ALGORITMOS DE ORDENAÇÃO

Matheus N. R. da Silva

Faculdade de Computação - Universidade Federal do Mato Grosso do Sul (UFMS)
Campo Grande, Mato Grosso do Sul
Atividade 1 - Implementação Algorítmica
Prof. Carlos Henrique Aguena Higa

nantes.matheus@ufms.br

Abstract: *This report presents the development of Activity 1, where 6 sorting algorithms were implemented: SelectionSort, InsertionSort, MergeSort, HeapSort, QuickSort and CountingSort, applied to different types of vectors.*

Resumo: *Este relatório apresenta o desenvolvimento da Atividade 1, onde foi implementado 6 algoritmos de ordenação: SelectionSort, InsertionSort, MergeSort, HeapSort, QuickSort e CountingSort, aplicados em diferentes tipos de vetores.*

1. Introdução:

O intuito desta atividade foi desenvolver os 6 algoritmos de ordenação solicitados, bem como gerar 4 tipos diferentes de vetores, os quais são: vetor aleatório (identificado com "[[RANDOM]]), vetor reverso, ou seja, ordenado de forma decrescente (identificado com "[[REVERSE]]), vetor ordenado crescentemente (identificado com "[[SORTED]]), e um vetor parcialmente ordenado crescentemente, com cerca de 10% de seus elementos em posições incorretas (identificado com "[[NEARLY SORTED]]). Os algoritmos de ordenação foram explicados ao decorrer das aulas da disciplina. Esta atividade, assim como citado pelo professor, tem como objetivo analisarmos os algoritmos de ordenação, aplicados em diferentes conjuntos de dados, e ver como eles se comportam, para que também possamos compará-los e visualizarmos quais são mais e menos eficientes em diferentes circunstâncias. Tal metodologia é considerada muito interessante por parte de alguns alunos, pois o conceito de análise assintótica lecionado à turma até o dado momento, em disciplinas como estrutura de dados e Algoritmos e programação 2, foi de maneira muito teórica e conceitual, onde víamos os algoritmos e seus tempos unitariamente, sem nenhuma aplicação ou comparação de fato, e tais conteúdos eram retratados apenas como uma lei, que apenas deveríamos aceitar, no estilo "QuickSort é $O(n*\log n)$ no melhor caso, e $O(n^2)$ no pior, porque sim, e ponto.", obviamente tal frase nunca foi dita explicitamente, mas foi essa a impressão que alguns alunos obtiveram em relação à análise assintótica. Já logo no início do desenvolvimento deste trabalho foi perceptível a motivação e curiosidade por parte de alguns alunos em implementar e descobrir de fato como tais algoritmos de ordenação se comportam paralelamente sob diferentes circunstâncias.

2. Metodologia:

Foram definidas alguns pontos no início do desenvolvimento: a IDE de desenvolvimento a ser utilizado seria o VS CODE, a linguagem de programação seria a C, e seria utilizado o GitHub para controle de versionamento e possibilidade de acesso ao código de forma remota, caso houvesse necessidade, e os gráficos seriam gerados através do GNUPLOT, entretanto, este último ponto foi modificado posteriormente pois migramos para a biblioteca MATPLOTLIB do python, apenas para a geração dos gráficos. Para organização de código, foi definido também que cada algoritmo de ordenação teria seu respectivo "arquivo.c", bem como um arquivo para a geração dos vetores, e por fim um arquivo main.c para realizar todas as outras operações necessárias. Tais arquivos são conectados pela biblioteca "biblioteca.h". Após essas definições, foi iniciado o desenvolvimento. Também seriam gerados arquivos, cujos os nomes seriam a junção do algoritmo de ordenação e qual tipo de vetor foi usado, facilitando a plotagem dos gráficos posteriormente, por exemplo "selectionRandom.txt", que contém os tempo do SelectionSort na ordenação de um vetor aleatório. O repositório desenvolvido pode ser visualizado no seguinte link: https://github.com/matheus-nantes/IMPLEX.

Foi estabelecida também uma sequência de trabalho, elencando prioridades e complexidade. A ordem definida foi: construir vetores, métodos de ordenação, na seguinte ordem: SelectionSort, InsertionSort, MergeSort, HeapSort, QuickSort CountingSort, criação de arquivos e plotagem dos gráficos.

Primeiramente, na geração dos vetores foi feito:

Vetor aleatório [[RANDOM]]: com o uso método rand, da biblioteca stdlib, foi possível obter valores aleatórios. Porém, durante o desenvolvimento, foi visualizado que o uso deste método sucessivamente acaba gerando valores repetidos, então foi preciso utilizar o método time(null) dentro do srand, para fazer com que a geração de números aleatórios fosse reiniciada, gerando de fato número aleatório. Para gerar os valores foi definido que, os valores iriam até 1.5 vezes o tamanho do n, para que assim a variedade de números fosse maior. Por exemplo, ao gerar um vetor com 1000 elementos, os valores iriam variar de 0 até 1499. Isso foi possível através de "rand()%(tamanho*1.5)".

```c
void gerarAleatorios(int tamanho, int * vetor){

    srand(time(NULL));

    for(int i = 0; i < tamanho; i++){
        vetor[i] = rand() % (int)(tamanho*1.5);
    }
}
```

Vetor reverso [[REVERSE]]: também com o método rand, juntamente com srand e time, este vetor é construído da última posição até a primeira, da seguinte forma: geramos um valor entre 0 e 10 e colocamos ele na última posição do vetor, com isso, as posições anteriores são definidas através da soma da posição imediatamente posterior à ela + um acréscimo gerado aleatoriamente que varia entre 0 e 10.

Com isso, os valores são gerados crescentemente, porém inseridos inversamente no vetor, do final para o começo.

```c
void gerarReverso(int tamanho, int * vetor){
    srand(time(NULL));
    int acrescimo = 0;
    vetor[tamanho-1] = rand() % 10;
    for(int i = 2; i <= tamanho; i++){
        acrescimo = rand()%10;
        if(acrescimo == 0){
            acrescimo = rand()%10;
        }
        vetor[tamanho-i] = vetor[tamanho-i+1] + acrescimo;
    }
    // printf("----ALEAT----- + %d - %d\n",vetor[0], vetor[tamanho-1]);
}
```

Vetor ordenado [[SORTED]]: a geração deste vetor é muito similar com a geração do vetor reverso, o único detalhe é que é definida a primeira posição do vetor, e as posições subsequentes são geradas utilizando a posição imediatamente anterior à ela + um acréscimo aleatório que varia entre 0 e 10. Este método gera um vetor que têm valores entre 0 e n*10 ordenados crescentemente.

Vetor quase ordenado [[NEARLY SORTED]]: este método utiliza o método para gerar um vetor ordenado para definir seus valores, porém, é realizado um processamento em cima desses valores. Para “bagunçar” os valores em 10% foi calculado que: para bagunçar os valores realizamos uma troca entre duas posições, logo, como uma troca envolve 2 posições, para bagunçar por exemplo 10% de 100 elementos, é preciso realizar 5 trocas, pois 10 posições seriam “bagunçadas”, logo, a quantidade de trocas a serem realizadas é equivalente à 5% da quantidade de elementos. Logo, precisamos dividir o tamanho por 20 para obter o equivalente a 5%. pois (x/100)*5, dividindo por 5 temos (x/20)*1. Com isso, temos mais um detalhe, precisamos garantir que as posições já não tenham sido trocadas e não sejam iguais, o que resulta em uma condição um pouco extensa, mas é basicamente: enquanto o valor da posição for menor que seu antecessor ou maior que seu sucessor, ou igual à outra posição selecionada, precisamos selecionar outra posição. Se ela for maior que seu antecessor ou menor que seu sucessor, significa que ela já foi trocada, pois essas posições não estão ordenadas. Porém, precisamos adicionar também uma condição que, caso a posição selecionada seja a primeira, ela não possui antecessor, logo, não podemos realizar a verificação com o antecessor senão ocorrerá erro, logo, verificamos se a posição é diferente da primeira, pois se ela for igual, resulta em FALSE, e juntamos ela com && com a comparação do antecessor, pois assim, por causa do and, a segunda condição nem será calculada. A lógica para a comparação com o sucessor é a mesma, só que verificamos se a posição não é igual ao tamanho-1. Após a definição correta das posições a serem trocadas, garantido que as posições selecionadas já não tenham sido trocadas, podem enfim realizar a troca e garantir que 10% dos elementos do vetor estejam de fato “bagunçados”. A seguinte lógica pode ser visualizada na seguinte imagem, nas linhas 51 e 54.

```c
42  void gerarQuaseOrdenado( int tamanho, int *vetor){
43      srand(time(NULL));
44
45      int aux, p1, p2;
46      gerarOrdenado(tamanho, vetor);
47      for(int i = 0; i < tamanho/20; i++){
48          p1 = rand()%tamanho;
49          p2 = rand()%tamanho;
50
51          while((p1 != 0 && vetor[p1] < vetor[p1-1]) || (p1 != tamanho-1 && vetor[p1] > vetor[p1+1]) || p1 == p2
52              p1 = rand()%tamanho;
53
54          while((p2 != 0 && vetor[p2] < vetor[p2-1])|| (p2 != tamanho-1 && vetor[p2] > vetor[p2+1]) || p1 == p2)
55              p2 = rand()%tamanho;
56
57          aux = vetor[p1];
58          vetor[p1] = vetor[p2];
59          vetor[p2] = aux;
60      }
61  }
```

Após a geração dos 4 tipos de vetores foram verificados todos os vetores gerados e eles são de fato aleatórios, cada um seguindo sua respectiva lógica, e a cada execução um conjunto de valores diferente é gerado.

Com a geração de vetores funcional, podemos prosseguir para o desenvolvimento de alguns algoritmos de ordenação. Utilizamos os materiais disponibilizados nas aulas da disciplina, porém, também utilizamos como auxílio vídeos do youtube para relembrar a lógica de cada algoritmo, no caso principalmente do MergeSort, QuickSort e CountingSort.

No desenvolvimento dos algoritmos de ordenação não tivemos muita dificuldade pois eles possuem uma estrutura e sequência lógica bem definida, ocorreram apenas erros de desenvolvimento básicos, como nomeação de variáveis e posicionamento incorreto de chamada de recursão. Porém esses erros foram facilmente identificados.

Com os algoritmos de ordenação implementados, realizamos os testes deles com vetores de tamanho reduzido para que pudéssemos realizar a verificação visual unitária, e verificamos que os algoritmos estavam realizando as ordenações corretamente.

Com os algoritmos implementados podemos realizar o cálculo do tempo de execução de cada um, com isso, precisamos implementar a estrutura principal do programa, onde realizamos as leituras das variáveis solicitadas pela descrição da atividade, que são inc, fim, stp e rpt. Com isso definimos estruturas de repetição com os valores dessas variáveis, onde temos um for que inicia em inc e termina em fim, e o incremento é o valor de stp. E para os vetores aleatórios possuímos mais um for para realizar rpt vezes as ordenações e então extrair a média dos tempos de execução. Para garantir que os algoritmos de ordenação utilizem exatamente os mesmos valores, para que a comparação seja de fato válida, realizamos a geração dos valores em um vetor, e depois copiamos cada posição desse vetor na respectiva posição de cada um dos outros 6 vetores, onde cada um será utilizado pelo seu respectivo algoritmo de ordenação, por exemplo, o vetorQ é utilizado pelo QuickSort, e etc.

Para realizar o cálculo do tempo utilizamos a biblioteca “sys/time”, onde através dela conseguimos usar o método gettimeofday, que retorna o tempo atual com precisão de microssegundos, e armazena o retorno em uma variável do tipo do struct timeval, providenciado pela mesma biblioteca. Com isso podemos subtrair os segundo e microssegundos do tempo final pelo tempo inicial e calcular quanto tempo demorou para executar a ordenação. Um detalhe é que para os vetores [[RANDOM]] nós precisamos extrair a média dos tempos,ou seja, acumulamos o resultado dessa subtração em uma variável e depois da estrutura de repetição dividos esse valor por rpt, enquanto que nos outros vetores temos como tempo o resultado imediato subtração citada, “fim-inicio”.

Após a obtenção do tempos de execução foi visualizado um erro, pois, apesar de todos os algoritmos de execução estarem ordenando corretamente, foi verificado que o QuickSort estava tendo um desempenho medíocre, equiparável ao SelectionSort e InsertionSort e alguns casos, principalmente no vetor reverso, com isso, verificamos que a maneira como implementamos a definição do pivô acaba resultando em mais ocorrências do pior caso para este algoritmo, que resulta em tempo $O(n^2)$. Com isso, realizamos algumas pesquisas e definimos que o pivô seria definido aleatoriamente, ao invés da última posição do vetor, que era o modo que estávamos fazendo, e com isso obtivemos uma diferença considerável na execução, garantindo que na maioria dos casos o QuickSort fosse realmente muito eficiente.

Outro detalhe foi que os testes foram realizados inicialmente com valores até 20000, mas vimos que os tempos estavam realmente muito baixos, sendo quase imperceptível o crescimento do tempo nos gráficos, com isso começamos a aumentar gradativamente os

valores de inc e fim. Com isso, foram definidos valores até 100.000, cujos quais apresentaram tempo e gráficos consideráveis.

Porém, um dos maiores empecilhos que tivemos na trajetória desta atividade foi que, na execução, especificamente do QuickSort em um vetor reverso, com tamanho maior que 44 mil, a execução simplesmente parava, não acusava nenhum erro, simplesmente finalizava e voltava para o terminal como se nada houvesse ocorrido. Mas com o uso do *debbuger* gdb foi visualizado que estava ocorrendo segmentation fault em algum momento, o que finalizava a execução como se nada tivesse ocorrido, porém, do ponto de vista lógico não havia sentido este erro não ter sido indicado em nenhum dos outros 43.999 vetores gerados e ordenados anteriormente, pois se fosse erro de lógica de código esse erro seria indicado em vetores com até mesmo 2 ou 3 elementos. Mas através de muita pesquisa foi verificado que isso era resultado de um conjunto de fatores, como limitações do VS Code juntamente com o SO, Windows no caso, referente a limite de memória, recursão e outras métricas, e com isso, encontrei uma flag para o gcc, que foi a “-O3”, “-Os” também serve, que remove a maioria desses impedimentos, e então finalmente foi possível realizar ordenações de vetores com até 5.000.000. Porém, como os tempos resultantes dessas ordenações eram extremamente extensos,por conta do SelectionSort e do InsertionSort, definimos que era inviável utilizá-los nos valores finais para o relatório, além de que valores a partir de 100.000 elementos já resultaram em tempos satisfatórios. Logo, os valores máximos definidos foram de 100.000, que foi o valor usado na última execução para realizar este relatório.

Após toda a lógica de programação estrutura e testa corretamente, implementamos os códigos para a geração dos arquivos para a alimentação dos gráficos no formato que o GNUPLOT utiliza, bem como a geração da tabela solicitada na descrição da atividade. Entretanto, os gráficos gerados pelo GNUPLOT não agradou em completude, então recorremos à biblioteca MATPLOTLIB do python, onde conseguimos obter gráficos satisfatórios, com isso, foi preciso realizar algumas adaptações nos arquivos de saída para que correspondessem aos parâmetros da MATPLOTLIB, sendo esse processo mais manual do que se tivesse sido usado o GNUPLOT.

Com o conjunto dos dados definido, realizamos a plotagem dos gráficos e análise dos mesmos. Os arquivos gerados pelo código, e utilizados para a plotagem dos gráficos também estão disponíveis no repositório no GitHub informado anteriormente. Utilizamos os dados gerados nos arquivos txt e posteriormente convertemos eles para um formato que a biblioteca MATPLOTLIB, do python, conseguisse ler e plotar os gráficos de maneira mais rápida e automática. Basicamente transformamos os valores dos tempos em um vetor e realizamos a plotagem com a MATPLOTLIB, onde cada vetor era correspondente à um algoritmo de ordenação. Achamos mais agradável e funcional a maneira que o MATPLOTLIB gerava seus gráficos, além de ser mais personalizável e de fácil acesso à tutoriais e materiais complementares.

3. Resultados:

Os resultados obtidos serão expressos através dos gráficos. Mas de modo geral, todos os tempos de execução estão presentes na seguinte tabela:

[[RANDOM]]

n	Selection	Insertion	Merge	Heap	Quick	Counting
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
5000	0.014938	0.010755	0.002157	0.000615	0.000700	0.000100
10000	0.029418	0.025160	0.002406	0.000696	0.001103	0.000100
15000	0.064786	0.054542	0.003400	0.001454	0.001500	0.000300
20000	0.107905	0.093099	0.004480	0.001666	0.001984	0.000150
25000	0.167700	0.144865	0.005755	0.002000	0.002659	0.000300
30000	0.244098	0.207846	0.006704	0.002399	0.003306	0.000200
35000	0.326339	0.283899	0.007954	0.002801	0.003712	0.000501
40000	0.425111	0.374591	0.008665	0.003447	0.004300	0.000400
45000	0.540652	0.467922	0.010908	0.004004	0.004761	0.000601
50000	0.663841	0.589588	0.011307	0.004201	0.005500	0.000400
55000	0.799993	0.711577	0.012471	0.004700	0.006158	0.000300
60000	0.952056	0.855449	0.013452	0.005300	0.006755	0.000500
65000	1.123440	0.993614	0.014563	0.005952	0.007601	0.000200
70000	1.294855	1.158138	0.015905	0.006251	0.008052	0.000500
75000	1.487295	1.333832	0.016952	0.007000	0.008396	0.000605
80000	1.695422	1.521510	0.018218	0.007047	0.009255	0.000700
85000	1.911791	1.713211	0.019267	0.008056	0.009748	0.000600
90000	2.161116	1.918715	0.020324	0.008202	0.010502	0.000801
95000	2.416567	2.149543	0.021772	0.008752	0.011168	0.000505
100000	2.651015	2.359845	0.022606	0.009266	0.011700	0.000801

[[SORTED]]

n	Selection	Insertion	Merge	Heap	Quick	Counting
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
5000	0.015014	0.000000	0.000000	0.000000	0.001011	0.000000
10000	0.035499	0.000000	0.002032	0.000000	0.000000	0.000000
15000	0.073686	0.000000	0.003001	0.000000	0.001058	0.000000
20000	0.111622	0.000966	0.003007	0.001046	0.000999	0.000000
25000	0.178987	0.000000	0.004038	0.001059	0.002007	0.000000
30000	0.246935	0.000000	0.005007	0.002004	0.002000	0.001001
35000	0.339270	0.000000	0.006046	0.002000	0.003002	0.000998
40000	0.496615	0.000000	0.007047	0.002008	0.002058	0.000000
45000	0.608246	0.000000	0.007045	0.002000	0.003036	0.000999
50000	0.681197	0.000000	0.009038	0.002008	0.003046	0.001000
55000	0.813114	0.000000	0.008506	0.003008	0.003999	0.001002
60000	0.961694	0.000000	0.010011	0.004000	0.004000	0.001009
65000	1.113739	0.000000	0.011057	0.002999	0.004046	0.001009
70000	1.311975	0.000000	0.011036	0.004007	0.005044	0.001001
75000	1.490218	0.000000	0.012056	0.004042	0.005011	0.001049
80000	1.703085	0.000000	0.012626	0.003998	0.004992	0.002001
85000	1.912201	0.000000	0.014045	0.005000	0.006007	0.001041
90000	2.158519	0.000000	0.014998	0.005008	0.007004	0.000997
95000	2.406264	0.000000	0.019999	0.007006	0.008010	0.001001
100000	2.861918	0.001001	0.016008	0.006011	0.007000	0.002000

[[REVERSE]]

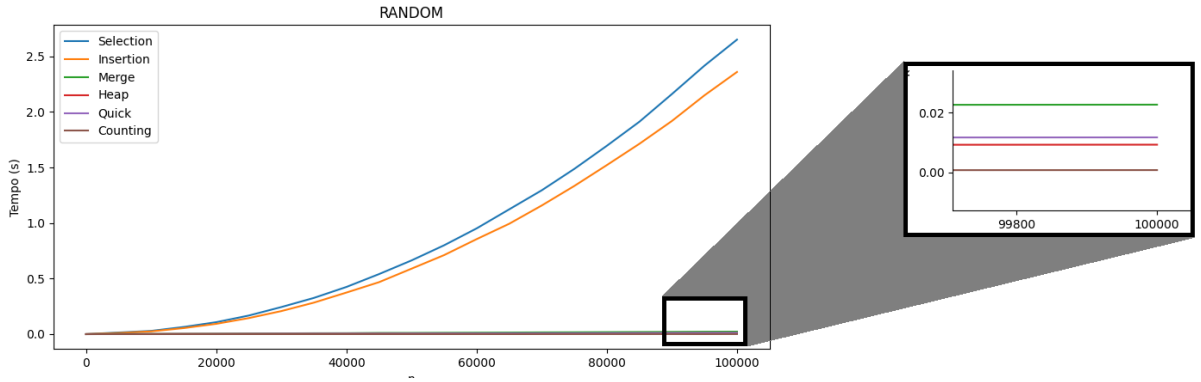
n	Selection	Insertion	Merge	Heap	Quick	Counting
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
5000	0.010002	0.013034	0.001000	0.000000	0.000000	0.000000
10000	0.035617	0.052995	0.002001	0.000000	0.000000	0.001299
15000	0.077661	0.118575	0.003050	0.001048	0.001001	0.001000
20000	0.132196	0.181664	0.003528	0.000508	0.002004	0.000000
25000	0.183216	0.297647	0.005003	0.000999	0.002000	0.000000
30000	0.298575	0.428921	0.005045	0.001963	0.002000	0.000000
35000	0.366591	0.567769	0.006003	0.001998	0.002009	0.001001
40000	0.471619	0.737638	0.006469	0.002007	0.003000	0.001002
45000	0.593725	0.925796	0.007045	0.003002	0.003007	0.001002
50000	0.719910	1.156403	0.008007	0.003045	0.004004	0.001001
55000	0.864073	1.382441	0.008588	0.004005	0.004044	0.001004
60000	1.012182	1.683725	0.010008	0.004036	0.004008	0.001001
65000	1.198790	1.956634	0.011039	0.003511	0.005006	0.001008
70000	1.382236	2.268470	0.011232	0.003954	0.005045	0.001009
75000	1.584503	2.625570	0.012614	0.004517	0.005045	0.002003
80000	1.801374	2.957823	0.013113	0.004999	0.006458	0.002007
85000	2.033471	3.324987	0.014044	0.005000	0.006014	0.001994
90000	2.280318	3.776294	0.014511	0.005057	0.006990	0.001999
95000	2.552040	4.174751	0.016054	0.005997	0.007007	0.002058
100000	2.822678	4.572439	0.016047	0.006008	0.007011	0.001997

[[NEARLY SORTED]]

n	Selection	Insertion	Merge	Heap	Quick	Counting
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
5000	0.014001	0.001003	0.001002	0.000000	0.000000	0.000000
10000	0.033034	0.003012	0.002003	0.000998	0.001000	0.000000
15000	0.072666	0.007055	0.003056	0.001006	0.000997	0.000000
20000	0.117662	0.010007	0.003103	0.001049	0.000986	0.000000
25000	0.178201	0.017010	0.004008	0.002000	0.001999	0.000000
30000	0.275058	0.028448	0.006048	0.002009	0.003000	0.001003
35000	0.412431	0.038058	0.006992	0.002048	0.003009	0.001060
40000	0.462982	0.040101	0.008963	0.002508	0.004000	0.002001
45000	0.580057	0.045612	0.009061	0.002948	0.004055	0.001001
50000	0.719007	0.050704	0.010961	0.003502	0.005000	0.001039
55000	0.857318	0.054048	0.011002	0.004036	0.005057	0.001000
60000	1.003351	0.053560	0.011296	0.003978	0.005009	0.001001
65000	1.158776	0.060570	0.012060	0.004055	0.005999	0.001992
70000	1.328584	0.065107	0.014008	0.005006	0.006037	0.001036
75000	1.505319	0.070569	0.014056	0.005058	0.005987	0.000997
80000	1.703093	0.074698	0.015011	0.005056	0.006011	0.002039
85000	1.924757	0.079617	0.016002	0.006007	0.006008	0.002001
90000	2.216215	0.083968	0.015584	0.005058	0.007009	0.002042
95000	2.401887	0.087677	0.016011	0.005997	0.007007	0.002001
100000	2.668978	0.091621	0.018968	0.006008	0.009000	0.002001

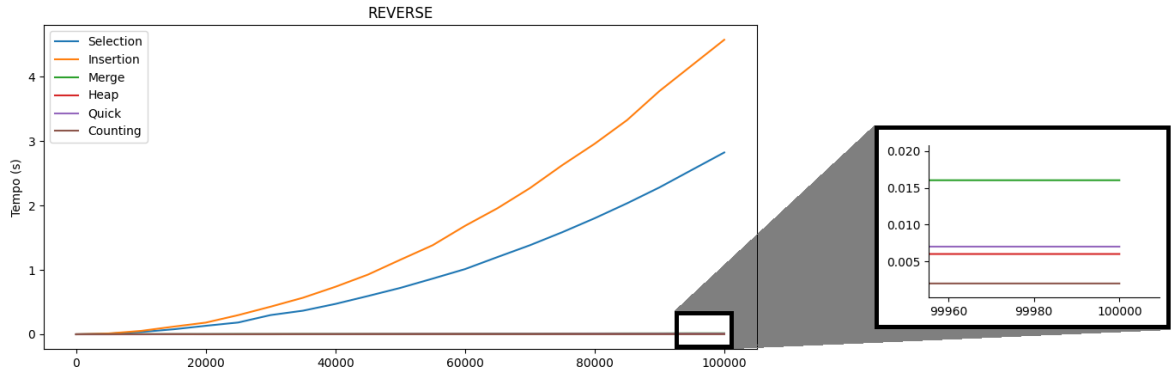
Como é possível deduzir com os valores da tabela, os parâmetros informados foram: inc = 0, fim = 100.000, stp = 5.000. Já o rpt informado foi 10.

Ordenação com vetor aleatório:



Como é possível visualizar, com um vetor de números aleatórios, elencando do mais eficiente para o menos, obtemos: CountingSort, HeapSort, QuickSort, MergeSort, InsertionSort e SelectionSort. Lembrando que esses tempos obtidos é resultado da média de 10 execuções de cada algoritmo em cada vetor com n elementos. Podemos ver também que a discrepância de tempo entre os 4 mais eficientes foi mínima, sendo ainda mais imperceptível entre Heap e o QuickSort sendo necessário dar um zoom bem considerável para poder perceber uma diferença.

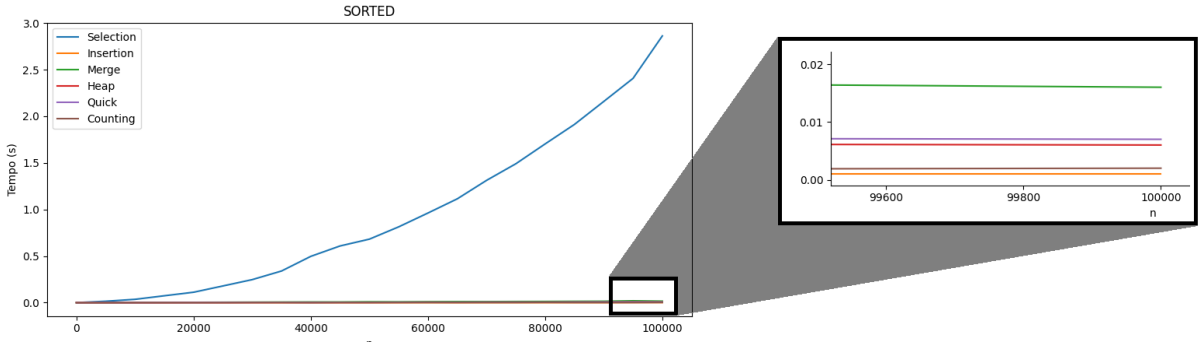
Ordenação com vetor reverso:



Já com o vetor reverso tivemos algumas alterações em relação ao aleatório, pois agora o algoritmo que desempenhou pior tempo foi o Insertion, pois esse é o seu pior caso. Do mais lerdo ao mais rápido nessa configuração de vetor temos: Insertion, Selection, MergeSort, QuickSort Heap e Counting.

Olhando agora de outra perspectiva, apenas para o 4 mais rápidos, podemos visualizar que o que acontece aqui é bem parecido ao que acontece com um vetor com valores aleatórios, mas com um pequeno detalhe, os tempos reduziram relativamente, pois enquanto o MergeSort estava passando por pouco a casa de 0,2 s em um vetor aleatório, nesse caso ela aproximou-se de 0,15s, uma redução de aproximadamente 25%, essa redução acontece também no QuickSort e no HeapSort.

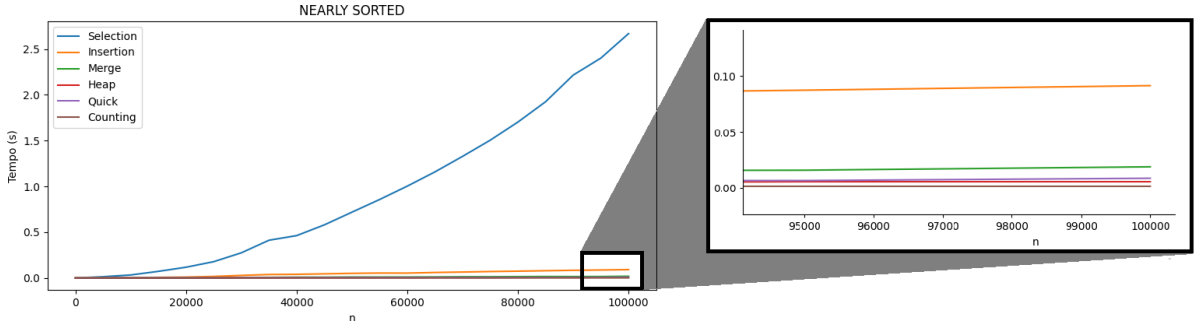
Ordenação com vetor ordenado:



Com um conjunto de elementos já ordenados tivemos uma alteração muito mais perceptível. Podemos ver que agora apenas o SelectionSort possui uma gráfico acentuado, inclusive, o InsertionSort, que nos outros dois casos estava presente, desempenhando tempos expressivos, agora já nem aparece no gráfico praticamente. Acontece que ele, assim como os outros, exceto o SelectionSort, realizam as operações muito rapidamente, muito próximos à zero, como é possível ver no gráfico.

Com isso, ainda podemos elencar os algoritmos conforme a eficiência sobre um conjunto de dados dispostos da forma ordenada, onde temos o InsertionSort como mais rápido, atingindo um tempo muito próximo à zero, seguido por CountingSort, HeapSort, QuickSort e por fim temos o SelectionSort como o menos eficiente neste caso.

Vetor parcialmente ordenado:



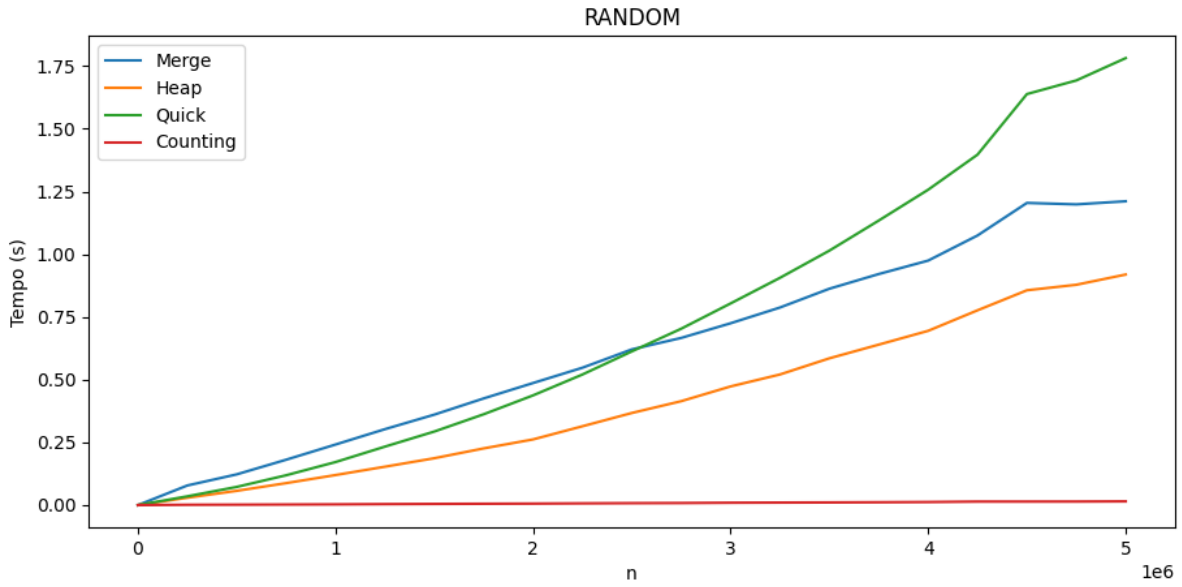
Este caso mostrou-se bem similar ao anterior, onde apenas o SelectionSort desempenha um tempo considerável, e os outros algoritmos apresentam desempenho muito rápido. Destaca-se que o InsertionSort, diferentemente do exemplo anterior começa a se distanciar dos 4 mais rápidos, e os 4 mais eficientes também apresentam uma inclinação levemente mais angulada.

Podemos ver que agora o mais rápido é o countig, depois QuickSort e HeapSort muito próximos, então temos o MergeSort, o InsertionSort, e bem distante temos o SelectionSort.

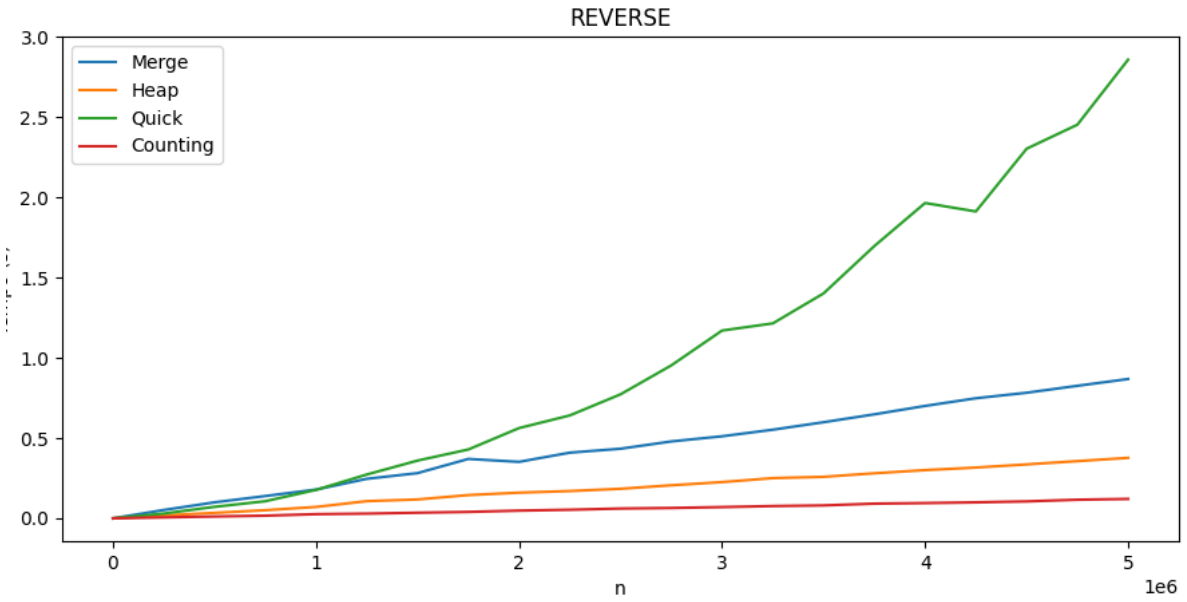
***Entradas consideravelmente altas apenas nos mais eficientes:**

*Para obtenção de resultados mais expressivos referentes ao algoritmos mais eficientes de cada caso, fez se preciso “chutar o balde” para que consigamos obter comparações mais significativas, logo, testes com vetores de 5.000.000 de elementos foram realizados com os algoritmos MergeSort, Heap, QuickSort e Counting seguindo o formato anterior. Para a realização destes testes foram apenas desativados os algoritmos SelectionSort e InsertionSort, bem como seus respectivos prints, pois seus tempos estavam atingindo valores muito altos, porém essa versão não estará disponível no github, pois é apenas para realização de teste e obtenção de resultados mais satisfatórios relacionados aos algoritmos mais eficientes. No arquivo .zip estão contidos apenas dados dos testes realizados com todos os algoritmos, que é o de 0 à 100.000 elementos.

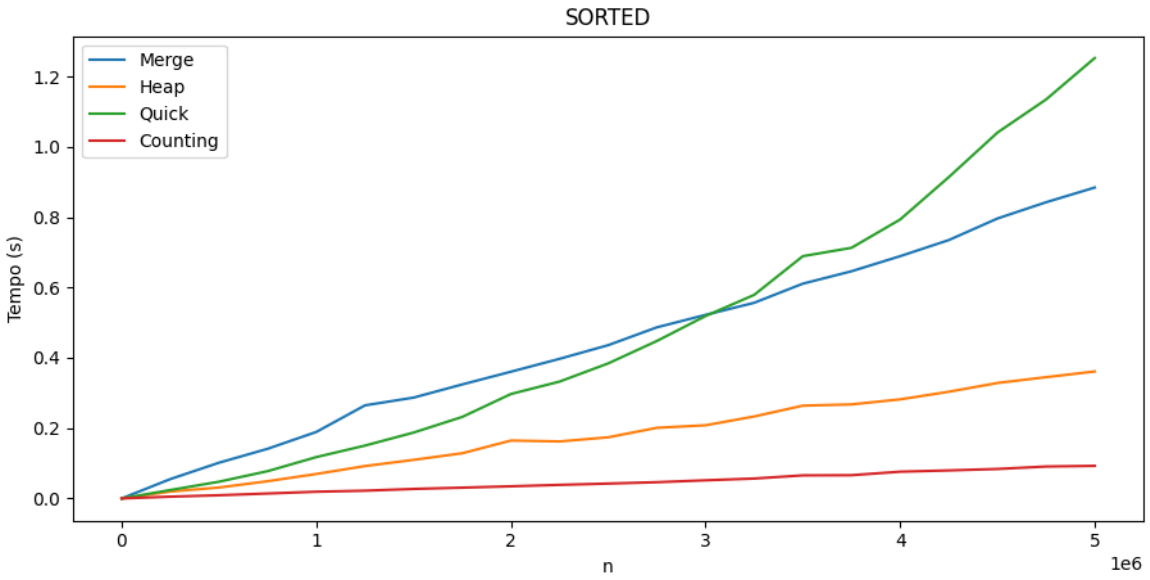
As variáveis para esses testes foram definidas com os seguintes valores: inc = 0; fim = 5.000.000; stp = 250.000; rpt = 10.



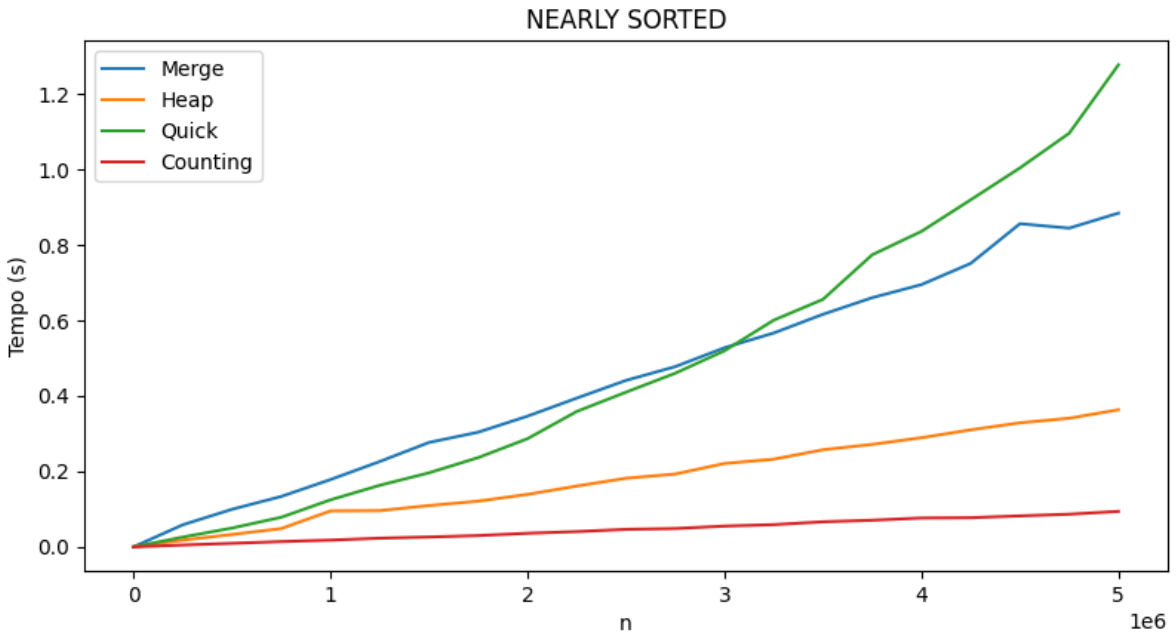
Com entradas altas, podemos perceber que um vetor com números aleatórios proporciona outra perspectiva, pois a linearidade deles desaparece. Podemos perceber que o QuickSort por um momento se mantém como segundo mais devagar, porém, conforme a entrada aumenta, percebemos que ele se torna o mais lerdo, ultrapassando até mesmo o MergeSort. Outro detalhe é que de fato o CountingSort se mantém constante, enquanto o HeapSort e o MergeSort aparentam apresentar coeficientes angulares muito similares.



Já com um vetor ordenado reversamente, o fato do QuickSort ultrapassar o MergeSort e se tornar o mais rápido se repete aqui, porém, isso acaba acontecendo mais cedo do que em vetores aleatórios, enquanto o HeapSort e o MergeSort se aproximam mais do CountingSort do que do QuickSort.



Já em um vetor ordenado, acontece o inverso do vetor reverso, o QuickSort acaba demorando mais para ultrapassar o MergeSort, enquanto o HeapSort se mantém próximo ao CountingSort, e o MergeSort se mantém mais próximo do QuickSort que do CountingSort.



Já em um vetor parcialmente ordenado, esse 10% de “desordem”, em grande escala acaba afetando muito pouco, pois os tempos acabam sendo minimamente maiores, entretanto, quando olhamos de uma perspectiva maior, percebemos que acaba não impactando significativamente, pois os gráficos (sorted e nearlly sorted) ficam praticamente idênticos.

4. Conclusões:

Com todos os dados, arquivos e gráficos gerados, nós podemos tirar várias conclusões. Primeiramente é que nem tudo é “branco ou preto”, temos uma “grande área cinza” quando o assunto é eficiência e ordenação, pois por exemplo a afirmação “o CountingSort sort é mais rápido”, e de fato essa é uma afirmação verdadeira nesses casos apresentados, porém, temos que levar em conta que, o CountingSort possui vários “pré-requisitos” para que ele possa ser de fato implantado e funcione corretamente, por exemplo, é preciso saber o tamanho do vetor, entre outros pontos; Assim como o InsertionSort, que desempenhou algumas vezes tempos expressivamente demorado, porém, ao dar um vetor ordenado, que é seu melhor caso, ele leva tempo praticamente tempo 0 para ordenar, o que é impressionante. Mas algumas afirmações que podemos concluir com esses dados é que de fato, em média, alguns algoritmos são melhores que outros na maioria dos casos, como o QuickSort e o CountingSort, pois foram os que mais permaneceram como “os mais rápidos”. Com apenas os testes até 100.00 elementos, essa afirmação seria verdade, mas assim como foi apresentado, o QuickSort a partir de certo ponto, cerca de 2.500.000/3.000.000 elementos começa a ter um desempenho inferior até mesmo ao MergeSort. Logo, a conclusão que temos é que para conseguir ter respostas concretas com esses algoritmos, precisamos realizar diversos tipos de testes, com variadas entradas, pois assim conseguimos extrair o melhor, e o pior desempenho também, de cada algoritmo de ordenação.

Outro detalhe percebido é que mesmo o QuickSort e o MergeSort, que utilizam da técnica de divisão e conquista, acabam se distanciando em determinado momento, pois mesmo que utilizem a mesma técnica “primária”, a lógica por trás de cada um é completamente diferente.

5. Referências:

Slides e materiais disponibilizados pelo professor.

Algoritmo de ordenação Countingsort (ordenação por contagem). Youtube, 22 de julho de 2020. Disponível em: https://www.youtube.com/watch?v=5enmTcTnJ7U

Classificação de pilha em 4 minutos. Youtube, 2 de agosto de 2016. Disponível em: https://www.youtube.com/watch?v=2DmK_H7IdTo

Counting Sort. Youtube,30 de abril de 2023. Disponível em: https://www.youtube.com/watch?v=EItdcGhSLf4&t=142s

Lógica do algoritmo de ordenação merge sort. Youtube,16 de fevereiro de 2018. Disponível em: https://www.youtube.com/watch?v=BnsYGiYYdnQ

Lógica do algoritmo de ordenação QuickSortsort. Youtube,16 de fevereiro de 2018. Disponível em: https://www.youtube.com/watch?v=WP7KDljG6IM

Lógica do algoritmo de ordenação selection sort. Youtube,16 de fevereiro de 2018. Disponível em: https://www.youtube.com/watch?v=pDY2rGdsYAE

Lógica do algoritmo de ordenação insertion sort. Youtube,16 de fevereiro de 2018. Disponível em: https://www.youtube.com/watch?v=7GUwzd_h3pI

Mesclar classificação em 3 minutos. Youtube,30 de julho de 2016. Disponível em: https://www.youtube.com/watch?v=4VqmGXwpLqc&t=28s