ANDREY GOGORA DOS SANTOS
GUSTAVO TORRES CUSTÓDIO

Algoritmos de Ordenação

BC1521 – Algoritmos e Estruturas de Dados I
Professor: Letícia Bueno

SANTO ANDRÉ – SP
2013

Apresentação e análise dos resultados

Nesse trabalho realizamos a ordenação de de vetores aleatórios de 10.000, 30.000, 90.000, 270.000, 810.000, 2.430.000, 7.290.000, 21.870.000 e 65.610.000 de posições. Realizamos 6 testes para cada número de posições com 6 seeds aleatórios diferentes (4, 81, 151, 1601, 2307 e 4207) e cronometramos o tempo. Também realizamos um teste com um vetor de 50.000 elementos invertido.

Limitamos o tempo de execução de cada algoritmo para 20 minutos. Os testes que ultrapassaram esse tempo foram apenas os testes com os algoritmos quadráticos. Para esses testes, estimamos o tempo de execução do algoritmo.

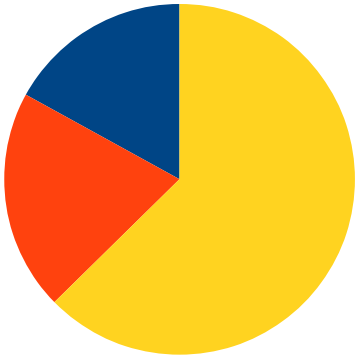
Para fazer a estimativa do tempo, consideramos que os algoritmos quadráticos têm tempo de execução n^2. Como o número de posições do vetor de teste é multiplicado por três para cada teste, o tempo de execução se torna 9 vezes maior ($3^2 = 9$).

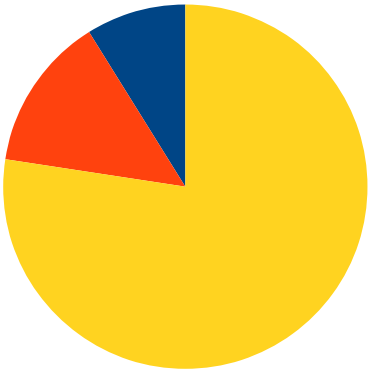
Houve a necessidade de alterar o código do método “merge” Ao invés de alocar estaticamente os vetores R e L, utilizamos ponteiros. Isso foi decidido porque, a partir do teste com o vetor de 270.000 posições o programa travava. Ao declararmos os vetores dinamicamente e depois utilizar “free” para liberar a memória, o programa deixou de travar. Concluímos que o problema era decorrente do uso excessivo do espaço de memória que não era liberado após ser usado.

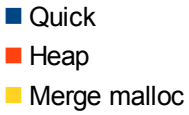
Antes de alterar o método “merge” realizamos testes com o mesmo com vetores de até 90.000 posições. Realizamos novamente esses testes após a alteração do método. Notamos que a alteração do método fez com que o tempo de execução do merge sort fosse maior. Por causa disso, usamos dois gráficos para os algoritmos eficientes: um gráfico para o merge sort estático e um para o merge sort dinâmico.

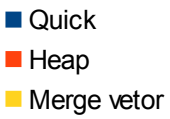
Tabelas com a média do tempo de execução de cada algoritmo dado em segundos, junto com gráficos representando a média dos tempos são mostrados abaixo:

Para um vetor de 10.000 posições:



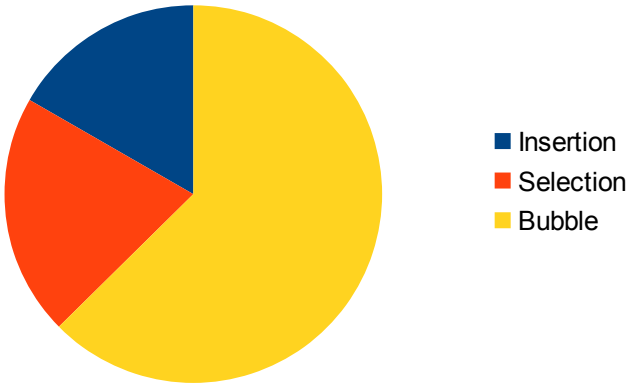


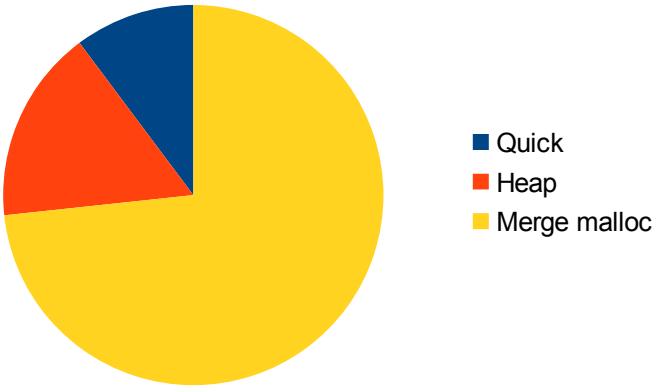


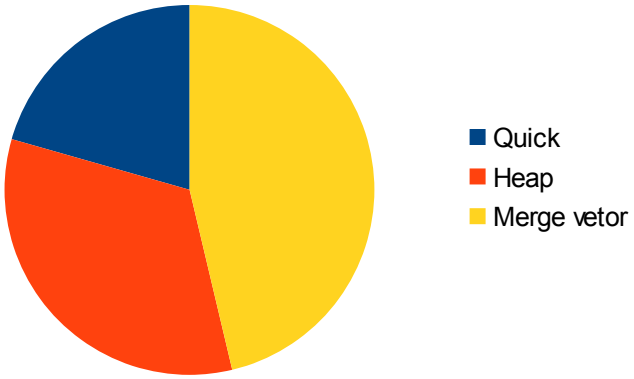


Seed	Insertion	Selection	Bubble	Quick	Heap	Merge vetor	Merge malloc
1	0.122	0.145	0.45	0.002	0.003	0.003	0.008
2	0.123	0.149	0.449	0.002	0.003	0.003	0.01
3	0.121	0.146	0.449	0.002	0.003	0.003	0.02
4	0.122	0.146	0.446	0.001	0.003	0.003	0.015
5	0.122	0.145	0.451	0.002	0.002	0.004	0.012
6	0.12	0.145	0.448	0.002	0.003	0.003	0.031
Média	***0.1216666667***	***0.146***	***0.4488333333***	***0.0018333333***	***0.0028333333***	***0.0031666667***	***0.016***

Para um vetor de 30.000 posições:

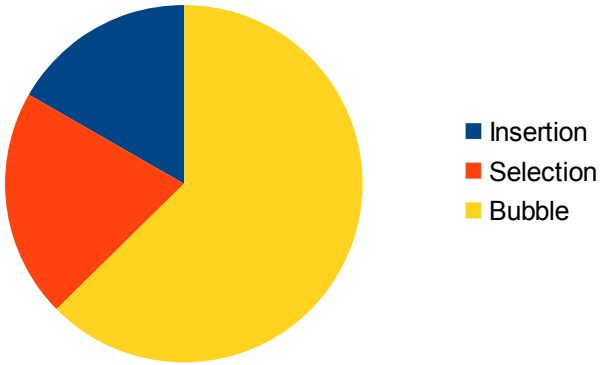


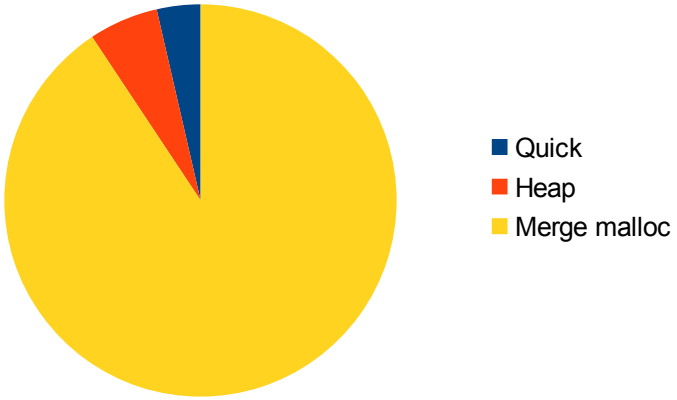


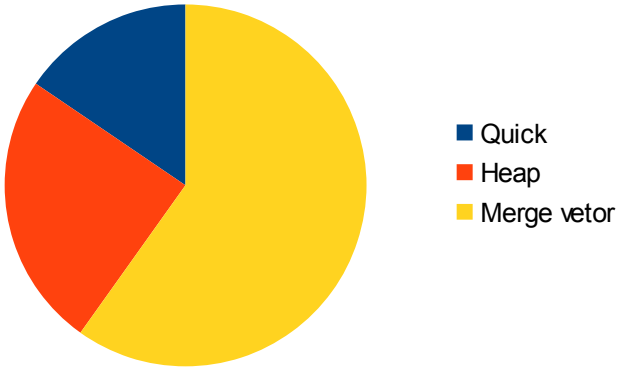


Seed	Insertion	Selection	Bubble	Quick	Heap	Merge vetor	Merge malloc
1	1.069	1.373	4.042	0.006	0.01	0.013	0.09
2	1.079	1.332	4.198	0.006	0.009	0.015	0.058
3	1.129	1.339	4.034	0.006	0.01	0.013	0.027
4	1.079	1.341	4.044	0.006	0.01	0.013	0.027
5	1.075	1.343	4.071	0.006	0.009	0.013	0.028
6	1.079	1.341	4.029	0.006	0.01	0.014	0.028
Média	***1.085***	***1.3448333333***	***4.0696666667***	***0.006***	***0.0096666667***	***0.0135***	***0.043***

Para um vetor de 90.000 posições:



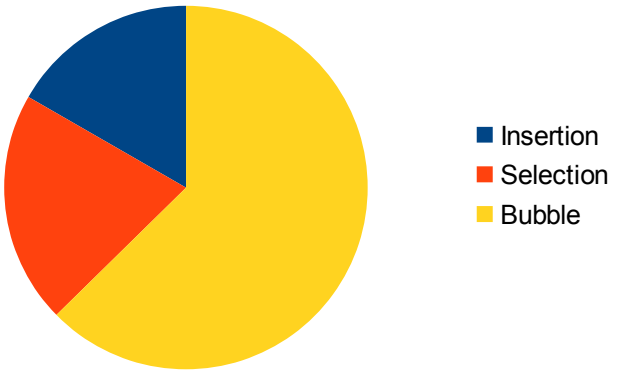


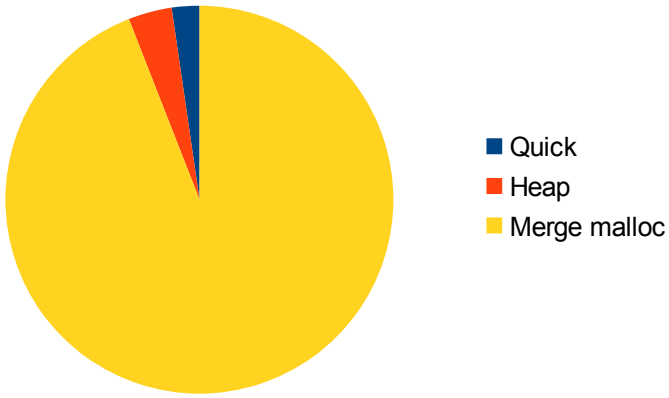


Seed	Insertion	Selection	Bubble	Quick	Heap	Merge vetor	Merge malloc
1	9.627	12.076	37.104	0.019	0.032	0.075	0.495
2	9.648	12.069	36.532	0.02	0.033	0.082	0.724
3	9.678	12.054	36.635	0.021	0.032	0.078	0.451
4	9.799	12.196	36.71	0.021	0.033	0.079	0.468
5	9.817	12.053	36.289	0.02	0.031	0.075	0.462
6	9.968	12.042	36.301	0.02	0.031	0.078	0.447
Média	***9.7561666667***	***12.0816666667***	***36.5951666667***	***0.0201666667***	***0.032***	***0.0778333333***	***0.5078333333***

Para um vetor de 270.000 posições:

A partir desse teste utilizamos o merge sort com alocação dinâmica de vetores.

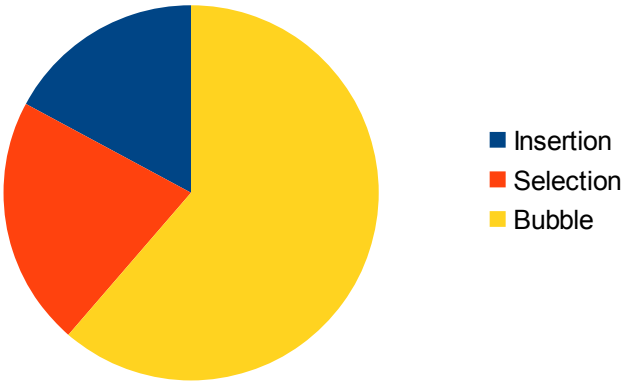


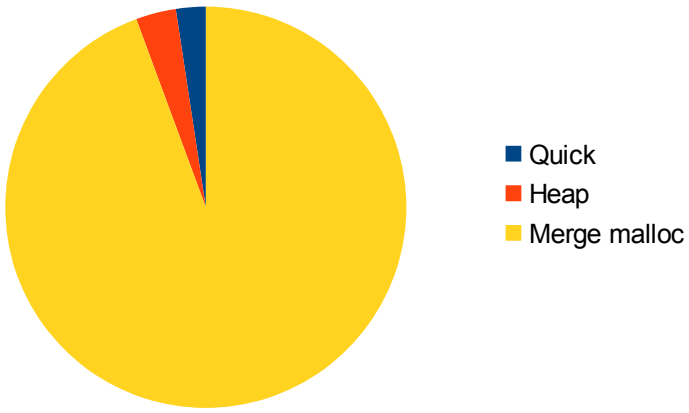


Seed	Insertion	Selection	Bubble	Quick	Heap	Merge malloc
1	87.438	109.774	313.891	0.069	0.109	2.39
2	87.946	109.883	315.426	0.068	0.109	2.933
3	87.644	109.797	313.598	0.07	0.107	2.869
4	87.715	110.212	314.948	0.069	0.108	2.868
5	88.366	109.356	314.59	0.068	0.108	2.946
6	87.341	110.304	313.726	0.069	0.111	2.87
Média	***87.7416666667***	***109.8876666667***	***314.3631666667***	***0.0688333333***	***0.1086666667***	***2.8126666667***

Para um vetor de 810.000 posições:

A partir desse teste, o tempo de execução do bubble sort ultrapassou os 20 minutos, portanto, estimamos o tempo de execução do mesmo.

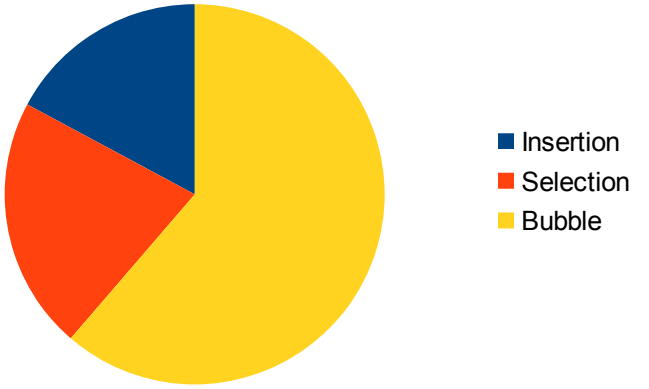


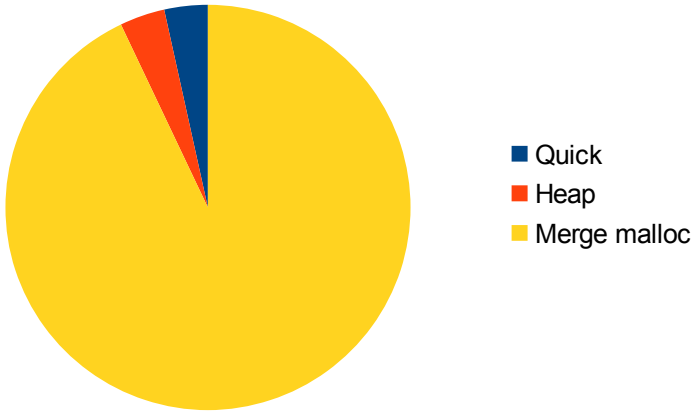


Seed	Insertion	Selection	Bubble	Quick	Heap	Merge malloc
1	792.09	997.376	2825.019	0.27	0.379	10.903
2	792.15	995.236	2838.834	0.282	0.364	10.681
3	792.303	994.778	2822.382	0.265	0.365	10.848
4	794.641	996.541	2834.532	0.287	0.401	11.456
5	791.186	996.361	2831.31	0.271	0.372	10.702
6	782.965	978.418	2823.534	0.276	0.366	10.759
Média	***790.8891666667***	***993.1183333333***	***2829.2685***	***0.2751666667***	***0.3745***	***10.8915***

Para um vetor de 2.430.000 posições:

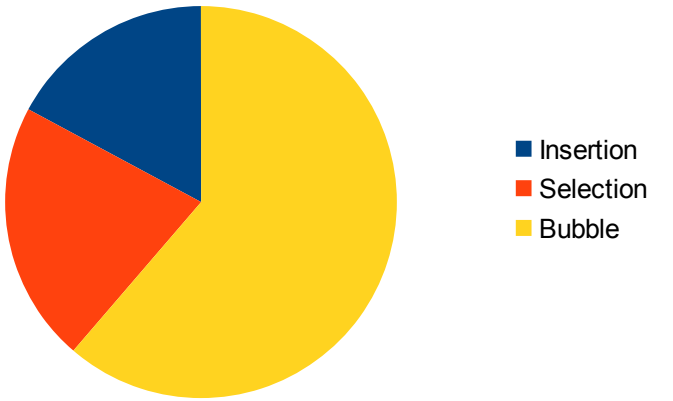
A partir desse teste, os tempos de execução do insertion sort e do selection sort ultrapassaram os 20 minutos determinados, então estimamos o tempo de execução dos mesmos.

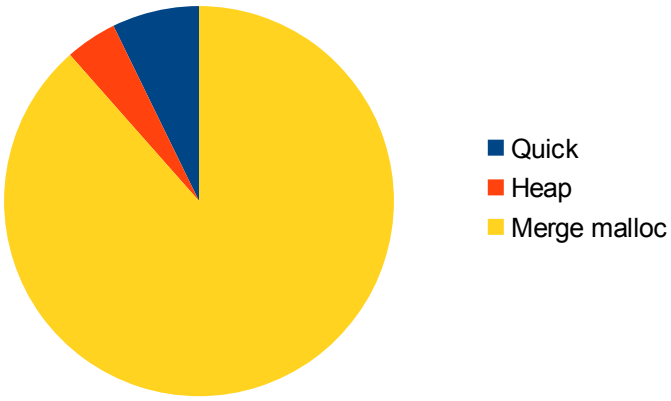




Seed	Insertion	Selection	Bubble	Quick	Heap	Merge malloc
1	7128.81	8976.384	25425.171	1.325	1.385	36.388
2	7129.35	8957.124	25549.506	1.33	1.384	35.483
3	7130.727	8953.002	25401.438	1.363	1.391	35.83
4	7151.769	8968.869	25510.788	1.336	1.399	35.229
5	7120.674	8967.249	25481.79	1.323	1.379	36.421
6	7046.685	8805.762	25411.806	1.352	1.403	36.179
Média	***7118.0025***	***8938.065***	***25463.4165***	***1.3381666667***	***1.3901666667***	***35.9216666667***

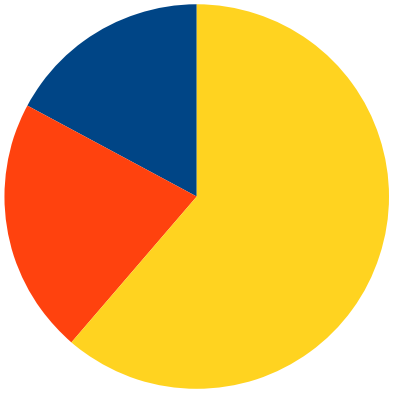
Para um vetor de 7.290.000 posições:

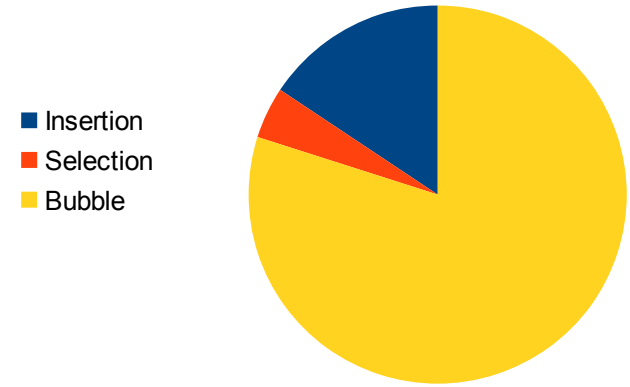




Seed	Insertion	Selection	Bubble	Quick	Heap	Merge malloc
1	64159.29	80787.456	228826.539	8.602	5.289	114.249
2	64164.15	80614.116	229945.554	8.621	5.304	110.583
3	64176.543	80577.018	228612.942	10.258	5.275	110.927
4	64365.921	80719.821	229597.092	9.832	5.459	111.101
5	64086.066	80705.241	229336.11	8.706	5.779	110.669
6	63420.165	79251.858	228706.254	8.692	5.322	112.405
Média	***64062.0225***	***80442.585***	***229170.7485***	***9.1185***	***5.4046666667***	***111.6556666667***

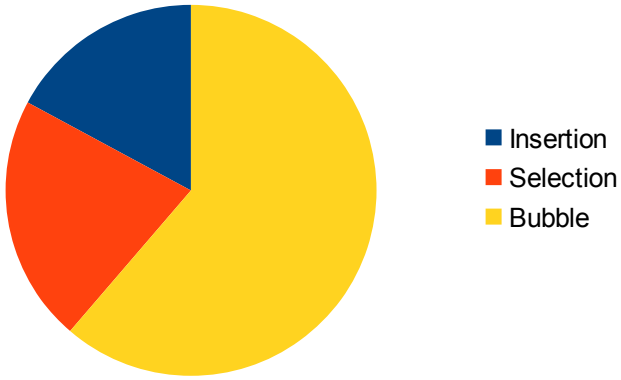
Para um vetor de 21.870.000 posições:

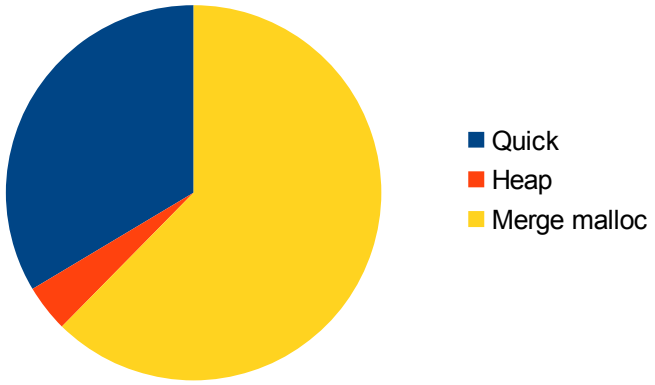




Seed	Insertion	Selection	Bubble	Quick	Heap	Merge malloc
1	577433.61	727087.104	2059438.851	69.302	20.097	348.701
2	577477.35	725527.044	2069509.986	68.183	18.935	351.881
3	577588.887	725193.162	2057516.478	68.694	18.948	350.473
4	579293.289	726478.389	2066373.828	68.438	19.562	347.565
5	576774.594	726347.169	2064024.99	68.207	19.087	353.374
6	570781.485	713266.722	2058356.286	68.417	19.329	348.462
Média	***576558.2025***	***723983.265***	***2062536.7365***	***68.5401666667***	***19.3263333333***	***350.076***

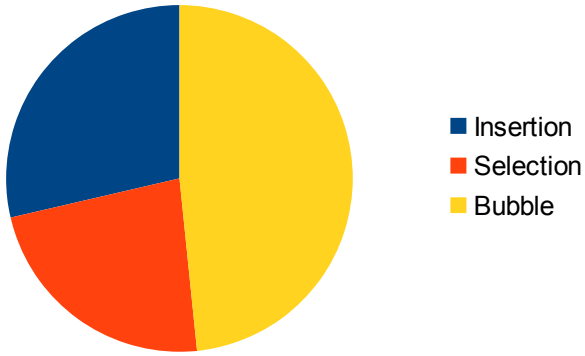
Para um vetor de 65.610.000 posições:

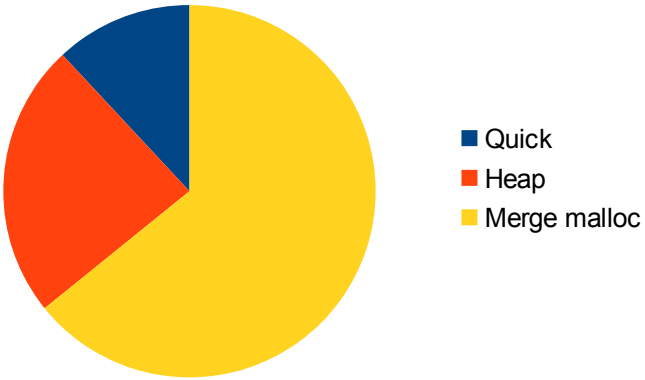


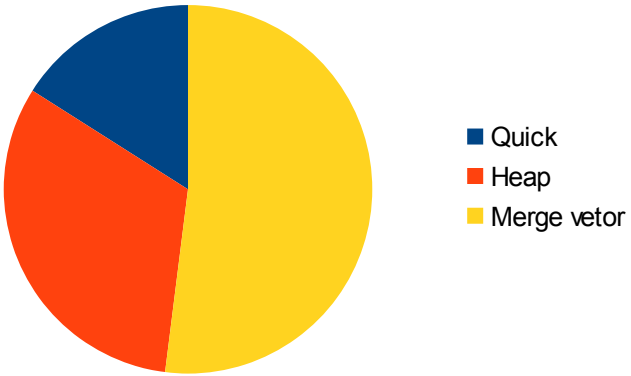


Seed	Insertion	Selection	Bubble	Quick	Heap	Merge malloc
1	5196902.49	6543783.936	18534949.659	581.157	69.539	1082.23
2	5197296.15	6529743.396	18625589.874	583.095	70.827	1086.39
3	5198299.983	6526738.458	18517648.302	582.239	70.535	1080.28
4	5213639.601	6538305.501	18597364.452	583.297	70.352	1083.61
5	5190971.346	6537124.521	18576224.91	580.866	70.773	1078.21
6	5137033.365	6419400.498	18525206.574	582.153	70.612	1080.4
Média	***5189023.8225***	***6515849.385***	***18562830.6285***	***582.1345***	***70.4396666667***	***1081.853333333***

Para o vetor de 50.000 posições invertido:







Seed	Insertion	Selection	Bubble	Quick	Heap	Merge vetor	Merge malloc
Média	***6.058***	***4.863***	***10.234***	***0.008***	***0.016***	***0.026***	***0.043***

Os dados foram separados em dois grupos : os algoritmos eficientes e os quadráticos. Isso porque os algoritmos quadráticos possuem tempo de execução muito superior ao dos eficientes. Assim, em um gráfico que contém o tempo de todos os algoritmos é difícil visualizar o tempo dos algoritmos eficientes.

Em todos os testes, os algoritmos eficientes possuem tempo de execução menor do que os quadráticos. Para os algoritmos quadráticos, o bubble sort é o algoritmo mais lento em todos os casos. O insertion sort é mais eficiente do que o selection sort em todos os casos, exceto no caso de entrada invertida.

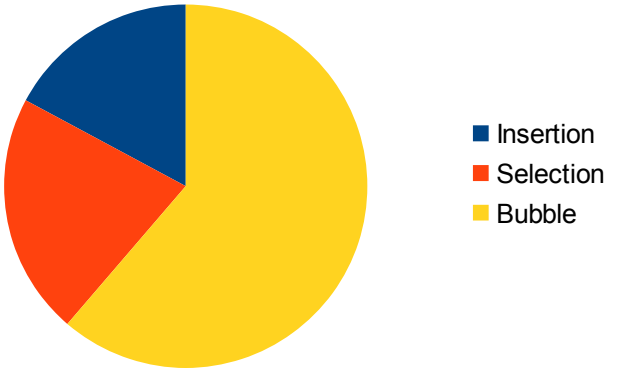
O quick sort se mostra o algoritmo de ordenação mais eficiente até o teste com 7.290.000 elementos. A partir desse teste, o algoritmo com menor tempo de execução é o heap sort. No entanto, para a entrada invertida o heap sort não é tão eficiente e tem tempo de execução maior do que o esperado para outra entrada.

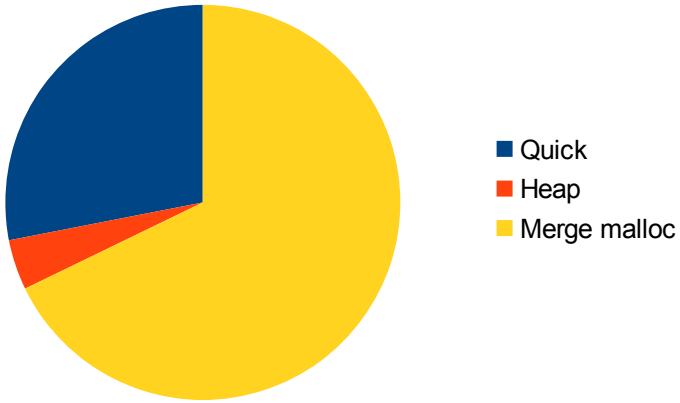
Pode-se observar que conforme o número de posições do vetor aumenta, o merge sort tem tempo de execução cada vez maior em relação aos outros algoritmos de ordenação eficientes até certo ponto. Mas a partir do teste de 810.000 elementos, o tempo de execução do mergesort para entradas maiores cresce menos do que o dos outros algoritmos eficientes de ordenação.

Além da média agrupada por número de elementos, fizemos também uma média do tempo de execução agrupada pelo seed.

Abaixo temos as tabelas e gráficos correspondentes:

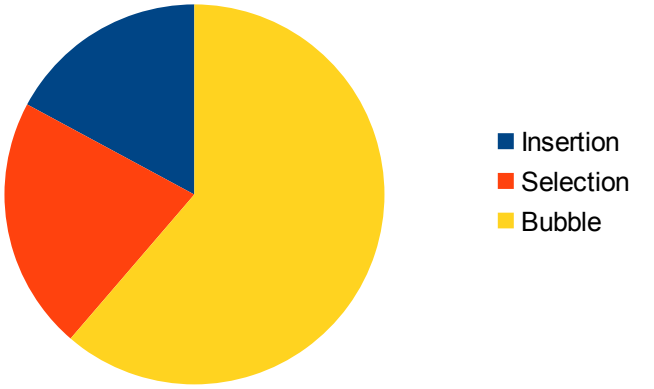
Para o seed 1:

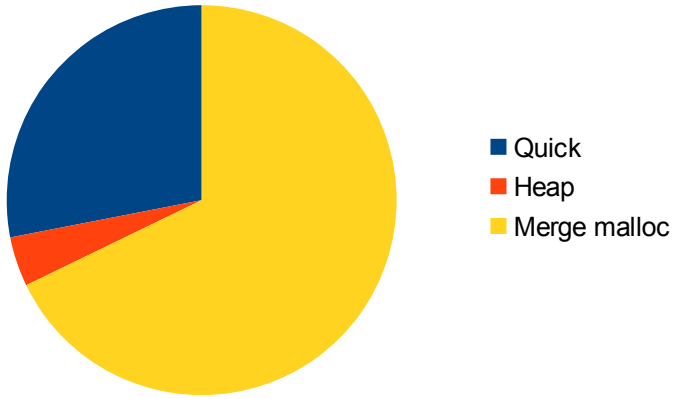




N elementos	Insertion	Selection	Bubble	Quick	Heap	Merge malloc
10000	0.122	0.145	0.45	0.002	0.003	0.008
30000	1.069	1.373	4.042	0.006	0.01	0.09
90000	9.627	12.076	37.104	0.019	0.032	0.495
270000	87.438	109.774	313.891	0.069	0.109	2.39
810000	792.09	997.376	2829.2685	0.27	0.379	10.903
2430000	7118.0025	8938.065	25434.4165	1.325	1.385	36.388
7290000	64062.0225	80442.585	228909.7485	8.602	5.289	114.249
21870000	576558.2025	723983.265	2060187.737	69.302	20.097	348.701
65610000	5189023.823	6515849.385	18541689.63	581.157	69.539	1082.23
Média	***648628.04406***	***814481.56044***	***2317711.8097***	***73.416888889***	***10.760333333***	***177.2726666667***

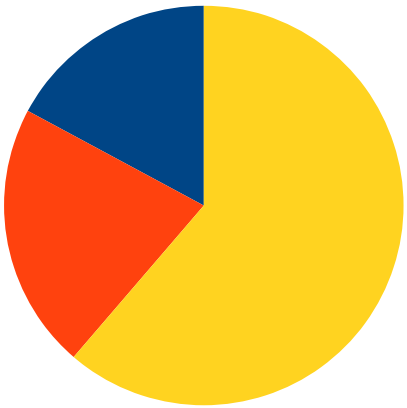
Para o seed 2:

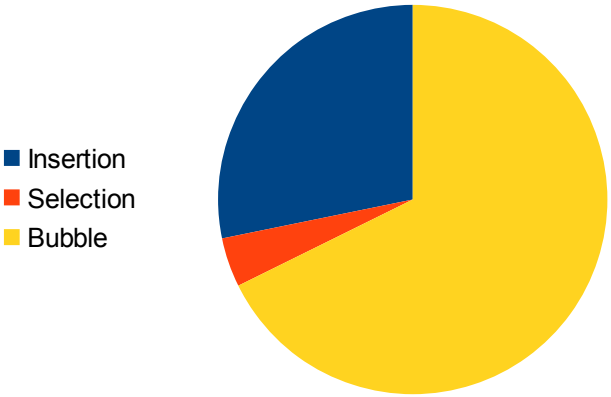




N elementos	Insertion	Selection	Bubble	Quick	Heap	Merge malloc
10000	0.123	0.149	0.449	0.002	0.003	0.01
30000	1.079	1.332	4.198	0.006	0.009	0.058
90000	9.648	12.069	36.532	0.02	0.033	0.724
270000	87.946	109.883	315.426	0.068	0.109	2.933
810000	792.15	995.236	2829.2685	0.282	0.364	10.681
2430000	7118.0025	8938.065	25434.4165	1.33	1.384	35.483
7290000	64062.0225	80442.585	228909.7485	8.621	5.304	110.583
21870000	576558.2025	723983.265	2060187.737	68.183	18.935	351.881
65610000	5189023.823	6515849.385	18541689.63	583.095	70.827	1086.39
Média	***648628.11072***	***814481.32989***	***2317711.9339***	***73.511888889***	***10.774222222***	***177.6381111111***

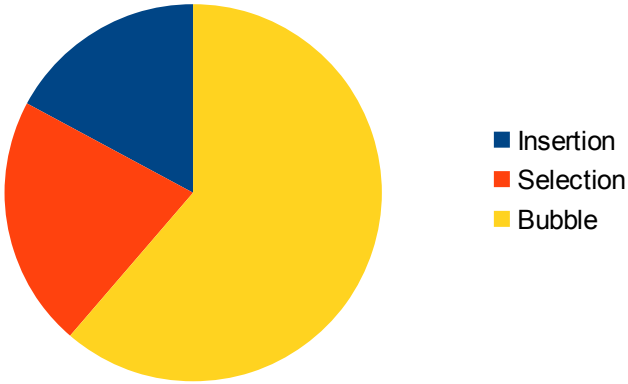
Para o seed 3:

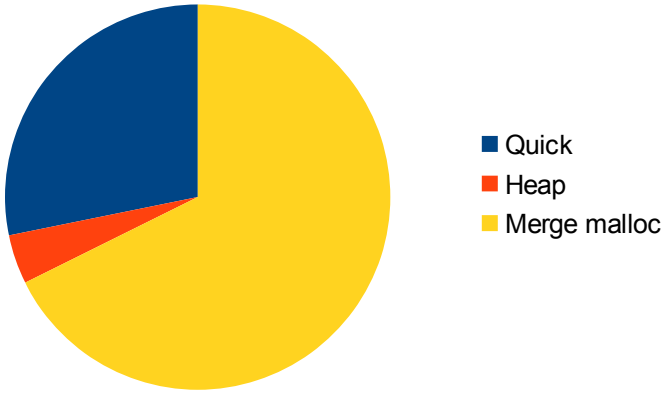




N elementos	Insertion	Selection	Bubble	Quick	Heap	Merge malloc
10000	0.121	0.146	0.449	0.002	0.003	0.02
30000	1.129	1.339	4.034	0.006	0.01	0.027
90000	9.678	12.054	36.635	0.021	0.032	0.451
270000	87.644	109.797	313.598	0.07	0.107	2.869
810000	792.303	994.778	2829.2685	0.265	0.365	10.848
2430000	7118.0025	8938.065	25434.4165	1.363	1.391	35.83
7290000	64062.0225	80442.585	228909.7485	10.258	5.275	110.927
21870000	576558.2025	723983.265	2060187.737	68.694	18.948	350.473
65610000	5189023.823	6515849.385	18541689.63	582.239	70.535	1080.28
Média	***648628.10283***	***814481.26822***	***2317711.7241***	***73.657555556***	***10.740666667***	***176.8583333333***

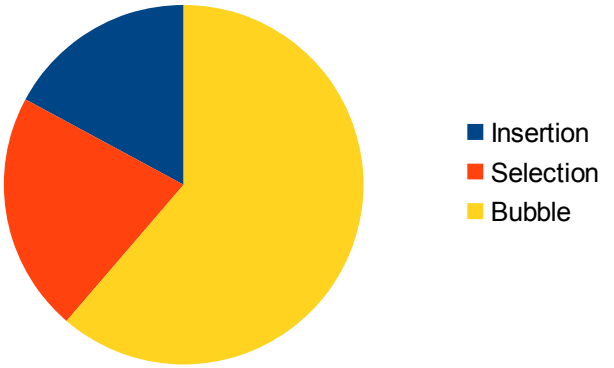
Para o seed 4:

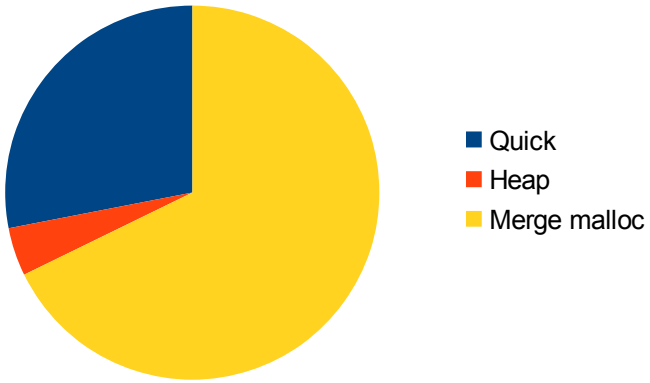




N elementos	Insertion	Selection	Bubble	Quick	Heap	Merge malloc
10000	0.122	0.146	0.446	0.001	0.003	0.015
30000	1.079	1.341	4.044	0.006	0.01	0.027
90000	9.799	12.196	36.71	0.021	0.033	0.468
270000	87.715	110.212	314.948	0.069	0.108	2.868
810000	794.641	996.541	2829.2685	0.287	0.401	11.456
2430000	7118.0025	8938.065	25434.4165	1.336	1.399	35.229
7290000	64062.0225	80442.585	228909.7485	9.832	5.459	111.101
21870000	576558.2025	723983.265	2060187.737	68.438	19.562	347.565
65610000	5189023.823	6515849.385	18541689.63	583.297	70.352	1083.61
Média	***648628.3785***	***814481.52622***	***2317711.8832***	***73.698555556***	***10.814111111***	***176.9265555556***

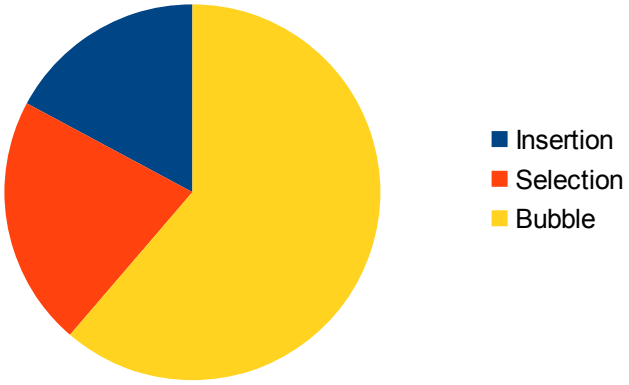
Para o seed 5:

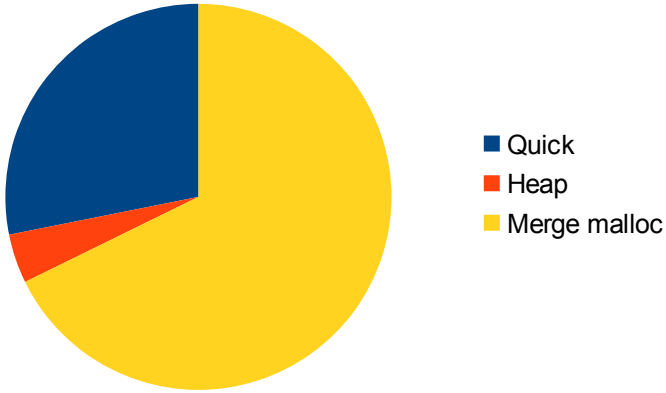




N elementos	Insertion	Selection	Bubble	Quick	Heap	Merge malloc
10000	0.122	0.145	0.451	0.002	0.002	0.012
30000	1.075	1.343	4.071	0.006	0.009	0.028
90000	9.817	12.053	36.289	0.02	0.031	0.462
270000	88.366	109.356	314.59	0.068	0.108	2.946
810000	791.186	996.361	2829.2685	0.271	0.372	10.702
2430000	7118.0025	8938.065	25434.4165	1.323	1.379	36.421
7290000	64062.0225	80442.585	228909.7485	8.706	5.779	110.669
21870000	576558.2025	723983.265	2060187.737	68.207	19.087	353.374
65610000	5189023.823	6515849.385	18541689.63	580.866	70.773	1078.21
Média	***648628.0685***	***814481.39533***	***2317711.8002***	***73.274333333***	***10.837777778***	***176.9804444444***

Para o seed 6:





N elementos	Insertion	Selection	Bubble	Quick	Heap	Merge malloc
10000	0.12	0.145	0.448	0.002	0.003	0.031
30000	1.079	1.341	4.029	0.006	0.01	0.028
90000	9.968	12.042	36.301	0.02	0.031	0.447
270000	87.341	110.304	313.726	0.069	0.111	2.87
810000	782.965	978.418	2829.2685	0.276	0.366	10.759
2430000	7118.0025	8938.065	25434.4165	1.352	1.403	36.179
7290000	64062.0225	80442.585	228909.7485	8.692	5.322	112.405
21870000	576558.2025	723983.265	2060187.737	68.417	19.329	348.462
65610000	5189023.823	6515849.385	18541689.63	582.153	70.612	1080.4
Média	***648627.05817***	***814479.50556***	***2317711.7005***	***73.443***	***10.798555556***	***176.8423333333***

Para todos os seeds, existe pouca diferença na visualização dos gráficos.

Os gráficos estão divididos entre os algoritmos quadráticos e os eficientes. Todos os eficientes têm tempo de execução menor do que todos os quadráticos.

Analisando, os resultados, é possível perceber que o algoritmo que apresenta a menor média de tempo de execução é o heap sort e o que apresenta maior tempo é o bubble sort. Colocando todos os algoritmos em ordem crescente pela média do tempo de execução temos: heap sort, quick sort, merge sort, insertion sort, selection sort e bubble sort.

Conclusão

A partir de um número grande de entradas, todos os algoritmos eficientes (quick sort, merge sort e heap sort) são mais eficientes do que os algoritmos quadráticos (bubble sort, selection sort, insertion sort). Quando precisamos ordenar uma grande quantidade de dados, o ideal é utilizar um algoritmo eficiente.

Para entradas muito grandes, o merge sort parece ter um tempo de execução cada vez menor em relação aos outros algoritmos. Faz sentido concluir que para um vetor muito maior do que os utilizados, ele pode ser o algoritmo mais eficiente.

O tempo de execução do quick sort cresce muito para vetores muito grandes, mas até certo ponto, ele é o algoritmo mais eficiente. O heap sort é o algoritmo eficiente que apresentou menor tempo de execução médio, no entanto, não é eficiente para entradas invertidas. O insertion sort é outro algoritmo que foi pouco eficiente com uma entrada invertida.

Pela análise feita, podemos concluir que nenhum dos algoritmos de ordenação apresentados é melhor para qualquer caso. Para escolher o melhor algoritmo deve-se considerar o número de elementos que queremos ordenar e se teremos entrada invertida.