Performance em algoritmos de ordenação em Java
Análises e comparações

Pedro Lunardelli Antunes[1]

[1]Escola Politécnica – Pontifície Universidade Católica do Paraná (PUC-PR)

LINK DO GITHUB: https://github.com/Pow3rfulR2307/Ordenacao-em-Java

Resumo. *Este documento tem como objetivo servir como uma análise do tempo de ordenação de vetores preenchidos aleatóriamente com 50,500,1000,5000,10000, usando três algoritmos de ordenação diferentes, Bubble Sort, Shell Sort e Heap Sort. Esses aloritmos também serão análisados levando em conta a quantidade de iterações realizadas bem como a quantidade de trocas de valores realizadas para efetuar a ordenação. Também é possível encontrar tabelas comparando cada valor analisado.*

1. Bubble Sort

O algoritmo de Bubble Sort funciona com base na troca sequencial de valores de um conjunto de dados. A cada iteração do loop é verificado se o valor atual é maior do que o valor da posição seguinte, isso em uma ordenação do menor para o maior não é permitido, ocorre então a troca da posição entre os dois valores. Bubble Sort acaba sendo uma operação muito custosa, realiza uma quantidade enorme de iterações sobre os valores do conjunto de dados, o que poderia ser evitado com outros algoritmos. A escolha do uso do Bubble Sort foi devido à sua grande disparidade de performance em relação aos outros algoritmos presentes nesse documento, muito bom para a comparação de sua eficiência. Abaixo há uma tabela que mostra o tempo médio de execução de ordenação de diferentes vetores pelo seu tamanho, bem como a quantidade de iterações realizadas sobre o vetor e a quantidade de trocas realizadas. Como é possível observar, o Bubble Sort é um péssima escolha para vetores muito grandes, a quantidade de iterações é absurdamente desnecessária.

Table 1. Relatório Bubble Sort

	A	B	C	D
1	TAMANHO	ITERAÇÕES	TROCAS	MÉDIA DE TEMPO DE ORDENAÇÃO
2	50	2400	563	0.08275999 ms
3	500	249000	62726	2.8684 ms
4	1000	998000	249583	1.6541601 ms
5	5000	24990000	6316974	20.1606 ms
6	10000	99980000	25036405	83.56743 ms
7				

2. Shell Sort

O algoritmo de Shell Sort tem como um de seus principais objetivos otimizar processos de ordenação principalmente em estruturas de dados com quantidade de valores grande. Ele

funciona com base na definição de um ”gap” entre elementos do vetor o fragmentando em subconjuntos, ao invés de uma comparação sequencial como o que ocorre com o Bubble Sort. Cada subconjunto é ordenado e a iteração repete até que os elementos sejam todos ordenados. Shell Sort foi escolhido pela sua fácil implementação, entretando seus resultados surpreenderam com relação aos outros algoritmos. Sua rápida capacidade de ordenação permite que vetores muito grandes sejam ordenados por uma fração do tempo que levaria para o Bubble Sort. Abaixo se encontra os dados coletados das operações de Shell Sort, e como é possível observar, a quantidade de trocas e iterações é incrivelmente menor que o algoritmo de Bubble Sort, já que a ordenação de subconjuntos evita que cada elemento busque sua posição no vetor individuamente percorrendo-o e realizando comparações a cada movimento. Para o tamanho 50, Shell Sort levou apenas 0.01 ms para a ordenação, Bubble Sort levou cerca de 0.08 ms, para o tamanho 10000, Shell Sort levou cerca de 0.5 ms enquanto o Bubble Sort levou cerca de 83 ms.

Table 2. Relatório Shell Sort

	A	B	C	D
1	TAMANHO	ITERAÇÕES	TROCAS	MÉDIA DE TEMPO DE ORDENAÇÃO
2	50	203	3	0.0105 ms
3	500	3506	453	0.25444 ms
4	1000	8006	711	0.31917998 ms
5	5000	55005	3786	0.52108 ms
6	10000	120005	5441	0.59524 ms
7				

3. Heap Sort

Heap Sort é um agoritmo de ordenação que envolve a construção de uma árvore binária chamada ”heap” que irá representar os dados de um conjunto de dados, no caso deste trabalho, um vetor. A árvore heap deverá ter seu nó raiz como sendo o maior valor do vetor, e cada nó pai sendo maior que seus filhos. Essa construção é realizada pela função ”heapify”, que irá construir essa árvore de maneira recursiva levando em conta a comparação entre os valores. Após a conclusão da árvore, cada elemento é extraído de forma que os valores serão preenchidos do final ao começo do vetor, percorrendo do nó raiz, para o pai, e depois ao filhos, a cada iteração o heap é reduzido já que os elementos inseridos novamente no vetor serão desconsiderados da árvore. O algoritmo de Heap Sort foi escolhido já que sua implementação aparentou ser mais fácil do que o Merge Sort, entretanto seus resultados fazeram valer a pena já que sua análise demonstrou tempos de execução e trocas decentes, mesmo que não tão eficientes quanto aos do Shell Sort, ainda se destacam em relação à algoritmos como o Bubble Sort. Levou apenas 1.4 ms para ordenar um vetor de tamanho 10000 enquanto o Bubble Sort levou cerca de 83 ms, 0.9 ms para ordenar um vetor de 5000, enquanto o Bubble Sort levou cerca de 20 ms. Abaixo se encontra a tabela com os resultados obtidos.

Table 3. Relatório Heap Sort

	A	B	C	D
1	TAMANHO	ITERAÇÕES	TROCAS	MÉDIA DE TEMPO DE ORDENAÇÃO
2	50	74	73	0.0507802 ms
3	500	749	748	0.13738 ms
4	1000	1499	1497	0.1554602 ms
5	5000	7499	7498	0.9381 ms
6	10000	14999	14998	1.4102602 ms
7				