Relatório EP3 MAC0121 - IME - USP

Daniel Silva Lopes da Costa

23 de novembro de 2020

Resumo

O presente trabalho apresenta a consolidação dos conhecimentos adquiridos no 2º semestre de 2020 na disciplina MAC0121 Algoritmos e Estruturas de Dados I. Serão comparados 4 algoritmos de ordenação: Bubblesort, Insertion sort, Mergesort, Quicksort, em diferentes cenários, a fim de entender suas complexidades, forças e limitações.

Sumário

1 Introdução

A ordenação é organização de um grupo de objetos, números, palavras, estruturas, em ordem crescente ou decrescente. Ter as informações ordenadas é de suma importância quando você quer fazer a busca por um elemento específico ou fazer análises estatística como mediana e quartis, assim como ter os dados ordenados pode representar um grande ganho em diversos algoritmos e heurísticas. Um exemplo disso é o Permutation Flow Shop Problem, que envolve um número de tarefas que precisam passar em sequência por várias máquinas, nesse problema o objetivo é encontrar o menor makespan, ou seja o menor tempo para finalização da última tarefa, tal problema é no-hard, portanto não pode ser resolvido de maneira ótima, a solução é implementar um heurísticas, nesse sentido ter as tarefas ordenadas influencia diretamente na otimização da heurística e quão bom será o resultado. Diante disso fica claro a importância de ter os conjuntos de dados ordenados. Para tanto pode-se usar algoritmos de ordenação, existem vários algoritmos com foco em ordenação. Nesse relatório vamos analisar quatro desse algoritmos: Bubblesort, Ordenação por inserção, Mergesort, Quicksort, analisando sua complexidade em diferentes situações, com listas formadas por conjuntos em ordem crescente, em ordem decrescente, semi-ordenado, constante e aleatório.

1.1 Motivação

Como falado acima, os algoritmos de ordenação são suma importância para a computação, com implicações em diversos problemas da realidade, organização de bancos de dados, buscas e heurísticas para problemas np-hard. Diante disso, entende-los e saber os pontos forte e fracos de cada solução é essencial para um bom cientista da computação. Portanto a motivação principal é entender mais sobre cada uma das soluções, identificando seus pontos forte e fracos de cada modelo.

1.2 Objetivos

Nas aulas em classe chegamos a provas da complexidade de cada algoritmo, em quais casos ele é mais eficiente ou não. O objetivo desse relatório é traçar comparações entre essas diferentes soluções e testar se para casos reais se algoritmo realmente possui a mesma complexidade e comportamento que foi obtido na teoria. Além disso serão gerados gráficos e tabelas a fim de comparar cada tipo de solução com as demais, facilitando assim a identificação das melhores soluções e em quais situações cada algoritmo se sai melhor.

2 Materiais e métodos

Para geração dos algoritmos foram utilizados a linguagem C, uma das mais tradicionais linguangens de programação. Foram usadas as seguintes bibliotecas padrão do C:

- stdio.h - Para gerenciar os comandos de entrada e saída

- stdlib.h - Para alocar espaço na memórias paras as listas e gerar os número aleatórios

Já na confecção dos gráficos e tabelas foi utilizado Jupyter Notebook com a linguagem python, foram usadas as seguintes bibliotecas:

- pandas - Para a geração das tabelas.

- matplotlib.pyplot - Para gerar os gráficos.

2.1 Instâncias

As quatro soluções foram testadas para cinco instância diferentes: crescente, decrescente, semi-crescente, aleatório e constantes. Em cada um delas, foram realizados testes com diferentes tamanhos de lista: 250, 500, 1000, 2000, 4000, 8000 e 16000 termos. A seguir temos as características e como foi formada cada uma dessas instâncias:

- Crescente - Foi criada uma lista que vai de 0 até n-1, em ordem crescente. Vale destacar que eu alguns gráficos e tabelas para representar crescente, abreviou-se a palavra para "cresc". Exemplo, para n=250 temos:

$$listaCresc = [0, 1, 2, 3, 4, ...100, 101, 102, ...245, 246, 247, 248, 249]$$

- Decrescente - Foi criada no laço do for, uma lista que vai de n-1 até 0, sempre diminuindo 1. A representação abreviada para a lista decrescente é "decresc". Exemplo, para n = 250 temos:

$$listaDecresc = [249, 248, 247, 246, 245, ..., 102, 101, 100, ..., 4, 3, 2, 1, 0]$$

- Semi-Crescente - Para essa instância foi criada uma lista ordenada crescente igual a anterior, só que em seguida foi realizada a troca entre alguns elementos aleatórios. Para tanto, foi utilizada uma função que sorteia dois número entre 0 e n-1, e troca os elementos correspondente aos índices sorteados na lista. Esse processo é repetido n/10 vezes, portanto temos no máximo 20% dos elementos trocados e o restante fica ordenado de maneira crescente. Cabe ainda destacar que foi usado na representação "semi-cresc", em algumas tabelas para representar essa lista. Exemplo, para n=250 digamos que foi sorteado os índices 2 e 101, então teríamos:

$$listaSemi - cresc = [0, 1, 101, 3, 4, ...100, 2, 102, ...245, 246, 247, 248, 249]$$

- Aleatório - Foi usado s função sorteia que possui como base a função random e foram gerados números entre -n e n, de maneira pseudo-aleatória. Não foi usado uma chave de tempo para aleatorizar, pois mesma sequência foi gerada várias vezes para diferentes métodos de ordenação. Foi definido "aleat"como abreviação para a instância aleatória. Exemplo, para n=250 temos portanto:

$$listaAleat = [(n < x_0 < n), (n < x_1 < n), ...(n < x_{100} < n), ..., (n < x_249 < n)]$$

- Constante - Foi preenchido todas as posições do vetor com 1. A representação de constante foi "const". Exemplo, para n=250 temos:

$$listaConst = [1, 1, 1, 1, 1, 1, ..., 1, 1, 1, 1, 1, 1, ..., 1, 1, 1, 1]$$

Segue alguns valores importantes para a análise dos resultados e identificação da complexidade:

n	n^2	$n(n-1) \div 2$	$nlog_2(n)$
250	62500	31125	$1991,44$
500	250000	124750	$4482,89$
1000	1000000	499500	$9965,78$
2000	4000000	1999000	$21931,56$
4000	16000000	7998000	$47863,13$
8000	64000000	31996000	$103726,27$
16000	256000000	127992000	$223452,54$
32000	1024000000	511984000	$478905,09$

2.2 Algoritimos de ordenação

Foram usadas duas funções auxiliares que se repetiram em vários comandos, a função sorteia e a função trocas.

- Função Sorteia - Recebe dois valores que representa o intervalos onde vai ser sorteado os números, e faz interpolação com a função rand(), da biblioteca stdib.h.

- Função Troca - Recebe um vetor e o índice dos dos termos que serão trocados, um deles é salvo em uma variável auxiliar, são realizadas três movimentações.

- Bubblesort - O algoritmo de bolhas, percorre todo o vetor várias vezes sempre comparando dois valores consecultivos deixando os dois ordenados quando necessário. Dessa forma ele leva sempre os maiores valores para o final de ele começar sempre no início. Existem maneiras mais sofisticadas de implementá-lo mas para os fins deste relatório, a implementação mais simple do bubblesort é suficiente. No melhor caso quando o vetor já está ordenado ele realiza n-1 comparações e nenhuma troca. Já no pior caso, com a lista em ordem decrescente temos $\frac{n(n-1)}{2}$ comparações e $\frac{n(n-1)}{2}$ trocas.

	Pior Caso	Complexidade
Comparações	$\frac{n(n-1)}{2}$	$O(n^2)$
Movimentações	$\frac{n(n-1)}{2}$	$O(n^2)$

- Ordenação por inserção - A inserção vai ordenando uma parte do vetor a cada passo, ele pega um primeiro elemento da parte desordenada e encontra em que lugar ele ficaria na parte ordenada. No melhor dos casos esse algoritmo realiza n-1 comparações e 0 trocas quanto o vetor já está ordenado de maneira crescente. Para o pior caso temos que $\frac{n(n-1)}{2}$ comparações e $\frac{n(n-1)}{2}$ trocas, quando o vetor está ordenado de maneira crescente. Tem como melhorar esse algoritmo implementando uma busca binária, o que reduz o número de comparações, mas nesse caso, foi utilizado o algoritmo mais simples.

	Pior Caso	Complexidade
Comparações	$\frac{n(n-1)}{2}$	$O(n^2)$
Movimentações	$\frac{n(n-1)}{2}$	$O(n^2)$

- Mergesort - Um dos algoritmos mais eficiente, utiliza um conceito de partição do vetor em tamanhos menores e assim realizar a ordenação, mesclando as partes separadas. Apesar da grande eficiência tal algoritmo utiliza espaço extra de memória. No pior caso temos que o número de comparações e o número de trocas é limitado superiormente por $nlog_2n$.

	Complexidade
Comparações	$O(nlog_2n)$
Movimentações	$O(nlog_2n)$

- Quicksort - Outro algoritmo muito eficiente e famoso, assim como o merge ele usa uma ideia de particionar o vetor, só que não usa espaço extra da memória. Nesta impelementação foi usada a implementação do Particiona do Sedric. Outra particularidade é que foi utilizado o pivô aleatório. Foi demonstrado em aula que a complexidade dessa algoritmo no caso médio, ou seja quando temos uma vetor aleatório é de $O(nlog_2n)$.

	Complexidade
Comparações	$O(nlog_2n)$
Movimentações	$O(nlog_2n)$

2.3 Parâmetros

Vale destacar quais parâmetros serão utilizados para metrificar cada algoritmo e assim estabelecer comparações. Serão computados a quantidade de comparações e a quantidade de movimentações, uma vez que isso indicaria o tempo computacional do algoritmo, uma vez que esses todas essas soluções são em base comparações e movimentações no vetor. Em relação a comparação, sempre que comparamos dois elementos do vetor, ou um elemento do vetor com um fator externo é contabilizado uma comparação. Se tivermos uma cadeia formada por if e else, é contabilizado apenas uma comparação. Sobre movimentação, foram contadas todas as vezes em que o algoritmos colocar um valor no vetor de interesse ou tira um elemento dele, por isso quando realizamos um troca, entre dois elementos do mesmo vetor são contabilizados 3 movimentações.

3 Resultados Experimentais

Os algoritmos foram implementados e serão analisados sob duas perspectivas, primeiramente será analisado cada solução separadamente para diferentes instâncias em seguida serão comparadas diferentes os algoritmos para a mesma instância.

3.1 Comparação:Algoritmo de ordenação para instâncias distintas

Nessa seção vamos analisar cada algoritmo de ordenação separadamente, comparando para as diferentes instâncias, como o algoritmo se comporta, em quais caso ele é mais efeiciente, ou menos. O gráficos foram feitos apenas para três insstâncias pois como é possível perceber o padrão sempre se repete para os diferentes valores das istâncias, portanto foi selecionado um intervalo mais restrito para as análises para melhor vizualização das informações contidas no gráfico.

3.1.1 Bubblesort

Com o Bubblesort obtivemos as seguintes resultados para as movimentações e comparações:

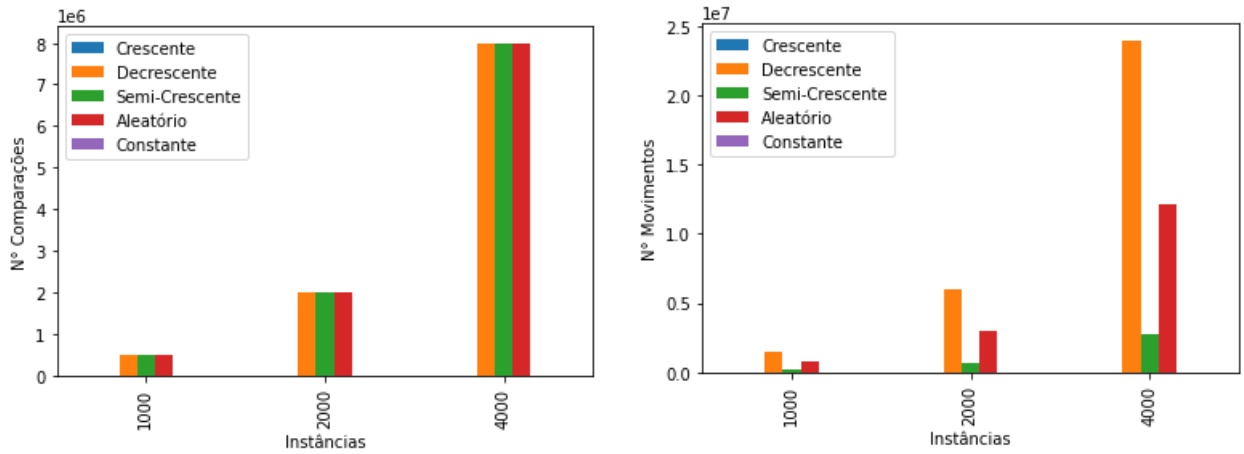
Figura 1: Bubblesort: Comparações(esquerda) e Movimentações(direita)

	Cresc	Decresc	Semi-Cresc	Aleat	Const
250	249	31125	30719	30915	249
500	499	124750	122920	124425	499
1000	999	499500	497960	499290	999
2000	1999	1999000	1990222	1996722	1999
4000	3999	7998000	7996824	7996289	3999
8000	7999	31996000	31844475	31988497	7999
16000	15999	127992000	127759779	127971090	15999
32000	31999	511984000	511459200	511973560	31999

	Cresc	Decresc	Semi-Cresc	Aleat	Const
250	0	93375	10707	47256	0
500	0	374250	44946	183456	0
1000	0	1498500	179496	776760	0
2000	0	5997000	608760	3006603	0
4000	0	23994000	2731314	12095832	0
8000	0	95988000	11061054	48360732	0
16000	0	383976000	45327930	192776817	0
32000	0	1535952000	179056428	766544715	0

Pela análises desse valores é possível perceber que o melhor caso, é quando a sequência já está ordenada de maneira crescente ou é constante, pois no caso temos n-1 comparações e zero trocas o que confirma a análise teórica. Além disso, vale destacar que no pior caso ele realmente realiza $\frac{n(n-1)}{2}$ comparações e $\frac{n(n-1)}{2}$. Com base nisso é possível destacar que os resultados teóricos obtidos em aula, são realmente verdadeiros, e que o Bubblesort é uma algoritmo extremamente eficiente para crescente, constantes ou semi-ordenados que é onde ele se saiu melhor em número de comparações e trocas, como fica evidente no gráfico a seguir:

Figura 2: Bubblesort: Comparações(esquerda) e Movimentações(direita)



Outro fato interessante é que na instância semi-crescente ele realizou quase o mesmo tanto de comparações que os piores casos, mas isso é compensado na número de movimentações que é bem menor do que o caso com vetor decrescente. Com isso podemos concluir que as instâncias para as quais o algoritmo é mais eficiente para essas instâncias é a seguinte, começando com os casos de maior eficiência e finalizando com o pior caso:

$$Crescente = Constante > semi - crescente > Aleatorio > Decrescente$$

Vale destacar ainda que existem forma de melhorar o bubblesort, a fim de torná-lo mais eficinte para o pior dos casos, como aliar dois bubblesort um indo e outro voltando, o que é denominado shakersort.

3.1.2 Ordenação por inserção

/ Sobre a Ordenação por inserção foram contabilizados os seguintes valores para comparações e trocas:

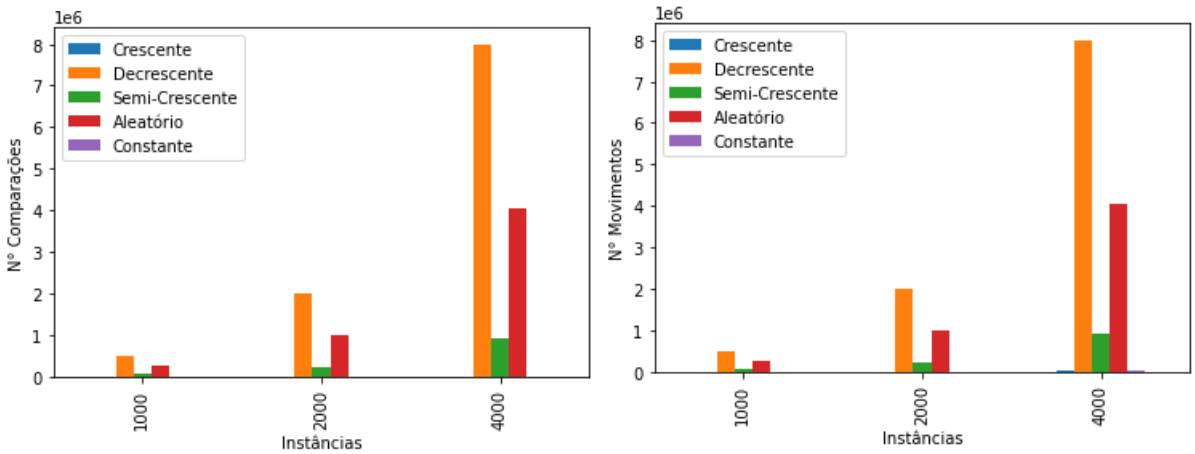
Figura 3: Bubblesort: Comparações(esquerda) e Movimentações(direita)

	Cresc	Decresc	Semi-Cresc	Aleat	Const
250	249	31125	3818	15995	249
500	499	124750	15481	61644	499
1000	999	499500	60831	259914	999
2000	1999	1999000	204919	1004191	1999
4000	3999	7998000	914435	4035937	3999
8000	7999	31996000	3695017	16128237	7999
16000	15999	127992000	15125309	64274930	15999
32000	31999	511984000	59717475	255546893	31999

	Cresc	Decresc	Semi-Cresc	Aleat	Const
250	498	31623	4067	16250	498
500	998	125748	15980	62150	998
1000	1998	501498	61830	260918	1998
2000	3998	2002998	206918	1006199	3998
4000	7998	8005998	918436	4039942	7998
8000	15998	32011998	3703016	16136242	15998
16000	31998	128023998	15141308	64290937	31998
32000	63998	512047998	59749474	255578903	63998

Pela tabela fica evidente que o melhor caso para a inserção é quando ele é crescente ou constante onde ele realiza n-1 comparações e 2(n-1) movimentações e o pior caso é quando o vetor está ordenado de maneira decrescente, onde temos $\frac{n(n-1)}{2}$ comparações e algo muito próximo de $\frac{n(n-1)}{2}$ movimentações. No gráfico a seguir fica evidenciado essa relação:

Figura 4: Inserção: Comparações(esquerda) e Movimentações(direita)



É perceptível a eficiência do algoritmo quando ele já está ordenado de maneira crescente e constante, uma vez que nesses caso os valores são tão pequenos que nem são visíveis nos gráfico. Nesse caso temos que a efeciência segue a seguinte ordem de maior para menor:

$$Crescente = Constante > Semi - ordenado > Aleatorio > Decrescente$$

3.1.3 Mergesort

Sobre o mergesort foram contabilizados os seguintes resultados para o número de comparações e movimentações:

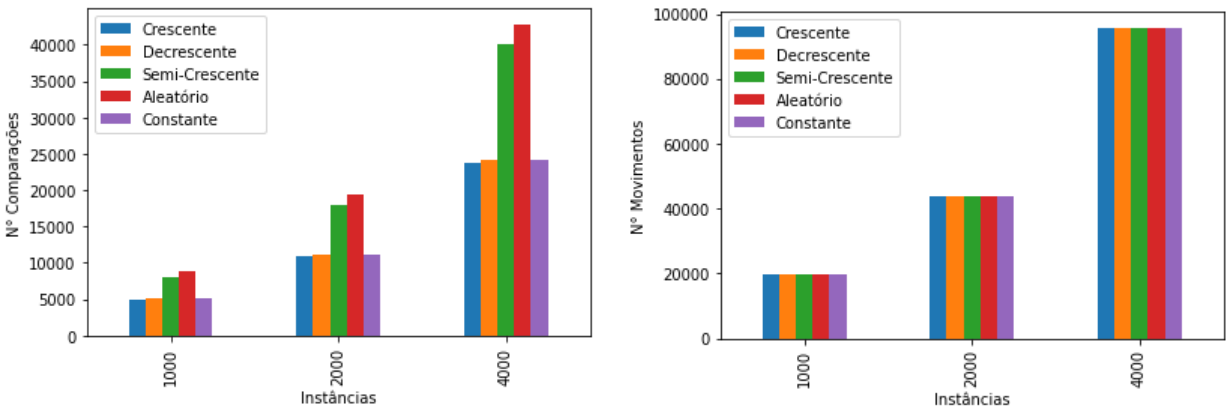
Figura 5: Mergesort: Comparações(esquerda) e Movimentações(direita)

	Cresc	Decresc	Semi-Cresc	Aleat	Const
250	983	1011	1523	1659	1011
500	2216	2272	3482	3845	2272
1000	4932	5044	7991	8727	5044
2000	10864	11088	17957	19427	11088
4000	23728	24176	40041	42846	24176
8000	51456	52352	88470	93642	52352
16000	110912	112704	192245	203228	112704
32000	237824	241408	415438	438587	241408

	Cresc	Decresc	Semi-Cresc	Aleat	Const
250	3988	3988	3988	3988	3988
500	8976	8976	8976	8976	8976
1000	19952	19952	19952	19952	19952
2000	43904	43904	43904	43904	43904
4000	95808	95808	95808	95808	95808
8000	207616	207616	207616	207616	207616
16000	447232	447232	447232	447232	447232
32000	958464	958464	958464	958464	958464

Nota-se que o melhor caso do merge é quando o vetor está ordenado de maneira crescente, pois é o caso com o menor número de comparações, além disso vale destacar que o número de movimentações é constantes para todas os casos. Podemos identificar que em todos os casos o número de comparações é sempre menor de $nlog_2(n)$, e que o número de movimentações é sempre $2nlog_2(n)$, para todos os casos. Podemos comparar melhor os resultados encontrados nos gráficos a seguir:

Figura 6: Mergesort: Comparações(esquerda) e Movimentações(direita)



É perceptível que o merge é mais eficiente para os casos ordenados ou constantes, mas nos casos aleatório e semi-crescente apesar de um pouco menos eficiente, ele consegue ordenar com

um número não muito alto de comparações. Temos a seguinte sequência de eficiência para o mergesort, do caso em que ela é mais eficiente, para o caso de menor eficiência:

$$Crescente > Constante > Decrescente > Semi - ordenado > Aleatorio$$

Portanto, fica claro a consistência do algoritmo merge, que não é extremamente bom para nenhuma das instâncias, mas também não é extremamente ruim, tendo sempre o número de comparações e troca com complexidade $O(log_2(n))$.

3.1.4 Quicksort

Com relação ao algoritmo quicksort, tivemos os seguintes resultados na análise de comparações e movimentações:

Figura 7: Quicksort: Comparações(esquerda) e Movimentações(direita)

	Cresc	Decresc	Semi-Cresc	Aleat	Const
250	31125	31125	4482	2036	31125
500	124750	124750	7458	4850	124750
1000	499500	499500	21282	12892	499500
2000	1999000	1999000	63203	25377	1999000
4000	7998000	7998000	158549	56021	7998000
8000	31996000	31996000	249461	125327	31996000
16000	127992000	127992000	670105	256490	127992000
32000	511984000	511984000	1179823	618659	511984000

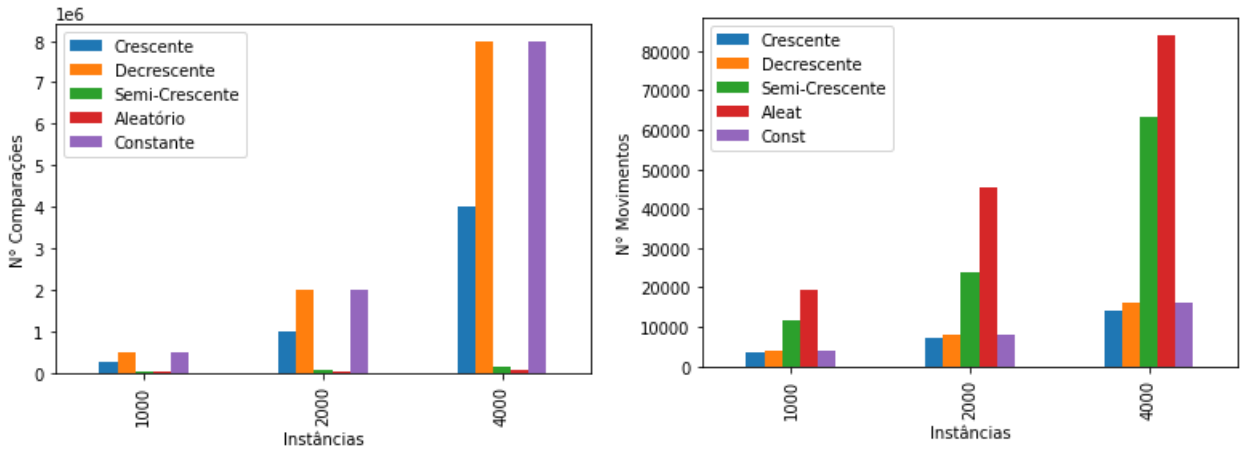
	Cresc	Decresc	Semi-Cresc	Aleat	Const
250	996	996	2666	3469	996
500	1996	1996	4274	6863	1996
1000	3996	3996	9697	19438	3996
2000	7996	7996	25618	43342	7996
4000	15996	15996	60164	83960	15996
8000	31996	31996	148345	166323	31996
16000	63996	63996	272607	387778	63996
32000	127996	127996	776773	982502	127996

Com isso, é perceptível a eficiência do algoritmo para o caso aleatório onde temos, aproximadamente $nlog_2(n)$ comparações e $2nlog_2(n)$ movimentações, este resultado foi alcançado em aula e mostra que no caso médio com vetor aletatório o quicksort é extremamente eficiente. Outro ponto interessante é o número de comparações para os casos ordenados e o constante, que foram extremamente elevados, como fica evidente nos gráficos abaixo:

$$Aleatorio > Semi - ordenado > Crescente > Decrescente > Constante$$

Vale destacar que essa ineficiência do quicksort com os algoritmos ordenados deve-se ao fato de o pivô ser aleatório, se o pivô começasse do início a melhor sequência mais rápida seria a ordenada crescente, de maneira análoga um vetor com pivô no fim seria muito eficiente para listas em ordem decrescente.

Figura 8: Quicksort: Comparações(esquerda) e Movimentações(direita)



O número de comparações no algoritmo ordenado e constante é na ordem de $O(n^2)$ o que é extremamente ruim, mas como geralmente não queremos ordenar algo que já está ordenado, e na maioria das vezes temos uma vetor aleatório, para a maioria das aplicações práticas ele é muito eficaz. Segue a relação de eficiência, como o número de movimentações não é tão díspar, o número de trocas pesou mais para essa relação:

3.2 Comparação: Entre algoritmos com instâncias distintas

Nessa seção vamos comparar as soluções entre si para cada uma das instâncias a fim de identificar qual algoritmo é mais eficiente em cada caso, novamente serão apresentados gráficos para as instâncias 1000, 2000, 4000, mas é perceptível que eles seguem um padrão portanto não a necessidade de analisar para as demais quantidades calculadas.

3.2.1 Crescente

Para a lista ordenada crescente temos as seguintes tabelas com os resultados das comparações e movimentações:

Figura 9: Crescente - Comparações(esquerda) e Movimentações(direita)

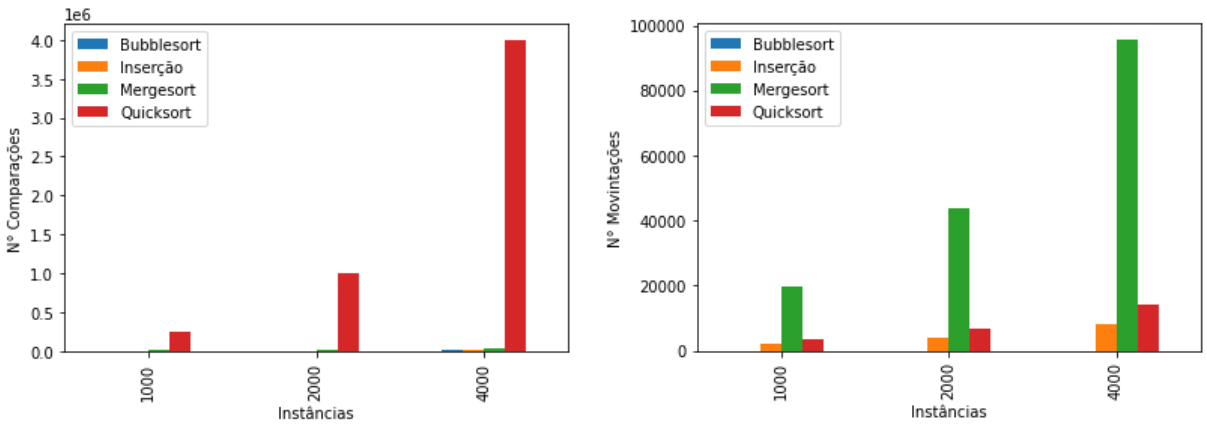
	Bubble	Inserção	MergeSort	Quicksort
250	249	249	983	31125
500	499	499	2216	124750
1000	999	999	4932	499500
2000	1999	1999	10864	1999000
4000	3999	3999	23728	7998000
8000	7999	7999	51456	31996000
16000	15999	15999	110912	127992000
32000	31999	31999	237824	511984000

	Bubble	Inserção	MergeSort	Quicksort
250	0	498	3988	996
500	0	998	8976	1996
1000	0	1998	19952	3996
2000	0	3998	43904	7996
4000	0	7998	95808	15996
8000	0	15998	207616	31996
16000	0	31998	447232	63996
32000	0	63998	958464	127996

É possível perceber nesse caso, a superioridade do Bubblesort, que realiza a menor quantidade de comparações e de movimentações, tal resultado já era esperado, pois esse é o melhor caso para esse algoritmos. Outro algoritmo também muito eficiente nesse caso é o de ordenação por inserção. Apesar de o inserction não realizar trocas nesse caso ele realiza duas movimentações a cada chamada, pois o elemento do vetor é salvo em um vetor auxiliar e depois recolocado no mesmo lugar do vetor, tal fato explica o número de movimentações ser 2(n-1).

Outro resultado interessante é como os algoritmos Mergesort e Quicksort são ruins para essa instância, especialmente o quicksort que realiza um número gigantesco de comparações, na ordem de $O(n^2)$ comparações. O mergesort como se previa teve com resultado das comparações sempre valores menores que $nlog_2(n)$ e as movimentações foi sempre igual a $2nlog_2(n)$ que é da ordem de $O(nlog_2(n))$.

Figura 10: Crescente - Comparações(esquerda) e Movimentações(direita)



Diante desse resultados fica claro quais algoritmos são mais eficientes, nesse caso temos do

melhor algoritmo para o pior em número de comparações e movimentações a seguinte sequência:

$$Bubblesort > Inserction > Mergesort > Quicksort$$

Reforçando que um abordagem diferente do quicksort poderia fazê-lo funcionar para essa instância de maneira mais eficaz.

3.2.2 Decrescente

Em relação ao caso decrescente temos os seguintes resultados para a quantidade de comparações e movimentações:

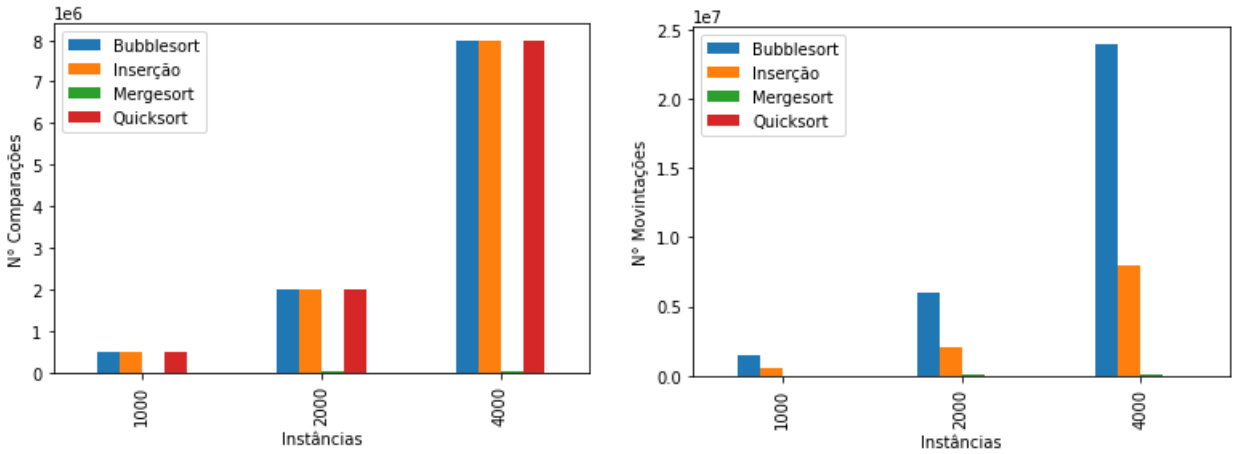
Figura 11: Decrescente - Comparações(esquerda) e Movimentações(direita)

	Bubble	Inserção	Mergesort	Quicksort
250	31125	31125	1011	31125
500	124750	124750	2272	124750
1000	499500	499500	5044	499500
2000	1999000	1999000	11088	1999000
4000	7998000	7998000	24176	7998000
8000	31996000	31996000	52352	31996000
16000	127992000	127992000	112704	127992000
32000	511984000	511984000	241408	511984000

	Bubble	Inserção	MergeSort	Quicksort
250	93375	31623	3988	996
500	374250	125748	8976	1996
1000	1498500	501498	19952	3996
2000	5997000	2002998	43904	7996
4000	23994000	8005998	95808	15996
8000	95988000	32011998	207616	31996
16000	383976000	128023998	447232	63996
32000	1535952000	512047998	958464	127996

Como a instância decrescente é o pior cenário para o bubblesort e para a ordenação por inserção, temos que a estabilidade do mergesort de sobressai com o número de comparações sempre menor que $nlog_2(n)$, enquanto os outros três algoritmos realizam aproximadamente $\frac{n(n-1)}{2}$ comparações. Já em relação a movimentações temos que o Bubblesort e o algoritmo de inserção realizam respectivamente $\frac{3n(n-1}{2}$ e $\frac{n(n-1)}{2}$, enquanto merge realiza $2nlog_2(n)$ movimentações e o quicksort realiza menos que $nlog_2(n)$. Tais resultados ficam evidenciados nas gráficos a seguir:

Figura 12: Decrescente - Comparações(esquerda) e Movimentações(direita)



Pela interpretação dos gráficos e tabelas para essa instância a ordem de eficiência do maior para o menor é :

$$Mergesort > Quicksort > Inserction > Bubblesort$$

Novamente vale destacar que é muito pouco provável que instância desse tipo ocorram no mundo real, e essa análise tem muito mais um viés de entendimento teórico dos melhores e piores casos de cada algoritmo.

3.2.3 Semi-Crescente

Para a lista como 80% ou mais dos seus elementos ordenados em ordem crescente temos os seguinte resultados para o número de comparações e movimentações:

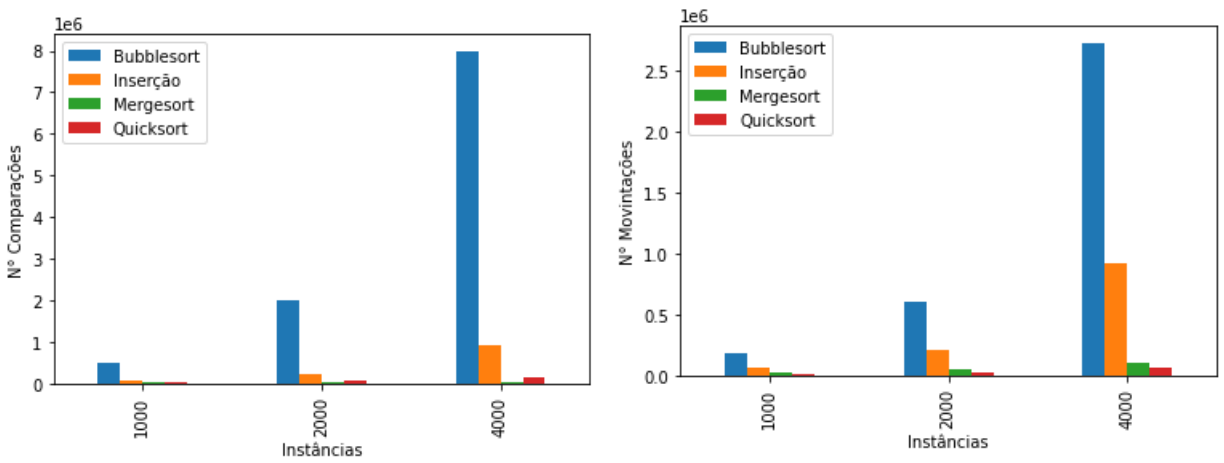
Figura 13: Semi-Crescente - Comparações(esquerda) e Movimentações(direita)

	Bubble	Inserção	MergeSort	Quicksort
250	30719	3818	1523	4482
500	122920	15481	3482	7458
1000	497960	60831	7991	21282
2000	1990222	204919	17957	63203
4000	7996824	914435	40041	158549
8000	31844475	3695017	88470	249461
16000	127759779	15125309	192245	670105
32000	511459200	59717475	415438	1179823

	Bubble	Inserção	MergeSort	Quicksort
250	10707	4067	3988	2666
500	44946	15980	8976	4274
1000	179496	61830	19952	9697
2000	608760	206918	43904	25618
4000	2731314	918436	95808	60164
8000	11061054	3703016	207616	148345
16000	45327930	15141308	447232	272607
32000	179056428	59749474	958464	776773

Diante do exposto é cabível destacar a eficiência do mergesort e do quicksort, que realizam trocas e movimentações com complexidade $O(nlog_2(n))$, ambos funcionam muito bem. A ordenação pro inserção apesar de melhor que o bubblesort é bem puco eficientes como pode-se conculir pelo gráfico abaixo:

Figura 14: Semi-Crescente - Comparações(esquerda) e Movimentações(direita)



Com base nisso podemos traçar a relação de eficiência entre os algoritmos da seguinte forma, baseado no número de comparações e movimentações:

$$Mergesort > Quicksort > Insertion > Bubblesort$$

Esse caso já é mais possível de acontecer em situações reais, onde se sabe que a sequência está parcialmente ordenada, com apenas alguns termos fora do lugar, para esse caso os algoritmos mais indicados são o mergesort e o quicksort, vale destacar ainda que a escolha entre o merge e quick depende muito das características do problema, o merge faz uso de memória auxiliar, enquanto o quicksort não, isso pode ser uma fator a ser levado em consideração. Outro fator é a quantidade de termos desordenados, pois como já vimos acima quando a lista está ordenada o quicksort é bem ineficiente, portanto se temos uma lista quase ordenada a melhor escolha seria o merge.

3.2.4 Aleatório

Com relação a instância de número aleatórios que foram gerados entre $(-n, n)$ temos os seguintes índices para a quantidade de comparações e trocas:

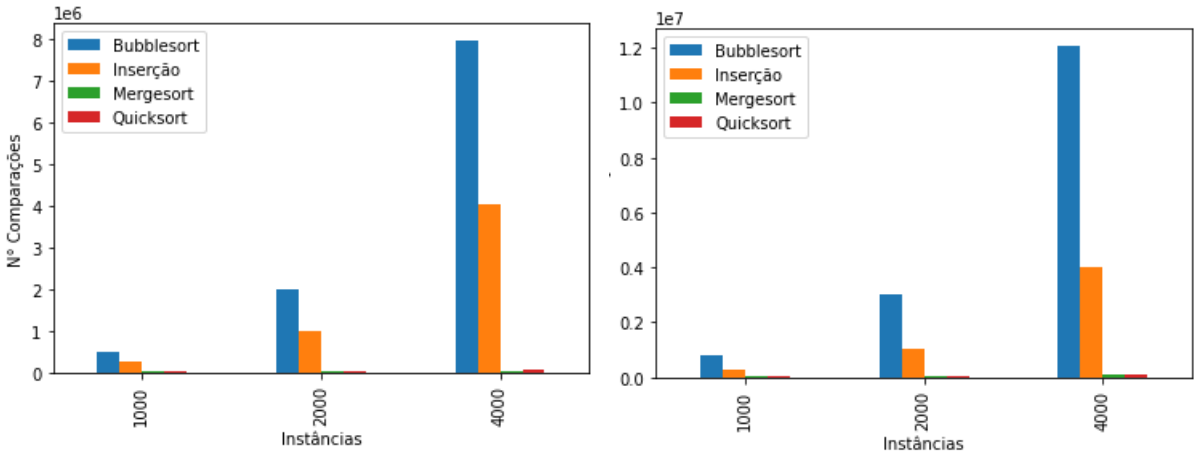
Figura 15: Aleatório - Comparações(esquerda) e Movimentações(direita)

	Bubble	Inserção	MergeSort	Quicksort
250	30915	15995	1659	2036
500	124425	61644	3845	4850
1000	499290	259914	8727	12892
2000	1996722	1004191	19427	25377
4000	7996289	4035937	42846	56021
8000	31988497	16128237	93642	125327
16000	127971090	64274930	203228	256490
32000	511973560	255546893	438587	618659

	Bubble	Inserção	MergeSort	Quicksort
250	47256	16250	3988	3469
500	183456	62150	8976	6863
1000	776760	260918	19952	19438
2000	3006603	1006199	43904	43342
4000	12095832	4039942	95808	83960
8000	48360732	16136242	207616	166323
16000	192776817	64290937	447232	387778
32000	766544715	255578903	958464	982502

Pela análise da tabela fica evidente a eficiência do mergesort e do quicksort, ambos possuem o número de comparações e movimentações com complexidade $O(nlog_2(n))$. Já o bubblesort e a ordenação por inserção foram bem ineficiente com complexidade próxima de $O(n^2)$ para as movimentações e trocas. Tal relação pode ser evidenciada no gráfico abaixo:

Figura 16: Aleatório - Comparações(esquerda) e Movimentações(direita)



Pelas informações trazidas nas tabelas e nos gráficos podemos chegar a seguinte classificação para os algoritmos em relação a eficiência quando os termos são completamente aleatórios:

$$Quicksort > Mergesort >> Insertion > Bubblesort$$

Esse é o caso mais comum de acontecer em situações práticas e poderia dizer a instância mais importante, nesse caso ficou claro e eficiência dos algoritmos mergesort e quicksort, vale novamente destacar que apesar da grande proximidade entre a quantidade de movimentações

e comparações dos dois o merge possui a desvantagem de utilizar memória adicional. Portanto o algoritmo mais eficiente quando a instância é realmente aleatória é o quicksort.

3.2.5 Constante

Para o vetor formado apenas por 1 temos as seguinte configuração de comparações e movimentações:

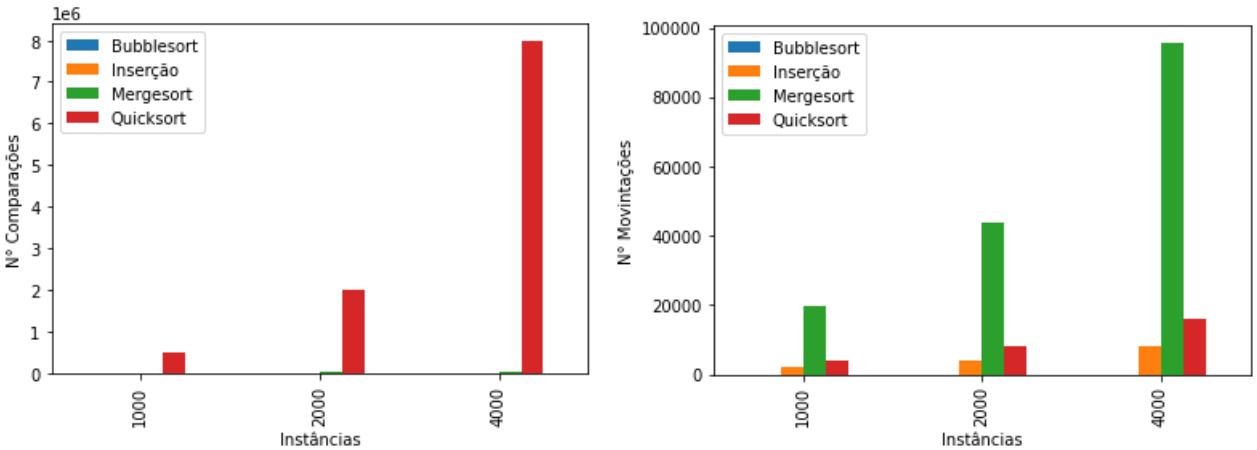
Figura 17: Constante - Comparações(esquerda) e Movimentações(direita)

	Bubble	Inserção	MergeSort	Quicksort
250	249	249	1011	31125
500	499	499	2272	124750
1000	999	999	5044	499500
2000	1999	1999	11088	1999000
4000	3999	3999	24176	7998000
8000	7999	7999	52352	31996000
16000	15999	15999	112704	127992000
32000	31999	31999	241408	511984000

	Bubble	Inserção	MergeSort	Quicksort
250	0	498	3988	996
500	0	998	8976	1996
1000	0	1998	19952	3996
2000	0	3998	43904	7996
4000	0	7998	95808	15996
8000	0	15998	207616	31996
16000	0	31998	447232	63996
32000	0	63998	958464	127996

Novamente, é perceptível a eficiência do bubblesort e da ordenação por inserção, que possuem comparações e trocas com complexidade linear. O merge novamente muito estável, não muito bom, mas também não muito ruim. Já o quicksort apresentou uma quantidade gigantesca de comparações como mais de $\frac{n(n-1)}{2}$ comparações. Na figura 18 fica explícito essas disparidades.

Figura 18: Constante - Comparações(esquerda) e Movimentações(direita)



Tivemos novamente um caso muito semelhante ao caso dos números em ordem crescente, com apenas alguns ligeiras diferenças. Ma ficou nítido a grande eficiência do bubblesort nesse tipo de cenário. Temos portanto, a seguinte classificação:

$$Bubblesort > Insertion > Mergesort > Quicksort$$

Novamente é válido pontuar que é muito pouco provável esse cenário em casos reais.

4 Conclusão

Concluímos que os resultados alcançados em sala se concretizaram, houveram algumas diferenças que se deu em virtude do que foi considerado como movimentação, que diferiu um pouco do parâmetro visto em aula, apesar disso, foi possível notar a complexidade de todos os algoritmos estavam corretos. Além disso novas informações foram descobertas, como a extrema ineficiência do quicksort em vetores ordenados ou constantes que não foi discutido tão profundamente.

Diante do que foi exposto, fica claro que existem algoritmos mais eficientes, ou menos, de acordo com as características das informações que precisam ser ordenadas, sendo que o quicksort e mergesort são os mais eficientes para situações reais onde possivelmente haverá uma lista semi-ordenado ou completamente desordenada, nesse dois casos os dois realizam o trabalho com o menor número de comparações e movimentações.

Outro ponto de muito interesse foi a estabilidade do mergesort, que funciona de maneira razoável para todas as instâncias e portanto seria o mais indicado entre os quatro se você não conhece nada das características do vetor.

Próximos passos para esse relatório, seria implementar para outros algoritmos de ordenação como selection sort, counting sort, heapsort e também implementar os algoritmos vistos acima com pequenas diferenças como a busca binária na ordenação por inserção, o uso de dois bubble sort - shakersort, usar pivô do quicksort no início e no final, ao invés de aleatório, existem várias técnicas que podem ser avaliadas e validadas com esse tipo de análise.