Universidade Federal do Pampa | Organização e Arquivos de Dados – 2023/ 01

Manoela Resende Gomes da Cunha | 2210100235

29/05/2023

RELATÓRIO: AVALIAÇÃO COMPARAÇÃO DE ALGORITMOS SORT

---

1. OBJETIVO E FUNCIONALIDADE:

O objetivo deste código é realizar uma comparação entre diferentes algoritmos de ordenação, avaliando o número de comparações realizadas por cada algoritmo em diferentes tamanhos de vetores. A funcionalidade principal do código é calcular e registrar o número de comparações feitas pelos algoritmos em arquivos de saída, permitindo assim uma análise comparativa do desempenho dos algoritmos.O código implementa os seguintes algoritmos:

- Grupo 1 (complexidade $O(n^2)$):
  - Bubble Sort
  - Insertion Sort
  - Selection Sort
- Grupo 2 (complexidade $O(n \log n)$):
  - Merge Sort
  - Quick Sort
  - Heap Sort

Além desses algoritmos, o código também implementa o algoritmo Radix Sort, que possui complexidade $O(w + n)$, onde w é o número de dígitos dos elementos e n é o tamanho do vetor.

2. FUNÇÕES UTILIZADAS:

O código fornecido contém várias funções para implementar algoritmos de ordenação. As funções incluídas são:

- **vetor_aleatorio():** preenche um vetor com valores aleatórios;
- **maior_elemento():** retorna o maior elemento de um vetor;

- **bubbleSort(), insertionSort(), selectionSort(), mergeSort(), quickSort(), heapSort(), radixSort(), countingSort():** ordenação;

3. EXECUÇÃO:

Primeiramente, é aberto o arquivo "dados_comparacoes.dat" em modo de escrita, utilizando a função ‘fopen’. Esse arquivo será usado para armazenar os resultados das comparações dos algoritmos de ordenação. Logo, é iniciado um loop para comparar os algoritmos de ordenação, que percorre diferentes tamanhos de vetores. Para cada tamanho de vetor, é criado um vetor de tamanho correspondente, e é preenchido com valores aleatórios utilizando a função ‘vetor_aleatorio()’.

Para cada tamanho de vetor, são chamadas as funções dos algoritmos de ordenação (bubbleSort(), insertionSort(), selectionSort(), mergeSort(), quickSort(), heapSort(), radixSort()) e a variável ‘comparacoes’ é atualizada com o número de comparações realizadas por cada algoritmo. Os resultados das comparações são escritos no arquivo "dados_comparacoes.dat" utilizando a função ‘fprintf’. O tamanho do vetor e o número de comparações de cada algoritmo são escritos em cada linha do arquivo.

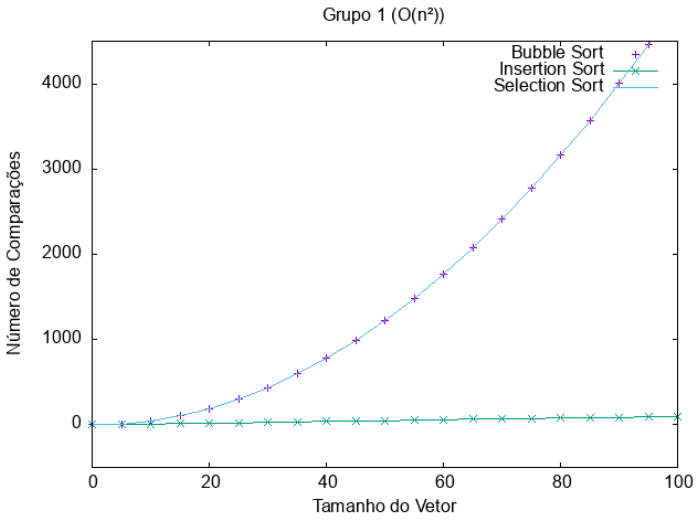
Para facilitar a compreensão, foram abertos arquivos independentes com os grupos 1 e 2, e o radix, seguindo os passos anteriormente mencionados, para retornar as comparações dos algoritmos de cada grupo nos arquivos ‘dados_grupo1.dat’, ‘dados_.grupo2.dat’ e ‘dados_radix.dat’.

4. ANÁLISE DOS GRÁFICOS:

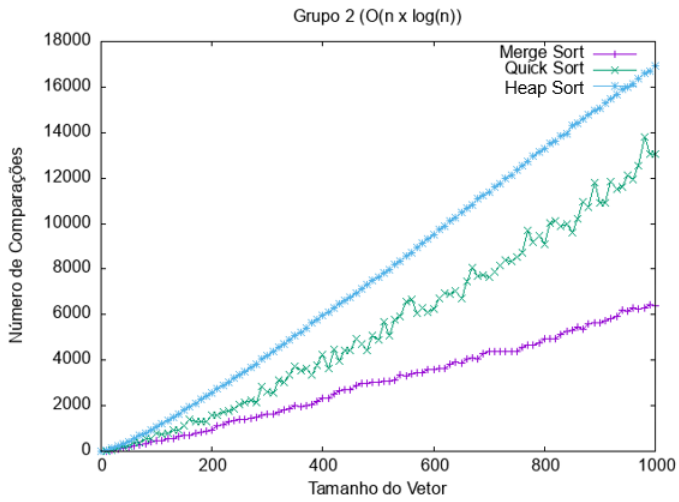
O eixo x representa o tamanho do conjunto de dados, enquanto o eixo y representa o número de comparações o para ordenar o conjunto usando cada algoritmo. Os gráficos, em geral, exibem uma curva ascendente à medida que o tamanho do conjunto aumenta para cada algoritmo.

4.1.GRUPO 1 (O(n²)):

Bubble Sort é o menos eficiente dos três, e o gráfico refletirá isso com uma curva acentuada de crescimento. O número de comparações aumenta rapidamente à medida que o tamanho do conjunto de dados aumenta. Já o Insertion Sort mostra um gráfico com um crescimento muito mais suave, o número de comparações aumentará à medida que o tamanho do conjunto de dados aumenta, mas em uma taxa bem menor, sendo mais eficiente. E o Selection Sort tem um gráfico com um padrão igual ao do Bubble Sort, com um crescimento acentuado à medida que o tamanho do conjunto de dados aumenta.

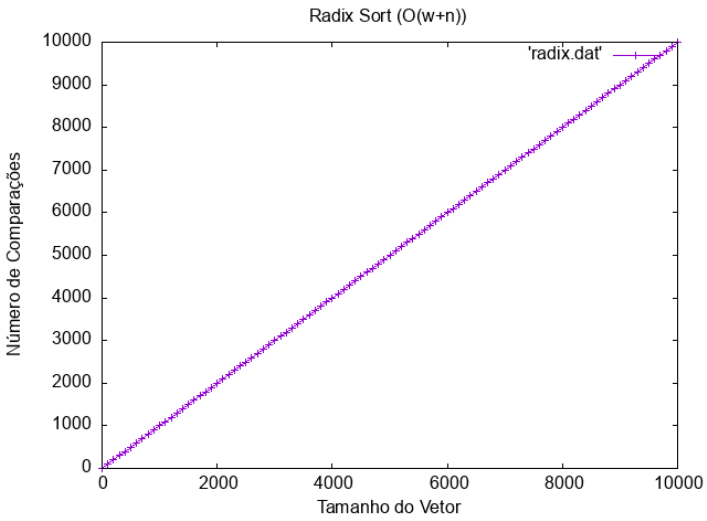


4.2.GRUPO 2 (O(n x log(n)):



Para tamanhos de entrada menores. o Merge Sort e o Quick Sort têm números de comparações. À medida que o tamanho do vetor de entrada aumenta, o Merge Sort mostra um desempenho melhor em comparação com os outros dois algoritmos. O Quick Sort também apresenta um bom desempenho, mas é um pouco mais lento que o Merge Sort.

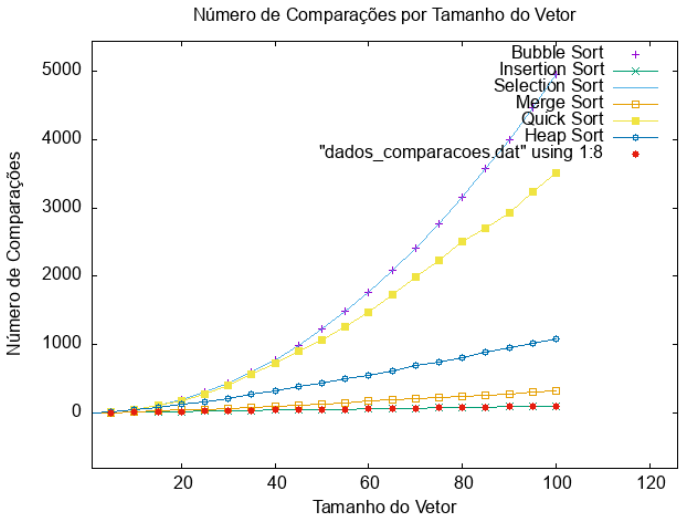
4.3.RADIX SORT (O(w+n)):



Podemos observar que o número de comparações aumenta linearmente com o tamanho do vetor de entrada. Essa é uma característica esperada do Radix Sort, pois possui uma complexidade de tempo linear. Portanto, é esperado que o número de comparações do Radix Sort aumente proporcionalmente ao tamanho do vetor.

4.4.COMPARAÇÃO ENTRE GRUPOS (TOTAL):

É possível observar que os algoritmos do grupo 1 têm um crescimento acentuado no tempo de execução à medida que o tamanho do vetor aumenta. Isso confirma a complexidade quadrática desses algoritmos. Os do grupo 2 mostram um crescimento mais moderado no tempo de execução, o que está de acordo com a complexidade de tempo de O(n log n) desses algoritmos. O Radix Sort (vermelho) apresenta um crescimento linear. Então, podemos concluir que o grupo 1 é formado por algoritmos menos eficientes, especialmente para tamanhos grandes de vetor. Os do grupo 2 são mais eficientes



. O Radix Sort se destaca por ter um desempenho linear, tornando-o uma opção eficiente para ordenar grandes quantidades de dados.