UNIVERSIDADE FEDERAL DE OURO PRETO – UFOP

Daniel Monteiro Valério

Filipe Ramos de Souza Santos

Mateus Pevidor Reis

Relatório

Trabalho prático – Projeto e análise de algoritmos

Ouro Preto – Minas Gerais

Outubro 2021

Daniel Monteiro Valério, Filipe Ramos de Souza Santos, Mateus Pevidor Reis

Relatório
Trabalho prático – Projeto e análise de algoritmos

Relatório apresentado ao professor Anderson Almeida Ferreira, sobre a avaliação empírica de três algoritmos de ordenação, como requisito para obtenção de pontos na disciplina BCC 241 – Projeto e análise de algoritmos.

Ouro Preto – Minas Gerais
Outubro 2021

SUMÁRIO

RESUMO

A ordenação de conjuntos de dados é uma operação recorrente em sistemas computacionais. A visualização e interpretação de largas escalas de dados muitas vezes depende de sua organização, de forma a destacar informações mais relevantes. Neste contexto, foi realizada a implementação de três diferentes algoritmos de ordenação, que foram executados sobre diferentes instâncias de diferentes tamanhos. Essas instâncias foram geradas aleatoriamente, sendo compostas por vetores de valores inteiros, com o objetivo de ordená-los em ordem crescente. Posteriormente, foi realizado o teste t com 95% e 70% de confiança para avaliar o desempenho de cada um dos algoritmos estatisticamente, a fim de compará-los. Por fim, foi realizada a análise de complexidade de cada um dos algoritmos com a finalidade de verificarmos os resultados obtidos por meio da avaliação empírica.

INTRODUÇÃO

Na disciplina de Projeto e análise de algoritmos são estudadas formas de análise de algoritmos com o objetivo de inferir sua complexidade de tempo e espaço. Dessa forma, foi proposta a análise de três algoritmos de ordenação, juntamente da avaliação empírica por meio da execução deles em diferentes instâncias de diferentes tamanhos.

Inicialmente os algoritmos foram implementados utilizando a linguagem de programação C e, em seguida, foram realizadas 20 execuções de cada um dos algoritmos com instâncias de tamanho de 100 a 1.000.000 de elementos, medindo o tempo de execução em cada uma delas. Posteriormente, foi realizado o teste t de Student com 95% de confiança para verificar, estatisticamente, qual dos três algoritmos obteve melhor desempenho.

METODOLOGIA

2.1 Algoritmos:

2.1.1 InsertionSort:

O InsertionSort é um algoritmo que utiliza o método de percorrer todo o vetor, verificando e inserindo corretamente cada valor em sua devida posição. Para realizar o método ele subdivide o vetor em uma parte ordenada e outra não ordenada, de modo que a cada iteração ele pega um valor da parte não ordenada e insere na parte ordenada do vetor até que a parte não ordenada não possua mais nenhum valor.

Como já dito, a execução do programa se dá a partir de um loop externo que vai percorrer todas as posições da parte não ordenada, e um loop interno que vai percorrer e ordenar o vetor da parte ordenada, sendo que, este loop depende inteiramente da posição em que o loop da parte não ordenada se encontra. Dessa forma, o algoritmo pode ser expresso pela seguinte relação:

$$T(n) = n + \sum_{i=1}^{n-1} \quad i \Rightarrow T(n) = n + \frac{(n-1)(n)}{2} \Rightarrow T(n) = \frac{n^2+n}{2} = O(n^2)$$

2.1.2 RadixSort e CountingSort:

2.1.2.1 Counting Sort

Esta é uma técnica de ordenação O(n + k), onde k é o maior elemento do array, e com o pré requisito de que os elementos sejam no máximo do tamanho do vetor, exemplo:

Se temos um vetor com 1000 posições, o tamanho máximo dos elementos ali contidos deve ser 1000. Caso os elementos ultrapassem o tamanho do array, é possível criar um array auxiliar de tamanho k para fazer a ordenação, por isso o algoritmo é O(n + k), depende de qual dos valores é maior.

O CountingSort armazena no vetor auxiliar a frequência de cada elemento, e depois, aloca esses elementos no vetor de saída seguindo três passos básicos:

1. Registra a frequência dos elementos do vetor de entrada em um vetor auxiliar
2. Calcula a soma cumulativa nesse vetor auxiliar, com isso conseguimos saber quantos elementos deve ter antes de cada elemento do vetor.
3. Fazer uma iteração sobre o vetor de entrada, olhando em qual posição deve ficar com base no vetor auxiliar e armazenar no vetor de saída.

2.1.2.2 RadixSort

A ideia do RadixSort é muito parecida com o CountingSort, basicamente a ideia é pegar o maior elemento do vetor, quebrá-lo em k parte, onde k é seu tamanho (123 seria quebrado em 1 2 3). E ir processando um counting sort olhando, a cada iteração do dígito menos significativo para o mais significativo.

Então ele funciona rodando k vezes, o algoritmo de CountingSort. E como vimos o algoritmo de CountingSort é O(n + k). Então seu tempo de resolução é O(nk).

2.1.3 MergeSort:

O MergeSort é um algoritmo que faz uso da técnica de divisão e conquista, reduzindo o problema em subproblemas menores. Para realizar o processo de ordenação, o algoritmo divide o vetor em dois subvetores menores até que eles estejam trivialmente ordenados. Depois de ordenados, cada par de subvetores é transformado em um único vetor ordenado por meio de um processo de intercalação.

A execução do algoritmo se dá por meio da divisão do vetor de tamanho n em duas metades, realizando a chamada recursiva do método para cada uma das metades. Após a execução de cada chamada, é realizado o processo de intercalação das metades. Dessa forma, o algoritmo pode ser expresso pela seguinte relação:

$$T(n) = 2T\left(\frac{n}{2}\right) * (intercalação)$$

O processo de intercalação percorre cada uma das metades, comparando os elementos seguintes de cada subvetor para determinar qual deles é menor. Dessa forma, a complexidade de tempo do processo de intercalação é da ordem de $O(n)$. Portanto, o MergeSort pode ser descrito pela seguinte relação:

$$T(n) = 2T\left(\frac{n}{2}\right) * O(n)$$

Pelo teorema mestre simplificado, se tem que $a = 2, b = 2$ e $d = 1$.

Como $d = \log_b a \rightarrow 1 = \log_2 2 \rightarrow 1 = 1$, então $T(n) = O(n^d \log n) = O(n \log n)$

2.2 Avaliação experimental

2.2.1 Configuração dos experimentos

Todos os experimentos foram realizados no mesmo dispositivo sob as mesmas circunstâncias, provendo o mesmo poder computacional para cada uma das execuções. Cada execução foi realizada separadamente, de forma que instâncias de tamanhos de 100 a 1.000.000 foram ordenadas por cada um dos três algoritmos analisados. Para maior diversidade de dados

de análise, esse procedimento foi repetido 20 vezes com instâncias geradas aleatoriamente no momento de execução. O processo de geração das instâncias não foi contabilizado no tempo de execução.

2.2.2 Métrica de avaliação

Para avaliar os resultados obtidos, foi realizado o teste t de Student com 95% de confiança. Em uma primeira avaliação foi medido o intervalo de confiança individualmente para cada um dos algoritmos analisados, todos os cálculos foram feitos de acordo com o que se encontra no PDF disponibilizado pelo professor. Nesta primeira análise foi verificado que o intervalo de confiança do InsertionSort estava com uma diferença enorme em relação aos outros dois algoritmos, enquanto o MergeSort e o RadixSort mantiveram um intervalo quase que semelhante. Sendo assim, houve uma segunda avaliação levando em consideração apenas os algoritmos MergeSort e RadixSort utilizando o teste pareado entre as duas por possuírem a mesma quantidade de execuções testadas, e mesmo após os cálculos, não é possível determinar que um se sobressai ao outro de modo concreto. Mas se utilizarmos o teste t de Student com 70% de confiança é possível determinar que o MergeSort alcança resultados melhores que o RadixSort.

2.2.3 Resultados

Instância	InsertionSort	RadixSort	MergeSort
100	0,00000s	0,00000s	0,00010s
1.000	0,00085s	0,00005s	0,00115s
10.000	0,05910s	0,00330s	0,01065s
100.000	7,14595s	0,02480s	0,07280s
1.000.000	771,17040s	0,20175s	0,51400s

Tabela 1 - Tempo médio de execução

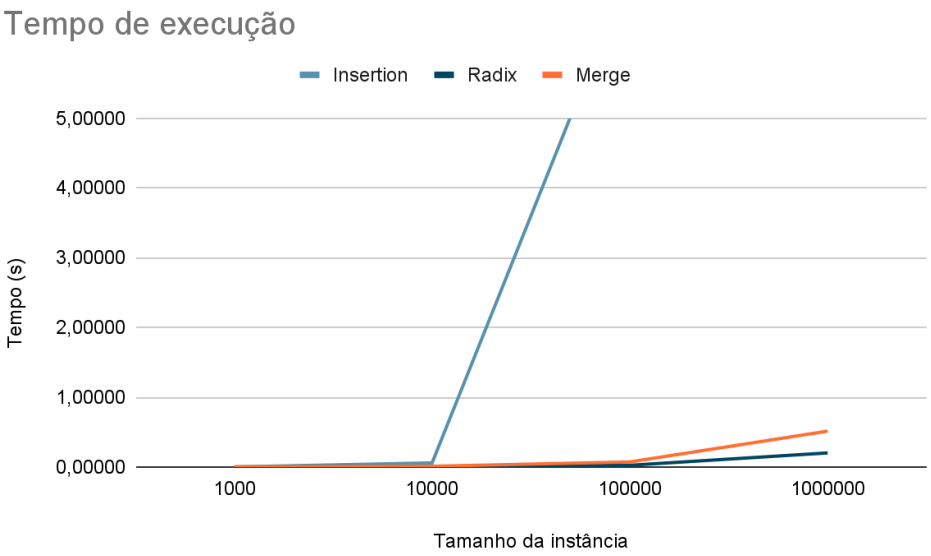

Gráfico 1 - Tempo médio de execução

A partir da análise do gráfico 1, é possível perceber que o InsertionSort é notavelmente pior que os demais algoritmos analisados, visto que para instâncias grandes o tempo de execução é significativamente maior.

Em relação aos algoritmos RadixSort e MergeSort, o tempo de execução é similar. Dessa forma, foi realizado o teste t pareado com 95% de confiança a fim de compará-los estatisticamente entre si. Foi obtido o intervalo de confiança (-0,241, 0,093), levando à conclusão de que os dois algoritmos têm desempenho equivalente nesse nível de confiança.

A partir do resultado anterior, comparamos os tempos de execução por meio do teste t pareado com 70% de confiança com o objetivo de verificarmos qual dos dois algoritmos apresentam melhor desempenho. Neste caso, obtemos o intervalo de confiança (-0,145, -0,002), indicando que o MergeSort, nesse nível de confiança, apresenta melhor desempenho.

CONCLUSÃO

A partir dos resultados obtidos e da análise realizada sobre os algoritmos quanto à complexidade de tempo, foi notado que o algoritmo InsertionSort não traz bons resultados, dado que para instâncias maiores o tempo de execução aumenta rapidamente, convergindo com a análise de complexidade de tempo realizada de $O(n^2)$. O MergeSort e o RadixSort apresentaram resultados similares e satisfatórios para instâncias maiores, dado que ambos apresentam complexidade de tempo logarítmica (em alguns casos, para o RadixSort). Por meio do teste t com 95% de confiança, chegamos à conclusão de que ambos os algoritmos têm desempenho equivalente, mas pudemos inferir que o MergeSort apresenta desempenho superior ao RadixSort para níveis de confiança menores que os propostos 95%, no caso, 70%.

REFERÊNCIAS

Insertion Sort: Disponível em: <https://en.wikipedia.org/wiki/Insertion_sort>. Acesso em 23/10/2021

Tabela T: Distribuição de t-Student segundo os graus de liberdade e uma dada probabilidade num teste bicaudal. Disponível em: <http://www.epi.uff.br/wp-content/uploads/2015/05/Tabela-T.pdf>. Acesso em: 27/10/2021

https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/radixsort.pdf

Merge Sort - GeeksforGeeks. 2021. Disponível em: <https://www.geeksforgeeks.org/merge-sort/>. Acesso em: 23/10/2021

Material da professora Amanda Sávio Nascimento - 2019.