Exercício 1 – Levantamento dos conhecimentos prévios dos alunos em relação a programação e testes

Instruções

- Individual
- Data limite: 09.ago.2024 (entrega e debate)
- Fazer um programa que leia 1000 números e exiba em ordem crescente a sequência digitada
- Descrever como fazer para testar para o programa gerado

Vamos testar a eficiência de 11 métodos de ordenação:

1. Selection Sort
2. Bubble Sort
3. Merge Sort
4. Heap Sort
5. Insertion Sort
6. Counting Sort
7. Radix Sort
8. Bucket Sort
9. Quick Sort
10. Shell Sort
11. Dual Pivot Quicksort

Tabela de Complexidade Temporal e Espacial

	Complexidade Temporal			**Complexidade Espacial**
	Pior	**Médio**	**Melhor**	
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
Merge Sort	$O(n\ log(n)$	$O(n\ log(n))$	$O(n\ log(n))$	$O(n)$
Heap Sort	$O(n\ log(n)$	$O(n\ log(n))$	$O(n\ log(n))$	$O(1)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(n + k)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n + k)$
Bucket Sort	$O(n^2)$	$O(n + k)$	$O(n + k)$	$O(n + k)$
Quick Sort	$O(n^2)$	$O(n\ log(n))$	$O(n\ log(n))$	$O(n\ log(n))$
Shell Sort	$O(n^2)$	$O(n\ log(n))$	$O(n\ log(n))$	$O(1)$
Dual Pivot Quicksort	$O(n^2)$	$O(n\ log(n))$	$O(n\ log(n))$	$O(log(n))$

Teste:

Registrar o runtime de ordenação crescente dos 11 métodos nas 10 situações abaixo

1. Vetor com 10 números inteiros entre 0 e 10 dispostos aleatoriamente
2. Vetor com 100 números inteiros entre 0 e 100 dispostos aleatoriamente
3. Vetor com 1.000 números inteiros entre 0 e 1000 dispostos aleatoriamente
4. Vetor com 100.000 números inteiros entre 0 e 100.000 dispostos aleatoriamente
5. Vetor com 250.000 números inteiros entre 0 e 250.000 dispostos aleatoriamente

6. Vetor com 10 números inteiros entre 0 e 10 dispostos em ordem decrescente
7. Vetor com 100 números inteiros entre 0 e 100 dispostos em ordem decrescente
8. Vetor com 1.000 números inteiros entre 0 e 1.000 dispostos em ordem decrescente
9. Vetor com 10.000 números inteiros entre 0 e 10.000 dispostos em ordem decrescente
10. Vetor com 50.000 número inteiros entre 0 e 50.000 dispostos em ordem decrescente

Método de testes:

1. **Criação do vetor:** Um vetor de inteiros de tamanho X é criado com valores aleatórios entre 0 e X usando a função `public static int[] generateRandom(int qnt, int min, int max)`

2. **Cópias do vetor:** 11 cópias do vetor original são feitas para serem usadas por cada algoritmo de ordenação.

3. **Medição do tempo:** Para cada algoritmo de ordenação (selectionSort, bubbleSort, mergeSort, heapSort, insertionSort, countingSort, radixSort, bucketSort, quickSort, shellSort e dualPivotQuickSort), o tempo de execução é medido usando System.nanoTime(). O tempo inicial é registrado antes da execução do algoritmo e o tempo final é registrado após a conclusão.

4. **Repetição:** O processo de criação do vetor, cópia e medição do tempo é repetido ciclos vezes para obter resultados mais confiáveis.

5. **Cálculo da média:** A média do tempo de execução para cada algoritmo é calculada somando os tempos de execução de todas as repetições e dividindo pelo número de repetições.

6. **Exibição dos resultados:** Os tempos médios de execução para cada algoritmo são impressos no console.

Resultados

Tabela de Runtime com vetor de inteiros dispostos aleatoriamente

	V[10]	**V[100]**	**V[1.000]**	**v[10.000]**	**V[100.000]**	**V[250.000]**
Nº Ciclos	100.000.000	10.000.000	1.000.000	10.000	100	5
Tempo Total Ciclos	458,973 s	583,059 s	1.293,218 s	1.668,32 s	1.900,583 s	635.2551 s
Tempo Total	1.507,9683 s	700,971 s	1.314,683 s	1.670,01 s	1.900,818 s	635.3343 s
SelectionSort	197 ns	6.809 ns	211.862 ns	27,4855 ns	2.37895 ms	21,8617 s
Bubble Sort	300 ns	16.387 ns	696.283 ns	119,7336 ms	15,8173 s	97,9149 s
Merge Sort	653 ns	7.131 ns	71.776 ns	1,2821 ms	14,7968 ms	57, 6676 ms
Heap Sort	354 ns	5.345 ns	60.154 ns	1,1046 ms	13,088 ms	45,7802 ns
Insertion Sort	191 ns	2.040 ns	50.778 ns	7,3226 ms	735,6806 ms	6, 9741 s
Counting Sort	202 ns	515 ns	3.376 ns	75.954 ns	1,0929 ms	10, 0850 ms
Radix Sort	556 ns	2.969 ns	23.136 ns	560.229 ns	5,4913 ms	41,1678 ns
Bucket Sort	938 ns	5.552 ns	39.759 ns	669.375 ns	8,37653 ms	49,9767 ms
Quick Sort	347 ns	4.071 ns	45.327 ns	787.152 ns	8,796 ms	27,8047 ms
Shell Sort	251 ns	4.005 ns	54.636 ns	1,15341 ms	15,0184 ms	39, 6974 ms
Dual Pivot Quicksort	207 ns	3.094 ns	35.588 ns	617,169 ns	7,1469 ms	27, 99634 ms

Tabela de Runtime com vetor de inteiros dispostos decrescentemente

	V[10]	**V[100]**	**V[1.000]**	**V[10.000]**	**V[20.000]**	**V[22.200]**
Nº Ciclos	100.000.000	10.000.000	100.000	5.000	1.000	1.000
Tempo Total Ciclos	446,694 s	484,8831 s	409,6592 s	1.120,1162 s	1.178,3097 s	1,866.15361
Tempo Total	1.627, 498 s	708,6123 s	435,8564 s	1.129,169 s	1.184,6029 s	1,874.52
SelectionSort	226 ns	11257 ns	1.92134 ms	116,9446 ms	652,2181 ms	998,4243 ms
Bubble Sort	214 ns	7526 ns	1,05025 ms	65,4296 ms	245,1442 ms	513,4293 ms
Merge Sort	672 ns	5703 ns	100064 ns	587030 ns	1,852885 ms	2413825 ns
Heap Sort	240 ns	3150 ns	104487 ns	718886 ns	2,2723 ms	2983307 ns
Insertion Sort	183 ns	3263 ns	262033 ns	13, 2784 ms	94,4468 ms	121,5806 ms
Counting Sort	219 ns	661 ns	9642 ns	62403 ns	173604 ns	225781 ns
Radix Sort	589 ns	3735 ns	63923 ns	460580 ns	1,8598 ms	2015684 ns
Bucket Sort	1047 ns	6275 ns	98138 ns	554917 ns	1729334 ns	2,1991 ms
Quick Sort	245 ns	4797 ns	447010 ns	25,6948 ms	177,6213 ms	221.79595 ms
Shell Sort	167 ns	1413 ns	33687 ns	243778 ns	883360 ns	955233 ns
Dual Pivot Quicksort	218 ns	251 ns	2546 ns	12927 ns	38544 ns	46094 ns

Conclusões

Cenário 1: Vetor de Inteiros Aleatórios:

1. **Algoritmos com Desempenho Consistente:** Merge Sort, Heap Sort, Radix Sort e Counting Sort demonstram um desempenho consistente e eficiente, com tempos de execução que crescem de forma relativamente suave à medida que o tamanho do vetor aumenta.

2. **Algoritmos com Desempenho Variável:** Quick Sort e Dual Pivot Quicksort apresentam tempos de execução geralmente bons, mas podem ser afetados por casos desfavoráveis, como dados quase ordenados. Shell Sort também se sai bem, mas seu desempenho pode variar dependendo da sequência de gaps utilizada.

3. **Algoritmos com Desempenho Limitado:** Selection Sort e Bubble Sort são significativamente mais lentos que os outros algoritmos, especialmente para vetores grandes. Insertion Sort se sai melhor para vetores pequenos, mas seu desempenho degrada rapidamente à medida que o tamanho do vetor aumenta.

4. **Bucket Sort:** O desempenho do Bucket Sort depende da distribuição dos dados. Se os dados estiverem uniformemente distribuídos, ele pode ser muito eficiente.

Cenário 2: Vetor de Inteiros Decrescente:

1. **Impacto da Ordem Inicial:** A ordem inicial dos dados afeta significativamente o desempenho de alguns algoritmos. Selection Sort, Bubble Sort e Insertion Sort são particularmente afetados, pois seus tempos de execução aumentam drasticamente em relação ao cenário com dados aleatórios.

2. **Robustez de Merge Sort e Heap Sort:** Merge Sort e Heap Sort demonstram robustez em relação à ordem inicial dos dados, mantendo tempos de execução consistentes em ambos os cenários.

3. **Quick Sort e Dual Pivot Quicksort:** Quick Sort e Dual Pivot Quicksort são mais suscetíveis a casos desfavoráveis quando os dados estão ordenados em ordem decrescente.

Considerações:

1. **Complexidade Assintótica:** Os resultados dos testes confirmam as complexidades assintóticas teóricas dos algoritmos. Algoritmos com complexidade O(n log n), como Merge Sort, Heap Sort, Quick Sort e Dual Pivot Quicksort, geralmente superam algoritmos com complexidade $O(n^2)$, como Selection Sort, Bubble Sort e Insertion Sort, para vetores grandes.

2. **Fatores Adicionais:** O desempenho real dos algoritmos pode ser influenciado por fatores como a implementação específica, o hardware utilizado e a natureza dos dados.

Recomendações:

1. Para vetores grandes e com ordem inicial desconhecida, Merge Sort, Heap Sort ou Quick Sort são boas opções.

2. Para vetores pequenos ou quase ordenados, Insertion Sort pode ser eficiente.

3. Se os dados possuem um range limitado, Counting Sort ou Radix Sort podem ser muito rápidos.

4. Bucket Sort é útil quando os dados estão uniformemente distribuídos.

Cenários de testes futuros:

1. **Vetor com Muitos Valores Repetidos:**
    - **Objetivo:** Avaliar como os algoritmos lidam com vetores que possuem muitos valores repetidos.

    - **Cenário:** Crie um vetor onde um número limitado de valores se repete muitas vezes. Esse tipo de distribuição pode impactar a eficiência de alguns algoritmos, especialmente aqueles que dependem de comparações.

2. **Vetor Quase Ordenado (Com Pequenas Permutações):**
    - **Objetivo:** Verificar o desempenho de algoritmos quando o vetor está quase ordenado, com apenas alguns elementos fora de ordem.

    - **Cenário:** Um vetor que está 95% ordenado, com pequenos swaps. Isso é útil para avaliar algoritmos como Insertion Sort, que tende a ser muito eficiente em vetores quase ordenados.

3. **Vetor com Dados Aleatórios em Diferentes Intervalos:**
    - **Objetivo:** Analisar o desempenho em vetores onde os valores têm uma grande variedade de intervalos, como grandes números misturados com números pequenos.

    - **Cenário:** Um vetor com números que variam, por exemplo, de 0 a 10^9, pode ajudar a entender como algoritmos como Bucket Sort e Counting Sort se comportam.

4. **Vetor com Número de Elementos Não Potência de 2:**
    - **Objetivo:** Testar o impacto do tamanho do vetor nos algoritmos, especialmente em casos onde o tamanho não é uma potência de 2, o que pode afetar o desempenho de alguns algoritmos como Radix Sort.

    - **Cenário:** Vetores com tamanhos como 3, 7, 15, 31, etc.

5. **Vetor com Ordem Alternada (Zig-Zag):**
    - **Objetivo:** Avaliar como os algoritmos se comportam quando os dados têm uma ordem alternada (e.g., [1, 3, 2, 4, 3, 5]).

    - **Cenário:** Este tipo de arranjo pode ser desafiador para algoritmos que dependem de comparações locais.

6. **Vetor com Dados Distribuídos em um Padrão Específico (e.g., Gaussiano):**
    - **Objetivo:** Testar o desempenho com distribuições não uniformes, como a distribuição normal (Gaussiana).

    - **Cenário:** Vetores com valores distribuídos de acordo com uma curva normal podem revelar como os algoritmos lidam com aglomerações de valores em uma faixa específica.