Universidade Federal do Maranhão
São Luís, 28 de novembro de 2021
Alunos: Matheus Levy de Lima Bessa; Italo Luigi Cerqueira Dovera
Disciplina: DEIN0083 - ESTRUTURA DE DADOS II (CP) (2021.2 - T01)
Professor: João Dallyson Sousa de Almeida

1. Testes

Dado 1						
			Ordem Crescente		Ordem Decrescente	
Método	Repetições	Tamanho	Operações	Tempo(ms)	Operações	Tempo(ms)
MergeSort Padrão	100	1.000	24.947	2,510 ms +-(2,359)	24.947	2,220 ms +-(2,232)
		10.000	7.876.168	73,260 ms +-(15,247)	7.876.168	57,450 ms +-(3,073)
	15	100.000	3.837.851	3980,667 ms +-(288,407)	3.837.851	3505,467 ms +-(22,535)
MergeSort (Otimizado)	100	1.000	9.107	1,220 ms +-(0,871)	8.933	1,270 ms +-(0,908)
	15	100.000	1.580.651	181,200 ms +-(39,699)	1.580.373	140,400 ms +-(28,643)
	20	1.000.000	19.300.892	1996,100 ms +-(187,167)	867.974.104	1686,900 ms +-(125,414)
QuickSort (com InsertSort)	100	1.000	11.962	1,570 ms +-(0,977)	13.473	2,130 ms +-(1,228)
	15	100.000	2.293.903	279,533 ms +-(82,426)	2.350.845	186,800 ms +-(31,623)
	20	1.000.000	28.558.073	2949,000 ms +-(509,130)	1.281.385.735	2330,400 ms +-(119,014)
HeapSort	100	1.000	8.066	1,610 ms +-(0,909)	8.062	2,490 ms +-(2,580)
	15	100.000	1.474.907	246,533 ms +-(69,484)	1.475.061	241,000 ms +-(62,859)
	20	1.000.00	18.047.262	3758,450 ms +-(346,180)	18.047.912	3410,000 ms +-(226,121)
	100	1.000	10.912	0,800 ms +-(0,616)	10.912	0,800 ms +-(0,523)

	15	100.000	2.016.728	197,750 ms +-(62,466)	2.016.728	157,150 ms +-(31,743)
TreeSort	20	1.000.000	24.780.931	2788,600 ms +-(199,544)	24.780.931	2907,500 ms +-(202,370)
	100	1.000	-	1,690 ms +-(0,861)	-	1,620 ms +-(0,801)
	15	100.000	-	219,600 ms +-(83,779)	-	153,067 ms +-(33,527)
JDK MergeSort	20	1.000.000	-	2044,550 ms +-(121,225)	-	1817,400 ms +-(176,006)
	100	1.000	-	0,760 ms +-(0,653)	-	1,260 ms +-(0,661)
	15	100.000	-	175,067 ms +-(35,121)	-	157,200 ms +-(38,006)
JDK QuickSort	20	1.000.000	-	2173,500 ms +-(272,855)	-	1703,100 ms +-(107,916)

Dado 2						
			Ordem Crescente		Ordem Decrescente	
Método	Repetições	Tamanho	Operações	Tempo(ms)	Operações	Tempo(ms)
		1.000	24.947	1,490 ms +-(1,078)	24.947	0,930 ms +-(0,573)
	100	10.000	7.876.168	59,380 ms +-(7,794)	7.876.168	53,890 ms +-(3,712
MergeSort Padrão	15	100.000	3.837.851	3834,733 ms +-(258,010)	3.837.851	3289,467 ms +-(94,979)
	100	1.000	9.107	0,480 ms +-(0,785)	8.933	0,380 ms +-(0,546)
	15	100.000	1.580.651	53,533 ms +-(12,794)	1.580.373	45,933 ms +-(4,334)
MergeSort (Otimizado)	20	1.000.000	19.300.892	892,150 ms +-(379,513)	867.974.104	1686,900 ms +-(125,414)
QuickSort (com InsertSort)	100	1.000	11.962	0,420 ms +-(0,554)	13.473	0,890 ms +-(0,803)

	15	100.000	2.293.903	53,200 ms +-(6,316)	2.350.845	48,467 ms +-(6,151)
	20	1.000.000	28.558.073	1023,400 ms +-(295,612)	1.281.385.735	922,150 ms +-(160,458)
HeapSort	100	1.000	8.066	0,650 ms +-(0,796)	8.062	1,050 ms +-(1,546)
	15	100.000	1.474.907	83,600 ms +-(14,382)	1.475.061	77,733 ms +-(13,833)
	20	1.000.00	18.047.262	1383,650 ms +-(108,396)	18.047.912	1265,200 ms +-(57,361)
TreeSort	100	1.000	10.912	0,550 ms +-(0,510)	10.912	0,500 ms +-(0,513)
	15	100.000	2.016.728	74,600 ms +-(19,367)	2.016.728	69,550 ms +-(17,325)
	20	1.000.000	24.780.931	1287,700 ms +-(17,664)	24.780.931	1343,150 ms +-(100,480)
JDK MergeSort	100	1.000	-	1,570 ms +-(1,066)	-	1,380 ms +-(0,801)
	15	100.000	-	65,200 ms +-(17,387)	-	56,600 ms +-(7,308)
	20	1.000.000	-	939,400 ms +-(151,458)	-	919,850 ms +-(142,276)
JDK QuickSort	100	1.000	-	0,360 ms +-(0,482)	-	1,260 ms +-(0,661)
	15	100.000	-	70,333 ms +-(10,655)	-	56,600 ms +-(7,308)
	20	1.000.000	-	1220,850 ms +-(357,810)	-	840,650 ms +-(66,486)

Dado 3						
			Ordem Crescente		Ordem Decrescente	
Método	Repetições	Tamanho	Operações	Tempo(ms)	Operações	Tempo(ms)
	100	1.000	24.947	1,700 ms +-(1,534)	24.947	1,250 ms +-(0,730)
		10.000	7.876.168	54,820 ms +-(4,439)	7.876.168	51,320 ms +-(2,274)

MergeSort Padrão	15	100.000	3.837.851	3883,067 ms +-(188,575)	3.837.851	3296,200 ms +-(68,900)
MergeSort (Otimizado)	100	1.000	9.107	0,480 ms +-(0,522)	8.933	0,510 ms +-(0,522)
	15	100.000	1.580.651	78,600 ms +-(19,478)	1.580.373	65,000 ms +-(15,579)
	20	1.000.000	19.300.892	1021,000 ms +-(326,333)	867.974.104	791,650 ms +-(64,113)
QuickSort (com InsertSort)	100	1.000	11.962	0,350 ms +-(0,520)	13.473	0,290 ms +-(0,518)
	15	100.000	2.293.903	95,867 ms +-(28,012)	2.350.845	71,533 ms +-(15,376)
	20	1.000.000	28.558.073	946,450 ms +-(187,641)	1.281.385.735	844,100 ms +-(59,877)
HeapSort	100	1.000	8.066	0,280 ms +-(0,514)	8.062	0,420 ms +-(0,572)
	15	100.000	1.474.907	107,667 ms +-(26,375)	1.475.061	120,933 ms +-(27,889)
	20	1.000.00	18.047.262	1717,150 ms +-(138,258)	18.047.912	1649,200 ms +-(114,441)
TreeSort	100	1.000	10.912	0,450 ms +-(0,510)	10.912	0,250 ms +-(0,444)
	15	100.000	2.016.728	96,950 ms +-(20,920)	2.016.728	90,800 ms +-(23,761
	20	1.000.000	24.780.931	1516,250 ms +-(171,117)	24.780.931	1378,400 ms +-(59,657)
JDK MergeSort	100	1.000	-	1,040 ms +-(1,063)	-	1,400 ms +-(1,255)
	15	100.000	-	78,867 ms +-(18,788)	-	90,067 ms +-(24,458)
	20	1.000.000	-	944,900 ms +-(83,758)	-	857,000 ms +-(106,075)
	100	1.000	-	0,590 ms +-(0,570)	-	0,750 ms +-(1,381)
	15	100.000	-	89,733 ms +-(16,377)	-	79,200 ms +-(15,566)

JDK QuickSort	20	1.000.000	-	981,750 ms +-(84,739)	-	820,700 ms +-(60,749)

2. Conclusão

Após o levantamento da complexidade e desempenho dos algoritmos de ordenação pode-se então analisar a eficiência e particularidades dos mesmos com inferência dos dados retirados.

Os dados criados para compor as arrays a serem testadas foram instanciadas em ordem completamente aleatória, sendo assim impossível testar os melhores e piores casos de cada algoritmo. Portanto, todas as inferências realizadas são pela complexidade de tempo médio do algoritmo.

A princípio, é possível perceber que os resultados obtidos conferem com a complexidade assintótica de suas respectivas funções. Por exemplo, para o algoritmo HeapSort, a quantidade de operações realizadas é respectivamente 8.066, 1.474.907 e 18.047.262, enquanto, sendo a função $O(n\,log(n))$, a quantidade de operações previsto para cada tamanho é dado por:

- Pequeno: $n\,log(n) = 1000 * log(1000) = 1000 * 3 = 3000$;
- Médio: $n\,log(n) = 100000 * log(100000) = 100000 * 5 = 500000$;
- Grande: $n\,log(n) = 1000000 * log(1000000) = 1000000 * 6 = 6000000$;

logo, é possível entender que os resultados encontrados nos testes podem ser definidos por $k * n * log(n)$, k representando uma constante. Verificando que k varia de 3 a 4, o valor previsto para os três tamanhos de dados, quando multiplicados por k, se aproximam dos valores obtidos.

Acerca dos dados, estes possuem uma chave e um valor. A ordenação se dá através das chaves, sendo o valor apenas um peso de memória. Os dados estão disposta da seguinte forma:

- Dado1: String(chave), double(valor)
- Dado2: double(chave), string(valor)
- Dado3: double(chave), int(valor)

A influência do tipo de dado se demonstra através do aumento do tempo de execução. Por exemplo, no MergeSort Padrão temos 3505,467 ms +-(22,535) para o Dado1 e 3289,467 ms +-(94,979) para Dado2. Isso se dá pois o tipo de dado Dado1 é o mais complexo, por sua chave se tratar de uma string, enquanto os outros dados possuem inteiro ou double como chave, sendo assim mais fácil de se comparar duas chaves. As diferenças entre o Dado2 e o Dado3 não são tão relevantes, pois ambos apresentam uma chave do tipo double, portanto, teremos apenas algumas flutuações pequenas nos tempos de execução. O principal desequilíbrio observado foi sobre o Dado1 em relação ao Dado2 e Dado3.

O MergeSort foi dividido em dois métodos: MergeSortPadrão e MergeSort(Otimizado). Sendo o MergeSortPadrão a versão original do método de ordenação e o MergeSort(Otimizado), como o nome sugere, utilizando algumas técnicas para melhorar o tempo de execução do método. Entre essas técnicas temos:

1. A utilização de InsertSort para subvetores com tamanho menor do que 15.

2. A utilização de uma checagem para verificar se os subvetores já estão ordenados. Caso os vetores já estejam ordenados, ele copia o vetor para o auxiliar, caso contrário executa o merge normalmente. Essa verificação é feita da seguinte forma: podemos reduzir o tempo de execução para ser linear para vetores que já estão ordenados, adicionando um teste para pular a chamada para o método merge() se A[meio] é menor ou igual a A[meio+1]. Com essa mudança, ainda fazemos todas as chamadas recursivas, mas o tempo de execução para qualquer subarray ordenado é linear.
3. Podemos utilizar um truque recursivo, para alternar o vetor ordenado entre o vetor principal e o vetor auxiliar, assim embora não economizemos memória economizamos o tempo de copiar do vetor auxiliar para o vetor principal. O que para vetores grandes é uma grande economia. Isto é feito da seguinte forma: Para fazer isso, fazemos duas chamadas ao método de ordenação: uma obtém sua entrada do array fornecido e coloca a saída ordenada no array auxiliar; a outra obtém sua entrada do vetor auxiliar e coloca a saída ordenada no vetor fornecido. Com essa abordagem, em um pouco de truque recursivo, podemos organizar as chamadas recursivas de modo que a computação troque os papéis da matriz de entrada e da matriz auxiliar em cada nível.

Com essas otimizações podemos obter resultados melhores até mesmo que a função Collections.sort() do java que implementa uma versão de TimSort o Arrays.sort() que implementa uma versão de QuickSort(). Como demonstrado na tabela de testes. Para o Dado1 temos 1686,900 ms +-(125,414) para o MergeSort(Otimizado) com 10^6 elementos contra 1817,400 ms +-(176,006) com o Collections.sort(). Apresentando um tempo menor e uma variância menor.

Acerca da ordem crescente e decrescente essa base de dados obtidas do csv demonstra uma facilidade para ser ordenada de forma crescente. Isso se mostra através da comparação dos tempos e operações nos dois casos. Apesar de a base ser gerada aleatoriamente por um script em python, é utilizada essa única base. Portanto, temos que o tempo de execução vai variar de acordo com as bases, pois assim teremos flutuações no tempo médio de ordenação ao variar a base, com tanto que a base não seja nem o pior nem o melhor caso.

Também é possível notar uma diferença de performance entre os métodos providenciados pela JDK do Java e seus respectivos pares. Para o MergeSort, temos que o desempenho geral no caso médio, considerando a quantidade operações necessárias para ordenar os dados, é:

MergeSort Padrão > MergeSort do JDK > MergeSort (Otimizado),

onde o primeiro é o menos otimizado para tratar os dados recebidos e o último o mais otimizado. Isso ocorre devido às diferenças de implementação dos três métodos. Para o QuickSort, tem-se que aquele fornecido pela JDK tem desempenho melhor que o desenvolvido na atividade.

As principais dificuldades encontradas durante o desenvolvimento foram encontrar uma metodologia para obter os tempos de execução e o número de operações. Uma vez que os tempos de execução flutuam muito quando feitos isoladamente devido as

configurações da máquina que está rodando e o escalonamento de processos do sistema operacional, foi instituído, para minimizar as flutuações dos tempos, um número de repetições do método. Assim, um método ordenava um vetor e depois o vetor sobrescrito e feita novamente a ordenação. A cada ordenação se capturava o tempo de execução e se alocava em um vetor com todos os tempos de execução resultantes. Ao final se tirava a média aritmética dos tempos e a variância dos tempos. Assim pode-se obter resultado menos enviesado.
A maior parte dos códigos já haviam sido desenvolvidos e constavam no github (https://github.com/MatheusLevy/MergeSort). Dado isso, foi necessário apenas pequenas alterações e desenvolvimento dos métodos de TreeSort e HeapSort.

3. Referências

Algoritmos de Ordenação em Java. Devmedia. Disponível em: <https://www.devmedia.com.br/algoritmos-de-ordenacao-em-java/32693>. Acesso em: 26 de nov. de 2021.

Arrays.sort() in Java with examples. Geeks for Geeks, 2021. Disponível em: <https://www.geeksforgeeks.org/arrays-sort-in-java-with-examples/>. Acesso em: 26 de nov. de 2021

Collections.sort() in Java with Examples. Geeks for Geeks, 2018. Disponível em: <https://www.geeksforgeeks.org/collections-sort-java-examples/>. Acesso em: 26 de nov. de 2021

HeapSort. Geeks for Geeks, 2021. Disponível em: <https://www.geeksforgeeks.org/heap-sort/>. Acesso em: 26 de nov. de 2021.

Merge Sort in Java. Baeldung, 2020. Disponível em: <https://www.baeldung.com/java-merge-sort>. Acesso em: 26 de nov. de 2021.

Merge Sort. Geeks for Geeks, 2021. Disponível em: <https://www.geeksforgeeks.org/merge-sort/>. Acesso em: 26 de nov. de 2021.

Ordenação de Dados - Quick Sort. Universidade de Java. Disponível em: <http://www.universidadejava.com.br/pesquisa_ordenacao/quick-sort/>. Acesso em: 26 de nov. de 2021.

Tree Sort. Geeks for Geeks, 2021. Disponível em: <https://www.geeksforgeeks.org/tree-sort/>. Acesso em: 26 de nov. de 2021.
.
Understanding merge sort optimization: avoiding copies. Stack Overflow, 2011. Disponível em: <https://stackoverflow.com/questions/7577825/understanding-merge-sort-optimization-avoiding-copies>. Acesso em: 26 de nov. de 2021.