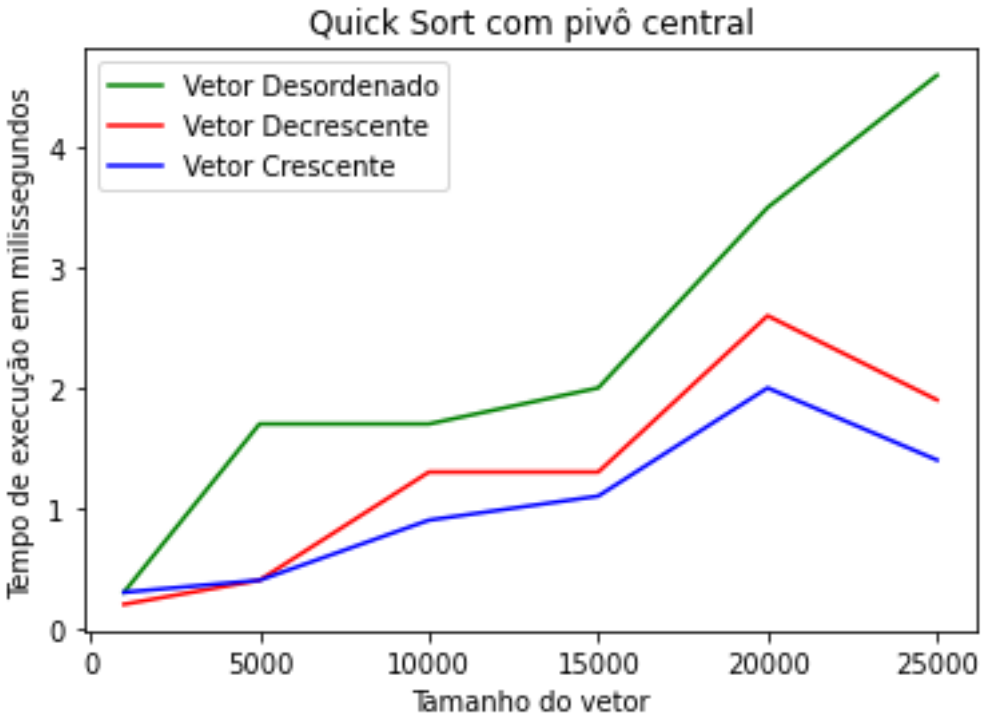


Figura 2 ASCENCIO, Ana; ARAÚJO, Graziela. Estrutura de dados algoritmos, análise de complexidade e implementações em Java e C/C++. (2010, p. 75)

Análise experimental





A nossa análise experimental é comprovada na análise assintótica, uma vez que temos um desempenho muito superior no caso em que o pivô do quick sort é central. Com pivô central o tempo de execução é de $T(n) = \Theta(n \times log\ n)$. No caso em que o particionamento começa no inicio do vetor gerando uma partição com n-1 o desempenho é inferior, com tempo de execução $\Theta(n^2)$. Contudo se mostrou melhor que o Bubble Sort.

Insertion Sort

Análise assintótica

O trecho do algoritmo onde ocorre a ordenação é:

```
1.      Para i ← 1 até dimensão faça
2.      Início
3.      Eleito ← X[i]
4.      J ← i-1
5.      Equanto(j >= 0 E X[j] > eleito)
6.         Início
7.         X[j+1] ← X[j]
8.         J ← j-1
9.         Fim
10.     X[j+1] ← eleito
11.     Fim
```

Na implementação acima do algoritmo temos duas estruturas de repetição. A primeira é executada n-1 vezes e a segunda estrutura ‘enquanto’ a princípio também é executada n-1 vezes contudo dependendo do vetor de entrada pode ser executado menos

vezes. O pior caso ocorre quando o vetor de entrada está na ordem inversa a ser ordenada. E o tempo de execução é dada pela fórmula:

$$T(n) = 2 + 3 + 4 + \cdots + n$$

$$T(n) = \left(\sum_{i=1}^{n} i\right) - 1$$

$$T(n) = \frac{(1+n)n}{2} - 1$$

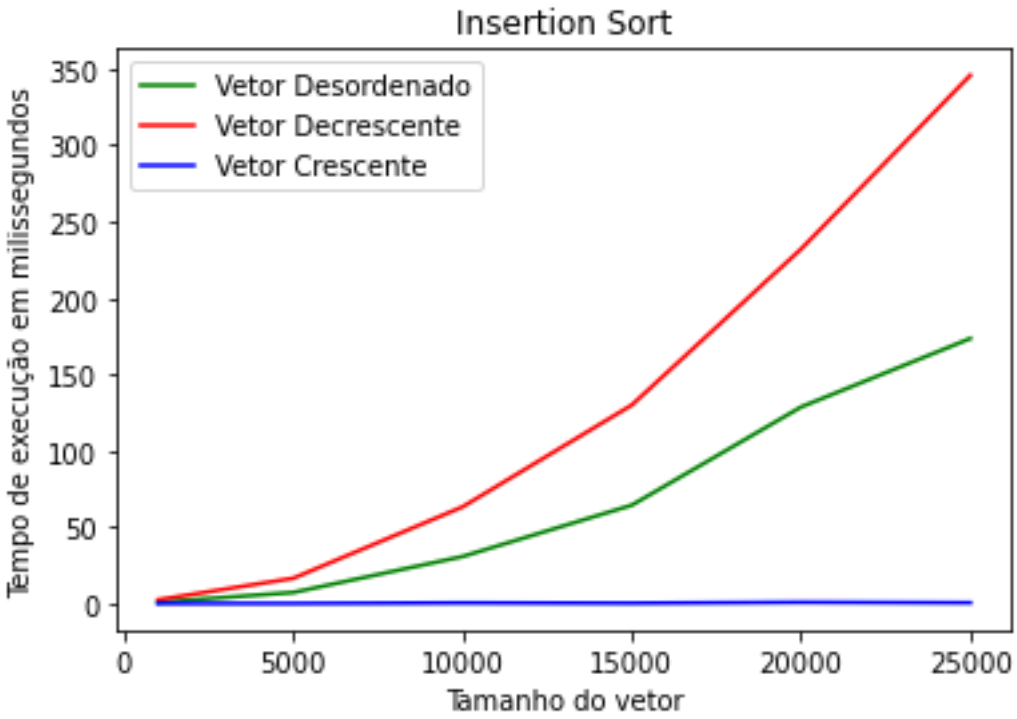
$$T(n) = \frac{n^2 + n}{2} - 1$$

$$T(n) = O(n^2), para\ c = 2, n \geq 1.$$

O melhor caso é quando o vetor de entrada já está ordenado e possui tempo de execução T(n) = O(n-1).

Figura 3 ASCENCIO, Ana; ARAÚJO, Graziela. Estrutura de dados algoritmos, análise de complexidade e implementações em Java e C/C++. (2010, p. 45)

Análise experimental



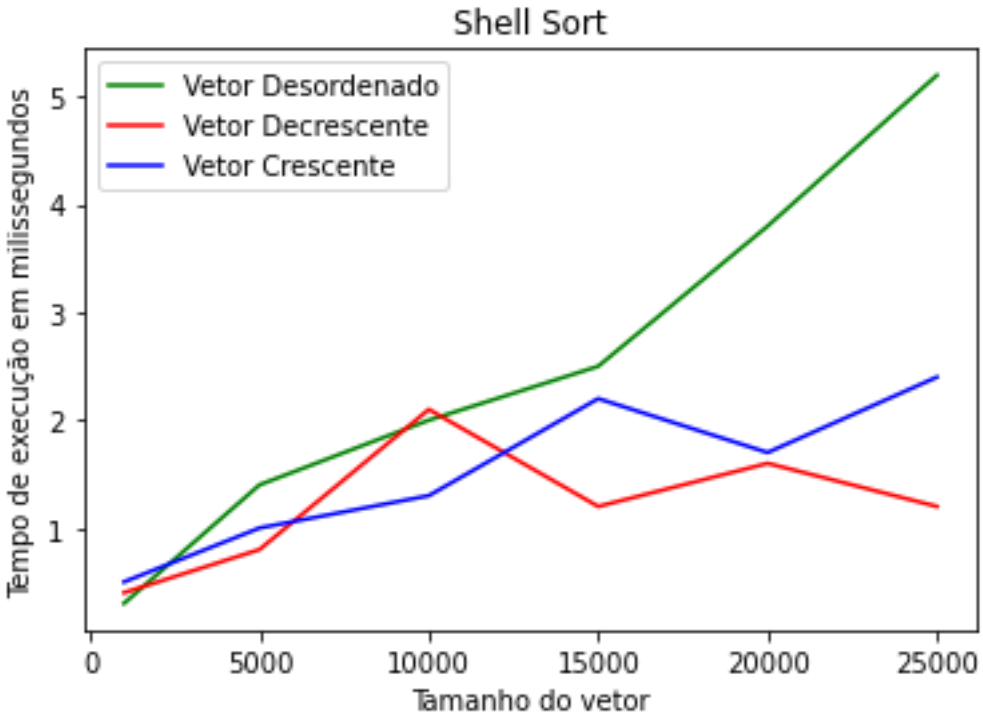
A análise assintótica pode ser visualizada no gráfico da análise experimental. Temos o pior caso quando o vetor está na ordem inversa e tem tempo de execução $O(n^2)$ que no gráfico é a linha vermelha 'vetor decrescente', o pior desempenho desse algoritmo. Já o melhor caso é quando o vetor já está ordenado, linha azul no gráfico com tempo de execução O(n-1).

Shell Sort

Análise assintótica

Esse algoritmo é uma versão melhorada do insertion sort que tenta evitar o pior caso quadrático. No melhor caso tem complexidade de tempo Θ(n lg2 n). No pior caso a complexidade é a mesma do insertion sort $\Theta(n^2)$.

Análise experimental



A nossa análise experimental concluiu que de fato o shell sort tem desempenho superior ao insertion sort e ele evita o pior caso para vetores inversos, que tem complexidade $\Theta(n^2)$.

Selection Sort

Análise assintótica

O trecho de código a seguir é o principal do selection sort:

```
1.      Para i ← 0 até 3 faça
2.      Início
3.         Eleito ← X[i]
4.         Menor ← X[i+1]
5.         Pos ← X[i+1]
6.         Para j ← i+1 até 4 faça
7.            Início
8.            Se(X[j]<menor)
9.            Então início
10.             Menor ← X[j]
11.             Pos ← j
12.           Fim
13.     Fim
14.     Se(menor < eleito)
15.     Então início
16.           X[i] ← X[pos]
17.           X[pos] ← eleito
18.           Fim
19.     Fim
```

Para i = 0 no primeiro loop (linha 1) o segundo loop (linha 6) por consequência é executado n – 1 vezes. Já para i = 1 no primeiro loop o segundo loop (mais interno) será executado n – 2 vezes. Para i = 2 no for mais externo o for mais interno será executado n – 3 vezes e assim sucessivamente até o último valor de i. Logo o tempo de execução do Selection Sort é $\Theta(n^2)$ independente do vetor de entrada.

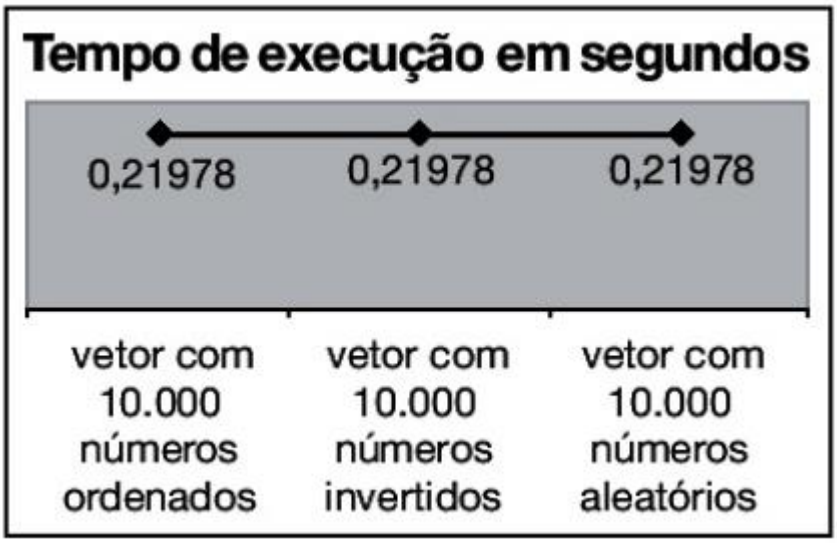
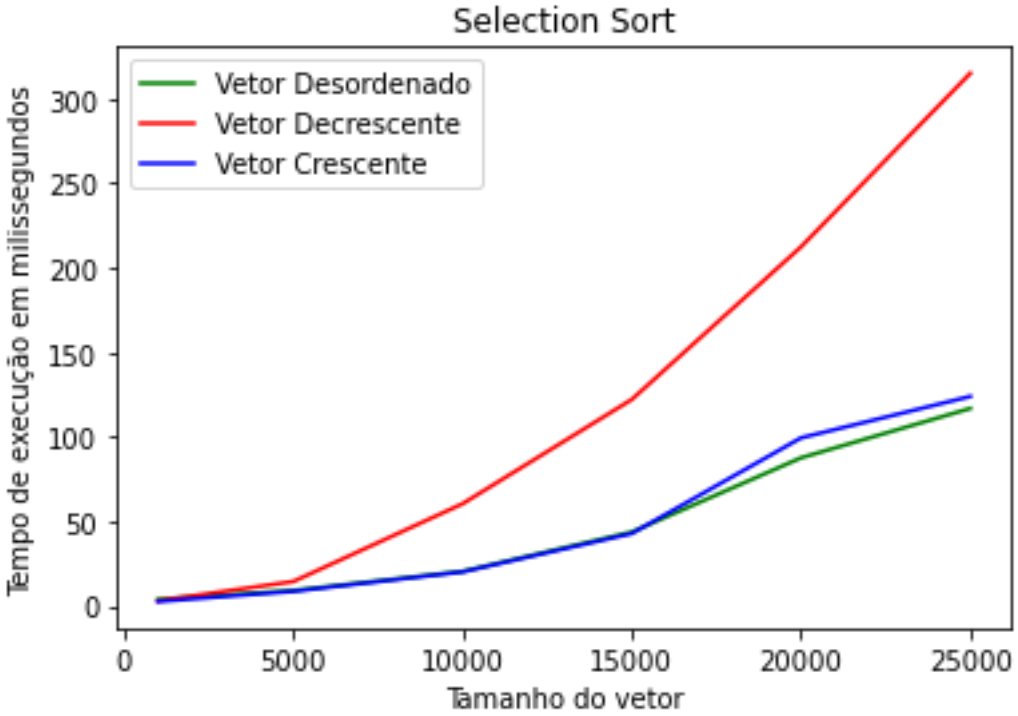


Figura 4 ASCENCIO, Ana; ARAÚJO, Graziela. Estrutura de dados algoritmos, análise de complexidade e implementações em Java e C/C++. (2010, p. 52)

Análise experimental



Apesar da análise assintótica afirmar que o desempenho é o mesmo para diferentes tipos de vetores, na análise experimental o vetor ordenado decrescente teve um pior desempenho em comparação com o vetor ordenado crescentemente e o desordenado.

Heap sort

Análise assintótica

O algoritmo heap utiliza-se de outros dois procedimentos que vamos começar a analisar que é o heap_fica e transforma_heap.

O procedimento heap_fica é aplicado a um elemento do vetor e 'afunda' esse elemento ou nó da arvore, uma vez que aqui o vetor é visto como uma árvore, até que a propriedade heap seja válida. Seu pior caso ocorre quando o procedimento é aplicado ao nó raiz da árvore. Logo o número de trocas realizadas corresponderá à altura da árvore que é log n. Portanto o procedimento heap_fica tem complexidade O(log n).

Já o procedimento transforma_heap faz uso do heap_fica. Temos que qtde é o número de elementos do vetor e o loop é executado qtde/2 vezes. Sabendo que o tempo de execução do heap_fica é log n, portanto o tempo de execução deste procedimento é $T(n) = \frac{n}{2}\log n = \ \mathrm{O}(n \log n)$.

```
1.      Para i ← qtde/2 até 1 faça passo -1
2.      Início
3.         Heap_fica(i,qtde)
4.      Fim
```

Com o vetor já transformado a ordenação de fato ocorre, por meio do procedimento ordena que é descrito abaixo:

```
Fução ordena(qtde numérico)
1.      Início
2.      Declare i, aux, ultima_posi numérico
3.      Para i ← qtde até 2 faça passo -1
4.      Início
5.         Aux ← X[1]
6.         X[1] ← X[i]
7.         X[i] ← aux
8.         Ultima_posi ← i-1
9.         Heap_fica(1, ultima_posi)
10.     Fim
11.     Fim_funcao_ordena
```

O tempo de execução do procedimento é $T(n) = (n-1)\log n = \mathrm{O}(n \log n)$.

Apesar da transformação do vetor em heap antes da ordenação em si, que tem custo n log n, o tempo do algoritmo heap sort não ultrapassa o limitante n.log n.

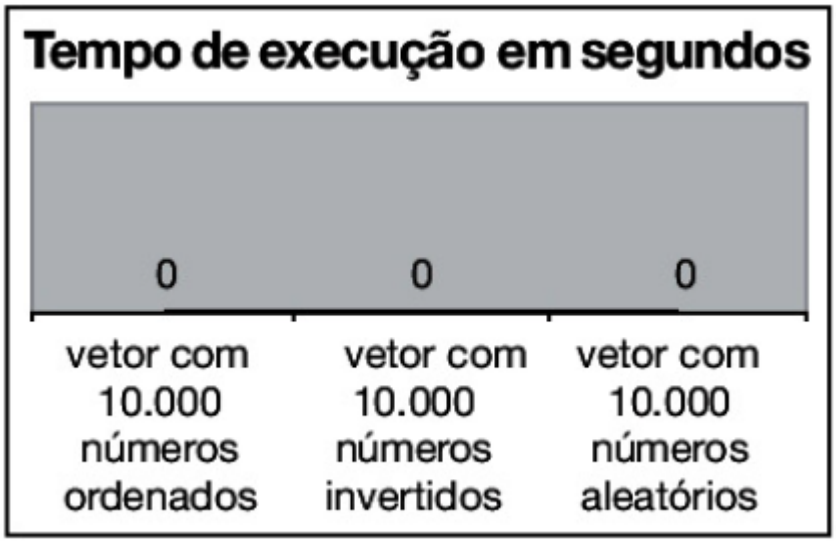
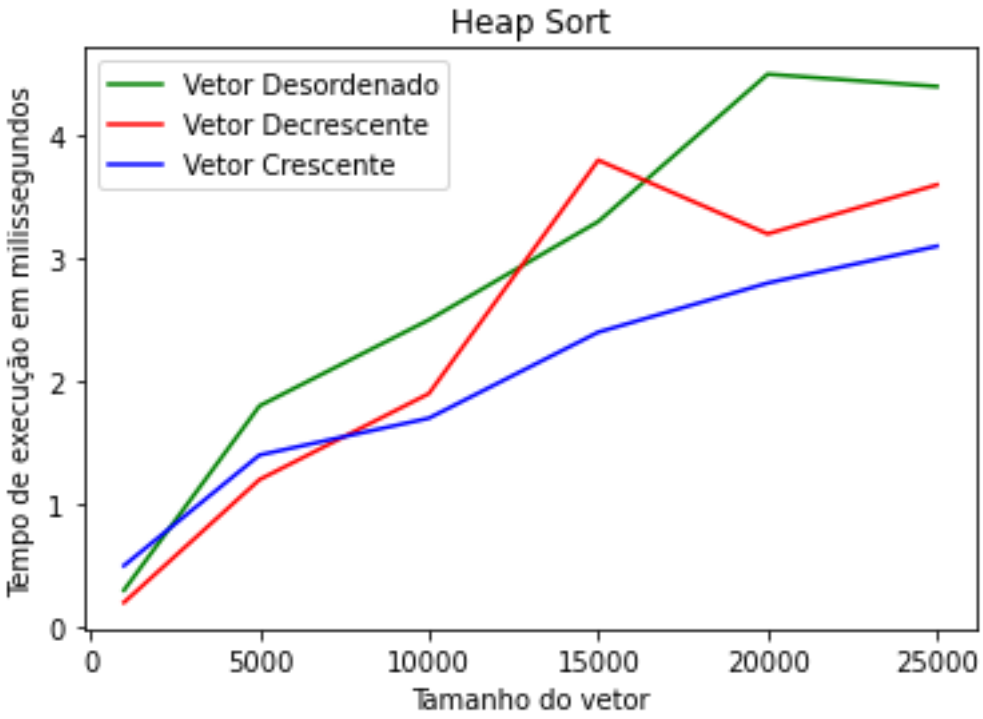


Figura 5 ASCENCIO, Ana; ARAÚJO, Graziela. Estrutura de dados algoritmos, análise de complexidade e implementações em Java e C/C++. (2010, p. 90)

Análise experimental



Como dito na análise assintótica, na nossa análise experimental é constatado a eficiência desse algoritmo e para os três casos temos um comportamento semelhante.

Merge sort

Análise assintótica

Trecho de código relevante para o merge sort:

```
1.      Funcao merge(X, início, fim)
2.       Início
3.         Declare meio numérico
4.         Se(inicio < fim)
5.            Então início
6.                  Meio ← parteinteira((inicio+fim)/2)
7.                  Merge(X,início,meio)
8.                  Merge(X,meio+1,fim)
9.                  Intercala(X, início, fim, meio)
10.                 Fim
11.     Fim_funcao_merge
```

A função intercala realiza a intercalação de dois vetores, cujos tamanhos sejam m_1 e m_2 ela faz a varredura de todas as posições dos dois vetores gastando com isso n = $m_1 + m_2$.

Como o merge é um algoritmo recursivo para obtermos o seu tempo de execução precisamos da expressão de recorrência. Então, desconsiderando a princípio as funções piso e teto, a expressão de recorrência é dada por $T(n) = 2T\left(\frac{n}{2}\right) + n$. Utilizando o método mestre para resolver a recorrência. Os valores necessários para resolução por esse método são: a=2, b=2, f(n)=n. Como:

$$f(n) = \Theta\left(n^{\log_b a}\right)$$

$$n = \Theta\left(n^{\log_2 2}\right)$$

$$n = \Theta(n^1)$$

Então

$T(n) = \Theta(n \log n)$

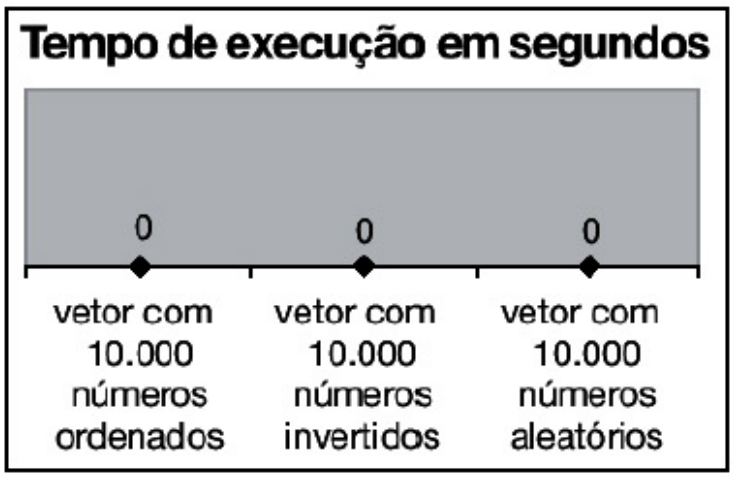
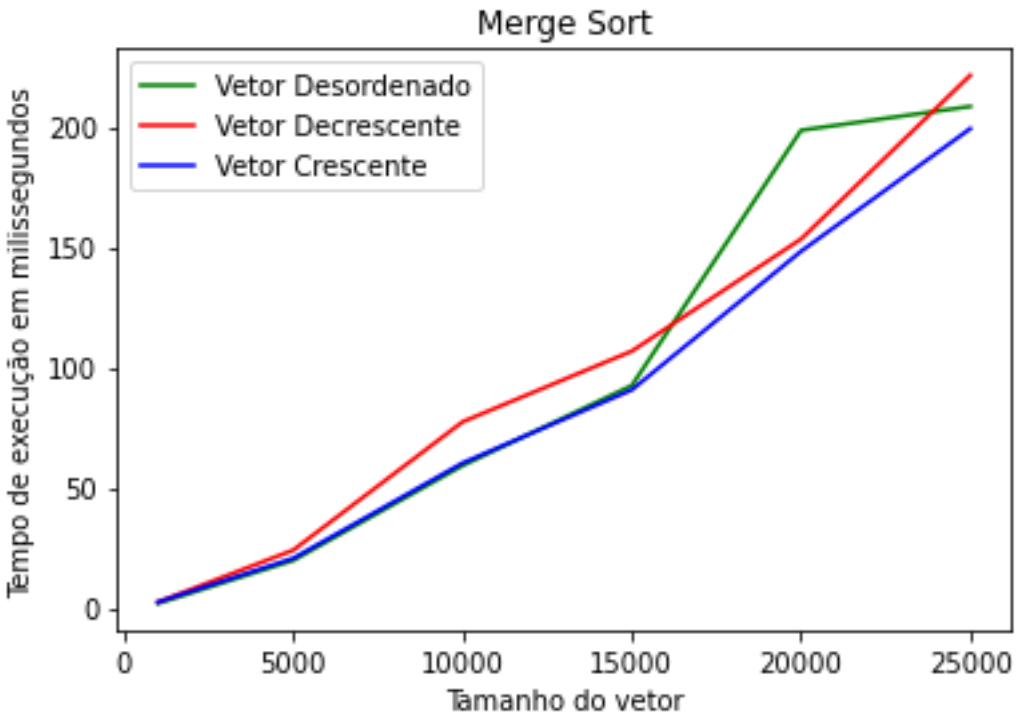


Figura 6 ASCENCIO, Ana; ARAÚJO, Graziela. Estrutura de dados algoritmos, análise de complexidade e implementações em Java e C/C++. (2010, p. 60)

Análise experimental



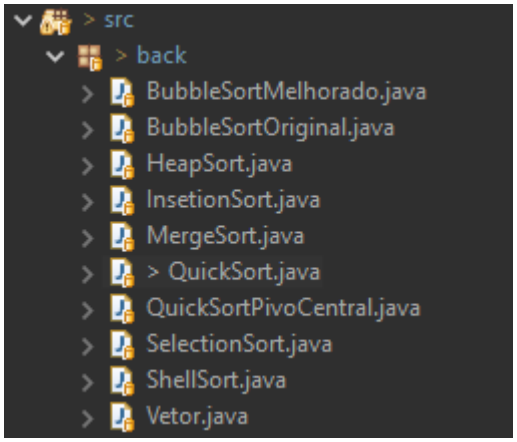
Na análise experimental, o desempenho do merge ficou inferior ao heap mesmo ambos tendo custo n log n. O comportamento se manteve mais ou menos o mesmo para os três tipos diferentes de vetores.

Observações sobre a implementação

A linguagem escolhida para a implementação foi o Java e o ambiente de desenvolvimento utilizado foi o eclipse. Além disso, usamos as bibliotecas pandas e matplotilib para tratamento dos dados de tempo e geração dos gráficos.

O programa desenvolvido para o teste dos algoritmos é composto por dois pacotes, o primeiro deles é o ‘back’ que contém as classes dos algoritmos de ordenação e a classe que gera os vetores. Os algoritmos analisados foram implementados em classes que são compostas por atributos necessários para o algoritmo e o vetor que se desejaria ordenar. Além dos atributos as classes dos algoritmos implementam o método ordena que implementa de fato o algoritmo de ordenação, recebe como parâmetro o vetor a ser ordenado e retorna o vetor já ordenado. Outra classe presente no pacote ‘back’ é a classe vetor, que tem como tarefa gerar os vetores que serão ordenados, ela implementa o atributo ‘vet’ e os métodos ‘gerar’ que gera um vetor com números aleatórios e recebe como parâmetro a dimensão do vetor, ‘gerarCrescente’ inicializa o atributo vet com

números crescentes e recebe como parâmetro a dimensão e o último método é o ‘gerarDecrescente’ que inicializa o atributo ‘vet’ em ordem decrescente e recebe como parâmetro a dimensão.



No pacote ‘controlador’ temos o programa em si. Instanciação das classes e testamos para vetores de dimensão mil, cinco mil, dez mil, quinze mil, vinte mil e vinte e cinco mil. Cada algoritmo foi testado para todas essas entradas e testado dez vezes para cada entrada. Para gerar os gráficos obtemos a média dos dez testes para cada tamanho de vetor. Ou seja, o merge foi testado dez vezes para ordenar o vetor com mil posições e armazenado todos os tempos de resposta, mais dez vezes para ordenar o vetor com cinco mil e assim sucessivamente, tanto para vetor com números aleatórios quanto para vetor em ordem crescente e decrescente. Esse processo foi feito para todos os algoritmos e os dados foram armazenados em uma planilha como essa:

1	Tamanho	Bubble Sort	Bubble Sort (Melhorado)	Quick Sort	Quick Sort (Pivo Central)	Insetion Sort	Shell Sort	Selection Sort	Heap Sort	Merge Sort
2	1000	8	11	0	1	3	1	6	1	3
3	1000	5	2	0	0	1	1	2	0	3
4	1000	5	5	0	1	1	0	2	1	3
5	1000	4	1	0	0	1	0	4	0	2
6	1000	2	1	0	0	1	1	4	0	0
7	1000	1	1	1	0	1	0	4	0	0
8	1000	2	1	0	0	1	0	3	1	3
9	1000	1	1	0	1	1	0	4	0	3
10	1000	1	1	0	0	1	0	9	0	1
11	1000	2	1	0	0	0	0	1	0	1
12	5000	45	61	2	2	20	5	40	2	41
13	5000	39	36	1	7	17	2	11	2	17
14	5000	34	38	1	1	6	1	5	4	11
15	5000	35	40	1	3	4	1	5	2	34
16	5000	41	36	0	0	4	1	5	1	41
17	5000	45	37	1	1	5	1	5	1	9
18	5000	34	38	0	1	5	1	5	2	12
19	5000	35	37	1	1	5	0	5	1	10
20	5000	39	36	1	1	5	1	4	2	12
21	5000	34	39	0	0	4	1	6	1	11
22	10000	176	192	3	3	86	5	28	3	109
23	10000	174	194	2	3	65	2	18	3	66
24	10000	161	143	2	2	20	2	20	3	78
25	10000	174	177	1	2	21	3	20	2	106

Referências

ASCENCIO, A. F. G.; ARAÚJO, G. S. **Estruturas de dados:** algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson, 2010.

BAELDUNG. Shell Sort in Java. **Baeldung**, 2020. Disponivel em: <https://www.baeldung.com/java-shell-sort>. Acesso em: 20 Janeiro 2022.

BRUNET, J. A. Ordenação por Comparação: Quick Sort. **Estruturas de dados e algoritmos**, 2019. Disponivel em: <https://joaoarthurbm.github.io/eda/posts/quick-sort/>. Acesso em: 20 Janeiro 2022.

DEVMEDIA. Algoritmos de ordenação. **Devmedia**, 2006. Disponivel em: <https://www.devmedia.com.br/algoritmos-de-ordenacao/2622>. Acesso em: 20 Janeiro 2022.

SOUZA, R. M.; OLIVEIRA, F. S.; PINTO, P. E. Análise Empírica do Algoritmo Sehllsort. **XXXVI Congresso da Sociedade Brasileira de Computação**, Rio de Janeiro, p. 903-906.