Eduardo Figueredo Pacheco

Nº USP : 13672832

Exercício Programa 3 de Algoritmos e Estrutura de Dados

2022

São Paulo

Eduardo Figueredo Pacheco
Nº USP: 13672932

- Introdução e Descrição

Com objetivos de análise crítica sobre os diferentes algoritmos de ordenação de strings, comparações e descrições específicas, esse trabalho traz um repertório grande de estudos de códigos, tempo de atuação de algoritmo, tempo máximo e mínimo de execuções em um código de vetores com palavras ordenadas, desordenadas aleatoriamente ou não. De modo a acrescentar nessa análise.

A princípio, será analisado o tempo que cada algoritmo consome no geral, o número de comparações que são feitas a depender do teste, análise do melhor e do pior caso também serão feitas nessas seções. Os resultados foram feitos com a análise dos testes que serão descritos abaixo.

Nesse Exercício programa foram testados diversos métodos para se realizar uma análise completa e relacionada aos diferentes algoritmos de ordenação. entre eles:

- Teste com 250 palavras de um texto normalmente utilizado no dia a dia, com repetições e palavras aleatórias e depois 2x o valor anterior, até onde for possível.
- Teste com 250 palavras ordenadas e depois 2x o valor anterior, até onde for possível.
- Teste com 250 palavras ordenadas de maneira oposta e depois 2x o valor anterior, até onde for possível.
- Teste com 250 palavras iguais, e depois 2x o valor anterior, até onde for possível.

Eduardo Figueredo Pacheco
Nº USP: 13672932

● Desempenho - QuickSort

O tempo depende fortemente de como a partição é feita, o que depende da escolha do pivô. Ou seja, aquele algoritmo que deixa mais desproporcionais os tamanhos dos vetores divididos é o que detém de maior tempo de execução, pois serão necessárias mais comparações para se ordenar.

É possível inferir que o desempenho do código é alterado de acordo com o número de comparações entre elementos do vetor, ou seja, o algoritmo com melhor desempenho é aquele que realiza menos comparações, que, nesse caso, são entre caracteres de uma string. Além disso, nesse projeto, como estão sendo comparadas strings de tamanhos aleatórios entre 1 e 10 caracteres, é possível extrair que, com um maior número de caracteres iguais no começo das strings, maior vai ser o tempo de execução, na medida que cada vez mais comparações serão feitas.

Levando em conta o melhor caso, que é aquele em que cada palavra tem apenas um caractere, ou que todas as palavras começam com um caractere diferente um do outro, sabe-se que o consumo médio de tempo deste algoritmo nessas circunstâncias é de O(n lg n).

Já no caso médio, percebe-se também que o tempo de execução é de O(n lg n). Isso se dá pois nos casos quaisquer, o vetor principal vai ser dividido pelo pivô separando (sendo 100% o tamanho do vetor) em x% e (100%-x%). desse modo, há um número linearítmico de comparações, assim como no melhor caso de execução analisado acima.

Por fim, o pior caso ocorre quando o pivô é, ou igual a “q”, ou igual “r”, ao passo que seria necessário realizar mais comparações que os casos normais. O tempo de execução é de $O(n^2)$, e um exemplo desse caso ocorre quando o vetor já está ordenado.

● Desempenho InsertionSort

O insertion Sort é um algoritmo de ordenação por inserção e o melhor, pior e tempo médio de execução desse algoritmo serão analisados abaixo:

No melhor caso, que ocorre quando o vetor já está ordenado, o teste do laço, enquanto é executado somente uma vez para cada valor de i do laço para, totalizando $n-1 = O(n)$ execuções.

No pior caso, é possível analisar que, no laço, ocorre a execução i vezes para cada valor de i do laço, totalizando $2 + \cdots + n = n(n + 1)/2 - 1 = O(n^2)$ execuções. Esse caso ocorre quando o vetor está em ordem decrescente.

Já no caso médio, qualquer uma das n! permutações dos n elementos pode ser o vetor de entrada. Nesse caso, cada número tem a mesma probabilidade de estar em quaisquer das posições do vetor. Assim, em média, metade dos elementos em $A[1..i - 1]$ são menores do que A[i], de modo que o laço enquanto é executado cerca de i/2 vezes, em média. De modo

que, temos em média por volta de $n(n-1)/4$ execuções do laço enquanto, e, então, no caso médio, o tempo é semelhante ao pior caso: de $O(n^2)$.

● Desempenho MergeSort

Para o MergeSort é possível analisar que o algoritmo não possui pior caso pois temos que o algoritmo executa:

- A ordenação recursiva dos $\lfloor n/2 \rfloor$ primeiros elementos da lista.
- A ordenação recursiva dos $\lfloor n/2 \rfloor$ últimos elementos da lista.
- Intercala as duas sublistas previamente ordenadas
  De modo que, então, o tempo de execução seja sempre O(n log n).

● Desempenho HeapSort

O HeapSort possui implementação:

- Troca-se o último elemento do *heap* com o primeiro.
- O novo último elemento fará parte do vetor ordenado, logo, esse deve sair do *heap*, sempre diminuindo em 1 o tamanho do HEAP.
- Como o primeiro elemento, depois de realizada a troca, provavelmente não atende às propriedades do *heap*, o corrige-descendo é acionado para reordenar o *heap* a partir da raiz modificada.

O tempo de execução do Heapsort é $O(n \lg n)$ pois, como foi demonstrado, o tempo de execução do constroiHeap é $O(n)$, adicionalmente o do corrige-descendo é $O(\lg n)$, mas ele é chamado n-1 vezes, o que torna todo o processo $O(n \lg n)$.

Eduardo Figueredo Pacheco
Nº USP: 13672932

- # Resultados

Como visto anteriormente, na introdução, os testes seguiram aqueles tópicos e os gráficos foram feitos a partir da medição do tempo de execução utilizando a função clock() da biblioteca <time.h>. Assim, é possível mensurar a eficiência de cada código a partir da análise de dados dessas tabelas e, posteriormente, dos gráficos. A princípio foi analisado o tempo de execução de cada algoritmo de ordenação, e depois a quantidade de comparações realizadas.

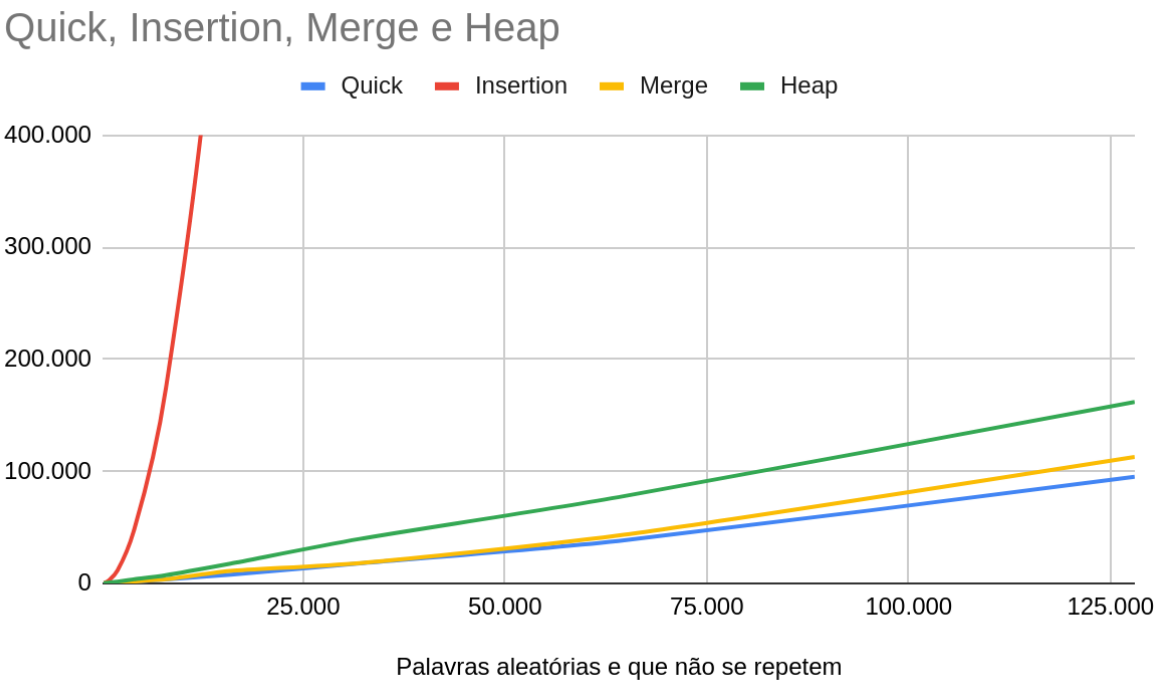
- ## Análise por tempo de execução:

Os gráficos foram medidos em diferentes textos com 250 palavras até 128.000 palavras e o tempo foi aplicado em milissegundos pela função clock() da biblioteca <time.h>. A função clock_t clock(void) retorna o tempo de processamento desde o início do programa em uma unidade de processamento. Basta dividir por CLOCKS_PER_SEC para obter o valor em segundos. time_t time(time_t *timer) salva a hora atual no formato time_t.

Além disso, a escala dos gráficos também foi ajustada para garantir melhor visualização e análise gráfica.

Eduardo Figueredo Pacheco
Nº USP: 13672932

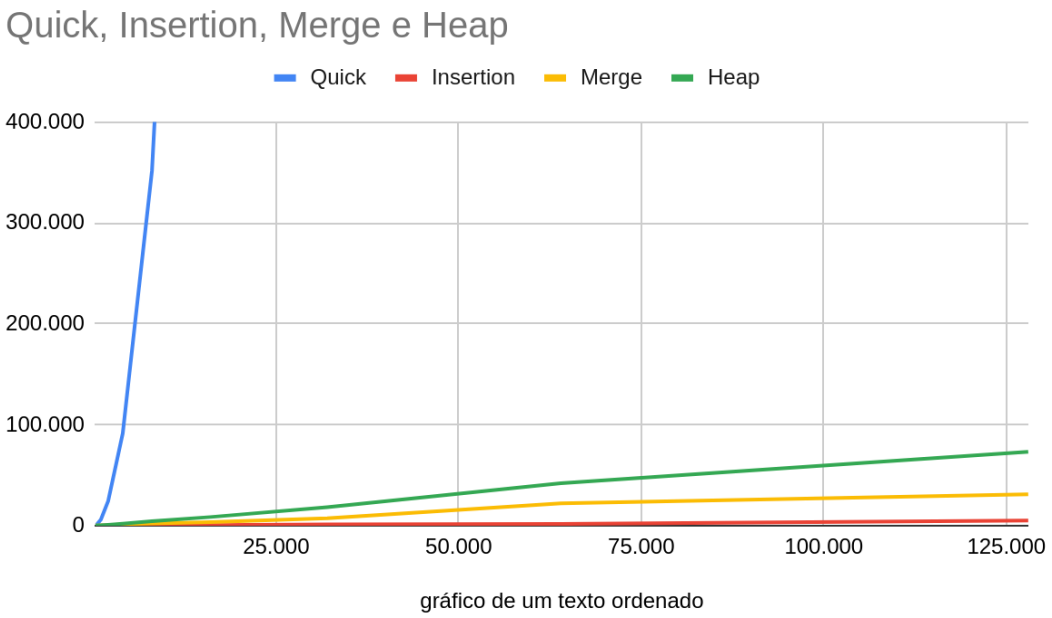
Palavras aleatórias e que não se repetem				
Nº de Palavras:	Quick	Insertion	Merge	Heap
250	62	174	94	138
500	145	675	165	315
1.000	303	3.010	507	729
2.000	694	11.635	768	1.591
4.000	1.589	46.488	1.689	3.592
8.000	3.587	174.538	3.604	7.377
16.000	7.669	679.648	11.010	17.684
32.000	17.810	2.802.314	18.119	39.685
64.000	37.739	13.182.827	42.622	76.891
128.000	95.104	62.021.400	112.819	162.015



Como visto anteriormente, já era esperado que o maior tempo de execução de textos não ordenados fossem do algoritmo do InsertionSort [$O(N^2)$] e que o tempo médio do quickSort, MergeSort e HeapSort fosse de O(n log n) .

Eduardo Figueredo Pacheco
Nº USP: 13672932

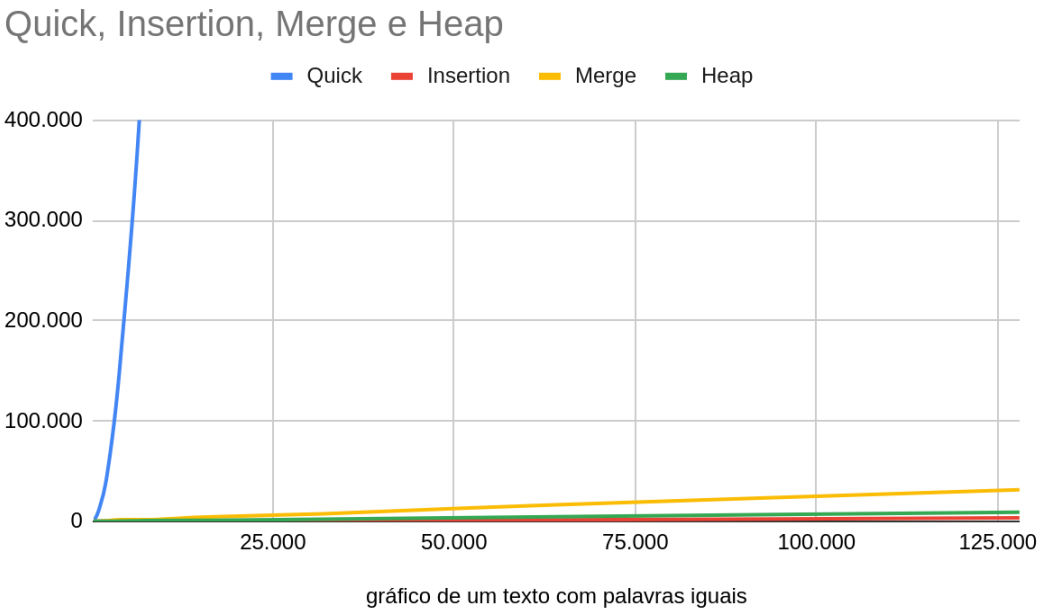
gráfico de um texto ordenado				
Nº de Palavras:	Quick	Insertion	Merge	Heap
250	468	8	50	86
500	1.645	14	110	190
1.000	6.017	27	219	389
2.000	24.418	55	391	834
4.000	91.571	176	1.312	1.945
8.000	351.918	305	2.242	4.311
16.000	1.450.306	666	3.475	8.540
32.000	6.224.860	1.271	7.204	18.195
64.000	23.467.937	1.643	22.146	42.008
128.000	92.257.638	5.139	30.957	73.273



Como visto nas análises de desempenho, já era esperado o menor tempo de execução do algoritmo insertionSort [O(N)] e o pior do Quicksort [$O(N^2)$].

Eduardo Figueredo Pacheco
Nº USP: 13672932

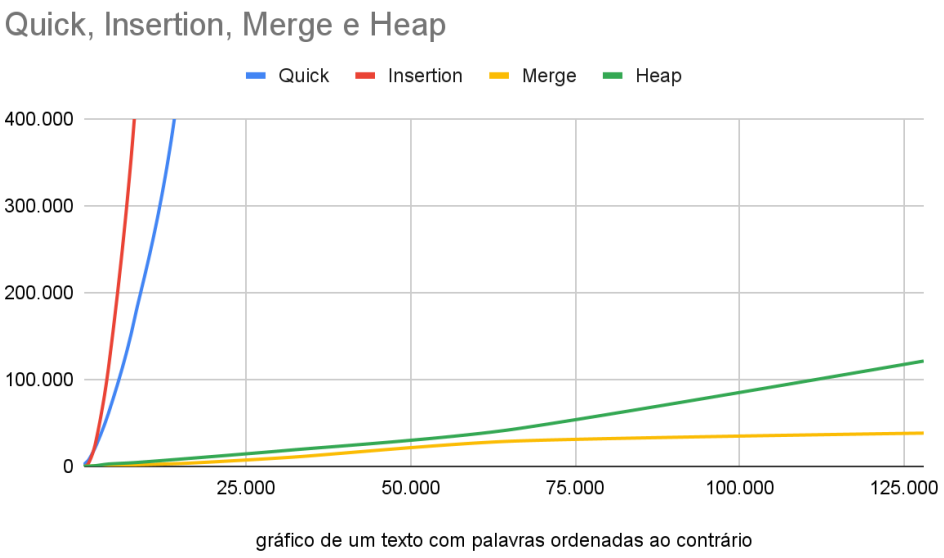
gráfico de um texto com palavras iguais				
	Quick	Insertion	Merge	Heap
250	643	7	45	18
500	2.858	12	98	54
1.000	11.432	23	193	69
2.000	40.658	48	590	146
4.000	163.667	89	1.681	280
8.000	602.612	200	1.695	866
16.000	2.635.079	616	4.346	1.033
32.000	9.668.089	699	7.600	2.150
64.000	38.440.592	1.407	16.477	4.692
128.000	155.360.710	3.604	31.528	9.001



Assim como em um texto com palavras ordenadas, o Quicksort segue no seu pior caso também para um texto com palavras iguais [$O(N^2)$], enquanto o tempo médio dos demais seguem bem menores e, já o InsertionSort segue com tempo O[n], o melhor dentre os 4 para esse caso.

Eduardo Figueredo Pacheco
Nº USP: 13672932

gráfico de um texto com palavras ordenadas ao contrário				
Nº de Palavras:	Quick	Insertion	Merge	Heap
250	882	9	51	26
500	3.452	17	105	53
1.000	7.139	3.433	218	267
2.000	21.200	24.468	413	754
4.000	60.163	110.711	873	2.698
8.000	169.487	401.768	1.771	4.328
16.000	523.136	1.396.348	3.629	8.972
32.000	2.316.858	5.171.662	10.600	18.906
64.000	6.214.341	20.891.210	28.560	41.039
128.000	26.747.684	87.398.524	38.281	121.187



Por fim, também como o esperado segundo as análises feitas anteriormente, esse caso faz com que os algoritmos de insertionSort e QuickSort sejam executados no maior tempo: $O(n^2)$.

Além de analisarmos esses diferentes casos em que alguns algoritmos têm desempenho pior que os demais, também foi perceptível a diferença entre o QuickSort e InsertionSort, e o MergeSort e HeapSort. Pois esses últimos sempre tiveram tempo de execução parecido ou igual, ao passo que há certa linearidade e constância em seus algoritmos, independente dos diversos casos.

Eduardo Figueredo Pacheco
Nº USP: 13672932

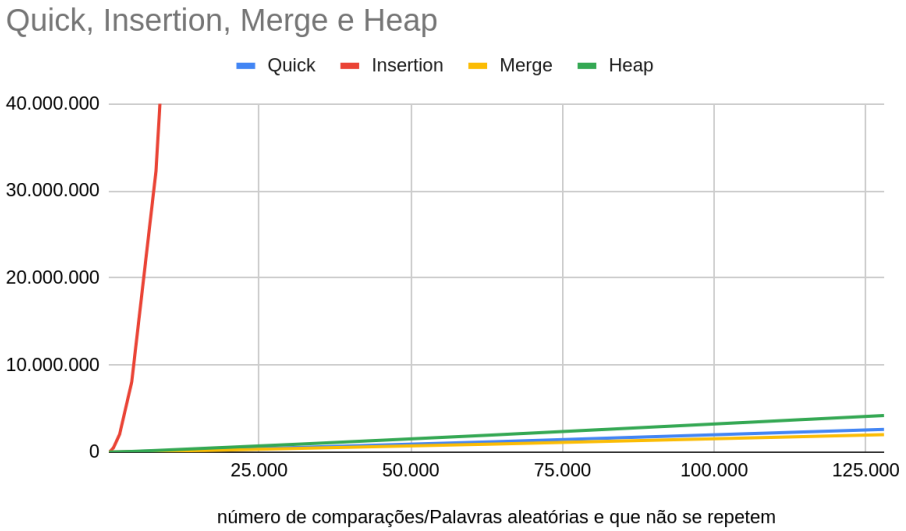
- Análise por número de comparações:

Os gráficos foram medidos em diferentes textos com 250 palavras até 128.000 palavras e o número de comparações foi medido em cada teste por meio de um contador que era acrescido a cada vez que a função “ComparaString” era chamada.

Outro fator importante é que o número de comparações às vezes pode ser impreciso, mas a coerência entre as comparações (proporcionalidade) de uma execução para outra com mais palavras deve se manter. Por isso é importante analisar os gráficos.

Além disso, a escala dos gráficos também foi ajustada para garantir melhor visualização e análise gráfica.

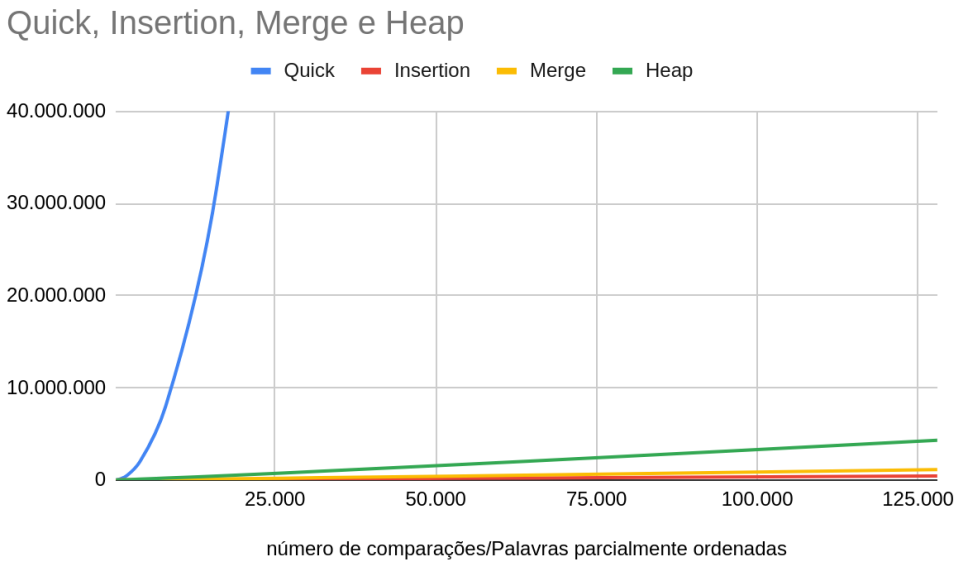
Palavras aleatórias e que não se repetem				
Nº de palavras:	Quick	Insertion	Merge	Heap
250	2.082	29.626	1.674	3.695
500	4.659	123.584	3.846	8.428
1.000	10.234	515.454	8.699	18.865
2.000	24.046	2.040.870	19.408	41.754
4.000	57.863	8.063.698	42.837	91.414
8.000	120.327	32.298.688	93.643	198.789
16.000	267.822	127.665.608	203.271	429.830
32.000	553.701	510.312.952	438.529	923.389
64.000	1.199.675	2.047.762.422	940.952	1.974.815
128.000	2.612.337	8.189.233.636	2.010.288	4.205.574



Eduardo Figueredo Pacheco
Nº USP: 13672932

Assim como já era esperado, que o maior tempo de execução de textos não ordenados fossem do algoritmo do InsertionSort, como também visto no tempo de execução o desempenho de [$O(N^2)$] e que o tempo médio do quickSort, MergeSort e HeapSort fosse de O(n log n) .

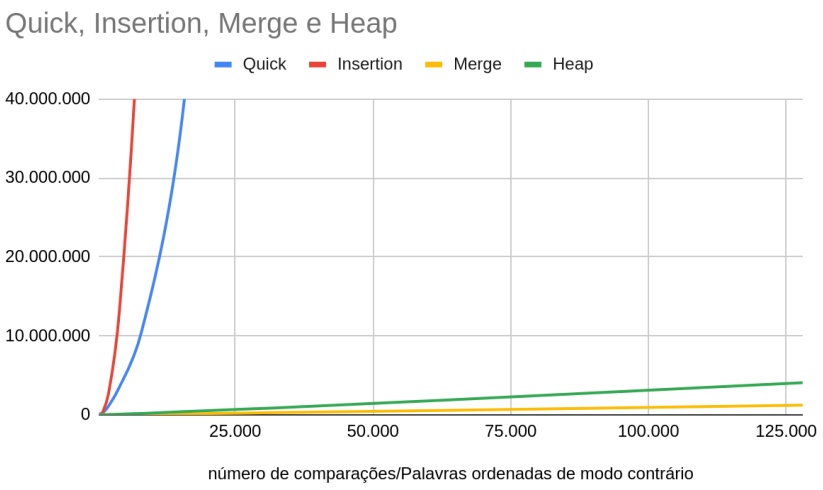
Palavras parcialmente ordenadas				
Nº de palavras:	Quick	Insertion	Merge	Heap
250	8.247	870	1.079	3.818
500	32.866	1.742	2.422	8.732
1.000	128.241	3.492	5.345	19.539
2.000	506.491	6.992	11.691	43.128
4.000	2.012.991	13.992	25.383	94.562
8.000	8.025.991	27.992	54.767	205.310
16.000	32.051.991	55.992	117.535	442.708
32.000	128.103.991	111.992	251.071	949.818
64.000	512.207.991	223.992	534.143	2.028.943
128.000	2.048.415.991	447.992	1.132.287	4.314.257



Assim como analisado anteriormente, sabe-se que quando o vetor já está ordenado, o pior desempenho é do quickSort $O(N^2)$ enquanto o melhor desempenho pertence ao insertionSort, que tem O(N). Além disso, tanto o HeapSort como o MergeSort têm seu desempenho médio de O(n log n).

Eduardo Figueredo Pacheco
Nº USP: 13672932

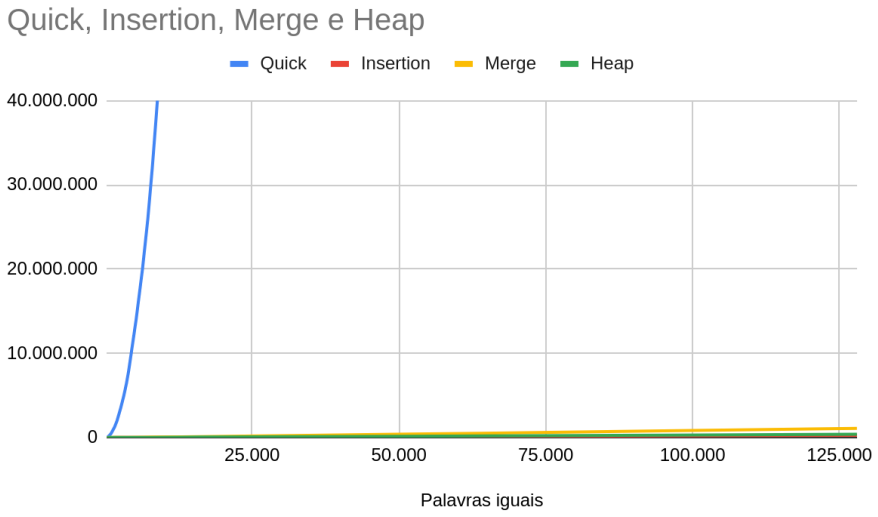
Palavras ordenadas de modo contrário				
	Quick	Insertion	Merge	Heap
250	31.125	498	1.011	747
500	124.750	998	2.272	1.497
1.000	282.942	436.474	5.520	10.679
2.000	1.102.278	2.658.340	12.179	30.101
4.000	3.548.436	13.290.684	27.392	80.094
8.000	10.534.454	58.539.560	60.381	180.587
16.000	41.454.721	245.246.944	129.279	405.769
32.000	190.982.365	1.002.104.144	273.096	875.211
64.000	540.129.720	4.052.476.266	575.668	1.901.919
128.000	2.408.536.642	16.298.626.010	1.213.863	4.078.390



Assim como no modo anterior, o quicksort também tem seu pior caso junto com o InsertionSort $O(n^2)$, enquanto os demais algoritmos seguem com seu tempo médio $O(n \log n)$.

Eduardo Figueredo Pacheco
Nº USP: 13672932

Palavras iguais				
Nº de Palavras:	Quick	Insertion	Merge	Heap
250	31.125	498	1.011	747
500	124.750	998	2.272	1.497
1.000	499.500	1.998	5.044	2.997
2.000	1.999.000	3.998	11.088	5.997
4.000	7.998.000	7.998	24.176	11.997
8.000	31.996.000	15.998	52.352	23.997
16.000	127.992.000	31.998	112.704	47.997
32.000	511.984.000	63.998	241.408	95.997
64.000	2.047.968.000	127.998	514.816	191.997
128.000	8.191.936.000	255.998	1.093.632	383.997



Já para o caso em que as palavras são iguais, é esperado e confirmado que o quicksort tem seu pior desempenho $O(N^2)$, enquanto o InsertionSort tem o seu melhor O(n) e os demais seguem com seus casos médios O(n log n).

Eduardo Figueredo Pacheco
Nº USP: 13672932

- Conclusão

A partir desses dados presentes nas tabelas e, com base na teoria analisada nos tópicos de desempenho desse documento, é possível notar que há coerência nos resultados obtidos e que demonstram que existem diversas utilidades para os inúmeros tipos de algoritmos de ordenação, tendo em vista as divergentes situações que podem ocorrer. Assim, é possível escolher a melhor opção para cada situação, sendo que os algoritmos de HeapSort e MergeSort têm um tempo médio menor que os demais e não variam muito diante dos diferentes casos, mas códigos como o do InsertionSort já possui o melhor tempo entre os demais nos casos em que o vetor já estava ordenado ou tinha muitas palavras repetidas.

É possível também analisar que os desempenhos de cada algoritmo, sejam no pior ou melhor desempenho, se mantiveram dentro da margem esperada para as situações que foram analisadas. Por exemplo, uma execução com 250 palavras tendo 432 comparações segue o modo O(n) assim como para 249 comparações.