**Laboratório – Ordenação.**

**Curso: Engenharia de computação.**

**Nome: Pedro Henrique Teixeira de Souza.**

#### 1 – Implementação.

#### Os algoritmos podem ser encontrados nesse repositório do GitHub: <https://github.com/phtsouza/AnaliseDeAgoritmos>

#### Link da apresentação: https://www.youtube.com/watch?v=wKJWzkiDclA

#### 2 – Análise de Desempenho Experimental

#### A) Algoritmo Bolha

O algoritmo funciona comparando um elemento da posição atual, com outro da próxima posição. Caso a posição seja menor que a outra já é realizada a troca entre eles.

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 0,007 | 0,017 | 0,022 |
| 4000 | 0,029 | 0,069 | 0,08 |
| 8000 | 0,111 | 0,243 | 0,277 |
| 16000 | 0,427 | 1,017 | 1,178 |
| 32000 | 1,868 | 3,888 | 5,206 |
| 64000 | 7,438 | 15,637 | 19,457 |
| 128000 | 29,559 | 63,475 | 73,422 |
| 256000 | 114,604 | 264,726 | 229,706 |

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 3,810 | 3,813 | 3,816 |
| 4000 | 3,850 | 3,848 | 3,852 |
| 8000 | 3,880 | 3,883 | 3,887 |
| 16000 | 3,910 | 3,914 | 3,918 |
| 32000 | 3,980 | 3,977 | 3,980 |
| 64000 | 4,100 | 4,098 | 4,113 |
| 128000 | 4,360 | 4,359 | 4,336 |
| 256000 | 4,820 | 4,816 | 4,820 |

**B) Algoritmo Seleção**

Esse algoritmo procura o menor elemento do vetor e realiza a troca com o elemento que está na primeira posição, após isso realiza o mesmo procedimento, porém a troca é feita em uma posição à frente da troca passada.

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 0,006 | 0,009 | 0,006 |
| 4000 | 0,03 | 0,032 | 0,03 |
| 8000 | 0,099 | 0,108 | 0,1 |
| 16000 | 0,401 | 0 | 0 |
| 32000 | 1 | 2 | 1 |
| 64000 | 6 | 7 | 6 |
| 128000 | 26 | 26 | 26 |
| 256000 | 106 | 105 | 102 |

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 3,816 | 3,816 | 3,820 |
| 4000 | 3,852 | 3,852 | 3,855 |
| 8000 | 3,887 | 3,887 | 3,891 |
| 16000 | 3,918 | 3,918 | 3,922 |
| 32000 | 3,980 | 3,980 | 3,984 |
| 64000 | 4,102 | 4,102 | 4,105 |
| 128000 | 4,363 | 4,363 | 4,367 |
| 256000 | 4,820 | 4,820 | 4,824 |

**C) Algoritmo Inserção**

Esse algoritmo funciona da seguinte forma, existem duas sequências, aquelas ordenadas e as que vão ser ordenadas. Se um elemento da posição atual for menor que a próxima posição não acontece nada, porém caso contrário o menor elemento é armazenado em uma variável temporária, com ela é feita as comparações até o valor ser alocado em sua posição correta.

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 0 | 0,012 | 0,005 |
| 4000 | 0 | 0,044 | 0,025 |
| 8000 | 0 | 0,197 | 0,078 |
| 16000 | 0 | 1 | 0 |
| 32000 | 0 | 2 | 1 |
| 64000 | 0.001 | 7 | 4 |
| 128000 | 0.001 | 31 | 15 |
| 256000 | 0.001 | 132 | 38 |
| 512000 | 0.004 | 483 | 85 |

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 3,816 | 3,816 | 3,820 |
| 4000 | 3,852 | 3,852 | 3,855 |
| 8000 | 3,887 | 3,887 | 3,891 |
| 16000 | 3,918 | 3,918 | 3,922 |
| 32000 | 3,980 | 3,980 | 3,984 |
| 64000 | 4,102 | 4,102 | 4,105 |
| 128000 | 4,352 | 4,363 | 4,355 |
| 256000 | 4,836 | 4,820 | 4,852 |
| 512000 | 5,813 | 5,797 | 5,801 |

**D) Algoritmo ShellSort**

Esse algoritmo funciona da seguinte maneira, o array inicial é separado em uma quantidade “h” de pseudo arrays, logo após é realizado o algoritmo de inserção em cada um. Depois é reduzido o valor de “h” e o processo é realizado de novo até que “h” seja igual a 1 e seja efetuado o processo de inserção pela última vez.

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 0.000 | 0.001 | 0.000 |
| 4000 | 0.001 | 0.001 | 0.001 |
| 8000 | 0.000 | 0.001 | 0.001 |
| 16000 | 0.002 | 0.006 | 0.005 |
| 32000 | 0.002 | 0.010 | 0.010 |
| 64000 | 0.008 | 0.016 | 0.021 |
| 128000 | 0.016 | 0.088 | 0.045 |
| 256000 | 0.033 | 0.069 | 0.091 |
| 512000 | 0.066 | 0.132 | 0.187 |
| 1024000 | 0.120 | 0.255 | 0.482 |
| 2048000 | 0.250 | 0.398 | 0.868 |
| 4096000 | 0.573 | 0.755 | 2,241 |
| 8192000 | 1,667 | 1,873 | 4,538 |
| 16384000 | 2,997 | 3,901 | 9,732 |

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 3,820 | 3,820 | 3,824 |
| 4000 | 3,855 | 3,855 | 3,859 |
| 8000 | 3,891 | 3,891 | 3,895 |
| 16000 | 3,922 | 3,922 | 3,926 |
| 32000 | 3,984 | 3,984 | 3,988 |
| 64000 | 4,105 | 4,105 | 4,109 |
| 128000 | 4,355 | 4,355 | 4,359 |
| 256000 | 4,840 | 4,840 | 4,844 |
| 512000 | 5,816 | 5,816 | 5,820 |
| 1024000 | 7,770 | 7,770 | 7,773 |
| 2048000 | 11,676 | 11,676 | 11,680 |
| 4096000 | 19,488 | 19,488 | 19,492 |
| 8192000 | 35,113 | 35,113 | 35,117 |
| 16384000 | 66,363 | 66,363 | 66,367 |

**E) Algoritmo MergeSort**

Esse algoritmo normalmente é implementado com recursividade e funciona dividindo o vetor inicial em sub vetores até que todos tenham o tamanho de um elemento. Após isso os vetores vão intercalando para serem ordenados entre si. Este algoritmo consome muita memória pela necessidade da criação de muitos vetores.

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 0,002 | 0,001 | 0,002 |
| 4000 | 0,001 | 0,003 | 0,003 |
| 8000 | 0,003 | 0,003 | 0,007 |
| 16000 | 0,006 | 0,004 | 0,012 |
| 32000 | 0,015 | 0,012 | 0,025 |
| 64000 | 0,031 | 0,027 | 0,042 |
| 128000 | 0,055 | 0,055 | 0,071 |
| 256000 | 0,112 | 0,101 | 0,115 |
| 512000 | 0,198 | 0,597 | 0,588 |
| 1024000 | 0,352 | 1,140 | 0,944 |
| 2048000 | 0,843 | 1,466 | 1,144 |
| 4096000 | 1,559 | 5,629 | 3,141 |
| 8192000 | 3,159 | 6,426 | 4,054 |

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 3,992 | 3,992 | 4,016 |
| 4000 | 4,363 | 4,363 | 4,387 |
| 8000 | 5,055 | 5,059 | 5,078 |
| 16000 | 6,410 | 6,410 | 6,434 |
| 32000 | 9,277 | 9,246 | 9,254 |
| 64000 | 15,371 | 15,379 | 15,434 |
| 128000 | 27,465 | 27,445 | 27,449 |
| 256000 | 53,074 | 52,992 | 53,012 |
| 512000 | 105,129 | 104,957 | 105,109 |
| 1024000 | 212,891 | 212,867 | 212,898 |
| 2048000 | 436,754 | 436,777 | 436,750 |
| 4096000 | 899,836 | 899,867 | 899,777 |
| 8192000 | 1.857,445 | 1.857,559 | 1.857,461 |

**F) Algoritmo QuickSort**

Nesse algoritmo se define um pivô (sempre recebe a posição do meio do vetor), pelo qual divide o vetor em dois pontos (a esquerda menores que o pivô e a direita maiores que o pivô). Cada lado da lista será ordenado para que no final elas juntas fiquem totalmente na ordem correta.

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 0,000 | 0,001 | 0,000 |
| 4000 | 0,001 | 0,000 | 0,000 |
| 8000 | 0,001 | 0,001 | 0,001 |
| 16000 | 0,001 | 0,001 | 0,003 |
| 32000 | 0,002 | 0,002 | 0,005 |
| 64000 | 0,005 | 0,003 | 0,013 |
| 128000 | 0,010 | 0,007 | 0,05 |
| 256000 | 0,016 | 0,014 | 0,054 |
| 512000 | 0,030 | 0,031 | 0,102 |
| 1024000 | 0,065 | 0,063 | 0,196 |
| 2048000 | 0,124 | 0,133 | 0,408 |
| 4096000 | 0,263 | 0,280 | 0,818 |
| 8192000 | 0,717 | 0,564 | 1,707 |
| 16384000 | 1,187 | 1,170 | 3,481 |

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 3,820 | 3,820 | 3,824 |
| 4000 | 3,855 | 3,855 | 3,859 |
| 8000 | 3,891 | 3,891 | 3,895 |
| 16000 | 3,922 | 3,922 | 3,926 |
| 32000 | 3,984 | 3,984 | 3,988 |
| 64000 | 4,105 | 4,105 | 4,109 |
| 128000 | 4,355 | 4,355 | 4,359 |
| 256000 | 4,840 | 4,840 | 4,844 |
| 512000 | 5,816 | 5,816 | 5,820 |
| 1024000 | 7,770 | 7,770 | 7,773 |
| 2048000 | 11,676 | 11,676 | 11,680 |
| 4096000 | 19,488 | 19,488 | 19,492 |
| 8192000 | 35,113 | 35,113 | 35,117 |
| 16384000 | 66,363 | 66,363 | 66,367 |

**G) Algoritmo Heapsort**

Se inspira no algoritmo de seleção, ele funciona por meio de uma heap (árvore de dados do vetor) invertida, ou seja, logo no início já será possível identificar o maior elemento do vetor. A ordenação é feita tendo como base a árvore de dados criada fazendo a ordenação debaixo pra cima.

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 0,000 | 0,000 | 0,001 |
| 4000 | 0,002 | 0,001 | 0,002 |
| 8000 | 0,003 | 0,002 | 0,004 |
| 16000 | 0,006 | 0,005 | 0,008 |
| 32000 | 0,017 | 0,010 | 0,014 |
| 64000 | 0,032 | 0,019 | 0,030 |
| 128000 | 0,061 | 0,041 | 0,052 |
| 256000 | 0,134 | 0,086 | 0,108 |
| 512000 | 0,224 | 0,191 | 0,225 |

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 3,820 | 3,820 | 3,824 |
| 4000 | 3,863 | 3,863 | 3,867 |
| 8000 | 3,910 | 3,910 | 3,914 |
| 16000 | 3,973 | 3,973 | 3,977 |
| 32000 | 4,098 | 4,098 | 4,102 |
| 64000 | 4,340 | 4,340 | 4,344 |
| 128000 | 4,836 | 4,836 | 4,840 |
| 256000 | 5,809 | 5,809 | 5,813 |
| 512000 | 7,762 | 7,762 | 7,766 |

**H) Algoritmo CountingSort**

Nesse algoritmo para inicializar ele procura o maior elemento presente no vetor, para assim, criar um novo vetor de contagem com o tamanho do valor de maior elemento (array de contagem). Nesse array de contagem é realizado o somatório da posição 0 até a posição atual. Exemplo: A posição atual é 2, o valor armazenado nela será da somatória da posição 0 e 1. Após preenchido o array de contagem, será produzido o vetor final, ele funciona de forma em que o valor armazenado na posição representa a posição em que o índice ficará no vetor final.

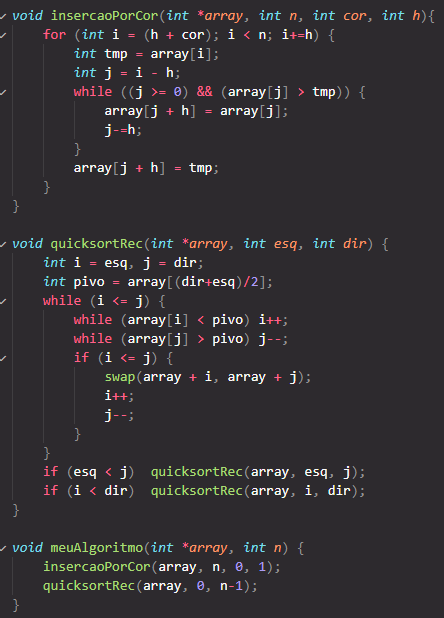
|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 0,000 | 0,000 | 0,000 |
| 4000 | 0,000 | 0,000 | 0,000 |
| 8000 | 0,000 | 0,001 | 0,000 |
| 16000 | 0,001 | 0,001 | 0,001 |
| 32000 | 0,001 | 0,001 | 0,002 |
| 64000 | 0,002 | 0,003 | 0,002 |
| 128000 | 0,005 | 0,008 | 0,007 |
| 256000 | 0,010 | 0,013 | 0,020 |

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 3,828 | 3,828 | 3,832 |
| 4000 | 3,875 | 3,875 | 3,879 |
| 8000 | 3,941 | 3,941 | 3,945 |
| 16000 | 4,035 | 4,035 | 4,039 |
| 32000 | 4,219 | 4,219 | 4,223 |
| 64000 | 4,586 | 4,586 | 4,590 |
| 128000 | 5,324 | 5,324 | 5,328 |
| 256000 | 6,785 | 6,785 | 6,789 |

**I) Meu algoritmo**

Para a implementação desse algoritmo, utilizei o pré-processamento por cores do algoritmo shellsort e após isso fiz a ordenação com o algoritmo quicksort.

Dessa forma:



Esses foram os resultados alcançados:

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 0,000 | 0,009 | 0,009 |
| 4000 | 0,000 | 0,061 | 0,043 |
| 8000 | 0,001 | 0,167 | 0,097 |
| 16000 | 0,001 | 0,640 | 0,311 |
| 32000 | 0,002 | 3,081 | 1,483 |
| 64000 | 0,003 | 12,709 | 7,531 |

|  |  |  |  |
| --- | --- | --- | --- |
| Valor para N | Ordenado | Não Ordenado | Aleatório |
| 2000 | 3,813 | 3,813 | 3,816 |
| 4000 | 3,848 | 3,848 | 3,852 |
| 8000 | 3,883 | 3,883 | 3,887 |
| 16000 | 3,914 | 3,914 | 3,918 |
| 32000 | 3,977 | 3,977 | 3,980 |
| 64000 | 4,098 | 4,098 | 4,102 |

**3 – Análise dos Resultados Analíticos e Experimentais**

#### A) Algoritmo Bolha

O algoritmo bolha é um dos piores casos em comparação com os outros, pois por ele não ter pior ou melhor caso, ele sempre faz n ao quadrado comparações, gastando mais tempo.

Função de custo: F(n) = θ(n^2).

**B) Algoritmo Seleção**

O algoritmo seleção de acordo com os testes não tem melhor ou pior caso. Ele sempre faz o número de (n^2)/2 + n/2.

Função de custo: F(n) = θ(n).

**C) Algoritmo Inserção**

O algoritmo inserção, diferente dos outros dois apresentados até agora tem um pior e melhor caso, pelo qual é quando o vetor não estar ordenado. Porém, quando o vetor estar ordenado ele se sai muito bem nos testes. No melhor caso é feito θ(n) comparações. Já no pior caso é feito θ(n^2).

Função de custo: No melhor caso θ(n). No pior caso θ(n^2).

**D) Algoritmo ShellSort**

A razão de funcionamento do algoritmo ainda não é conhecida, porém pode se afirmar que cada incremento não deve ser múltiplo do anterior. O shellsort é uma ótima escolha quando os arquivos de tamanho médio e ele é um algoritmo de fácil implementação pois utiliza poucas linhas de código. Porém sua desvantagem é quando não se sabe a ordem do vetor, pois ele é dependente da ordem inicial do arquivo.

Função de custo na conjectura de Knuth:

Conjectura 1: C(n) = θ(n^1,25).

Conjectura 2: C(n) = θ(n(ln n)^2).

**E) Algoritmo MergeSort**

Esse algoritmo não existe pior caso, sempre realiza θ(n(ln n)) comparações, porém seu ponto negativo é a quantidade de memória gasta para o seu processamento, pois é necessário criar vários vetores.

Função de custo: θ(n(ln n)).

**F) Algoritmo QuickSort**

O melhor caso desse algoritmo é quando o vetor é divido em partes iguais, já o pior caso é quando o pivô é o menor ou maior elemento do array.

Função de custo melhor caso: θ(n(ln n)).

Função de custo pior caso: θ(n^2).

**G) Algoritmo HeapSort**

Nesse algoritmo o melhor e pior caso são iguais, em ambos a sua função de custo será de θ(n(ln n)). Porém, mesmo assim ele perde para o quicksort em relação ao tempo de processamento.

**H) Algoritmo CountingSort**

Apresenta ótimos resultados em seus testes, porém tem um limitador pois ele só funciona com números inteiros e positivos. Portanto, também não apresenta melhor ou pior caso.

Função de custo: θ(n(ln n)).

**4 – Conclusão**

Dentre os algoritmos analisados, ainda assim é difícil apontar um que seja melhor que todos os outros em qualquer situação. Porém, analisando os casos em questão o quicksort é aquele que apresenta melhor resultado em geral. Ainda assim, podemos analisar o countingsort, pelo qual é aquele que apresenta os melhores resultados em questão do tempo gasto, porém ele é limitado a números positivos e inteiros. Agora, quando o vetor já se encontra ordenado, o melhor algoritmo de acordo com os testes feitos, é o de inserção.

Podemos então com esse trabalho, como são os resultados dos algoritmos na prática e com isso fica mais claro qual é a melhor opção para cada situação.

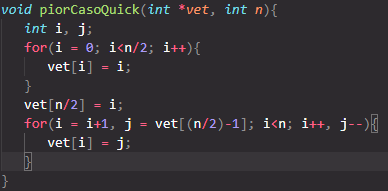
Agora, falando sobre a relação de análise de complexidade e análise experimental, temos algo curioso. As funções de custo do heapsort e quicksort são iguais, porém ao realizar os testes, é possível perceber uma grande superioridade de desempenho em todos os casos por conta do quicksort e isso só foi possível ser visto por conta da análise prática do algoritmo.

Temos como exemplo o mergesort, pelo qual apresenta um ótimo resultado de processamento, porém é limitado por conta da memória devido ao fato de ele criar uma alta quantidade de vetores, no entanto em casos pelo qual se trata de uma alta quantidade de dados, é totalmente inviável sua utilização.

**5 – Extra – QuickSort(Pior Caso) x Bolha**

Essa parte extra do trabalho é uma experiência muito interessante, pois no pior caso do quicksort ele tem a mesma função de custo do algoritmo bolha, θ(n^2). Porém na prática podemos ver a diferença real dos resultados.

Para a criação do pior caso, foi criado um procedimento:



Com isso, podemos analisar os seguintes resultados:

|  |  |  |
| --- | --- | --- |
| Valor para N | QuickSort | Bolha |
| 2000 | 0,004 | 0,007 |
| 4000 | 0,014 | 0,029 |
| 8000 | 0,068 | 0,111 |
| 16000 | 0,232 | 0,427 |
| 32000 | 0,871 | 1,868 |
| 64000 | 3,406 | 7,438 |

Contudo, após a análise do gráfico podemos ver que ainda assim com a mesma função de custo, na prático o algoritmo quicksort é muito superior.